Thèse de Doctorat de l'Université Paris VI

Spécialité Informatique

Présentée par Alexandre Fenyo
Pour obtenir le grade de
Docteur de l'Université Paris VI

Conception et réalisation d'un noyau de communication bâti sur la primitive d'écriture distante, pour machines parallèles de type «grappe de PCs»

Soutenue le 5 Juillet 2001,
devant le jury composé de

Isabelle Demeure	Rapporteur
Bernard Tourancheau	Rapporteur
Paul Feautrier	Examinateur
Alain Greiner	Examinateur
Philippe Lalevée	Examinateur
Jean-Louis Pazat	Examinateur

Résumé

Cette thèse présente l'étude et la réalisation de MPC-OS, un noyau de communication pour machines parallèles de type «grappe de PCs», bâti sur la primitive d'écriture distante.

Plus particulièrement, elle présente un empilement de protocoles implémentés dans le noyau Unix et accessibles autant aux applications en mode utilisateur qu'aux modules internes du système d'exploitation. Basé sur une primitive de DMA matérielle rudimentaire sur réseau d'interconnexion Gigabit, MPC-OS fournit des services à forte valeur ajoutée : communications sur canaux virtuels, sécurisation des transmissions, échanges de données entre espaces de mémoire virtuelle et gestion mémoire avec garanties d'intégrité des tampons de communication. La difficulté a consisté à concevoir ces services sans copie de tampon, pour une efficacité maximale.

L'allocation dynamique des ressources est externalisée auprès d'un gestionnaire distribué, basé sur un cœur d'ORB *multi-thread* développé spécifiquement pour MPC-OS.

Après une exploration détaillée des performances logicielles, on engage alors une approche nouvelle du problème de la sécurisation : une étude stochastique du phénomène de double-faute nous permet de montrer que, sous certaines conditions, un protocole de correction simpliste permet d'obtenir simultanément de bonnes performances et une sécurisation satisfaisante.

MPC-OS équipe aujourd'hui plusieurs plate-formes MPC réparties dans différentes universités de France. Il a permis de développer des portages optimisés des environnements de programmation parallèle PVM et MPI sur la machine MPC, et a été utilisé par des industriels (GEC Marconi Aerospace Systems et Parsytec Computer) pour la validation des développements matériels de machines construites à partir des mêmes composants que la plate-forme MPC.

Mots-clés

machine parallèle, environnement de programmation, écriture distante, sécurisation, gestion mémoire, allocation dynamique de ressources

Abstract

This Ph.D Thesis presents the design and construction of MPC-OS, a communication kernel built on the remote-write primitive and dedicated to parallel computers made of clusters of workstations.

Moreover, it discusses a stack of protocols implemented inside the Unix kernel, and available either to the user-level processes or to the operating system internal modules. Based on a simple Direct Memory Access hardware primitive for Gigabit interconnect networks, MPC-OS provides many value-added services : communications through virtual channels, reliable transmissions, data exchanges between virtual memory areas and memory management preserving the integrity of communication buffers. In order to get as much efficiency as possible, zero-copy mechanisms were used for all those services. This constraint represents the major difficulty of this work.

The dynamic allocation of resources is handled by a distributed manager, based on a multi-threaded Object Request Broker specially designed for MPC-OS.

After a detailed survey of the software performances, we studied the transmission reliability issue through a new approach : a probabilistic model in fault coupling allowed us to show that, if we fulfil some basic requirements, an elementary protocol may achieve both good performances and significant reliability.

Today, MPC-OS runs on various MPC clusters and is spread out over several French universities. It has allowed the optimised port of the major parallel programming environments such as PVM and MPI to the MPC computer. Some computer manufacturers (GEC Marconi Aerospace Systems and Parsytec Computer) have used it to validate their hardware developments based on hardware components of the MPC computer.

Keywords

parallel computer, cluster of workstations, programming environment, remote write, reliability, memory management, dynamic resource allocation

Remerciements

Je tiens tout d'abord à exprimer tous mes remerciements à Alain Greiner, Professeur à l'Université Paris VI et directeur du thème ASIM du LIP6, qui m'a accueilli au sein de son équipe en 1995, et qui a dirigé ma thèse. J'ai à cœur de lui signifier toute ma gratitude, notamment car il a toujours été disponible pour m'accompagner au cours de mes recherches. Ses conseils avisés et son constant suivi du déroulement de mes travaux m'ont été d'une aide plus que précieuse.

J'ai eu un grand plaisir à travailler avec les membres de l'équipe MPC du laboratoire ASIM, qui m'ont tous, à un moment ou à un autre, aidé à avancer dans mes travaux. En plus de cela, j'ai découvert auprès d'eux différentes personnalités toutes aussi attachantes et enrichissantes les unes que les autres, et je tiens donc à remercier Franck Wajsbürt, Vincent Reibaldi, Jean-Lou Desbarbieux, Cyril Spasevski, Amal Zerrouki, Emmanuel Coulon, Fabricio Silva, Olivier Glück et Frédéric Potter.

En outre, Franck, Jean-Lou et Cyril, géniaux concepteurs de quelques-uns des circuits de la machine MPC, ont toujours été disponibles pour m'apporter leurs lumières et compétences sur le fonctionnement du matériel spécifique, je ne les en remercierai jamais assez. Olivier Glück continue les développements logiciels de MPC, et j'ai eu avec lui de nombreux et fructueux échanges, qu'il en soit remercié.

J'ai encadré plusieurs étudiants de DEA et de Maîtrise dans le cadre de MPC-OS, et je les remercie de leur implication dans ce projet. Je ne peux tous les citer ici, mais je tiens néanmoins à remercier précisément Karim Mana, qui a travaillé de près sur les couches logicielles bas-niveau de MPC-OS et qui a effectué un travail particulièrement intéressant.

Mes travaux sur MPC-OS n'auraient bien sûr pas pu voir le jour en dehors du projet MPC, qui rassemble différentes équipes de recherche au sein d'un projet commun. Je remercie pour cela les différents membres de ces équipes et notamment ceux du PRiSM, de l'ENST, du LARIA, de l'INT et des thèmes ASIM et SRC du LIP6.

Plus particulièrement, je tiens à remercier Frédéric Potter et Pierre David pour leur participation à la conception des couches logicielles PUT et SLR/P, et Philippe Cadinot qui, en tant qu'utilisateur des couches logicielles MPC, s'est penché de très près sur leur fonctionnement. Ses remarques ont été de précieuses sources de réflexion.

Jean Jacod, directeur du DEA de probabilités de Paris VI, a pris le temps de relire le chapitre d'analyse stochastique : ses conseils et remarques judicieuses m'ont permis d'en améliorer sensiblement la forme et le contenu, et je l'en remercie.

Agnès, ma douce moitié, m'a non seulement soutenu tout au long de la rédaction, mais le réel intérêt qu'elle a porté à cette étude, et les discussions passionnées qui en ont découlé, m'ont beaucoup apporté. Je l'en remercie pour tout cela, ainsi que pour son fastidieux travail de vérification des calculs qui figurent dans le chapitre d'analyse stochastique.

J'ai enfin une pensée particulière pour Jérôme et Caroline qui m'ont toujours soutenu dans mes choix.

alex@fenyo.net
http://www.fenyo.net

mpc@mpc.lip6.fr
ftp://mpc.lip6.fr
http://mpc.lip6.fr

Table des matières

Table des figures

La science n'est peut-être que la forme
la plus élaborée de la littérature fantastique.
J-L. Borgès

Chapitre 1

Introduction

Sommaire

Le travail présenté dans cette thèse s'inscrit dans le cadre du projet de recherche MPC initié en 1995 à l'Université Pierre et Marie Curie. Il consiste à réaliser et à assembler les composantes autant matérielles que logicielles d'une machine parallèle à bas coût. Aujourd'hui, de nombreuses équipes participent au groupe de recherche MPC ; elles sont notamment issues du LIP6[1], de l'ENST[2], du PRiSM de l'Université de Versailles, du Laria[3] de l'Université de Picardie Jules Vernes, et de l'INT[4] d'Évry.

1.1 Un parallélisme nécessaire

Depuis les débuts de l'informatique, l'amélioration des performances matérielles et logicielles constitue un des objectifs majeurs de la recherche. En 1965, Gordon Moore fit une découverte mémorable [Moore, 1965] : il constata, en traçant la courbe de croissance des performances des microprocesseurs, que chaque génération de puce était à la fois deux fois plus puissante pour un délai de développement variant entre 18 et 24 mois.

Cette loi est toujours valable de nos jours : voilà seulement cinq ans, la fréquence d'un processeur grand public était de 100 MHz alors que son descendant est aujourd'hui cadencé à 1 GHz. Dans quelques mois, cette quantité sera obsolète. Bien sûr, on ne se contente pas d'accroître la puissance processeur des machines : les performances de la mémoire, des périphériques de stockage ainsi que des interfaces de communication suivent une progression semblable.

Et pourtant, la nécessité d'accroître les performances reste un problème primordial. Par exemple, la question des prévisions météorologiques, qui concerne tout-un-chacun, et qui consiste à prévoir les valeurs futures de la fonction *climat* à partir de l'observation de ses valeurs à un instant connu, est aujourd'hui résolue avec une prédiction à 7 jours, calculée en 24 heures : elle nécessite pour cela plus de 50 Gflops et une masse importante de données à traiter. Seules des machines parallèles peuvent aujourd'hui résoudre ce problème.

1.2 Des architectures parallèles variées

Depuis les années 50, date des premières tentatives de construction de machines parallèles, des architectures très diverses ont été proposées. On peut en citer quelques exemples :

✓ *Les machines vectorielles multi-processeurs* : elles disposent de processeurs puissants en petit nombre, ainsi que d'une mémoire partagée accessible depuis tous les processeurs. On peut citer parmi elles la Cray C98 [Cray Systems, 1993] à 8 processeurs vectoriels.

1. Laboratoire d'informatique de Paris 6
2. École nationale supérieure des télécommunications
3. Laboratoire de recherche en informatique d'Amiens
4. Institut national des télécommunications

✓ *Les machines multi-processeurs à mémoire distribuée* : constituées d'un grand nombre de processeurs ordinaires (jusqu'à 1024, voire plus), la mémoire y est répartie sur les différents processeurs.
La Paragon d'Intel [Cook, 1993] et la SP-2 d'IBM [Stunkel *et al.*, 1995] (jusqu'à 512 processeurs) en sont des représentants typiques.

✓ *Les machines synchrones* : elles disposent d'un très grand nombre de processeurs de faible puissance (jusqu'à 65536 processeurs 1 bit pour la CM-2 de Thinking Machines [Thinking Machines, 1998]). Ils sont commandés par un unique séquenceur et exécutent de manière synchrone une même instruction sur des données locales différentes.

✓ *Les machines SMP* : elles sont composées de 2 à quelques dizaines de processeurs CISC ou RISC identiques, interconnectés à une mémoire partagée centralisée et le matériel met en œuvre un mécanisme de cohérence de cache. Il en existe à base de carte mère Intel grand public, comme par exemple la récente Asustek P3C-D organisée autour du *chipset* Intel i820 et qui supporte jusqu'à deux processeurs Pentium III à 733 MHz. Aujourd'hui, la plupart des systèmes d'exploitation commerciaux et libres supportent de telles architectures. Leurs performances sont principalement liées à la gestion du verrouillage des structures noyau [Kaieda *et al.*, 2001], et parmi les Unix libres, Linux s'est énormément amélioré de ce point de vue récemment [Bryant *et al.*, 2000].

Les processeurs sont interconnectés par un réseau de communication spécialisé. Des topologies très diverses ont vu le jour. On peut par exemple citer l'architecture en tore de dimension trois de la Cray T3D [Cray Systems, 2000], le réseau à trois niveaux d'anneaux de la KSR-1 [Ramachandran *et al.*, 1996], la grille de processeurs de la Paragon, le réseau multi-étages de la SP-2, le réseau *fat-tree* de la CM-5, etc. ([Hwang *et al.*, 1997] compare les performances des SP2, T3D et Paragon).

1.3 Le modèle NOW

Face à ces machines parallèles complexes à réaliser et aux coûts prohibitifs, les réseaux de stations de travail offrent un rapport coût/performance très avantageux. C'est à partir de cette simple constatation qu'est né le projet NOW [Anderson *et al.*, 1995], à l'Université de Berkeley, USA.

Il consiste à utiliser un grand nombre de stations de travail individuelles (quelques centaines, voire plus), à les raccorder par un réseau rapide, et à y installer une bibliothèque de communication performante, afin d'obtenir des performances analogues à celles d'une machine massivement parallèle, à bas coût.

L'expérimentation NOW à Berkeley a pour objectif de rassembler 100 Ultra Sparc et 40 Sparc Stations Sun sous Solaris, 35 PC sous NT et Unix, 300 stations de travail HP, et entre 500 et 1000 disques, le tout connecté par un réseau de commutateurs Myrinet.

1.4 La machine MPC

Le projet MPC, initié en 1995 à l'Université Pierre et Marie Curie, s'inscrit dans la philosophie du projet NOW, et consiste à construire une machine massivement parallèle à faible coût à partir de nœuds de calcul standards.

La machine MPC [Greiner *et al.*, 1998], en configuration 4 nœuds bi-processeurs sur la figure 1.1 page ci-contre, se décompose ainsi :

▷ **Un ensemble de nœuds de calcul**, constitué de quelques dizaines à quelques centaines de cartes mères standards, chacune équipée d'un processeur Intel grand public, d'une mémoire locale de plusieurs centaines de Mo, d'un périphérique de stockage de masse local de quelques Go et d'un bus d'interface PCI 32 bits/33 MHz (il existe des versions plus rapides de l'interface PCI [Prakash, 2000]) ;

▷ **Un réseau de communication Gigabit** [Potter, 1996] constitué de l'interconnexion de cartes d'interface FastHSL. Chaque nœud de calcul comporte une de ces cartes, qui dispose de 7 liens Gigabits bidirectionnels vers le réseau et d'un accès direct à la mémoire locale à travers le bus PCI ;

▷ **Un réseau de contrôle** bas débit, raccordant l'ensemble des nœuds de calcul ainsi qu'une console ;

▷ **Un environnement logiciel** composé comme suit :

- Le système d'exploitation Unix dérivé de 4.4BSD et adapté aux processeurs Intel : FreeBSD [BSD Report, 2000] ;
- MPC-OS, un noyau de communication spécifique composé de processus gestionnaires de ressources et de modules noyau Unix proposant différentes API de communication par passage de messages ;
- Des implémentations optimisées des environnements de programmation parallèles PVM [Silva and Mana, 1998] et MPI [Ong and Farrell, 2000], bâties sur le noyau de communication MPC-OS.

L'environnement de calcul matériel, le réseau de contrôle et le système d'exploitation sont des composants standards. Le réseau Gigabit et le noyau de communication MPC-OS ont été développés par le groupe de recherche MPC.

> Construire, sur le matériel spécifique de la machine MPC, un environnement logiciel fournissant des moyens de communication propices à un développement efficace et rapide des applications constitue le centre de notre propos. Ce manuscrit de thèse présente dans ce cadre les services offerts par MPC-OS, son organisation interne et les choix architecturaux qui ont accompagné sa conception.

1.5 Le réseau Gigabit HSL

Le réseau Gigabit de la machine MPC est composé de cartes à interface PCI [Cyliax, 2000] présentes dans chaque nœud. La carte contrôleur réseau FastHSL possède deux fonctions principales : elle contient un processeur câblé qui exécute le protocole de communication,

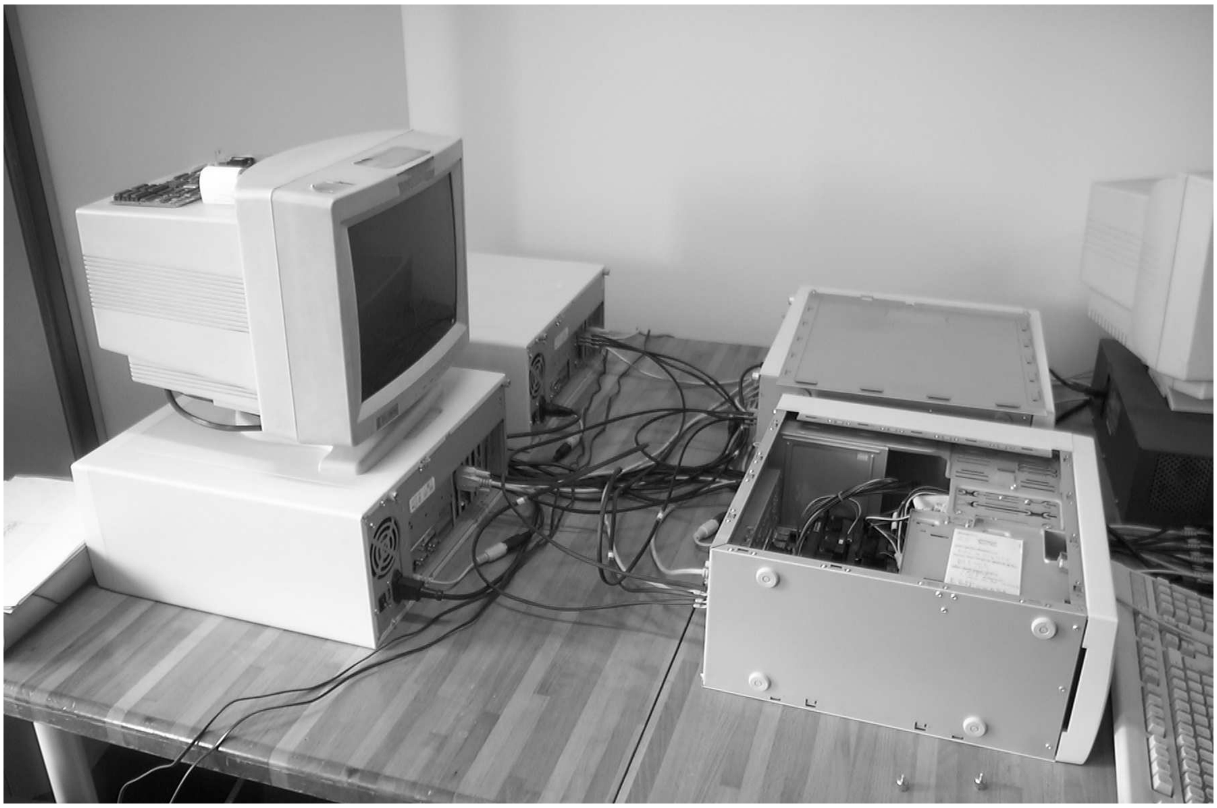


Fig. 1.1 *Machine MPC en configuration 4 nœuds bi-processeurs*

Fig. 1.2 *Connectique de la machine MPC*

ainsi qu'un routeur intégré qui permet de construire des réseaux de taille et topologie quelconques. Elles sont donc équipées de deux composants VLSI[5] développés au LIP6[6] :

✓ Le routeur **RCube** [Reibaldi, 1997]. Il s'agit d'un circuit CMOS d'environ 380000 transistors, qui offre une commutation de paquets de type *wormhole* entre huit ports Gigabit à la norme HSL IEEE-1355.

✓ Le contrôleur réseau **PCI-DDC**. Ce circuit CMOS d'environ 200000 transistors propose une primitive d'écriture distante en mémoire à travers le réseau HSL. Pour cela, il communique avec la mémoire du nœud local à travers le bus PCI, et avec son homologue sur le nœud distant à travers le réseau HSL.

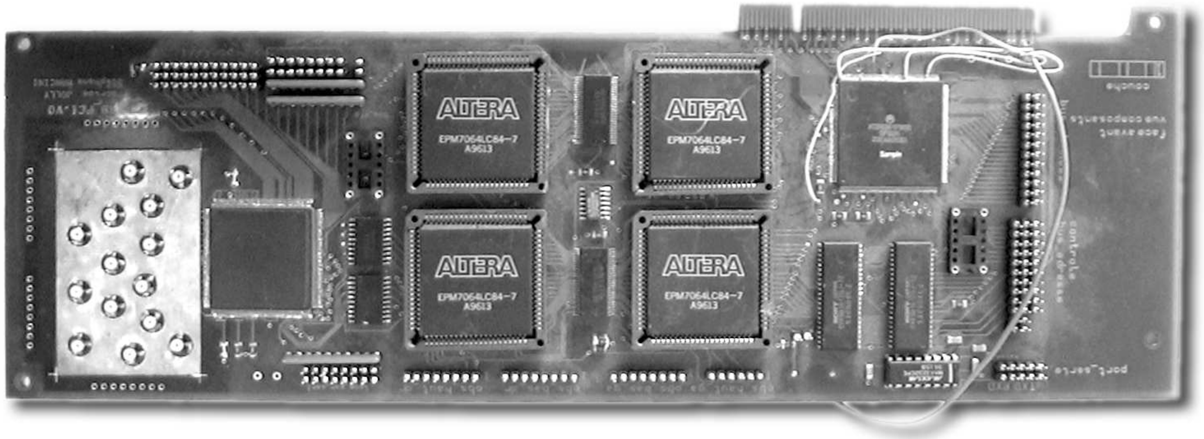


Fig. 1.3 *Première génération de carte MPC*

Les figures 1.3 et 1.4 page suivante présentent respectivement les premières et dernières générations d'adaptateur réseau MPC. Sur les deux cartes, on peut observer à gauche de la photographie, le composant RCube, et on peut noter que le circuit PCI-DDC est absent de la première génération de carte : il est remplacé par un ensemble de 5 composants : quatre Altera et un micro-contrôleur, qui émulent le fonctionnement de PCI-DDC.

La figure 1.5 page 34 présente une machine MPC à trois nœuds. Les liaisons HSL n'y sont pas représentées, seuls les connecteurs sont indiqués sur la figure : on peut construire la topologie de son choix en raccordant les connecteurs des différentes cartes FastHSL par des liens bidirectionnels constitués chacun d'une paire de câbles coaxiaux.

1.6 Le noyau de communication MPC-OS

Le système d'exploitation de la machine MPC est constitué, sur chaque nœud de calcul, de FreeBSD, système Unix dérivé de 4.4BSD, sur lequel vient se greffer le noyau de communication MPC-OS qui fait l'objet de cette thèse.

Il est composé d'une part de **couches de protocoles** de communication présentes sous

5. Very Large Scale Integration
6. Laboratoire d'informatique de Paris 6

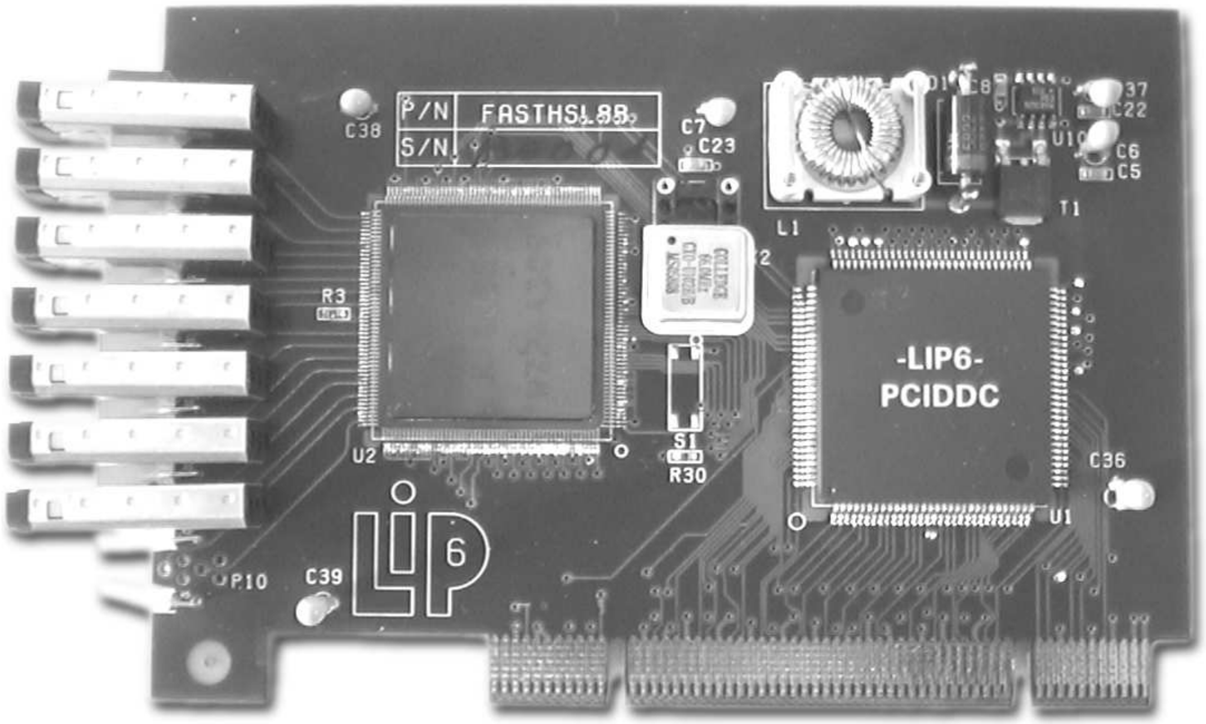


Fig. 1.4 *Dernière génération de carte MPC*

forme de modules noyau chargeables dynamiquement, et d'autre part de divers **processus de contrôle**.

Les protocoles implémentés dans les modules noyau permettent d'offrir des interfaces de communication par passage de message selon différents niveaux d'abstraction. Ces protocoles sont architecturés autour d'un modèle en couches, celle de plus bas niveau fournissant l'accès à la primitive d'écriture distante du composant PCI-DDC. Les couches de plus haut niveau assurent des communications par passage de messages à travers des canaux de communication virtuels.

Par ailleurs, des processus de contrôle sont chargés de configurer la machine et de gérer ses ressources :

✓ Ils assurent la **configuration matérielle et logicielle** de la machine, et dans ce but ils dialoguent entre eux à travers le réseau de contrôle. En effet, le réseau Gigabit est indisponible tant qu'il n'a pas été configuré.

  La configuration consiste à charger les tables de routage dans chaque routeur RCube, et à initialiser et configurer les composants PCI-DDC qui mettent en œuvre le protocole d'écriture distante.

✓ D'autres processus de contrôle sont chargés de gérer **l'attribution des ressources** de communication de la machine et ils utilisent pour cela le réseau Gigabit HSL. Ils assurent par exemple les négociations de canaux entre tâches d'une même application.

Le noyau de communication MPC-OS est utilisé par différentes équipes de recherche en France. Il équipe les machines MPC de l'Université de Versailles/St-Quentin en Yvelines,

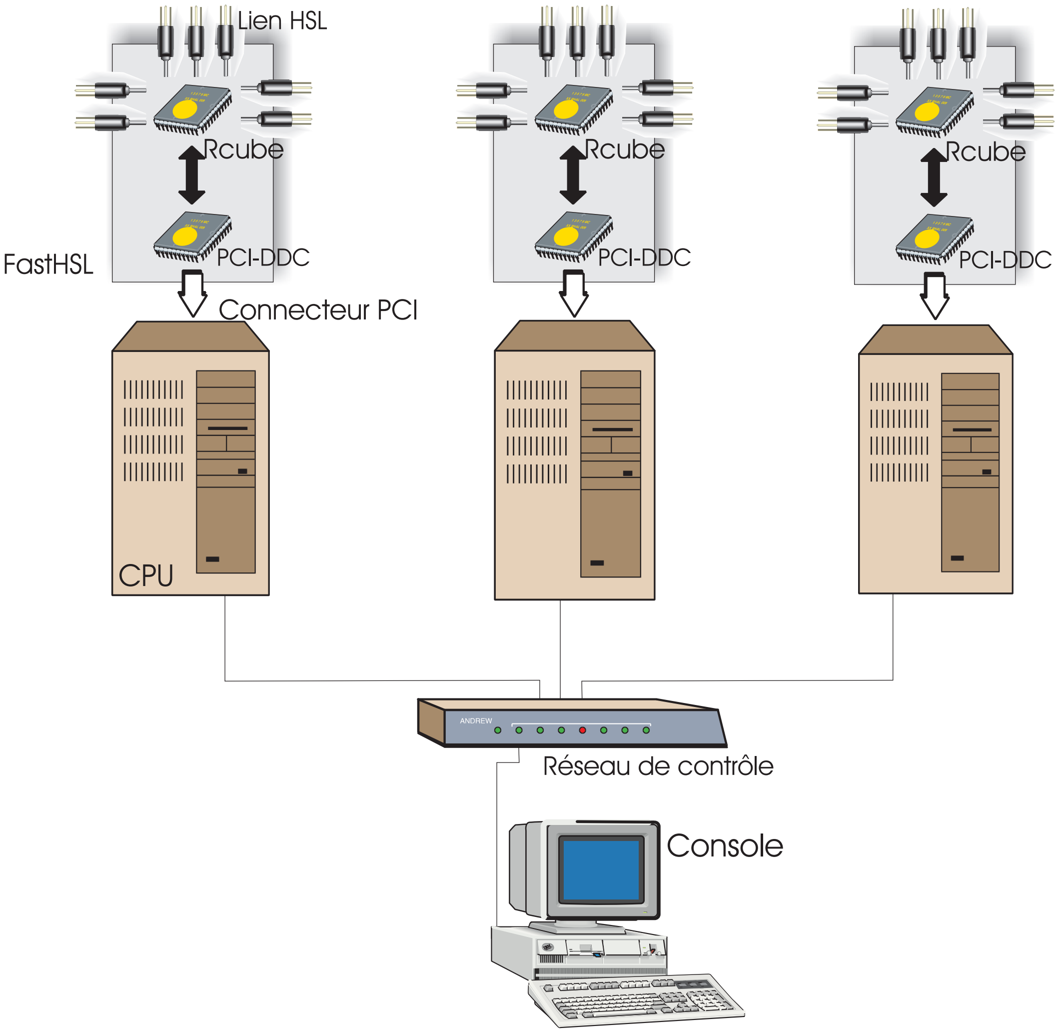


Fig. 1.5 *La machine MPC*

de l'Institut national des télécommunications, la machine "libre-service" du Laboratoire d'informatique de Paris VI, et la machine expérimentale de l'équipe *Architecture des systèmes intégrés et micro-électronique* du LIP6. Il fonctionne aussi en mode d'émulation du matériel sur une grappe de PCs à l'Université de Picardie Jules Vernes et sur une grappe de l'équipe *Systèmes répartis et coopératifs* du LIP6.

MPC-OS a ainsi permis à différentes équipes d'expérimenter des applications variées. On peut citer notamment le simulateur logique *cycle-based* PI-SIM [Pétrot *et al.*, 1997a] [Pétrot *et al.*, 1997b], le système de pagination distribué MAÎS [Cadinot *et al.*, 1997] et une implémentation de PVM [Silva and Mana, 1998].

Enfin, MPC-OS a été utilisé par des industriels pour des machines construites à partir de composants PCI-DDC et/ou RCube. MPC-OS a permis de valider les développements matériels de GEC Marconi Aerospace Systems (Grande-Bretagne), dans le cadre du projet EUROPRO [7], ainsi que de Parsytec Computer (Allemagne), dans le cadre du projet Arches [8].

1.7 Enjeux et objectifs

Le réseau de communication de la machine MPC vise les hautes performances. En effet, il est d'une part caractérisé par une faible latence de traversée (150 ns par routeur, ce qui signifie moins de 2 μs pour traverser un réseau de plusieurs centaines de processeurs) et de hauts débits de transfert de données (1 Gbit/s bidirectionnel pour chacun des ports). D'autre part, il possède un contrôleur réseau câblé qui intègre les protocoles de communication de bas niveau. Cela permet ainsi de décharger les processeurs des nœuds de calcul d'un grand nombre de tâches liées aux communications.

Bien sûr, il n'est pas question de laisser le programmeur utiliser directement les fonctions de communication du matériel, car elles sont d'un trop bas niveau d'abstraction pour être directement offertes aux applications. Il est nécessaire de proposer un support logiciel fournissant aux applications les traditionnelles garanties d'acheminement, de séquentialisation, d'intégrité, et de cohérence de gestion des ressources, qu'on trouve dans les protocoles d'usage courant.

Pour utiliser ce réseau, on pourrait donc imaginer faire usage de protocoles de communication traditionnels sur réseaux locaux, comme TCP/IP, en se contentant d'adapter les pilotes de périphériques bas-niveau. Malheureusement, cette solution simple et rapide à implémenter serait très préjudiciable aux performances du système : la latence de traversée logicielle des couches TCP/IP d'une machine Unix standard est déjà nettement supérieure à la latence du matériel spécifique à MPC, et les multiples copies de tampons intermédiaires, au cours de chaque transfert, sont rédhibitoires pour conserver des débits corrects.

Aux vues de ces constatations, l'objectif du travail présenté dans cette thèse est de spécifier et de réaliser un noyau de communication logiciel qui fournisse des services à haute valeur

7. European Multiprocessor System for Intensive Data Processing and Signal Processing
8. Application, Refinement and Consolidation of HIC Exploiting standards

ajoutée aux programmeurs d'applications (ex. : proposer des interfaces de communication simples d'utilisation, une gestion de la négociation des ressources et des tâches, etc.), **tout en préservant les bonnes performances fournies par le réseau de communication**.

1.8 Organisation du manuscrit

Ce manuscrit décrit les moyens mis en œuvre par MPC-OS pour fournir des services de communication de haut niveau à partir de la primitive d'écriture distante du réseau Gigabit. Il s'organise ainsi :

- ✓ chapitre 1, *Introduction*. Présentation des objectifs.
- ✓ chapitre 2, *Problématique et enjeux*. Le matériel fournit la primitive d'écriture distante. Celle-ci constitue l'unique interface proposée au système d'exploitation et aux applications pour communiquer à travers le réseau Gigabit. MPC-OS utilise cette primitive de bas niveau comme un composant de base en vue de construire des interfaces de communication plus abstraites. On analyse dans ce chapitre les difficultés à résoudre.
- ✓ chapitre 3, *Historique et état de l'art*. Des problèmes, analogues à ceux que nous avons rencontrés, se sont présentés lors de la conception de systèmes tels que BIP sur Myrinet, ou d'interfaces comme VIA sur réseau de type NOW ; le chapitre 3 étudie l'impact de ces problèmes et les méthodes de résolution proposées.
- ✓ chapitre 4, *Protocole de communication bas-niveau*. La couche de plus bas niveau fournie par MPC-OS, surnommée PUT, permet à plusieurs applications d'exploiter simultanément l'écriture distante, en leur proposant plusieurs modèles de programmation : l'utilisateur a le choix de la méthode de signalisation (interruption ou scrutation) et celui du comportement de l'API (bloquante ou non bloquante). Cette couche a aussi été utilisée pour masquer certains dysfonctionnements du matériel dans les premières versions. L'utilisateur s'adresse à elle en supposant que le circuit est tout à fait conforme à la spécification. On justifie tout au cours de ce chapitre les choix architecturaux qui ont mené à la conception de PUT pour atteindre ces objectifs.
- ✓ chapitre 5, *Canaux de communication*. Le modèle de communication sur canaux virtuels est largement plus répandu dans la communauté des programmeurs que celui de l'écriture distante, car il permet notamment de s'affranchir de la connaissance de la localisation physique des tampons dans les nœuds distants. Pour fournir ce type de communication, on a construit la couche SLR/P[9] au dessus de PUT, ce qui nous fait passer d'un mode de communication inter-nœuds avec PUT, à un mode de communication intra-application avec SLR/P. L'enjeu ici est de conserver de bonnes performances par rapport à PUT, notamment en s'imposant de réaliser un protocole sans copie.
- ✓ chapitre 6, *Sécurisation de la communication*. La couche SCP/P[10] est l'homologue de la couche SLR/P, à ceci près qu'elle offre, en plus, des garanties d'acheminement

9. SLR/P : StateLess Receiver protocol / using Physical addresses
10. SCP/P : Secure Channelized Protocol / using Physical addresses

(suppression des pertes de paquets et garantie d'intégrité des données). On expose dans ce chapitre la démarche suivie pour adapter SLR/P à ce nouvel enjeu qu'est la sécurisation vis-à-vis des fautes du réseau, tout en conservant, une fois encore, la caractéristique d'absence de copie. Pour réaliser ce protocole sécurisé, on a tout d'abord classifié l'ensemble des comportements pouvant être induits par une faute quelconque. On a alors défini un ensemble de sous-protocoles, basés sur l'utilisation de messages courts, destinés à fournir des opérations de contrôle comme le vidage du réseau entre deux nœuds. On montre enfin comment agencer ces sous-protocoles en un protocole global permettant d'acheminer tous types de messages à travers des liens non fiables.

✓ chapitre 7, *Mémoire virtuelle protégée et protocoles de haut niveau*. Disposer dans le noyau de canaux de communication sécurisés travaillant sur des zones de mémoire physique ne suffit pas à combler les *desiderata* des programmeurs d'application. On présente dans ce chapitre la réalisation de canaux de communications en adressage virtuel sur lesquels on a pu construire des protocoles de plus haut niveau d'abstraction. On y discute alors le système de protection des espaces de mémoire virtuelle vis-à-vis des dépôts de données provenant du réseau, réalisé au sein de la couche de communication SCP/V [11]. On montre notamment comment on a réalisé la garantie d'intégrité des tampons de communication même en cas de faute d'une application. Pour cela, on présente un mécanisme permettant de verrouiller les tampons pendant la durée totale d'un échange, même lorsqu'une application libère par erreur ces tampons avant la fin d'un transfert.

✓ chapitre 8, *Gestion dynamique des ressources*. Au cours des discussions qui précèdent, on a laissé de côté les problèmes d'allocation de ressources, pour se concentrer sur les protocoles et méthodes mises en jeu dans le cadre des communications entre tâches. On décrit dans ce chapitre un gestionnaire de ressources distribué implémenté au sein d'un processus présent sur chaque nœud. L'attribution des canaux et la gestion des tâches est à la charge de ce processus, construit sur un cœur d'ORB [12], au sein d'un environnement organisé selon la méthodologie objet et parallélisé avec le concept de *multi-threading*.

✓ chapitre 9, *Mesures et évaluation de performances*. Ce chapitre commence par la présentation des résultats d'une campagne de mesures de performances des couches de communication de la machine MPC. Il fournit alors un modèle physique dont la mise en équation permet de retrouver les valeurs expérimentales, ce qui permet d'identifier et de comprendre les différentes phases de comportement de la machine en fonction des stimuli auxquels elle est soumise.

✓ chapitre 10, *Analyse stochastique*. Le protocole de communication sécurisé présenté chapitre 6 s'appuie sur des canaux virtuels de communication pour proposer une sécurisation systématique face aux fautes du réseau, démarche traditionnelle dans le cadre de la sécurisation d'un protocole de communication. On propose ici une approche novatrice de ce problème, basée sur la proposition d'une adaptation sécurisée triviale de PUT permettant une reprise sur erreur en cas de faute simple du réseau, et justifiée par une analyse stochastique du phénomène de double-faute. Cette approche probabiliste propose un certain nombre de résultats généraux sur le phénomène de

11. SCP/V : Secure Channelized Protocol / using Virtual addresses
12. ORB : Object Request Broker

double-faute, permettant de quantifier ses occurrences en fonction des caractéristiques des processus stochastiques représentant l'apparition de fautes et du délai de reprise sur erreur. On se place alors dans le cadre de la machine MPC, et l'observation expérimentale des fautes du réseau, conjuguée à une analyse de leurs causes, permet de modéliser la distribution du processus de faute. On en déduit alors les caractéristiques que doit posséder le processus de reprise sur erreur pour obtenir un temps moyen entre deux double-fautes acceptable, et on montre finalement que la version de PUT sécurisée proposée ici est compatible avec ces contraintes.

✓ chapitre 11, *Conclusion et perspectives.* Les performances de l'interface PUT sont satisfaisantes, mais on paie au prix fort les services de plus haut niveau fournis par les différentes couches qui viennent s'empiler au dessus. L'expérience MPC-OS nous montre donc qu'une participation du matériel au support de ces services à valeur ajoutée est nécessaire si on veut conserver des performances analogues à celles de PUT. L'équipe de développement matériel de MPC a donc entrepris la réalisation d'un nouveau contrôleur réseau programmable [Desbarbieux, 2000]. On peut donc espérer introduire dans ce matériel programmable les opérations particulièrement coûteuses des couches logicielles actuelles, afin de profiter conjointement de bonnes performances et de services de haut-niveau.

Chapitre 2

Problématique et enjeux

Sommaire

La primitive d'écriture distante constitue l'unique interface de communication matérielle proposée au système d'exploitation et aux applications pour utiliser le réseau de communication MPC. MPC-OS implémente au dessus de cette primitive des interfaces plus abstraites. Nous allons examiner dans ce chapitre les difficultés rencontrées au cours de cette démarche.

2.1 La primitive d'écriture distante

2.1.1 Scénario général

Notre objectif général consistant à proposer des services de communication de haut niveau sur la primitive d'écriture distante matérielle de la machine MPC, nous présentons ses caractéristiques.

Lors du déroulement de cette opération (voir diagramme 2.2 page 43), le composant PCI-DDC d'un nœud émetteur extrait des données de la mémoire locale, les encapsule dans des paquets et les transmet sur le réseau de routeurs RCube à destination du nœud où ces données doivent être déposées. Le PCI-DDC du nœud destinataire reçoit ces paquets, en extrait les données, et les dépose dans sa mémoire locale.

2.1.2 Le réseau HSL

Les nœuds raccordés au réseau HSL sont représentés par des numéros compris entre 0 et 65535. Du point du vue du routeur, les paquets de données qui transitent sur le réseau comprennent une en-tête contenant le numéro du destinataire, puis des données et enfin un caractère de fin de paquet. Chaque routeur RCube dispose d'une table de routage qui indique la prochaine sortie pour atteindre un nœud donné.

2.1.3 Les messages

Les données correspondant à un ordre d'écriture distante sont encapsulées dans un **message**. Celui-ci contient en outre la localisation physique des plages de données dans les nœuds émetteur et récepteur. Notons que ces plages peuvent être discontiguës : Il n'y a *aucune contrainte de contiguïté*, ni du côté émetteur, ni du côté récepteur. C'est-à-dire qu'un message peut par exemple contenir des adresses physiques consécutives dans le nœud émetteur, mais pas dans le nœud récepteur.

De même, une plage de données du côté de l'émetteur peut se trouver désalignée avec son homologue du côté récepteur : il n'y a aucune contrainte d'alignement sur des frontières de mots à respecter.

2.1.4 Les pages réseau

Un message est décomposé en *pages réseau*, entité soumise à une double contrainte : c'est une plage de mémoire physique contiguë dans un nœud local, destinée à être déposée dans un nœud distant toujours dans un espace contigu. C'est à la charge du système d'exploitation de découper les messages en pages réseau et de les marquer d'un numéro unique, **l'identificateur de message**. Il permet, par exemple, à PCI-DDC de signaler la fin de réception d'un message en fournissant au système son numéro identificateur.

Par abus de langage, on utilisera le terme *page* en lieu et place du terme *page réseau*, lorsque le contexte permettra de différencier son sens de celui de la *page mémoire* de 4 Ko du système de mémoire virtuelle du processeur. Une *page réseau* peut avoir une longueur quelconque entre 1 et 65535 octets ; un message étant potentiellement constitué d'un nombre quelconque de *pages réseau*, sa longueur maximale n'est pas fixée.

2.1.5 Les paquets

PCI-DDC découpe les pages en paquets qu'il envoie d'un seul bloc sur le réseau HSL. Chaque paquet contient le numéro du nœud destinataire, la longueur des données ainsi que l'adresse initiale où doit avoir lieu le dépôt.

Le routage sur le réseau de RCube est de type *wormhole*. Ainsi, lorsque PCI-DDC doit attendre avant de pouvoir lire des données en mémoire, par exemple parce qu'un autre périphérique demande à acquérir le bus PCI, il doit clore le paquet en cours pour éviter de faire chuter les performances du réseau. Cela explique la nécessité de décomposer les pages en paquets.

La figure 2.1 présente les différents niveaux de découpage des messages en pages et paquets.

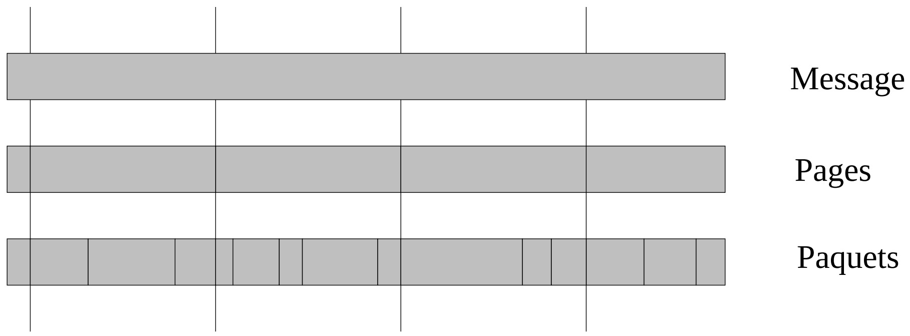


Fig. 2.1 *Découpage des messages en pages et paquets*

2.1.6 Les ordres d'écriture distante

Pour initier un ordre d'écriture distante, le système d'exploitation ajoute dans une table accessible à PCI-DDC la description des pages réseau constituant le message à transmettre, puis effectue un accès d'entrée/sortie vers PCI-DDC pour signaler le nombre d'entrées rajoutées à la table. Celle-ci se nomme **LPE**[1], Liste des Pages à Émettre et chacune de ses entrées contient 5 champs : l'identificateur du message associé ; l'adresse physique locale des données à émettre ; l'adresse distante où doit avoir lieu le dépôt ; la taille des données ; un registre d'indicateurs permettant d'activer ou non diverses fonctionnalités du PCI-DDC local ou distant. Ce dernier champ permet par exemple de demander la *génération d'interruption* indiquant sur le nœud émetteur la fin de l'émission d'une page ou d'un message et sur le nœud récepteur la fin de la réception de tous les paquets d'un même message.

En réception, PCI-DDC inscrit dans la table **LMI**[2] la liste des identificateurs des messages reçus.

Un message est donc constitué d'une suite d'entrées consécutives de la table **LPE** du nœud émetteur, toutes associées au même identificateur de message. Ce dernier est alors inséré dans la table **LMI** du nœud récepteur en fin de transmission.

2.1.7 Synopsis d'un échange standard

La figure 2.2 page suivante présente les différentes phases successives qui constituent une opération d'écriture distante :

❶ Le système d'exploitation rajoute dans la LPE les pages constituant le message et prévient PCI-DDC ;

❷ PCI-DDC encapsule les données de chaque page dans des paquets qu'il émet sur le réseau à destination du nœud récepteur ;

❸ PCI-DDC émetteur signale la fin de l'émission des données par une interruption ;

❹ PCI-DDC récepteur dépose les données en mémoire ;

❺ Une fois que tous les paquets sont reçus, PCI-DDC récepteur inscrit l'identificateur de message dans la LMI ;

❻ Il signale alors la fin de réception par une interruption.

2.1.8 Messages courts

PCI-DDC offre la possibilité d'envoyer des **messages courts**, d'au plus 8 octets. Le principe suit celui des messages traditionnels, à l'exception près qu'*il n'y a aucune adresse à fournir*, les données étant directement placées en émission dans la LPE, et se retrouvant en réception dans la LMI. PCI-DDC offre de plus une garantie d'atomicité pour ce type de message : ils sont transportés dans un unique paquet.

1. LPE : List of Pages to Emit
2. LMI : List of Message Identifiers

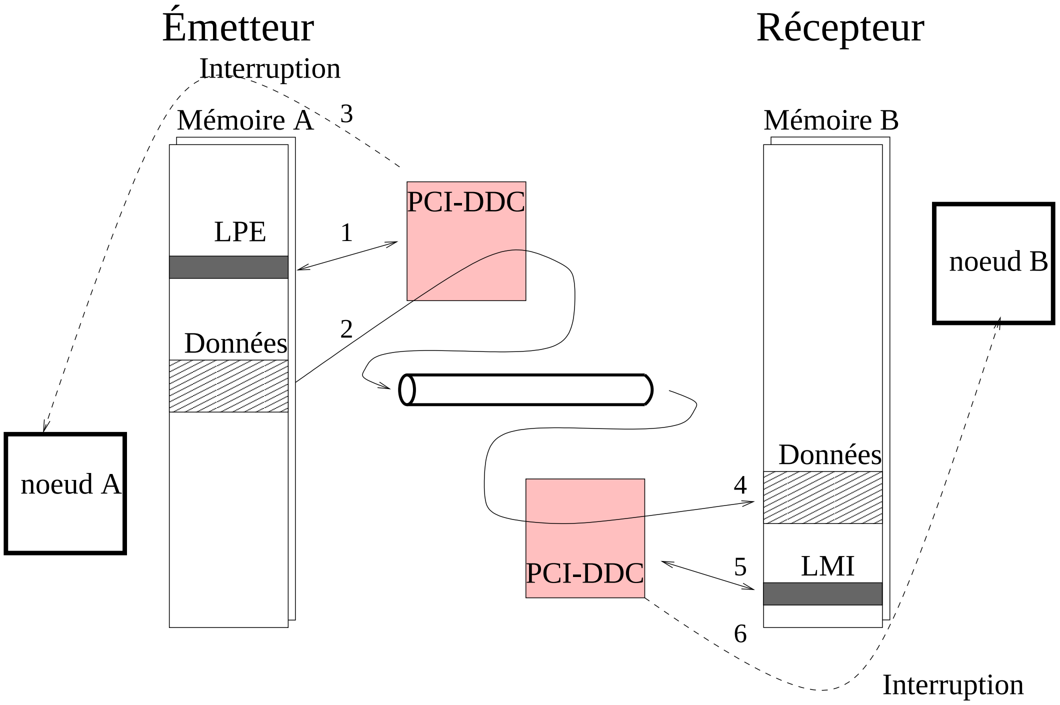


Fig. 2.2 *Synopsis d'un échange standard*

2.1.9 Liste des messages reçus

Suivant la configuration des tables de routage inscrites dans le réseau de RCube, le routage pourra être déterministe ou bien adaptatif.

Le routage adaptatif permet d'augmenter le seuil de bande passante utile du réseau HSL d'environ 10% [Bouaraoua, 1998]. Les simulations montrent que le seuil de bande passante utile se situe autour de 45% avec des tables non adaptatives et autour de 55% avec des tables adaptatives. Le routage adaptatif nécessite par contre une gestion plus complexe de la détection de fin de message : il ne suffit plus à l'émetteur de marquer le dernier paquet afin de permettre au récepteur de détecter la fin de transaction, car **les paquets peuvent se doubler dans le réseau**.

Pour résoudre ce problème, PCI-DDC émetteur doit compter les paquets qu'il génère et transmettre leur nombre total à son homologue en réception. On a vu que les paquets sont clos par PCI-DDC suivant l'état du bus PCI. Ce n'est donc que lorsque le dernier paquet constituant un message est généré qu'on peut connaître le nombre total de paquets mis en jeu : PCI-DDC indique dans ce dernier paquet le nombre total de paquets.

De ce fait, pour gérer l'adaptativité, il utilise en réception une table supplémentaire dans l'espace mémoire du nœud récepteur, accessible à travers un cache présent dans PCI-DDC. Il s'agit de la **LMR**, *Liste des messages reçus*. Cette table optionnelle, indexée sur les identificateurs de messages, est mise à jour dès qu'un paquet est reçu :

- ▷ PCI-DDC incrémente un compteur de paquets reçus dans l'entrée de LMR associée à l'identificateur de message correspondant ;
- ▷ Si ce paquet est le dernier généré par l'émetteur, il contient alors dans son en-tête

le nombre total de paquets du message correspondant. PCI-DDC inscrit donc ce nombre dans la LMR ;

▷ Il teste maintenant la fin de l'opération d'écriture distante : si l'on connaît le nombre total de paquets, c'est-à-dire si le dernier paquet a déjà été reçu, et si ce nombre est égal au compteur de paquets reçus, PCI-DDC effectue les opérations de fin de transmission : remise à zéro des compteurs de la LMR, mise à jour de la LMI, et activation d'une ligne d'interruption.

Deux contraintes d'utilisation de PCI-DDC sont dues à la gestion de cette table.

D'une part, la LMR étant indexée dans le nœud récepteur sur les identificateurs de messages, un ou plusieurs nœuds distincts ne doivent pas envoyer simultanément à un même destinataire des messages avec le même identificateur : le comptage des paquets en réception serait altéré, ce qui laisserait PCI-DDC récepteur dans un état invalide.

D'autre part, le compteur de paquets en émission est maintenu dans un registre unique au sein de PCI-DDC. Les pages constituant plusieurs messages ne doivent donc pas se retrouver *à cheval* dans la LPE : un message doit être constitué de pages successives dans la LPE.

2.1.10 Caractéristiques et contraintes à respecter pour l'écriture distante

Voici rassemblées ici les caractéristiques et contraintes que nous venons de rencontrer et qui définissent l'interface matérielle telle qu'elle est présentée au système d'exploitation :

- PCI-DDC fournit une primitive d'écriture distante sans intervention des processeurs de calcul ;
- La transmission est de type *zéro-copie* ;
- Les données sont encapsulés dans des messages, découpés en pages par le logiciel, celles-ci étant alors découpées en paquets par le matériel ;
- Aucune contrainte d'alignement ni de contiguïté n'est imposée au niveau message, tandis que la contiguïté est nécessaire au niveau page ;
- Chaque paquet contient des données et les adresses où elles doivent être déposées ;
- Deux types de messages sont disponibles : les messages traditionnels de taille quelconque et les messages courts, de taille limitée et transmis de manière atomique et sans nécessiter de gestion d'adresses ;
- Suivant les tables de routage inscrites dans les composants RCube, le réseau présentera un comportement soit déterministe, soit adaptatif ;
- Si le réseau est adaptatif, le logiciel doit garantir qu'un nœud donné ne peut recevoir simultanément plusieurs messages possédant le même identificateur ;
- Du fait de la garantie d'atomicité du transport des messages courts, cette dernière contrainte ne leur est pas applicable.

2.2 Des services de communication de plus haut niveau

On ne peut se contenter de fournir la primitive d'écriture distante aux programmeurs : celle-ci étant d'un trop bas niveau, elle se retrouve inadaptée à la plupart des applications.

Notre objectif est de fournir un environnement logiciel de communication à la fois adapté aux programmes destinés à la machine MPC et le plus efficace possible, basé sur l'interface d'écriture distante *zéro-copie* proposée par le matériel.

Cela se traduit par un noyau de communication qui doit être accessible tout à la fois pour le développement :

▷ *d'applications utilisateur :* on peut facilement isoler des caractères communs aux différentes bibliothèques de communication fournies aux programmeurs d'applications parallèles ou distribuées. Le modèle le plus traditionnel et le plus évident à manipuler par le concepteur d'une application est le modèle du tube bidirectionnel. Un tube de communication est d'abord ouvert entre deux entités, celles-ci vont alors pouvoir utiliser des primitives de type lecture/écriture pour y échanger des informations. Le comportement de ce canal doit être suffisamment permissif pour faciliter le travail du programmeur. Les opérations doivent pouvoir y être effectuées de façon asynchrones avec effet tampon (une écriture peut être effectuée même s'il n'y a pas de lecture en attente), et le caractère bloquant/non bloquant de chaque opération doit pouvoir être contrôlé à l'initiative du programmeur. Enfin, le système doit assurer des garanties d'acheminement (pas de perte ni duplication de données) et de séquentialité (les données sont reçues dans l'ordre d'écriture). L'interface Sockets BSD, l'interface TLI[3] System V, les tubes nommés, sont quelques exemples de systèmes de communication disposant des caractéristiques classiques présentées ci-dessus.

▷ *d'environnements de programmation parallèle :* les programmeurs d'applications parallèles utilisent le plus souvent les services de communication fournis par des environnement de programmation standards, comme les environnements définis par PVM (Parallel Virtual Machine, machine parallèle virtuelle) ou MPI (Message Passing Interface, interface de passage de messages). Il existe de nombreuses implémentations de ces environnements de programmation, chacune optimisée pour telle ou telle machine parallèle. Ces implémentations proposent un modèle de programmation standard, et sont construites à l'aide des bibliothèques de communication bas niveau propres à la machine parallèle pour laquelle ils sont destinés.

  La machine MPC est destinée d'une part à récupérer des applications parallèles déjà écrites, et d'autre part à faciliter la tâche des programmeurs déjà familiarisés avec les environnements standards. Pour cela, elle doit permettre de porter de tels environnements sans perte de performances : la difficulté principale lors du portage d'un environnement de programmation distribué tient dans la nécessité de trouver le moyen d'assembler cet environnement avec des primitives de communication propres à la machine cible, *tout en gardant les meilleures performances possibles.* Il faut donc pouvoir proposer au programmeur en charge d'un tel portage différentes

3. TLI : Transport Layer Interface

bibliothèques de programmation, peut-être moins souples à utiliser que le classique modèle de tube bidirectionnel, mais beaucoup plus performantes car plus proches du matériel.

▷ *de services noyau :* en plus des applications parallèles, on veut avoir la possibilité de favoriser le développement de services répartis sur la machine MPC, afin de pouvoir offrir une large gamme de modèles de communication. En effet, les tubes ou les passages de messages ne forment pas les seuls modèles de communication envisageables. Pour donner un exemple, une mémoire partagée répartie, ou un système de fichiers distribué cohérent peuvent s'avérer bien plus adaptés à certaines applications. Implémenter de tels modèles nécessite le plus souvent une modification d'une partie du code du noyau du système d'exploitation. On désire donc, au sein de MPC-OS, fournir des moyens de communication accessibles depuis le noyau, afin de permettre l'intégration des communications haut-débit au sein de services noyaux conçus spécifiquement pour la machine MPC.

Dans la suite du chapitre, on se propose d'identifier les problèmes qui découlent de notre objectif. On sera animé d'une volonté constante d'envisager des mécanismes qui minimisent le nombre de copies de données, afin de bénéficier de performances reflétant au mieux la caractéristique zéro-copie du dépôt direct.

2.3 Manipulation de messages : adaptativité vs simplicité

2.3.1 Pages et messages : deux niveaux d'abstraction

Comme nous l'avons remarqué en dégageant les caractéristiques de la primitive matérielle d'écriture distante, le logiciel doit simultanément manipuler des objets *pages réseau* et des objets *messages* pour commander les composants matériels qui gèrent le réseau MPC. Il s'agit néanmoins de deux niveaux d'encapsulation distincts, car les messages sont tout simplement composés de pages.

Les manipulations de pages sont soumises à des contraintes beaucoup plus lourdes que les manipulations de messages : les pages doivent être contiguës dans les espaces mémoire de l'émetteur et du récepteur, et leur taille est limitée.

2.3.2 Routage dans le réseau MPC

On l'a constaté section 2.1.9 page 43, l'adaptativité au sein du réseau de la machine MPC permet d'accroître dans une certaine mesure les performances.

Elle entre en jeu uniquement si les tables de routage fournissent un comportement adaptatif. Pour cela, une condition nécessaire, mais non suffisante, consiste à ce qu'il existe des chemins distincts et redondants entre deux nœuds du réseau. Par exemple, un réseau dans lequel les nœuds sont disposés en chaîne avec deux extrémités ne peut évidemment pas être adaptatif. Par contre, une topologie en anneau peut permettre un comportement adaptatif, car un message peut transiter dans les deux sens de l'anneau pour passer d'un nœud à un autre.

Vues les contraintes imposées aux tables de routage de RCube (routage par intervalles), toute topologie réseau présentant des chemins redondants ne permet pas forcément d'écrire des tables de routage adaptatives. Néanmoins, de nombreuses topologies régulières le permettent.

On sait de plus que, pour toute topologie connexe (le réseau est constitué d'une seule composante connexe), on peut trouver un ensemble de tables de routage non adaptatives [Bouaraoua, 1998].

Ainsi, l'administrateur d'une machine MPC peut toujours se ramener à un routage réseau déterministe, et dans de nombreux cas il peut choisir un routage adaptatif plus performant en terme de débit réseau brut dans des conditions de forte charge.

2.3.3 Contraintes d'adaptativité

Mais l'adaptativité induit, dans les protocoles de communication utilisant ce réseau, un coût non négligeable en terme d'accroissement de la complexité. Examinons dans quelle mesure.

L'entité minimale de routage dans le réseau est le paquet : un message est constitué de pages, à leur tour constituées de paquets. Ainsi, un message est constitué d'un certain nombre de paquets qui vont chacun pouvoir suivre un chemin distinct si le réseau est adaptatif. Sachant qu'il peut y avoir des engorgements dans certains routeurs, un paquet appartenant à un message peut être mis en attente pendant une durée importante, alors que l'ensemble des autres paquets de ce message ont déjà été déposés chez le récepteur. Ainsi, on ne peut pas garantir que les données sont déposées dans l'ordre croissant des emplacements mémoire, *on ne peut donc pas faire de scrutation (polling) sur la dernière case mémoire d'un tampon de réception pour en déduire que les données ont été déposées dans leur totalité.*

Pour la même raison, l'ordre de réception de messages successifs peut être inversé par rapport à l'ordre d'émission. Pire encore, *deux messages envoyés successivement peuvent voir leur réception s'entrecroiser.*

Comme on peut s'en douter, ces caractéristiques accroissent fortement la complexité des protocoles qu'on peut vouloir construire au dessus de la primitive d'échange de messages. Pour en donner un seul exemple, un nœud a parfois besoin de s'assurer que l'ensemble des messages qu'il a envoyé à un partenaire ont été reçus. Pour ce faire, la méthode la plus simple, dite de la voiture balais, consiste à émettre un dernier message vers ce partenaire, et à en attendre une réponse dans le sens contraire. Dès réception de cet aller-retour dans le nœud initiateur de cette opération, ce dernier a la garantie que tous les messages envoyés jusqu'à maintenant ont été reçus, si le réseau entre ces deux nœuds peut être considéré comme un tube. Au contraire, s'il existe plusieurs chemins entre ces deux partenaires, il n'y a pas de méthode alternative simple. On pourrait imaginer calculer un majorant du temps de transit d'un paquet dans le réseau, et que le nœud initiateur attende jusqu'au terme de ce délai sans émettre aucun message. Il aurait alors la certitude que le réseau est vide entre lui et son partenaire, dans le sens orienté vers son partenaire. Malheureusement, le temps de transit maximal théorique d'un paquet dans le réseau est très grand. En effet, le réseau étant de type wormhole, un paquet peut bloquer une artère vitale du réseau tant que le nœud auquel il est destiné est indisponible ; plus précisément, tant que la carte réseau de ce nœud n'a pas accès à la mémoire centrale, c'est à dire tant que cette carte n'a pas accès au bus PCI. Le contrôle d'accès au bus PCI étant coopératif, la carte réseau ne peut en exiger un accès permanent, et le temps de blocage du réseau peut être prohibitif en fonction des desiderata des autres cartes de ce nœud.

Le premier service que le système d'exploitation de la machine MPC doit fournir consiste donc à rendre opaque la gestion des pages, en se contentant de fournir des services d'échange de messages, avec les contraintes d'adaptativité qui leur sont associées : les protocoles de haut niveau ne manipulent plus que des messages, mais ils doivent être capables de prendre en compte les contraintes d'adaptativité s'ils veulent profiter d'un accroissement de performance. La complexité induite au sein du protocole ne vient-elle pas alors tempérer l'accroissement des performances ?

2.4 Dépôt direct et contrôle des messages reçus

Lorsque qu'un message est reçu par une carte réseau traditionnelle (Ethernet, Token Ring, ATM, etc.), il est d'abord assemblé dans la carte qui joue alors le rôle d'un tampon. Une ligne d'interruption est ensuite activée afin de signaler au processeur qu'il peut récupérer les données. Il les recopie donc en mémoire, directement ou par un mécanisme de DMA [4]. Lorsque le message suivant vient à être délivré, le processeur recopie ces données à un autre endroit inoccupé dans la mémoire. Le processeur constitue ainsi une file de tampons qui accueillent les données provenant du réseau, afin d'en autoriser un traitement ultérieur par les couches de protocoles adjacentes, comme illustré sur la figure 2.3.

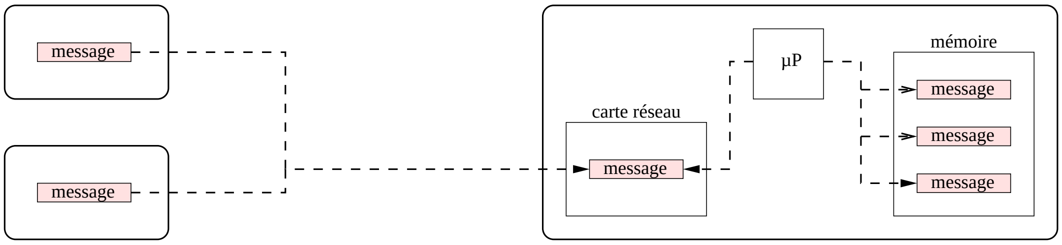


Fig. 2.3 *Dépôt de données à travers une carte réseau traditionnelle*

À l'inverse, lorsqu'un message est reçu par PCI-DDC, il va directement se déverser en mémoire **sans aucun contrôle possible de la part du processeur local**, du fait du mécanisme d'écriture distante par dépôt direct fourni par PCI-DDC.

Ce dernier ne peut donc pas organiser les données dans des files de tampons pour les traiter par la suite.

Pour s'adapter à cette contrainte, on pourrait imaginer que les différents messages à destination d'un même nœud soient tous déposés dans une zone fixe attribuée à l'initialisation

4. Direct Memory Access

de la machine par ce nœud, et communiquée au reste des processeurs.

Cette proposition n'est pas suffisante car il est possible que plusieurs messages se déposent *simultanément* dans la mémoire d'un même nœud : imaginons que deux nœuds distincts envoient chacun un message composé de multiples paquets vers un troisième nœud ; les paquets constituant les deux messages pourront être délivrés simultanément chez le récepteur. On se retrouverait donc dans la situation de la figure 2.4 où deux messages se déposent simultanément au même endroit, ce qui conduit évidemment à une perte d'intégrité des données.

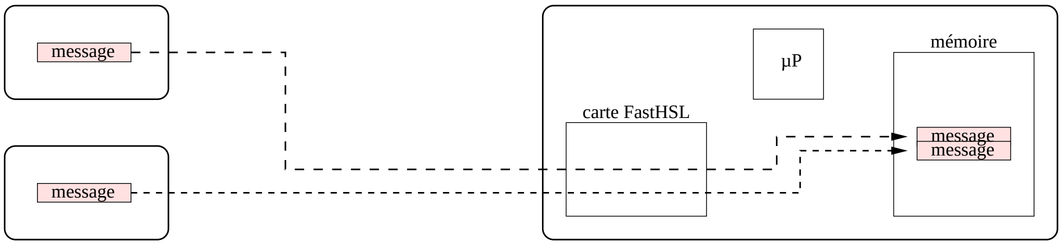


Fig. 2.4 *Dépôt direct des données sans action du processeur récepteur*

On ne peut pas non plus attribuer dans un nœud récepteur donné, une zone tampon pour chaque émetteur éventuel. Cela permettrait certes d'éviter que deux nœuds distincts viennent déverser leurs données au même endroit, mais il ne faut pas oublier que deux messages émis *par un même nœud* peuvent se déverser simultanément du fait du mécanisme d'adaptativité.

Dans le contexte du parallélisme intrinsèque de la machine MPC, c'est-à-dire d'émissions et de réceptions simultanées, se pose donc la question de savoir s'il est possible pour un nœud récepteur de contrôler la localisation des messages reçus afin d'éviter des dépôts simultanés aux mêmes emplacements ?

2.5 Identification et localisation des messages reçus

Après le dépôt d'un message, seules deux informations sont fournies au processeur récepteur :

- ▷ Une interruption lui indique qu'une opération vient d'être accomplie par PCI-DDC ;
- ▷ La consultation de la table LMI lui indique qu'il s'agit d'une réception, et le renseigne sur l'identificateur de message associé.

On constate donc que la seule information disponible relative au message reçu est l'identificateur de message qui lui est associé. Il n'y a donc aucune information directe sur la localisation physique des données reçues.

Nous avons en outre observé, section 2.1.10 page 44, que l'identificateur de message est soumis à une contrainte forte : si le réseau est adaptatif, le logiciel doit garantir qu'un nœud donné ne peut recevoir simultanément plusieurs messages possédant le même identificateur.

On peut donc légitimement se poser les deux questions suivantes :

– Peut-on localiser facilement les zones de données reçues, à l'aide de l'identificateur de message, seule information disponible dans les nœuds récepteurs ?
– Comment gérer l'attribution de ces identificateurs dans les nœuds émetteurs ?

2.6 Dépôt direct et acheminement des données au sein d'un nœud récepteur

Les couches de communication construites sur des interfaces réseau traditionnelles encapsulent leurs messages dans des trames, comportant une en-tête puis les données proprement dites.

Plusieurs phases d'encapsulation successives sont appliquées par les différentes couches de protocoles. Cela consiste à rajouter aux données plusieurs en-têtes, afin :

▷ d'une part de fournir les informations de routage nécessaires à l'acheminement correct des messages *vers le nœud destinataire* ;
▷ et d'autre part d'acheminer ces messages à l'intérieur du nœud récepteur *vers les processus auxquels ils sont destinés*, comme l'illustre la figure 2.5.

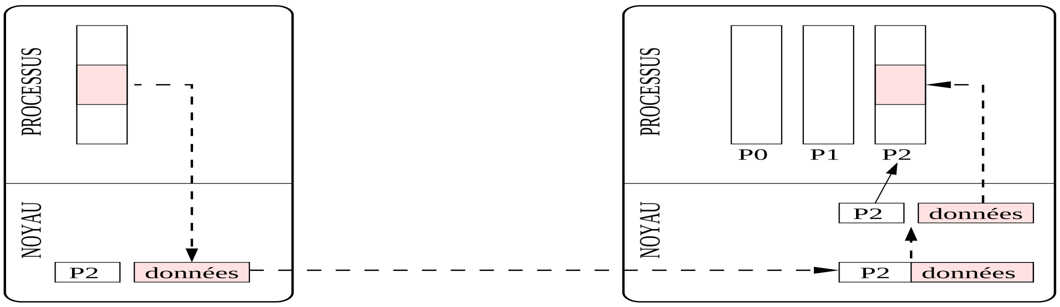


Fig. 2.5 *Acheminement des données au sein d'un récepteur, par encapsulation*

Dans la machine MPC, les en-têtes nécessaires pour la fonction de routage sont ajoutées directement par PCI-DDC au début de chaque paquet.

Par contre, contrairement à ce qui se passe lorsqu'il est muni d'une carte réseau traditionnelle, un nœud récepteur de la machine MPC **n'a aucun contrôle direct** sur les plages de mémoire où les données viennent se déposer, comme nous l'avons observé section 2.4 page 48 : le récepteur ne peut pas accumuler les messages dans une file de tampons afin d'analyser leur contenu, et notamment leurs en-têtes, pour déterminer à quels processus ils sont destinés.

Le principe du dépôt direct apparaît donc comme fondamentalement contradictoire avec les techniques d'acheminement des données au sein d'un récepteur.

Le noyau de communication de la machine MPC doit ainsi se passer de la technique traditionnelle d'encapsulation des données. De ce fait, on doit trouver une autre méthode pour acheminer dans un nœud les données jusqu'aux applications auxquelles elles sont destinées.

Pour aborder ce problème, il faut garder à l'esprit que le point fort de la primitive d'écriture distante de la machine MPC réside dans sa caractéristique *zéro-copie*. Nous sommes donc à la recherche d'un moyen efficace pour acheminer les données vers leur destination finale, tout en évitant d'imposer des copies intermédiaires.

Cela se traduit par les deux questions suivantes :

- Quelle méthode, se substituant à la traditionnelle encapsulation logicielle des données, peut-elle être mise en œuvre par le noyau de communication afin d'acheminer correctement les données au sein d'un nœud destinataire ?
- Existe-t-il une telle méthode, de faible coût et respectant la contrainte *zéro-copie* de bout en bout ?

2.7 Synchronisation des applications

L'écriture distante sans copie proposée par MPC a un impact important sur la synchronisation des applications qui communiquent sur le réseau HSL. Pour le constater, plaçons nous tour à tour du côté émetteur puis récepteur, et imaginons les contraintes de synchronisation à respecter pour qu'un transfert de message se déroule correctement.

✓ *Transfert vu du côté émetteur.* L'application ou le système doit indiquer au matériel la position locale du tampon qui contient le message à émettre, ainsi que sa localisation distante. Il s'agit donc, pour l'émetteur, de connaître au moment de la demande d'envoi, la position qu'occuperont les données chez le récepteur. Cela veut donc dire qu'au moment de l'émission, le tampon dans la mémoire du processus récepteur est déjà prêt. Dans le cas contraire, l'émission ne peut se produire, et le processus émetteur se retrouve bloqué. Bien sûr, si l'on est prêt à supporter le coût d'une copie, ce problème disparaît. Dans les systèmes de communication traditionnels, le retour de la primitive d'émission n'est pas conditionné par une demande de réception traduisant la présence d'un tampon d'accueil chez le récepteur. Cette synchronisation forcée dans le cadre de la machine MPC impose donc aux primitives de communication une sémantique non standard.

Continuons notre analyse : une fois que l'ordre d'émission est fourni à la carte réseau, le tampon d'émission n'en est pas pour autant accessible à nouveau. En effet, si un mécanisme de demande d'émission non bloquante est invoqué par un processus, celui-ci doit attendre la fin de l'émission avant de pouvoir réutiliser son tampon d'émission, contrairement encore une fois aux systèmes de communication traditionnels. Se pose alors la question du mode de signalisation de cet événement.

✓ *Transfert vu du côté récepteur.* Imaginons maintenant que les demandes de réception soient bloquantes. Comme on vient de le voir, l'émetteur doit connaître la localisation des tampons du récepteur avant de pouvoir entamer une émission. Le récepteur doit donc indiquer cette localisation par une demande de réception le plus tôt possible, le cas optimal consistant à ce que la demande de réception se produise avant la demande d'émission. Mais en terme de performances, il faut au contraire que la demande de réception, bloquante, se produise le plus tard possible, pour minimiser la durée pendant laquelle le processeur récepteur se contente d'attendre des données.

On constate donc une contradiction entre réception bloquante et performances. Le système doit donc aussi proposer des réceptions non bloquantes. Se pose alors à nouveau la question de la signalisation de fin d'opération (ici chez le récepteur).

On vient de constater que le caractère zéro-copie impose une synchronisation nouvelle (l'émission ne peut se produire que si la demande de réception a eu lieu). Se pose donc la question de savoir comment s'affranchir de cette contrainte de synchronisation pour pouvoir proposer au programmeur des primitives de communication qui suivraient un comportement traditionnel.
On a aussi constaté que pour des raisons de performances (recouvrement calculs/transfert réseau), les primitives de communications doivent pouvoir suivre un mode de comportement non bloquant. Ce caractère non bloquant, ainsi que le problème de la validité des tampons, nécessitent la définition d'une méthode de signalisation de fin d'opération.

Nous avons, dans cette section, conduit notre analyse en considérant des *processus* communicants, il faut néanmoins que nos solutions soient suffisamment générales pour être applicables dans le cadre de communications directement générées par le noyau, en conformité avec notre engagement section 2.2 page 45 de fournir un moyen de communication pour des services noyau tel qu'un système de fichier distribué par exemple.

2.8 Tolérance aux fautes matérielles du réseau

Quatre entités, regroupées au sein du matériel et du logiciel, peuvent être source de faute dans une machine parallèle : le matériel qui constitue les nœuds de calcul, le matériel qui constitue le réseau, le système d'exploitation et l'application. La machine MPC étant constitué de nœuds de calculs standards de type PC/Pentium, on considèrera que la probabilité de faute matérielle est suffisamment faible par rapport au temps de calcul d'une application. Pour des raisons analogues, on considèrera que le système d'exploitation est correctement construit. Il nous reste donc à examiner dans cette section le problème des fautes des composants du réseau, nous aborderons dans la section suivante celui des fautes de l'application.

2.8.1 Les causes

La machine MPC est destinée à rassembler, à terme, plusieurs dizaines, voire même des centaines de nœuds. La valence d'interconnexion d'une carte réseau FastHSL étant de 7, le nombre maximum de liens HSL dont peut disposer une machine MPC constituée de n nœuds est $E(\frac{7n}{2})$. Par exemple, le réseau Gigabit d'interconnexion d'une machine de 100 nœuds peut contenir jusqu'à 350 liens Gigabit HSL bidirectionnels.

Les paquets acheminés sur ces liens peuvent être altérés pour différentes raisons :

▷ *Problème mécanique sur un lien* : une mauvaise connexion involontaire entre deux noeuds peut subvenir. Elle peut être due à un défaut de l'un des deux câbles coaxiaux qui forment un lien, ou bien à un mauvais raccordement entre la prise mâle qui constitue une extrémité du lien, et la prise femelle sur la carte FastHSL.

▷ *Interférences magnétiques* : des interférences électro-magnétiques dans le domaine de fréquences du Gigabit peuvent altérer des données sur un lien HSL.

▷ *Contraintes d'horloges mal respectées* : la calibration d'un lien HSL est garantie par la récupération asynchrone de l'horloge à partir des bits qui parcourent le lien. Lorsque l'horloge externe qui alimente RCube ne suit pas les contraintes comportementales strictes imposées pour un bon fonctionnement du circuit, un lien peut se trouver décalibré. Les paquets en cours de transfert au moment où ce phénomène se produit se trouvent corrompus.

2.8.2 Effets potentiels

Les informations qui transitent sous forme binaire sur les liens HSL peuvent être altérées suite à une faute matérielle sur le réseau. Le matériel a été conçu pour pouvoir détecter ces événements à l'aide de sommes de contrôle diverses dans les en-têtes et à la suite des données. Néanmoins, la détection d'une faute, et éventuellement sa signalisation, ne peuvent se substituer à un mécanisme de reprise sur erreur, qui n'est, pour des raisons évidentes de complexité, pas fourni ici par le matériel.

On peut établir une classification exhaustive des possibles conséquences de fautes du réseau :

▷ Altération du fonctionnement du réseau :
  - interblocage ;
  - perte de performances (par exemple par la présence d'un paquet qui boucle dans le réseau) ;

▷ Perte d'intégrité des données :
  - pertes de données ;
  - duplication de données ;

▷ Défaut d'acheminement :
  - dépôt de données corrompues ;
  - acheminement de données vers un destinataire incorrect ;
  - dépôt de données dans un emplacement incorrect.

Il n'est pas évident que toutes ces conséquences puissent se produire, tout dépend des caractéristiques du matériel. Dans le cadre d'un réseau Ethernet [Metcalfe and Boggs, 1976], par exemple, une perte de données peut se produire, mais le dépôt de données corrompues est impossible. En effet, chaque trame transite par la carte Ethernet, qui s'assure de la validité des sommes de contrôle avant la transmission des données dans le tampon de l'application réceptrice. Au contraire, dans le cadre de la machine MPC, le circuit PCI-DDC dépose les données en mémoire centrale au fur et à mesure de leur réception, le calcul de la somme de contrôle d'un paquet n'étant terminé qu'après son dépôt complet. La carte FastHSL peut donc signaler la présence de données erronées, sans pour autant pouvoir éviter leur dépôt.

2.8.3 Intégration d'un mécanisme de reprise sur erreur

Notre travail, pour intégrer dans le système d'exploitation un mécanisme de reprise sur erreur, consiste à aborder les trois étapes successives suivantes :

❶ Il faut tout d'abord, aux vues de la spécification des composants matériels du réseau MPC déterminer leur comportement en cas de faute. Pour cela, on doit expliciter le format des paquets de données qui transitent sur chaque lien, et analyser **le comportement du matériel** en fonction du type de données altérées ;

❷ À partir du comportement matériel, on doit extraire **les effets produits au sein des nœuds récepteurs** de la classification 2.8.2 page précédente ;

❸ Il faut alors **déterminer s'il est possible de proposer un protocole** de recouvrement en cas de faute matérielle du réseau, pour compenser les effets des fautes dans les nœuds récepteurs. Évidemment, le surcoût dû au recouvrement des fautes doit être faible. Une méthode de recouvrement qui ferait perdre le caractère zéro-copie des échanges rendrait son utilisation rédhibitoire.

Le problème de la prise en compte des fautes matérielles du réseau se traduit donc par la question suivante :

> Est-il possible d'imaginer un protocole rendant transparentes à l'utilisateur les fautes matérielles du réseau, et dont le surcoût lié à cette contrainte serait négligeable ?

2.9 Tolérance aux fautes logicielles

Une fois de plus, nous sommes confrontés, dans le cadre de l'examen du problème des fautes logicielles, à des réactions du système qui sont à mille lieues de ce qui se produit dans les machines qui font usage de protocoles de communication standards.

Les causes de problèmes potentiels sont liées à la manipulation d'adresses et à la disponibilité des tampons de communication.

En effet, le matériel s'occupe d'effectuer les transferts de mémoire centrale locale à mémoire centrale distante. La seule interaction qu'il a avec l'application consiste, pour cette dernière, à fournir des informations d'adressage pour localiser les tampons de données, et à garantir la réservation effective de ces tampons pendant toute la durée du transfert.

Une application incorrecte ou erronée peut donc :

✓ *Fournir une adresse de tampon d'émission invalide*, par exemple allouée à une autre application. Le matériel attend en effet des adresses décrivant une position dans l'espace de mémoire physique, il n'a pas connaissance de la notion d'espace de mémoire virtuelle alloué à un processus, car il n'a notamment pas accès à la MMU[5] du processeur.
Une telle opération peut permettre à une application d'accéder aux données appartenant à d'autres applications, il s'agit donc ici d'un problème de confidentialité.

5. MMU : Memory Management Unit

✓ *Fournir une adresse de tampon de réception invalide.* Les composants du réseau ne font pas de vérification de droit d'écriture au moment de la réception, et le système d'exploitation d'un nœud récepteur n'est pas sollicité pendant un dépôt de données. Ainsi, une application peut aller modifier, a priori, n'importe quel emplacement mémoire de n'importe quel autre nœud de la machine MPC. Une application invalide dans un nœud peut donc corrompre l'intégrité d'un autre nœud.

✓ *Ne pas réserver un tampon d'émission pendant toute la durée du transfert.* Pendant tout le transfert, l'application doit garantir que le tampon d'émission est présent en mémoire physique, et qu'il ne change pas de position. Les systèmes d'exploitation modernes proposent un gestionnaire de mémoire virtuelle, souvent associé à un mécanisme d'évincement des pages (swap). Sous leur action, une zone de mémoire allouée à une application, peut se retrouver, au cours de la vie de cette application, à différentes positions dans la mémoire physique, ou bien même sur un disque. La machine MPC demande aux applications de se prémunir de telles migrations de données. Un mécanisme de verrouillage des pages est fourni par nombre de systèmes d'exploitation pour ce faire. Néanmoins, il n'est pas forcément suffisant, car une application qui provoquerait une faute grave pendant une émission (par exemple une division par zéro) serait amenée malgré elle à quitter le système, ses pages étant alors libérées pour que d'autres applications puissent en profiter. Le matériel du réseau MPC pourrait dans un tel cas être amené à transférer dans un nœud distant des données d'une application locale autre que celle qui a initié le transfert. On est donc à nouveau confronté à un problème de confidentialité.

✓ *Ne pas réserver un tampon de réception pendant toute la durée du transfert*, pour les raisons que l'on vient de voir (mécanisme d'évincement des pages ou application qui quitte le système), peut conduire à une perte d'intégrité du nœud récepteur. En effet, si une application quitte le système, et donc si son tampon de réception est libéré, le matériel va néanmoins continuer de déposer des données chez le récepteur.

Le tableau qui suit rassemble les résultats de cette analyse :

	action du gestionnaire de mémoire virtuelle	erreur d'adressage
émission	perte de confidentialité	perte de confidentialité
réception	perte d'intégrité	perte d'intégrité

Comme nous venons de le voir, la manipulation d'adresses physiques par l'application et la présence de mécanismes de gestion de mémoire dans les systèmes d'exploitation récents peuvent conduire à des problèmes de confidentialité et à des pertes d'intégrité du système. Ces difficultés sont dues, une fois de plus, au caractère zéro-copie du protocole de communication matériel de la machine MPC.

Pour pallier à ces difficultés, nous sommes amenés à nous poser la question suivante : peut-on modifier le fonctionnement du système de gestion de mémoire virtuelle et d'évincement des pages, et peut-on contrôler les informations d'adressage fournies par les applications, afin de garantir la confidentialité et l'intégrité du système global que constitue la machine MPC ?

2.10 Synthèse

Nous avons présenté au début de ce chapitre l'objectif global que l'on cherche à atteindre : fournir un environnement logiciel de communication à la fois adapté à l'élaboration d'applications utilisateur, d'environnements de programmation parallèle et de services noyau. Pour atteindre ce but, on veut s'appuyer sur la primitive d'écriture distante fournie par le matériel, primitive dont on a largement décrit les caractéristiques propres.

Pour dégager les difficultés rencontrées pour construire des services de communication simples à manipuler, et par voie de conséquence à même de nous aider à remplir notre objectif, on a examiné tour à tour le cadre de l'*acheminement des données*, de la *synchronisation des applications* ainsi que de la *tolérance aux fautes*.

On s'est alors rendu compte que **les possibilités d'adaptativité** lors du routage sur le réseau MPC, ainsi que **la caractéristique zéro-copie** de la primitive de dépôt direct fournie par le matériel, nous confrontaient à diverses difficultés dans ces trois cadres d'analyse successifs réunis.

On a donc été amené à se poser, au fur et à mesure de notre réflexion, un certain nombre de questions :

- ✓ Il faut tout d'abord construire, au dessus de la primitive d'écriture distante, un service d'échange de messages rendant opaque la gestion des pages réseau. Il nous faut alors construire, sur ce service de base, des protocoles de plus haut niveau fournissant des services plus évolués.
- ✓ A partir du service d'échange de messages, il faut construire des protocoles plus évolués, et pour ce faire examiner le cadre du contrôle de l'acheminement des données d'un nœud à un autre :
  - Est-il possible pour un nœud récepteur de contrôler la localisation des messages reçus afin d'éviter des dépôts simultanés aux mêmes emplacements ?
  - Peux-t-on localiser facilement les zones de données reçues, à l'aide de l'identificateur de message, seule information disponible dans les nœuds récepteurs ? Comment gérer l'attribution de ces identificateurs dans les nœuds émetteurs ?
  - Quelle méthode, se substituant à la traditionnelle encapsulation logicielle des données, peut-elle être mise en œuvre par le noyau de communication afin d'acheminer correctement les données au sein d'un nœud destinataire ? Existe-t-il une telle méthode, de faible coût et respectant la contrainte zéro-copie de bout en bout ?
- ✓ S'est alors posé le problème de synchronisation des applications communicantes. La contrainte zéro-copie est notamment source de difficultés quant à la durée de validité des tampons de transfert de données. Il faut, pour les résoudre, se poser le problème de la signalisation de fin d'opération.
- ✓ Enfin, partant du principe que ni le matériel constituant le réseau, ni les applications parallèles ne peuvent être garanties sans source d'erreur, on a cherché les questions qui découlaient de cet axiome. Tout d'abord, est-il possible d'imaginer un protocole

rendant transparentes à l'utilisateur les fautes matérielles du réseau? Ensuite, peut-on garantir la confidentialité et l'intégrité du système global que constitue la machine MPC, même en cas de faute logicielle?

On tâchera de trouver des réponses à ces interrogations dans la suite du manuscrit.

Chapitre 3

Historique et état de l'art

Sommaire

Les architectures rencontrées dans le modèle NOW sont le plus souvent composées d'un réseau physique propriétaire interconnectant des nœuds de calcul standards, auxquels on adjoint une carte d'interface entre le réseau propriétaire et le bus système. Pour initier les services de communication, on trouve d'abord des bibliothèques bas niveau, qui proposent un ensemble réduit de services, très efficaces et permettant d'accéder directement aux fonctionnalités de base du matériel. Construits au dessus d'une bibliothèque de ce type, des environnements de programmation haut niveau fournissent alors de nombreux modèles de communication et de synchronisation, et parfois même des primitives de gestion d'autres ressources que celles du réseau : placement des tâches, gestion des groupes, gestion de files de travaux, etc.

3.1 Les architectures matérielles

3.1.1 SCI : Scalable Coherent Interface

Le standard IEEE 1596-1992, SCI (Scalable Coherent Interface [Gustavason, 1992]), a pour but de fournir une mémoire distribuée globale à faible latence à travers un cluster, à l'aide d'un système de cohérence de cache [Chung *et al.*, 2000] à répertoire.

Il utilise des liaisons point-à-point unidirectionnelles électriques ou en fibres optiques, et permet d'atteindre des performances entre 200 Mo/s (CMOS) et 1 Go/s (BiCMOS), sur des distances de plusieurs dizaines de mètres avec des câbles électriques, et de plusieurs kilomètres avec des fibres optiques.

SCI a été conçu pour pouvoir s'interfacer avec différents bus standards : PCI [Cyliax, 2000], VME [VMEbus, 1996], SBus [Sun Microsystems, 2001], etc., et avec des connexions réseau telles que ATM [Foldvik and Meyer, 1995] ou Fiber Channel [Getchell and Rupert, 1992]. Il peut fonctionner en environnement hétérogène, ce qui est un problème plus ardu que la simple interconnexion d'un ensemble de nœuds identiques.

Pour les applications qui ne nécessitent pas les débits maximaux fournis par SCI, les protocoles standards supportent la topologie en anneau, ce qui revient à partager la bande passante disponible sur les liens entre plusieurs nœuds, en évitant le coût d'un commutateur.

On peut connecter, à un commutateur, soit un nœud seul, soit un anneau entier, ce qui permet de former diverses topologies réseau.

Une communication à travers SCI est initiée par un simple accès mémoire, en lecture ou écriture, effectué par un processus, vers un espace d'adressage global unique. L'accès mémoire génère typiquement une faute de cache, ce qui conduit le contrôleur de cache à accéder à la mémoire distante à travers SCI pour récupérer la donnée. Après quelques micro-secondes, celle-ci est alors placée dans le cache et le processeur reprend son exécution.

Aucune bibliothèque en mode utilisateur ni pilote de périphérique logiciel noyau n'entre en jeu dans la communication, ce qui permet d'une part d'obtenir une latence extrêmement faible, et d'autre part de décharger les processeurs du lourd travail de gestion logicielle distribuée d'un cache cohérent.

La société Dolphin produit des cartes SCI pour le bus SBus [Liaaen and Kohmann, 1999] des SPARC de Sun Microsystems, et a annoncé la disponibilité de cartes d'interface PCI [Acher *et al.*, 1999]. Dolphin fournit une version de MPI qui offre une latence de transfert d'un message vide de $12\mu s$ sur une plate-forme SPARC, et prévoit un portage vers le système d'exploitation de Microsoft. Il existe aussi une version de HPF [fos, 1996] adaptée à SCI, fournie par Portland.

Bien que les performances de SCI soient comparables aux autres acteurs de son domaine, il n'a pas réussi à émerger, à cause d'une part du coût important de ce matériel, et d'autre part des difficultés pour étendre à un grand nombre de nœuds un tel réseau.

3.1.2 Memory Channel

Memory Channel [Gillett, 1996] a été développé par Digital à partir de divers travaux dont le projet SHRIMP [Felten *et al.*, 1996]. La première version a été distribuée en 1996, et un an plus tard, ce fut le tour de Memory Channel 2, dont les performances et les fonctionnalités sont nettement accrues.

Un réseau Memory Channel est composé d'adaptateurs PCI et d'un commutateur optionnel [Gillett and Kaufmann, 1997] : comme pour SCI, on peut connecter deux nœuds directement.

Le principal défaut qu'on puisse attribuer à la première version de Memory Channel est d'avoir choisi d'implémenter le réseau en tant que média partagé, au travers duquel une seule transmission peut être active à un instant donné. Avec la seconde implémentation, on est donc passé d'un réseau half-duplex partagé à un crossbar 8x8 full-duplex.

La première version ne fournit que l'écriture distante, tandis que la deuxième propose une primitive de lecture distante.

Le tableau suivant présente une comparaison des caractéristiques des deux versions de Memory Channel :

Caractéristique	Memory Channel 1	Memory Channel 2
largeur du canal	37 bits (half-duplex)	16 bits (full-duplex)
fréquence du lien	33 MHz	66 MHz
longueur maximale du lien en cuivre	4 m	10 m
débit unidirectionnel maximum théorique	133 Mo/s	133 Mo/s
débit soutenu point-à-point	66 Mo/s	100 Mo/s
taille de paquet maximum	32 octets	256 octets
lecture distante	non	oui
taille de page supportée	8 Ko	4 Ko et 8 Ko
Architecture du commutateur	bus partagé	crossbar

Les adaptateurs hôtes communiquent entre-eux pour implémenter un contrôle de flux avec délai de garde afin de détecter les pannes des nœuds ou le blocage des transferts. La détection d'erreur est offerte par le matériel qui implémente la génération et la vérification d'un CRC 32 bits.

Pour permettre les communications, les applications effectuent un *mapping*, en lecture seule ou écriture seule, des pages physiques vers leur espace d'adressage virtuel. Chaque interface hôte contient deux tables (PCT, Page Control Tables) pour gérer ces correspondances : une pour les pages en lecture seule, l'autre pour celles en écriture seule. Ainsi, pour lire et écrire simultanément dans une même région, un nœud doit mettre en place deux *mapping* : l'un pour les lectures, l'autre pour les écritures.

On associe aux pages en lecture seule des pages verrouillées en mémoire physique locale, pour lesquelles on spécifie des attributs tels que *réception autorisée*, *interruption si réception*, *lecture distante*, etc.

Pour chaque page en écriture seule, on crée une entrée de table des pages du processeur l'associant à une page parmi les 128 Mo d'adressage PCI du contrôleur Memory Channel du nœud local. On peut alors définir des attributs du type *broadcast* ou point-à-point, demande d'acquittement, etc.

La figure 3.1 présente un transfert de données sur Memory Channel.

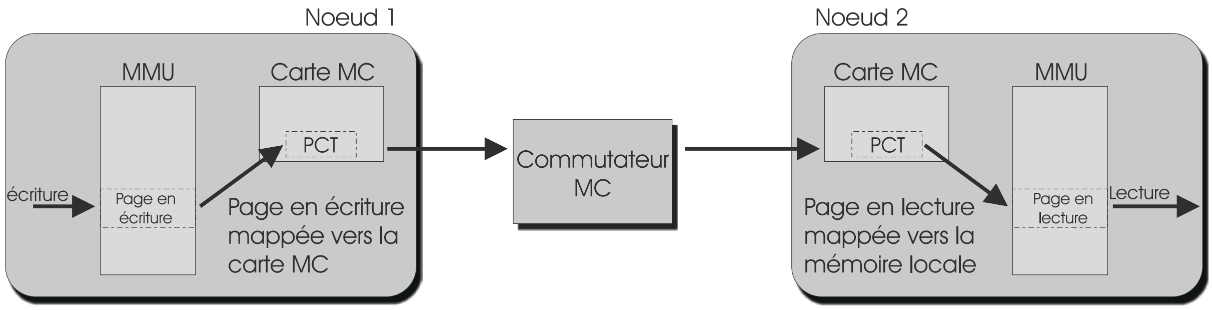


Fig. 3.1 *Transfert de données avec Memory Channel*

Pour assurer un comportement correct et reproductible, Memory Channel délivre les données séquentiellement, et invalide le cache du côté du lecteur pour chaque écriture.

Les *broadcast* sont diffusés vers *tous* les nœuds du réseau : lorsqu'un tel paquet entre dans le crossbar d'un commutateur, celui-ci attend pour le diffuser que toutes les sorties soient disponibles. Les nœuds concernés stockent ces données diffusées dans la page verrouillée en mémoire physique locale associée. Les autres nœuds ignorent simplement les données.

En plus de ces mécanismes de transferts simples, Memory Channel fournit une primitive de lecture distante, une barrière matérielle, ainsi qu'un verrou rapide.

Digital fournit deux interfaces logicielles pour Memory Channel : MCS (Memory Channel Service) et UMP (Universal Message Passing,[Hey, 1992]). Le premier est responsable de l'allocation et du *mapping* des pages de mémoire, tandis que le deuxième implémente des mécanismes simples de passage de message. UMP a été conçu comme un support de l'implémentation de bibliothèques de plus haut-niveau, telles que MPI, PVM et HPF.

MCS et UMP ont été tous-deux implémentés sur Digital Unix et Windows.

Les performances obtenues avec une configuration composée de deux nœuds Alpha 4100 (300 MHz, CPU 21164), raccordés directement sans commutateur, sont assez remarquables :

✓ latence d'un transfert unidirectionnel lors d'un test de ping-pong de 8 octets : 2,2μs (natif), 5,1μs (HPF) et 6,4μs (MPI) ;
✓ débit : 88 Mo/s pour des paquets de 32 octets.

Le reproche principal qu'on puisse faire à Memory Channel tient dans le nombre relativement faible de nœuds pouvant être interconnectés : la plus large configuration consiste à interconnecter 8 serveurs Alpha SMP de 12 processeurs chacun, ce qui constitue un cluster de 96 processeurs.

3.1.3 Myrinet

La technologie Myrinet [Boden *et al.*, 1995] est basée sur deux projets de recherche plus anciens, Mosaic et Atomic LAN [Felderman *et al.*, 1994] de Caltech. L'USC y a aussi participé. Mosaic est un super-ordinateur à grain fin, qui nécessitait un réseau d'interconnexion évolutif, et pour lequel fut créé le projet Atomic LAN, qui peut être vu comme le prototype de recherche ancêtre de Myrinet. Finalement, des membres des deux groupes Caltech et USC ont fondé la société Myricom pour convertir les résultats de leurs recherches en un produit commercial.

L'interface Hôte Myrinet est constituée de deux composants majeurs : le processeur LANai et sa mémoire SRAM associée. Le LANai est un circuit VLSI spécifique qui contrôle les transferts de données entre l'hôte et le réseau. Son composant principal est un microcontrôleur programmable qui contrôle les opérateurs de DMA responsables des transferts hôte/mémoire embarquée et mémoire embarquée/réseau, les messages de données transitant par la mémoire SRAM avant d'être injectés dans le réseau.

Cette SRAM stocke en outre le MCP (Myrinet Control Program), ainsi que différentes files de travaux qui permettent de faire communiquer le LANai avec les pilotes de périphériques ou bibliothèques en mode utilisateur résidant sur l'hôte. En plus du contrôle des données, Le LANai est aussi responsable de la configuration automatique du réseau ainsi que de la surveillance du bon fonctionnement du réseau. La configuration automatique est facilitée par les commutateurs qui sont capables de déterminer la présence ou l'absence d'un lien raccordé à un de leurs ports.

Comme on peut le constater sur la figure 3.2 page suivante, le LANai dispose de trois opérateurs de DMA : le premier pour les transferts entre l'hôte et la mémoire du LANai, le deuxième pour les transferts entre la mémoire du LANai et l'interface de paquetisation, et enfin un dernier pour les réceptions de données provenant de l'interface de paquetisation et à destination de la mémoire du LANai.

Les premières versions du réseau Myrinet étaient basées sur une interface hôte pour le SBus de Sun Microsystems, et Myricom commercialise maintenant des cartes hôte pour bus PCI 32 et 64 bits. Le LANai est cadencé de 33 MHz à 66 MHz.

Les premiers liens Myrinet bidirectionnels étaient constitués de canaux de 9 bits parallèles

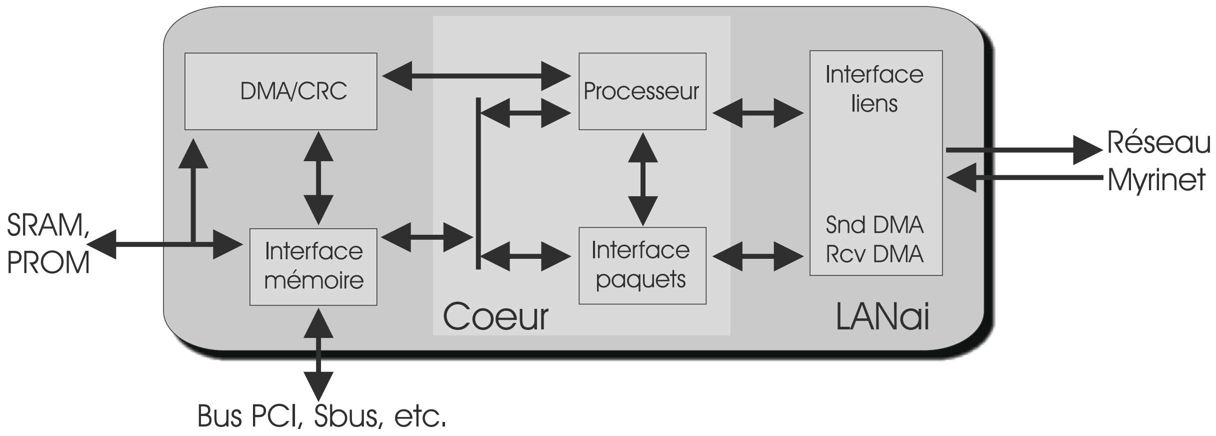


Fig. 3.2 *LANai*

par sens de transmission, cadencés à 80 MHz, ce qui permettait d'offrir une bande passante maximale de 160 Mo/s. Les câbles fournis pouvaient avoir une longueur de 10 m à vitesse maximale et jusqu'à 25 m à vitesse réduite de moitié. Des liens optiques sont aujourd'hui disponibles, et des cartes aux débits encore plus importants sont proposées par Myricom.

Les paquets de données, de longueur quelconque, sont acheminés à travers un commutateur de type wormhole, suivant le principe du routage par la source. Chaque paquet est constitué d'une en-tête de routage, d'un champ de typage, des données à transporter, ainsi que d'une somme de contrôle (CRC). Le commutateur retrouve dans le début de l'en-tête le numéro de port destination, et se contente de transmettre la suite du paquet, privé de ce numéro, sur le port de sortie. Lorsqu'il atteint l'hôte destinataire, l'en-tête de routage du paquet a été complètement consommée. Des symboles de contrôle (STOP/GO) sont utilisés pour implémenter un contrôle de flux en sens inverse.

Les commutateurs Myrinet sont disponibles en version 4, 8 et 16 ports, et peuvent être raccordés entre-eux, ce qui permet d'obtenir toute topologie de réseau.

La figure 3.3 présente un commutateur 16 ports et une carte hôte Myrinet.

Fig. 3.3 *Carte et commutateur Myrinet*

La somme de contrôle doit être calculée à chaque étape du transit d'un paquet dans le réseau, car l'en-tête du paquet y est modifiée (écourtée). Myricom annonce un MTBF de plusieurs millions d'heures pour ses commutateurs et ses interfaces hôtes. En cas de détection de fautes sur un câble, ou de panne d'un nœud, le LANai calcule des routes alternatives.

Les spécifications de Myrinet sont publiques, ce qui est lié pour une part à son succès. Notamment, les sources du pilote de périphérique ainsi que le programme du MCP sont distribués afin de servir de documentation, et de permettre le portage de nouveaux protocoles sur Myrinet. Cela a ainsi motivé de nombreuses équipes de recherche à implémenter leur propres couches de protocoles de passage de message sur Myrinet. Des pilotes de périphériques sont disponibles sur Linux, Solaris, Windows, Digital Unix, Irix et VxWorks, pour les architectures à base de Pentium, Sparc, Alpha, MIPS et PowerPC. On trouve même une version du compilateur C GNU, gcc, modifiée afin de compiler des programmes pour MCP.

Les performances de quelques bibliothèques de passage de messages construites sur Myrinet sont présentées dans le tableau suivant (source [Buyya, 1999]) :

Machine	API	Latence (μs)	Débit (Mbit/s)
Pentium Pro 200 MHz	BIP	4,8	1009
Pentium 166 MHz	PM	7,2	941
Ultra-1	AM	10	280
Pentium Pro 200 MHz	TCP (Linux/BIP)		293
Pentium Pro 200 MHz	UDP (Linux/BIP)		324
DEC Alpha 500/266	TCP (Digital Unix)		271
DEC Alpha 500/266	UDP (Digital Unix)		404

On peut noter que les meilleures performances sont atteintes par BIP. Myricom fournit maintenant une nouvelle couche de protocoles, GM, qui remplace dorénavant les anciens pilotes de périphériques et programmes MCP. Elle propose des accès protégés par l'intermédiaire du noyau, ou en mode utilisateur, en évitant ainsi les coûteux appels systèmes. GM permet d'atteindre un débit de 75 Mo/s pour des paquets de 4 Ko. La latence passe de $40\mu s$ pour un message de 4 octets, à $100\mu s$ pour 4 Ko, mesurée sur un PC Linux. Des messages de 200 octets utiles émis avec MPICH génèrent une latence de $40\mu s$ et de 30 Mo/s environ. L'interface VIA a été implémentée récemment sur GM [Chelius, 2001].

L'énorme avantage de Myrinet est de disposer sur la carte hôte d'un micro-contrôleur programmable. Il est dommage que ce contrôleur ne puisse être cadencé qu'à des fréquences relativement faibles. D'autre part, le passage intermédiaire des données par la mémoire embarquée empêche la construction de réels protocoles de type zéro-copie, et c'est une des raisons pour lesquelles les performances pour les paquets de petite ou moyenne taille sont modérées.

3.1.4 Fast Ethernet, ATM, HiPPI, ServerNet, Synfinity

Les trois architectures matérielles que nous venons d'étudier, Memory Channel, Myrinet et SCI, sont certainement parmi les plus en vogue aujourd'hui, mais elles ne sont bien sûr pas les seules solutions proposées pour construire des grappes.

Citons donc quelques-unes des autres solutions proposées :

- ▷ Fast Ethernet constitue l'évolution logique des réseaux moyen-débit Ethernet. Fast Ethernet offre un débit de 100 Mbit/s pour un coût de moins de 200 F par adaptateur, ce qui en fait la solution la moins chère du marché. Les nœuds peuvent partager leur média d'interconnexion, à travers un HUB, et c'est alors le protocole CSMA/CD (Carrier Sense Multiple Access with Collision Detection) qui est utilisé par l'attribution de la ressource réseau. Mais si l'on adjoint un commutateur, tous les nœuds pourront alors disposer du débit maximum.

  Pour interconnecter des commutateurs Fast Ethernet, on dispose maintenant de la technologie Gigabit Ethernet [Stallings, 2000]. Récemment, un prototype de communication optique supportant un trafic Ethernet à 10 Gbit/s a été mis en place avec succès [Woodward *et al.*, 2000].

  Dans [Kurmann *et al.*, 2001], les auteurs présentent une tentative pour accroître les performances des communications sur réseaux Ethernet en agissant au niveau logiciel : l'objectif est de modifier le pilote de la carte d'interface Ethernet, ainsi que les couches TCP/IP qui se greffent dessus, afin d'obtenir un acheminement des données de type zéro-copie au sein d'un nœud émetteur ou récepteur. Pour ce faire, les tampons de données ne sont donc pas recopiées mais remappés grâce au module de mémoire virtuelle du système d'exploitation. Une approche semblable pour implémentater TCP/IP sur la machine MPC a été conduite il y a cinq ans par le laboratoire PRiSM de l'Université de Versailles. Cela a permis de constater que le coût du remmaping mémoire était un frein important à l'obtention de bonnes performances avec une telle approche.
- ▷ ATM, Asynchronous Transfer Mode, est une technologie de commutation de paquets dans un mode orienté connexion. Elle est très utilisée dans le monde des télécommunications, peut fédérer un réseau de clusters [Foldvik and Meyer, 1995], ou bien même fournir l'interconnexion d'un machine massivement parallèle. ATM est néanmoins plus utilisé pour les réseaux moyenne ou longue distance. Son principal intérêt vis-à-vis des autres solutions étudiées ici réside dans les possibilités évoluées de gestion de la qualité de service. Il existe différentes architectures logicielles permettant de transporter TCP/IP au dessus de ATM ([Girardi *et al.*, 1996] et [Rindos *et al.*, 1996]), ce qui est pour une part probablement non négligeable dans l'essor de la technologie ATM, qui ne s'est pas imposée aussi vite que prévu [Cochran, 1999].
- ▷ HiPPI [Tolmie *et al.*, 1999], High Performance Parallel Interface, permet de transférer, sur du câble électrique, des données à un débit de 800 Mbit/s (solution avec 32 lignes parallèles) ou 1,6 Gbit/s (64 lignes parallèles).
- ▷ ServerNet [Avresky *et al.*, 1999] est une technologie de réseau rapide racheté par Compaq, qui permet d'atteindre 125 Mo/s entre deux nœuds. Dans la conception de ServerNet, l'accent a particulièrement été mis sur la fiabilité [Shurbanov *et al.*, 2000] et la tolérance aux fautes [Huang *et al.*, 1999], autant au niveau matériel qu'au niveau logiciel.
- ▷ Synfinity [Ito, 2000], développé par Fujitsu, supporte conjointement le passage de message et la mémoire partagée à des débits très importants : jusqu'à 1,6 Go/s. En plus des opérations send/receive traditionnelles, le support matériel des transactions

mémoire distantes et de la synchronisation permettent d'implémenter dans le logiciel des protocoles optimisés. Cela a ainsi permis la mise en place du support de l'interface de programmation optimisée VIA [Dunning *et al.*, 1998] à travers un ensemble de clusters Unix interconnectés par Synfinity [Kishimoto *et al.*, 2000].

3.2 Les bibliothèques de communication de bas niveau

3.2.1 Techniques d'optimisation

On trouve de très nombreuses bibliothèques de communication bas-niveau, certaines pour des architectures dédiées, d'autres généralistes et bénéficiant de plusieurs portages. On peut classer ces bibliothèques en deux catégories majeures : les bibliothèques en mode utilisateur, qui ne font pas appel à un pilote de périphérique noyau pour communiquer avec le matériel réseau, et les bibliothèques qui utilisent des services noyau.

Les bibliothèques en mode utilisateur bénéficient d'une très faible latence de par l'absence d'appel système, mais se pose alors le problème de la sécurité et de l'intégrité du système (l'interface de communication PAPI [Renault *et al.*, 2000], compilable au choix en mode utilisateur et en mode noyau, a permis d'analyser l'impact de la sécurité sur les performances dans [Éric Renault, 2000]).

Différentes autres techniques sont mises en œuvre par les bibliothèques de communication bas niveau pour optimiser les communications :

- ▷ Utilisation de plusieurs réseaux simultanément : cela demande plusieurs adapteurs par nœud, ce qui augmente drastiquement le coût du matériel ;
- ▷ Simplification à l'extrême des protocoles de communication : on peut par exemple se dispenser de contrôle d'erreur ou de flux, et laisser au matériel le soin de s'en charger ;
- ▷ Suppression des tampons intermédiaires : hormis pour les paquets de petite taille, les tampons intermédiaires sont toujours source de baisse de performances. Pour s'en affranchir, les bibliothèques bas-niveau peuvent privilégier un mode de communication à un autre (par ex. communications synchrones), ou utiliser des fonctionnalités particulières du matériel sous-jacent (par ex. DMA fourni par la carte hôte) ;
- ▷ Interruptions rapides : la signalisation par interruption induit une latence importante dans le traitement des communications. Certaines bibliothèques utilisent un gestionnaire logiciel d'interruption optimisé pour gérer leurs communications. Celui-ci peut par exemple se charger seul d'effectuer tout le traitement applicatif lié à l'interruption directement depuis le noyau, sans faire intervenir le processus concerné. On évite ainsi un signal ou tout autre moyen de communication noyau-processus ;
- ▷ Scrutation : pour éviter le surcoût lié aux interruptions, on peut par exemple s'affranchir complètement de celles-ci et utiliser des techniques de scrutation.

3.2.2 Beowulf

Le projet Beowulf [Sterling *et al.*, 1995] consiste à utiliser les protocoles de communication standards du monde Unix, sur une grappe de PC Linux. Pour accroître les performances, plusieurs interfaces réseau sont utilisées simultanément, grâce à un pilote noyau spécifique.

Une topologie classique avec Beowulf consiste à mettre en place une grille à deux dimensions : chaque nœud est connecté à deux réseaux Ethernet nommés respectivement *horizontal* et *vertical*. Ainsi, chaque nœud est adjacent à tous ses homologues des réseaux horizontal et vertical auxquels il appartient. D'autre part, chaque nœud agit comme un routeur logiciel : deux nœuds non adjacents quelconques disposent donc de deux plus courts chemins distincts les reliant. Ces deux chemins sont alors utilisés simultanément lors des communications entre ces deux nœuds non adjacents.

Il existe des plate-formes Beowulf contenant plusieurs centaines de processeurs. Dans [Michalickova *et al.*, 2000], les auteurs présentent une plate-forme composée de 216 processeurs interconnectés par des commutateurs Gigabit Ethernet. Pour administrer des clusters Beowulf de telles tailles, des outils d'administration spécifiques ont été conçus [Uthayopas *et al.*, 2000].

3.2.3 AM

Les Messages Actifs (AM, Active Messages, [Eicken, 1994]) se différencient du modèle traditionnel de type *send/receive* [Wallach *et al.*, 1995]. Il s'agit plutôt d'un modèle de communication unilatéral : quand un émetteur initie un échange, celui-ci est effectué indépendamment de l'activité courante du processus récepteur. Il n'y a donc pas d'opération *receive*.

Cette sémantique particulière des Messages Actifs permet de se passer de tampons de stockage temporaires, ce qui accroît considérablement les performances. Et, si l'on dispose d'un support matériel adéquat, on peut très facilement superposer les communications et les calculs.

En effet, dans les systèmes de communication standard par passage de message, on utilise un tampon de stockage temporaire chez le destinataire, en attendant que celui-ci consomme les données. Avec les Messages Actifs, l'arrivée d'un message chez un récepteur provoque l'invocation d'une fonction dans le processus récepteur. Cette fonction, nommée *receiver handler*, est exécutée dans le contexte d'un processus léger qui lui est propre, afin de consommer les données.

On découple ainsi la gestion du message de l'activité courante du fil d'exécution principal du processus destinataire. Le *receiver handler* se charge par exemple de positionner des structures de données, pour la gestion du message suivant, ou pour donner une information au fil d'exécution principal.

Pour faciliter le travail des processus émetteurs, l'espace d'adressage qui contient le code de l'application est partagé entre les émetteurs et récepteurs, il s'agit donc d'un modèle de fonctionnement SPMD (Single Program, Multiple Data).

L'implémentation commerciale des Messages Actifs la plus diffusée est la bibliothèque CMAML (Connection Machine Active Message Layer), destinée à l'architecture CM-5, mais on trouve différentes autres implémentations qui suivent les principes généraux définis dans cette interface, chacune s'adaptant à une architecture matérielle bien précise.

3.2.4 Fast Sockets

Les Fast Sockets constituent une implémentation au sein d'une bibliothèque utilisateur des Sockets POSIX en mode connecté, au dessus de l'interface des Messages Actifs. Lorsque des nœuds sont séparés par un réseau TCP/IP, l'interface Fast Sockets utilise alors les protocoles TCP/IP standards.

Les performances mesurées entre des machines UltraSPARC-1 sous Solaris, et interconnectées par Myrinet, sont plutôt faibles : 57,8μs de latence et 32,9 Mo/s de débit.

3.2.5 BIP

BIP [Prylli and Tourancheau, 1998] est une bibliothèque en mode utilisateur pour plate-forme Myrinet et nœuds de type Pentium/Linux qui permet d'atteindre de plus hautes performances que les bibliothèques natives fournies par Myricom, ce qui a contribué à son énorme succès. L'API de BIP est composée d'une trentaine de points d'entrée [Prylli, 1998].

La première raison pour expliquer ces performances réside dans la mise à disposition directe, sans protection, des registres de l'adaptateur Myrinet aux différents processus. Il n'y a donc pas de multiplexage par l'intermédiaire du noyau, ce qui accroît les performances, mais pose des problèmes de sécurité et d'intégrité du système.

BIP offre des communications de type send/receive, bloquantes ou non au choix, et implémentées suivant un mode de rendez-vous. Néanmoins, les très petits paquets peuvent s'affranchir du rendez-vous par l'utilisation de tampons intermédiaires.

Comme nous l'avons indiqué auparavant, les communications sur Myrinet sont très fiables et le contrôle de flux est géré par le matériel, deux conditions qui permettent à BIP de s'affranchir du traitement des fautes de transmission ou des pertes de données. BIP se contente donc de signaler les fautes ou pertes, mais il ne met en œuvre aucun protocole de contournement, ce qui permet d'alléger considérablement les opérations.

Bien que Myrinet n'impose pas de taille limite pour les paquets de données, BIP fragmente ceux-ci, de façon transparente, afin de mettre en œuvre un pipeline le long du chemin de données entre l'émetteur et le récepteur.

BIP permet d'obtenir 96 % du débit maximum théorique de Myrinet (c-à-d 126 Mo/s pour un maximum théorique de 132 Mo/s), avec une latence très faible (4,3μs).

Le portage de TCP/IP sur BIP conduit à une latence de 70μs et un débit de 35 Mo/s, ce qui montre l'inadéquation de TCP/IP au monde des machines parallèles. Le portage de MPICH sur BIP fournit quant-à-lui des résultats intéressants : 12μs de latence et un débit de 113 Mo/s.

3.2.6 Fast Messages

Les Fast Messages [Iannello *et al.*, 1998] de l'Université d'Illinois consistent en une bibliothèque de type Messages Actifs implémentée sur une plate-forme Myrinet et fournissant une garantie d'acheminement ordonné des données, un système de contrôle de flux, et la retransmission des paquets en cas de besoin [Pakin *et al.*, 1997].

La première version des FM était uniquement destinée aux grappes de stations SPARC. Les performances ont été accrues avec la venue de la deuxième version des FM, qui permet d'utiliser le bus PCI de nœuds de type Pentium. Les performances de la seconde version sont néanmoins réduites en comparaison de celles de BIP : $11 \mu s$ de latence et un débit de 77 Mo/s.

3.2.7 U-Net

Deux version majeures de U-Net [Eicken *et al.*, 1995] ont vu le jour : U-Net pour ATM, qui nécessite un support matériel spécifique, et U-Net pour Fast Ethernet qui fonctionne avec n'importe quel type d'adaptateur Ethernet.

U-Net pour ATM

U-Net pour ATM offre aux processus utilisateurs un accès direct au périphérique matériel d'interface réseau, d'ailleurs l'interface de programmation de U-Net est assez semblable à celle d'une carte réseau. Les couches de communication sont donc implémentées en mode utilisateur.

L'intérêt majeur d'U-Net, vis-à-vis de la plupart des autres bibliothèques en mode utilisateur, réside dans la persistance de la protection et de la garantie d'intégrité du système, malgré le choix du mode utilisateur. Pour ce faire, le multiplexage de l'accès au périphérique réseau est assuré par un processeurs Intel i960 localisé sur l'adaptateur réseau.

L'adaptateur est virtualisé par des *terminaisons* (*end-point*), distribuées aux différents processus. Chaque terminaison est constituée d'un tampon de mémoire noyau, d'une file d'émission et d'une file de réception, permettant la synchronisation processus/adaptateur. Le tampon des terminaisons constitue ainsi un stockage intermédiaire pour les opérations d'émission et de réception.

Le rôle du système d'exploitation se limite ainsi à établir des *mapping* mémoire entre les tampons et files des terminaisons et les espaces d'adressage des processus. Lors des communications, il n'y a alors plus aucune intervention du système d'exploitation.

Les performances de U-Net sont très proches des performances du lien matériel ATM à 155 Mb/s : $44{,}5 \mu s$ de latence et 15 Mo/s de débit.

U-Net pour Fast Ethernet

La version Fast Ethernet de U-Net est destinée aux adaptateurs Fast Ethernet standards. Son objectif est de conserver une interface sensiblement identique à U-Net pour ATM, et de préserver les garanties de sécurité et d'intégrité du système.

Mais l'absence de processeur programmable sur ce type d'adaptateur empêche la virtualisation en mode utilisateur des terminaisons. Ainsi, U-Net pour Fast Ethernet n'est pas un protocole en mode utilisateur, mais il dispose au contraire d'un support noyau spécifique pour multiplexer les accès à l'adaptateur réseau.

3.2.8 VIA

VIA, Virtual Interface Architecture, est une tentative de standardisation d'une interface en mode utilisateur, poussée par Intel, Compaq et Microsoft.

VIA spécifie une architecture de communication, proche de U-Net, en y ajoutant l'opération de DMA distant. Cela permet de s'affranchir de l'utilisation d'un tampon de mémoire intermédiaire lors des communications, pour accroître les performances [Dunning *et al.*, 1998].

La spécification VIA impose la détection d'erreur et la protection par multiplexage de l'interface entre les différents processus. Elle n'impose néanmoins pas (mais recommande), ni la fiabilité des communications, ni l'implémentation de la virtualisation de l'interface au sein de l'adaptateur réseau.

Sur VIA, on trouve des applications, des protocoles par passage de message, ou enfin des systèmes de communication par mémoire distribuée [Rangarajan and Iftode, 2000].

3.2.9 Chameleon, CHIMP, P4, LAM

De nombreuses autres bibliothèques en mode passage de message ont vu le jour, certaines généralistes, d'autres profitant d'optimisations permises par une architecture cible spécifique. Citons quelques-unes parmi les plus populaires :

- ▷ Chameleon permet de s'adapter à une topologie dynamique de la machine ;
- ▷ CHIMP (Common High-level Interface to Message Passing, [Alasdair *et al.*, 1994]) offre peu de possibilités, hormis le simple passage de messages point-à-point, mais est en contre-partie aisément portable ;
- ▷ P4 (Portable Programs for Parallel Processors, [Butler and Lusk, 1994]) supporte notamment les réseaux hétérogènes ;
- ▷ LAM (Local Area Multicomputer) propose au choix, et de manière complètement transparente, des communications directes entre tâches distantes ou à travers un daemon local *lamd* (un seul par nœud), ce qui facilite le debugage ainsi que la surveillance du bon déroulement de l'application.

3.2.10 Performances comparées

Le tableau suivant (source [Buyya, 1999]) montre les performances comparées de quelques-unes des bibliothèques étudiées précédemment, et les oppose aux performances de TCP/IP, pour des échanges de type ping/pong :

Plate-forme	Latence (μs)	Débit (Mo/s)
TCP/IP Linux 2.0.29, half-duplex (3COM 3c595 100base-T, Pentium 133 MHz, DMA par le CPU)	113,8	6,6
TCP/IP Linux 2.0.29, half-duplex (3COM 3c905 100base-T, Pentium II 300 MHz, DMA par l'adaptateur)	64,4	10,8
Fast Sockets (Myrinet, UltraSPARC)	57,8	32,9
U-Net/Fast Ethernet (DEC DC21140 100base-T)	30,0	12,1
U-Net/ATM (FORE PCA-200 ATM)	44,5	15,0
VIA Linux (DEC DC21140 100base-T)	33,0	11,9
BIP (Myrinet, Pentium Pro)	4,3	126,0
Fast Messages 2 (Myrinet, Pentium Pro)	11,0	77,0
MPC	4,0	61,8

3.3 Les environnements de programmation de haut niveau

3.3.1 MPI

MPI, Message Passing Interface, est, comme son nom l'indique, une spécification d'interface de passage de messages, et non une implémentation particulière d'une bibliothèque de communication. MPI a été défini par le MPI Forum, rassemblant des industriels, des équipes de développeurs ainsi que des utilisateurs. Évidemment, les différentes équipes de développement n'avaient pas toutes à l'esprit les mêmes environnements d'exécution, et les différents utilisateurs ne partageaient pas forcément les mêmes applications cibles. MPI est donc le fruit d'une âpre négociation, au sein du MPI Forum, qui a abouti à la spécification d'une interface généraliste, aux nombreuses possibilités, et adaptée à des applications très diverses.

Les choix primordiaux qui ont dicté la spécification de MPI peuvent être énumérés comme suit :

- ▷ MPI est une bibliothèque permettant d'écrire des applications parallèles, et non pas un système d'exploitation distribué. Ce choix a eu des conséquences importantes sur la gestion des ressources d'une machine parallèle par MPI.
- ▷ La spécification n'impose pas le support des processus légers, mais l'autorise, afin de supporter les grappes de nœuds SMP. Cela veut dire que toute notion de tampon courant, code d'erreur courant, etc., est proscrite dans MPI.

- ▷ La spécification MPI doit pouvoir évoluer et s'étendre. Pour cela, une approche orientée objet est privilégiée, mais sans imposer l'utilisation d'un langage orienté objet. Ainsi, les objets sont simplement manipulés par des fonctions de l'interface MPI, ce qui explique en partie le grand nombre de fonctions introduites dans la spécification.
- ▷ Sans l'imposer à toutes les implémentations, le support des architectures hétérogènes est pris en charge par l'interface.
- ▷ Tous les comportements de l'interface MPI doivent être prédictifs, sans laisser de choix à l'implémentation sous-jacente.

Parmi les nombreuses fonctionnalités définies par MPI, on trouve les opérations de groupe, différents mécanismes de communication, bloquants et non bloquants, et la gestion dynamique des tâches. Mais la profusion de fonctionnalités, et donc la multiplicité des points d'entrée dans l'interface MPI, conduisent le plus souvent les utilisateurs à n'utiliser qu'un sous-ensemble réduit de l'interface.

De nombreuses implémentations de MPI ont vu le jour, certaines propriétaires et optimisées pour une unique architecture matérielle, d'autres structurées en sous-modules permettant de faciliter le portage d'une architecture à une autre, en isolant les primitives de communication les plus rudimentaires au sein d'un même module de bas niveau. MPICH [ANL, 2001] et LAM/MPI sont les implémentations libres qui ont le plus de succès. Il existe des implémentations sur VIA de MPICH (MVICH [Ong and Farrell, 2000]) et de LAM/MPI ([Bertozzi *et al.*, 2001]).

FT-MPI [Fagg and Dongarra, 2000] est une implémentation de MPI tolérante aux fautes particulièrement adaptée aux grappes comprenant plusieurs centaines de processeurs. Elle n'est pas transparente pour le programmeur : le comportement en cas de fautes et leur traitement est controllé par l'application.

3.3.2 PVM

À la différence de MPI, PVM constitue avant tout une implémentation [Geist *et al.*, 1994], développée par une équipe de recherche unique. L'interface proposée par PVM est rudimentaire vis-à-vis de celle de MPI, et probablement pour cette raison plus facile d'accès.

PVM a longtemps évolué de façon anarchique, les utilisateurs apportant des modifications en rapport à leurs besoins directs. Mais cela ne l'a pas empêché d'obtenir un grand succès dans de nombreux domaines, car il s'agit d'un système rapidement utilisable pour un non informaticien.

Aujourd'hui, les interfaces MPI et PVM tendent à converger (la gestion dynamique des processus, originaire de PVM, est maintenant prise en charge au sein de MPI, et les groupes statiques et les contextes de messages, services initialement fournis par MPI, sont maintenant intégrés à PVM). Mais en terme de nombre d'utilisateurs, PVM est progressivement dépassé par MPI.

3.3.3 PM2

PM^2, Parallel Multithreaded Machine, propose un environnement d'exécution parallèle conçu dans le cadre du projet ESPACE (Execution Support for Parallel Applications in high-performance Computing Environments) de l'équipe GOAL (Groupe Objets et Acteurs de Lille).

Il se distingue radicalement de PVM et MPI, au sens où il ne se conforme pas au traditionnel modèle de passage de messages : les communications prennent place au sein d'appels de procédures distantes peu coûteux (LRPC, Lightweight Remote Procedure Call) entre processus légers (threads) [Namyst *et al.*, 1995].

La notion de processus légers étant intégrée à la bibliothèque de passage de messages, la migration et la préemption de threads est alors permise, ce qui autorise PM^2 à travailler en environnement hétérogène ou dynamique.

Il est composé de deux sous-systèmes :

✓ Marcel est une bibliothèque de gestion des processus légers en mode utilisateur, qui a été développée spécifiquement pour PM^2. C'est l'intégration d'une bibliothèque de processus légers au sein de la bibliothèque de communication qui permet la migration des processus communicants.
  Cela a permis d'autre part une grande flexibilité dans la paramétrisation de la gestion des threads, notamment au niveau des conditions de préemption, de la distribution des quanta de temps processeur, et de l'allocation des ressources mémoire entre les différentes activités.

✓ Madeleine est une interface de communication entre processus légers, qui implémente des mécanismes d'appel de procédure distante. Il s'agit d'une interface portable, qui a été mise en œuvre au dessus de différentes bibliothèques de passage de messages, telles que BIP, MPI et TCP/IP.
  Madeleine est décomposée en deux couches logicielles superposées, la plus haute définissant l'interface de programmation, et la plus basse se chargeant de l'interface avec le système de communication sous-jacent. Pour porter Madeleine, il suffit donc de porter la couche inférieure, ce qui constitue un avantage certain.
  Madeleine II [Aumage *et al.*, 2000], évolution de Madeleine, permet de contrôler simultanément plusieurs interface réseau (BIP, SISCI, VIA) et plusieurs types d'adaptateurs réseau (Ethernet, Myrinet, SCI), au sein d'une même session.

3.3.4 Linda

Linda constitue un modèle de programmation parallèle assez ancien mais très original, développé à l'université de Yale. Les processus communiquent à l'aide de Linda au travers d'une mémoire associative globale, appelée *espace des tuples* [Carriero *et al.*, 1994].

Un processus qui désire communiquer une information génère un nouveau tuple, et l'ajoute à l'espace des tuples. Tout autre processus peut alors effectuer une recherche associative sur la clé et certains des champs des tuples pour récupérer cette information.

Pour mettre à jour un tuple, il faut l'extraire de l'espace des tuples, le modifier, puis l'y réinsérer. De cette manière, on évite les problèmes d'accès concurrents et de verrouillage.

Un processus peut générer un *live tuple*, pour effectuer un calcul ou une opération intermédiaire. Cela crée un nouveau processus, qui s'exécute en parallèle de son processus générateur. Lorsque l'exécution est terminée, le *live tuple* est transformé en un tuple ordinaire, dont la valeur est le résultat du processus intermédiaire. Ce résultat est alors placé dans l'espace des tuples, et devient donc accessible par tous les processus, et notamment par le processus générateur.

Six opérations sont définies sur l'espace des tuples :

- ✓ *out* : insère un tuple dans l'espace des tuples ;
- ✓ *in* : extrait un tuple de l'espace des tuples ; le processus qui invoque *in* est bloqué jusqu'à ce qu'on puisse trouver, dans l'espace des tuples, un élément compatible avec les critères de la recherche associative fournis lors de l'appel à *in* ;
- ✓ *inp* : version non bloquante l'opération *in* ;
- ✓ *rd* : opération bloquante qui permet de lire la valeur d'un tuple, sans l'extraire de l'espace des tuples ;
- ✓ *rdp* : version non bloquante de l'opération *rd* ;
- ✓ *eval* : crée un *live tuple*, c'est-à-dire un nouveau processus qui se terminera par l'insertion de son résultat dans l'espace des tuples.

Linda est un modèle de programmation parallèle, et non un langage à part entière. Parmi les implémentations de Linda, Network Linda, de SCA (Scientific Computing Associates), fournit une version C et une version Fortran d'un compilateur proposant le modèle de programmation Linda (la version Fortran est construite au dessus du compilateur C). Network Linda communique à l'aide du protocole UDP, et en cas de configuration hétérogène, il utilise l'encodage Sun XDR pour permettre à des nœuds constitués d'architectures distinctes de communiquer entre eux, de manière transparente pour le programmeur.

3.4 Sécurisation

Dans les machines parallèles, le problème de la sécurisation des protocoles bas-niveau s'est toujours heurté aux exigences de performances. La plupart des machines s'appuient sur le matériel pour garantir l'absence de faute. Celui-ci est validé avec un MTBF (Mean Time Between Failures) très faible, ce qui permet de s'abstenir de prendre en compte les conséquences d'une faute de communication.

On trouve aussi d'autres approches, qui font intervenir le logiciel :

- Redondance : en optant pour une topologie de réseau dans laquelle on peut joindre deux nœuds par deux chemins distincts (topologies en double-anneau, ou grille à deux dimensions comme dans Beowulf), l'application peut, même en cas de panne grave d'un lien, continuer à fonctionner correctement.
- Correction de bout en bout par le protocole de communication : les réseaux ATM ou IP autorisent les routeurs à ne pas traiter une partie des paquets en cas de surcharge

de débit. De même, Ethernet est un protocole qui ne garantit pas l'acheminement correct des données. Les protocoles de communication implémentés dans les nœuds de calcul vont ainsi devoir prendre en compte, de bout en bout, les pertes éventuelles. Par exemple, TCP propose un mécanisme basé sur des numéros de séquence sur chaque canal de communication permettant de corriger de telles pertes.
- Prise en compte par la bibliothèque de programmation : certaines bibliothèques de programmation proposent un mécanisme de reprise sur erreur. Par exemple, PVM, quand il utilise le protocole UDP non fiable, effectue lui-même les opérations de prise en compte des pertes de données et de réémission.
- Correction assistée par l'application : pour des fautes plus graves qu'une simple perte de paquet, par exemple la perte définitive d'un lien permettant d'accéder à un nœud particulier, certains environnements de programmation parallèle se proposent d'en avertir l'application afin d'autoriser son bon fonctionnement même en cas de panne grave. FT-MPI [Fagg and Dongarra, 2000] en est un exemple typique.

3.5 Conclusion

Pour pouvoir fournir une machine parallèle généraliste, c'est-à-dire capable de résoudre des problèmes dans des domaines scientifiques bien différents, en s'adressant à des utilisateurs aux profils et besoins largement distincts, il est évident que l'interface fournie au programmeur d'application parallèle doit être, elle-aussi, généraliste. Il est utopique de demander à un tel programmeur de porter son application sur une bibliothèque de communication bas-niveau, si efficace soit-elle.

Mais pour tirer le meilleur parti d'une architecture matérielle innovante, on l'a vu dans la deuxième partie de ce chapitre, il faut nécessairement intégrer des mécanismes de communication spécifiques aux particularités matérielles de la machine utilisée.

Partant de ces considérations, le choix de fournir une bibliothèque de communication exploitant au mieux les caractéristiques propres au réseau rapide de la machine MPC s'imposait. C'est la conception de cette bibliothèque, et des protocoles qu'elle met en œuvre, que nous allons donc étudier dans la suite de cette discussion.

Chapitre 4

PROTOCOLE DE COMMUNICATION BAS-NIVEAU

Sommaire

La couche de communication PUT, dont nous décrivons et justifions les choix architecturaux dans ce chapitre, s'appuie sur la primitive matérielle d'écriture distante. Afin de pouvoir s'adapter au plus grand nombre de modèles de programmation, cette couche de plus bas niveau doit être capable de suivre tous les comportements exigibles par le programmeur.

4.1 Implantation mixte

4.1.1 Système d'exploitation standard

Dans [Heath *et al.*, 2001], les auteurs comparent l'augmentation constante de la vitesse de calcul des processeurs à l'évolution des performances des opérations d'entrée-sortie, pour en déduire à l'aide d'une analyse basée sur le modèle logP [Mei-E, 2000], que le fossé qui sépare les performances obtenues avec des protocoles standards tels que TCP ou UDP se rapprochent de celles obtenues avec des protocoles spécialisés. Néanmoins, force est de constater que le matériel disponible aujourd'hui nécessite des protocoles spécifiques pour obtenir les meilleurs performances. Nous nous sommes donc engagés dans la conception de protocoles spécifiques optimisés, mais en gardant un contexte standard, qui prend forme dans le système d'exploitation choisi.

La machine MPC est constituée de nœuds de type PC standard mono ou multi-processeurs (Intel/Pentium). On a pris le parti d'adapter le système d'exploitation Unix FreeBSD à notre machine parallèle. FreeBSD, proche de NetBSD et dérivé de BSD-4.4, est un système Unix libre particulièrement bien adapté aux machines de type PC/Intel.

Pourquoi ce choix de FreeBSD ?

Il nous fallait tout d'abord un système d'exploitation libre, c'est à dire un système dont les sources sont accessibles et modifiables à volonté. Il nous fallait aussi un système fiable et éprouvé.

Deux systèmes libres et éprouvés sont actuellement largement utilisés sur les machines à base de processeur Intel : Linux et FreeBSD.

Nous avons vu dans le chapitre 2 que la primitive d'écriture distante de la carte FastHSL impose un comportement de la carte réseau vis-à-vis de la mémoire souvent différent de celui qu'on rencontre dans les machines classiques. Il nous fallait donc un système d'exploitation dont le module de gestion mémoire propose une interface puissante et soit muni de structures de données internes correctement documentées.

Linux dispose d'un module de gestion mémoire écrit sur mesure, alors que FreeBSD utilise le système de mémoire virtuelle du micro-noyau MACH. Ce module de gestion mémoire est particulièrement performant et flexible. Sur MACH, on trouve notamment différentes expériences d'implémentation de gestion mémoire non conventionnelle suivant différents algorithmes de distribution ou de partage de mémoire.

Au contraire, le système de gestion mémoire de Linux est moins générique. Il utilise notamment des caractéristiques propres au processeur Intel Pentium comme la segmentation [1]. Dans un système Unix standard, l'adresse en mémoire virtuelle d'une donnée du noyau et l'adresse en mémoire virtuelle d'une donnée d'un processus sont toujours distinctes : il existe une frontière en deçà de laquelle toute adresse virtuelle désigne une donnée d'un processus, et au delà de laquelle toute adresse virtuelle représente une partie du noyau. Linux utilise des segments : un segment pour le noyau et un autre pour le processus courant. Deux adresses identiques peuvent représenter, en fonction du segment qui leur est associé, des données distinctes.

Les couches de communication MPC doivent pouvoir échanger des données en espace utilisateur (par exemple dans le cadre d'applications parallèles), et des données appartenant au noyau (par exemple dans le cadre de nouveaux services noyau, comme un système de pagination distribué). Elles doivent donc proposer une interface accessible dans le noyau, et aussi accessible depuis le mode utilisateur par le biais d'un appel système.

Dans un système mémoire utilisant la segmentation, les fonctions de communication devraient donc, à chaque passage de paramètre indiquant une adresse en mémoire virtuelle, rajouter un paramètre désignant le segment associé. Cette gymnastique devrait évidemment être répétée pour le traitement des structures de données utilisées par les protocoles de communication.

Éviter ce travail fastidieux nous a fait privilégier le système d'exploitation FreeBSD.

4.1.2 Implantation mixte des protocoles

On l'a déjà souligné plusieurs fois, on désire fournir des services de communication accessibles autant depuis un processus classique en mode utilisateur, que depuis un service noyau quelconque.

Nos protocoles de communication pourraient être implantés hors noyau, c'est-à-dire dans un processus traditionnel, comme cela se fait dans les systèmes basés sur des micro-noyaux (par exemple MACH, ou des versions anciennes de Windows NT).

Une implantation au sein d'un processus utilisateur simplifie énormément la phase de développement. En effet, une erreur de programmation dans le noyau peut nécessiter un redémarrage complet de la machine. D'autre part, il existe de nombreux outils pour déverminer et analyser le comportement d'un processus. Analyser du code noyau est beaucoup plus complexe.

Malheureusement, ceci introduit une dégradation des performances par rapport à une implantation directe dans le noyau. Enfin, la nécessité de pouvoir interagir avec le système de gestion de mémoire de FreeBSD, qui est intégré au noyau, plaide pour une solution construite au sein de ce noyau.

Pour profiter des meilleures performances, en ayant pleinement accès au gestionnaire de mémoire virtuelle, tout en facilitant la mise au point des mécanismes complexes, on a

1. Les versions récentes du noyau Linux ne font plus usage de la segmentation, mais il reste en circulation un très grand nombre de versions qui continuent à l'utiliser.

choisi une implantation mixte des protocoles :

- ✓ Les opérations de communication courantes, émission/réception de données, sont implantées dans un module noyau à chargement dynamique nommé HSLDRIVER ;
- ✓ Les opérations complexes et peu courantes, telles que les négociations entre nœuds pour l'attribution d'un canal de communication entre deux tâche distantes, sont implantées dans un processus utilisateur présent sur chaque nœud de calcul : le Manager local.

4.2 Allocateur de mémoire contiguë

Les plages de données qu'on veut échanger à travers le réseau HSL doivent être contiguës en mémoire physique, du côté de l'émetteur comme du côté du récepteur. On comprend donc qu'il peut être extrêmement pratique de disposer d'un outil d'allocation de plages de mémoire locale contiguës autant en mémoire virtuelle qu'en mémoire physique. L'objectif est de pouvoir éviter d'avoir à effectuer des découpages complexes en cas de plages de mémoire locales et distantes discontiguës et dont les morceaux ne sont pas tous de même taille chez l'émetteur et chez le récepteur. La figure 4.1 présente un tel exemple d'écriture distante entre deux zones de mémoire virtuelle contiguës (par exemple des zones de mémoire de processus), à l'aide d'un message qui transite entre deux nœuds. Vus les *mapping* (correspondances) en mémoire physique, ce message doit être découpé par le logiciel en quatre pages, chacune étant transmise séparément sur le réseau.

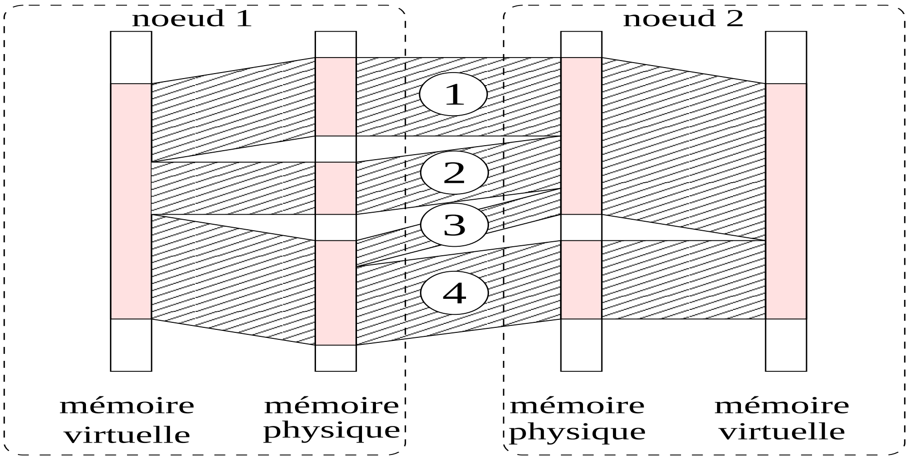


Fig. 4.1 *Découpages en quatre pages pour un échange entre deux zones virtuelles contiguës*

On a donc créé un allocateur de mémoire contiguë, CMEM, au sein d'un module noyau à chargement dynamique, que l'on a nommé CMEMDRIVER.

Son fonctionnement est simple : ce module est chargé au démarrage de la machine, juste après le chargement du noyau, et bien avant l'activation des daemons et autres processus Unix. À cet instant, on peut encore trouver, dans la mémoire d'un nœud, des plages de mémoire physique contiguë libres et de grande taille. CMEM se réserve donc, auprès du gestionnaire de mémoire de FreeBSD, une telle plage. Le gestionnaire mémoire la verrouille et l'insère dans la carte de mémoire virtuelle du noyau.

CMEM en redistribue alors des morceaux contigus, *via* l'interface accessible en mode noyau décrite dans le tableau suivant :

`slot_id = cmem_getmem(size, slot_name)`	réservation d'un emplacement contigu
`cmem_release_mem(slot_id)`	libération d'un emplacement
`phys_addr = cmem_physaddr(slot_id)`	adresse physique d'un emplacement
`virt_addr = cmem_virtaddr(slot_id)`	adresse virtuelle d'un emplacement

On pourrait aussi imaginer utiliser un mécanisme de ramasse-miette pour fabriquer une plage de mémoire physique contiguë libre, par des déplacements des zones de mémoire déjà allouées. L'avantage d'un tel mécanisme dynamique serait de ne réserver de la mémoire physique que selon les besoins des applications. C'est malheureusement impossible, car déplacer une zone de mémoire physique déjà allouée nécessite de mettre à jour toutes les variables du noyau qui pourraient la référencer. Cette zone, immobile en mémoire virtuelle du noyau, commence à un adresse physique qui se trouve notamment indiquée dans de nombreuses tables internes du système de gestion mémoire de MACH. Une telle modification des structures de données internes de ce gestionnaire mémoire serait une opération extrêmement complexe, car il n'a pas été conçu pour de telles manipulations. En outre, de nombreux pilotes de périphériques référencent la position physique de certaines zones de mémoire du noyau. Modifier les structures internes de tous ces pilotes nécessiterait de connaître l'ensemble de ces structures, chose quasiment impossible.

On comprend donc qu'il n'y a pas d'alternative au mécanisme proposé par CMEM : celui-ci alloue de façon statique et verrouille, au démarrage de la machine, une zone de mémoire et en redistribue par la suite des morceaux. Ce mécanisme n'étant pas dynamique, CMEM doit éviter de monopoliser trop de ressources mémoire dans la machine. Pour cela, la zone qu'il réserve est de taille limitée (quelques méga-octets), et ne doit être utilisée que dans des cas où la simplification attendue est significative. CMEM est par exemple utilisé par les protocoles de communication de plus haut niveau pour stocker des données internes. On évite d'utiliser CMEM pour stocker les données proprement dites des applications.

CMEM n'a pas été inclus au sein du module de gestion des protocoles, HSLDRIVER, mais au sein d'un module séparé, CMEMDRIVER car on veut pouvoir charger et décharger HSLDRIVER à volonté, par exemple lors d'une phase de développement de nouveaux protocoles noyau (décharger puis charger HSLDRIVER prend quelques secondes alors qu'il faut plusieurs minutes pour redémarrer un nœud). Au contraire, CMEMDRIVER ne peut pas être déchargé puis rechargé sur un nœud en cours de fonctionnement : quelques secondes après son déchargement, il devient impossible de retrouver suffisamment de mémoire physique contiguë libre, du fait des nombreuses activités présentes sur une machine Unix.

Si un processus utilisateur désire disposer d'une zone de mémoire contiguë allouée par CMEM, il lui suffit d'utiliser la méthode Unix standard pour remapper la zone correspondante du noyau, en utilisant l'appel système `mmap()` sur le fichier spécial `/dev/mem`, afin d'obtenir dans son espace virtuel propre une zone contiguë en mémoire physique, comme on peut le constater sur l'exemple de la figure 4.2 page suivante.

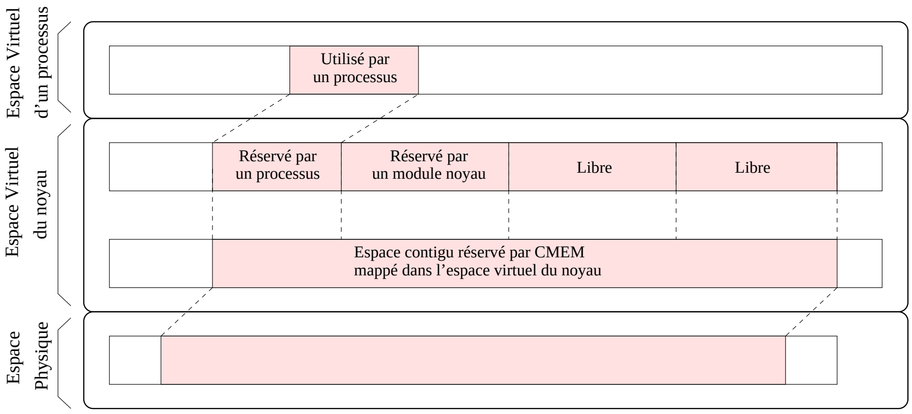


Fig. 4.2 *Accès depuis le noyau ou un processus aux zones fournies par CMEM*

4.3 Bootstrap

Sur chaque nœud de la machine MPC, les deux daemons hslclient et hslserver sont présents pour permettre l'initialisation des couches de communication logicielles, en leur permettant de s'échanger des informations tandis que le réseau HSL n'est pas encore disponible.

Au démarrage de la machine, le processus hslclient sur chaque nœud de calcul se connecte aux processus hslserver de l'ensemble des autres nœuds, *via* le protocole de communication RPC [2] à travers le réseau de contrôle. Le processus hslclient est un *client* des processus hslserver au sens RPC du terme. La figure 4.3 représente les connexions RPC mises en jeu par un nœud.

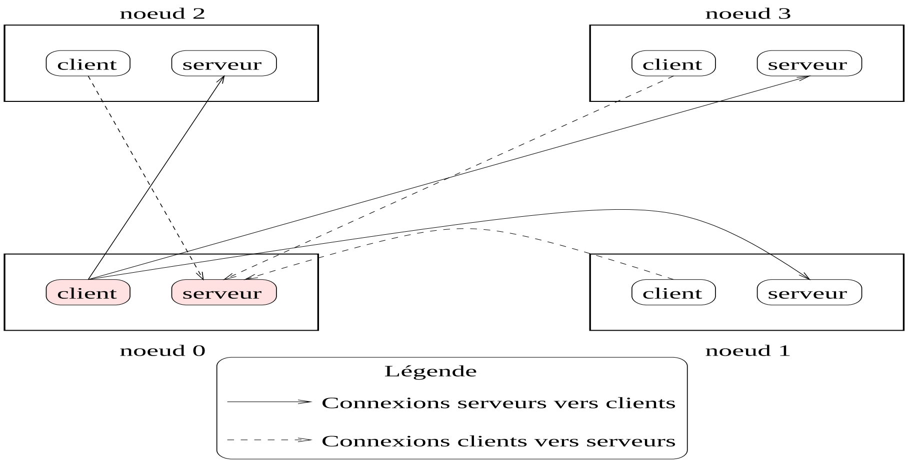


Fig. 4.3 *Connexions RPC mises en jeu par le nœud 0 sur une machine à 4 nœuds*

Chaque client invoque alors une procédure dans chaque serveur afin d'indiquer la localisation en mémoire physique des structures de données mises en jeu dans les protocoles de communication noyau, comme on le verra en détail section 5.3.3 page 109.

Après cette opération de *bootstrap*, les daemons hslclient et hslserver restent actifs pour offrir un second service : l'émulation du controleur de réseau FastHSL, à travers le réseau

2. RPC : Remote Procedure Call, RFC-1050

de contrôle. Le fonctionnement du mode d'émulation disponible sur la machine MPC est décrit en annexe A page 233.

4.4 Architecture globale

La figure 4.4 présente dans sa globalité l'architecture logicielle de chaque nœud de calcul. On y retrouve les différents composants qu'on a déjà présentés et les liens qui les relient :

✓ **Au sein du noyau** :

- Le module CMEMDRIVER inclut l'allocateur de mémoire contiguë CMEM, qui fournit aux couches de protocoles du module HSLDRIVER des tampons de communication faciles à exploiter ;
- Le module HSLDRIVER inclut la partie noyau des couches de communication. Certaines de ces couches interagissent notamment avec le système de mémoire virtuelle de MACH. D'autres font usage de CMEM.

✓ **En mode utilisateur** :

- Le Manager local implémente la partie *attribution des ressources* des protocoles de communication noyau ;
- Les daemons hslclient et hslserver dialoguent entre eux par RPC sur le réseau de contrôle, et avec le module HSLDRIVER du noyau pour échanger les informations de configuration lors du *bootstrap* de la machine ;
- L'application dialogue, à travers des bibliothèques, avec le Manager local et avec les protocoles de communication implantés dans le noyau.

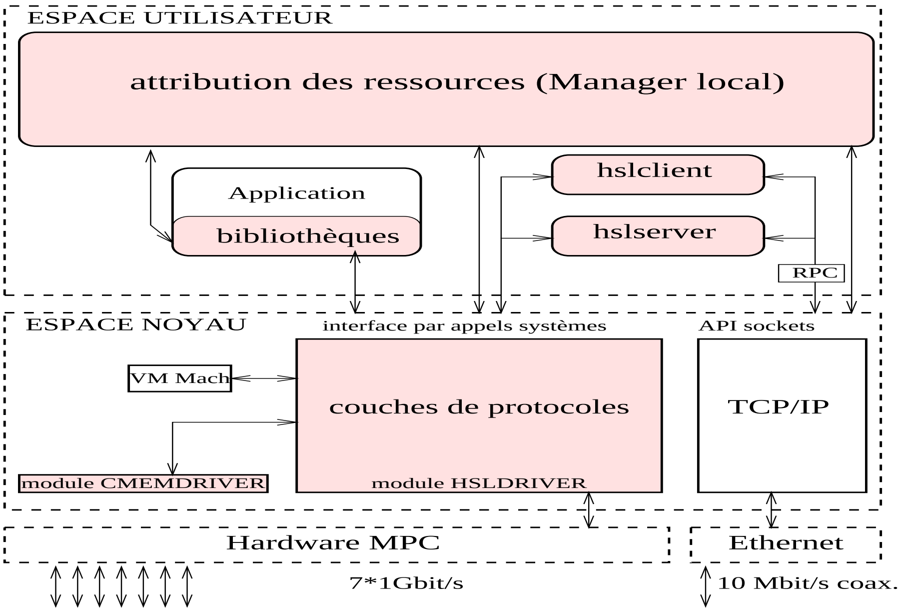


Fig. 4.4 *Architecture logicielle globale d'un nœud de calcul*

Le noyau de communication est donc constitué de cinq composants : le module noyau HSLDRIVER, le module noyau CMEMDRIVER, le Manager local, les daemons hsl-client/hslserver, et des bibliothèques destinées à l'application. On désigne par **MPC-OS** la réunion de ces cinq composants.

Chaque nœud de calcul est équipé de MPC-OS, et est relié aux autres nœuds à travers le réseau de contrôle Ethernet et le réseau HSL. Parfois, une machine supplémentaire, appelée *console*, uniquement connectée au réseau Ethernet, effectue les opération d'administration et de contrôle de l'ensemble des nœuds de calculs : gestion des utilisateurs et de files de travaux en mode *batch*, attribution des ressources de calcul aux différents travaux, activation des tâches, rapatriement des fichiers résultats. Les opérations de la console sont effectuées par un JMS [3] (Job Management System).

4.5 Échange de messages

4.5.1 Couche de communication PUT

La couche de communication PUT présente l'interface de plus bas niveau qui donne accès aux fonctionnalités d'écriture distante fournies par le contrôleur réseau PCI-DDC.

Trois objectifs fondamentaux ont présidé à sa définition :

- ✓ Plusieurs tâches doivent pouvoir utiliser PUT simultanément, et ce de façon transparente ;
- ✓ PUT doit pouvoir supporter un modèle de programmation par scrutation tout autant qu'un modèle de programmation par interruptions ;
- ✓ PUT doit fournir une interface d'accès unique, quelque soit le modèle de programmation.

PUT doit prendre en charge l'initialisation de la carte réseau FastHSL (initialisation et configuration de PCI-DDC et RCube, chargement des tables de routage dans RCube), gérer les ordres d'écriture distante et signaler les fins de transmission aux applications. La gestion des ordres d'écriture distante et de la signalisation consiste à manipuler la Liste des pages à émettre (LPE) dont chaque entrée représente un ordre d'écriture distante, et la Liste des identificateurs de messages (LMI) dont chaque entrée représente un message reçu. Rappelons qu'un message est constitué de plusieurs pages réseau, chacune étant représentée par une entrée dans la LPE munie d'un même numéro d'identificateur de message (MI [4]).

La figure A.2 page 236 présente les interactions entre la couche PUT et les composants matériels de la machine MPC (on étudiera les interactions avec les applications dans la section suivante). La couche PUT est composée de trois sous-modules :

- ▷ *Sous-module d'initialisation* : il permet de configurer les composants RCube et PCI-DDC, et de copier la table de routage de RCube dans ses registres internes (lien

3. Le JMS a été réalisé par un stagiaire de l'INT, Evry.
4. MI : Message Identifier

numéroté 1 de la figure 4.5) ;

- ▷ *Sous-module d'écriture distante* : il transmet des ordres d'écriture distante à PCI-DDC (lien numéroté 2 sur la figure) ;
- ▷ *Sous-module de traitement des interruptions* : le sous-module de traitement des interruptions reçoit et traite les interruptions provenant de PCI-DDC (lien numéroté 3 pour une indication de fin d'émission, et 4 pour une indication de réception). Une fois le traitement accompli, le module de traitement d'interruption de PUT signale l'événement à l'application, à travers un point d'accès au service PUT, décrit dans la section qui suit.

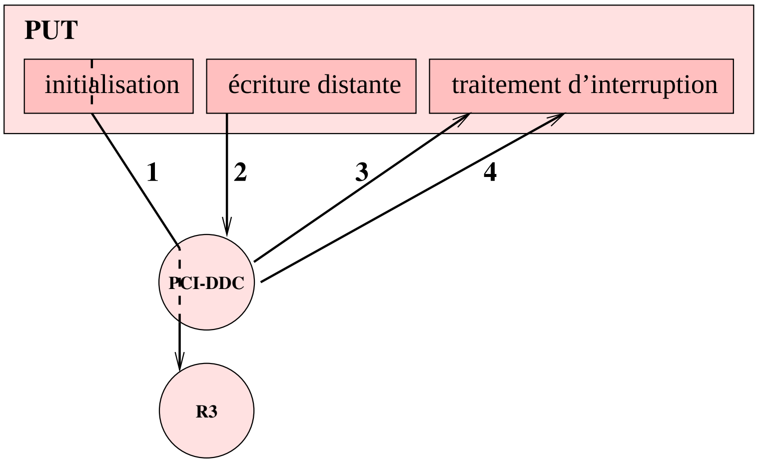


Fig. 4.5 *La couche PUT*

4.5.2 Points d'accès au service

Lors de la signalisation d'une fin de transmission côté émetteur, ou réception complète d'un message côté récepteur, la tâche qui a émis la requête associée à cette signalisation doit être avertie. PUT utilise pour cela l'identificateur de message, seul indicateur présent autant en émission qu'en réception lors d'une phase de signalisation. Il attribue donc à chaque application un ensemble d'identificateurs de messages.

On nomme Point d'accès au service (SAP[5]) un module noyau auquel PUT a attribué un ensemble de MI. Ce SAP est le plus souvent utilisé par un processus dont il est l'intermédiaire noyau pour les communications. Quand PUT désire émettre une signalisation à destination d'un SAP, et par voie de conséquence au processus associé, il se contente d'appeler une fonction qui fait office de point d'entrée de ce SAP, et lui fournit les informations dont il dispose, notamment l'identificateur de message associé à la signalisation en cours. A la charge de ce point d'entrée de SAP de propager l'information jusqu'au processus concerné.

La figure 4.6 page suivante présente un exemple d'utilisation de PUT par trois SAP : SLR/P (il s'agit d'une couche de communication construite au dessus de PUT, que nous

5. SAP : Service Access Point

examinerons au chapitre 5 page 99), et deux tâches directement utilisatrices de PUT nommées SAP 1 et SAP 2 (SAP 1 et SAP 2 sont intégrés dans le noyau, à la charge des deux tâches associées de communiquer avec SAP 1 ou SAP 2 à l'aide d'appels systèmes).

La couche HSLDRIVER, en dessous de PUT sur notre figure, contient diverses opérations de base, qui vont de l'acheminement de données ou commandes entre le noyau et les processus qui font usage des couches de communications, jusqu'à la gestion de statistiques d'utilisation des protocoles.

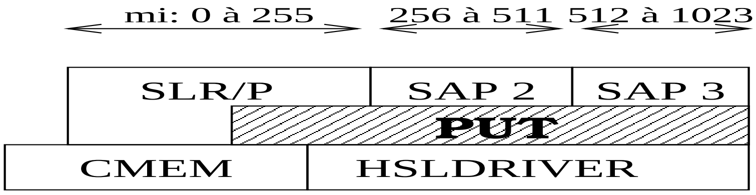


Fig. 4.6 *Points d'accès au service*

Un SAP parcourt un cycle à quatre étapes pour accéder au réseau HSL :

❶ **Enregistrement**
Le SAP appelle la fonction `put_register_SAP()` afin de récupérer un numéro identificateur de SAP. Lors de cet appel, il fournit deux points d'entrée (pointeurs sur fonction) qui seront invoqués par PUT en cas de signalisation de données émises pour l'un et de données reçues pour l'autre :

```
sap_id = put_register_SAP(minor, (*send)(...),
                                 (*recv)(...));
```

Le premier paramètre, `minor`, désigne la carte FastHSL à laquelle on veut s'adresser, pour le cas où plusieurs cartes seraient utilisées dans la même machine, par exemple pour rajouter des routeurs RCube sur le réseau. Les deux paramètres suivants, `send` et `receive`, sont les fonctions de signalisation dont nous venons de parler. Le SAP reçoit en retour un numéro, `sap_id`, qui le représente.

❷ **Attribution d'une plage d'identificateurs de messages**
La fonction `put_attach_mi_range()` permet de demander l'attribution d'une plage de MI. La longueur de la plage demandée, doit être une puissance de 2, nous la noterons 2^L. PUT attribue alors une plage de MI libre et commençant à un multiple de sa longueur. Ainsi, décomposés en binaire, les MI de la plage, nombres écrits sur 24 bits, seront représentés par une partie de poids faible et variable de L bits et une partie de poids fort fixée de $24 - L$ bits. Cette méthode d'attribution des MI offre au SAP la possibilité de partager les L bits de poids faible en plusieurs champs de bits, (dont par exemple un champs indiquant le numéro de nœud émetteur). L'émetteur et le récepteur peuvent donc associer des méta-données aux messages qu'ils s'échangent par écriture distante.
Le SAP commence par faire une demande d'attribution de MI, puis consulte le numéro du premier MI de la plage qui lui est attribuée :

```
put_attach_mi_range(minor, sap_id, range);
first_mi = put_get_mi_start(minor, sap_id);
```

❸ **Fonctionnement courant**
Pour émettre un message constitué de N pages, il faut ajouter N entrées successives dans la LPE, chacune étant décrite par une structure de type `lpe_entry_t`. La fonction `put_add_entry()` permet d'ajouter une entrée. Il s'agit d'une fonction non bloquante : si la LPE est pleine, elle rend la main immédiatement.

```
error = put_add_entry(minor, entry);
```

D'autre part, les entrées représentant un même message devant être consécutives dans la LPE, la fonction `put_get_lpe_free()` indique le nombre d'entrées libres et permet à un SAP d'être sûr qu'il pourra aller jusqu'à son terme :

```
nb_entries = put_get_lpe_free(minor);
```

À chaque fois qu'un événement asynchrone se produit (fin de réception ou d'émission), les fonctions de signalisation, fournies par le SAP à la première étape, sont invoquées. Voici leurs prototypes :

```
void (*send)(minor, mi, lpe_entry);
void (*recv)(minor, mi, data1, data2);
```

Le paramètre `minor` désigne la carte FastHSL en jeu. Les points d'entrée sont informés, par le paramètre `mi`, du MI du message qui a généré leur invocation. La fonction de signalisation d'émission reçoit l'entrée de LPE pour laquelle elle a été invoquée. En cas de réception d'un message court, la fonction de signalisation reçoit, dans `data1` et `data2`, les 8 octets de données qu'il contient.

❹ **Terminaison**
Pour libérer la plage de MI qui lui a été attribuée, un SAP utilise la fonction `put_unregister_SAP()` :

```
put_unregister_SAP(minor, sap_id);
```

4.5.3 Cohérence globale des plages de MI

Quand les tâches qui forment une application globale répartie communiquent, elles utilisent un SAP pour effectuer des appels à la fonction d'écriture distante, `put_add_entry()`. Le MI fourni par le SAP du nœud source doit appartenir à la plage des MI du SAP du nœud destinataire. Dans le cas contraire il n'y aura pas de signalisation en réception : en effet, lorsque PUT décide de signaler une fin de réception à un SAP, il avertit celui dont la plage de MI contient l'identificateur associé au message qui vient d'être reçu.

Ainsi, tous les SAP qui représentent une application donnée doivent disposer de la même plage de MI sur tous les nœuds. Comme l'algorithme d'attribution est déterministe, il suffit pour cela de charger sur chaque nœud toujours dans le même ordre les modules noyau qui s'enregistrent en tant qu'utilisateur de PUT. Il faut bien sûr que ces modules demandent la même taille de plage quelque soit le nœud. C'est d'ailleurs assez naturel pour les application parallèles découpées de manière symétrique.

4.5.4 Fonction d'émission

La fonction d'émission, `put_add_entry()`, reçoit en paramètre une structure de données de type `lpe_entry_t`. Celle-ci représente une entrée de LPE et est définie comme suit :

```
typedef struct _lpe_entry {
u_short page_length;    /* Longueur de la page de données à émettre */
u_short routing_part;   /* Numéro de noeud destinataire */
u_long control;         /* MI sur 24 bits et drapeaux divers */
caddr_t PLSA;           /* Adresse locale : Page Local Start Address */
caddr_t PRSA;           /* Adresse distante : Page Remote Start Address */
} lpe_entry_t;
```

L'entrée de LPE comporte toutes les informations nécessaires au matériel pour effectuer l'écriture distante. Le champ `control` est découpé en deux parties : la première sur 24 bits, contient le MI, la seconde sur 8 bits contient des drapeaux permettant de configurer page par page le comportement des cartes réseau émettrice et réceptrice. Ils sont explicités dans le tableau suivant (certains sont réservés, seuls 6 sont disponibles) :

Indicateur	**Sens**	**Utilisation**
LMP	Last Message Page	Dernière page d'un message
NOR	Notify Once Received	Demande d'interruption en réception
NOS	Notify Once Sent	Demande d'interruption en émission
SM	Short Message	Émission d'un message court
LMI	List of MI	Insertion d'une entrée de LMI en réception
LRM	List of Received Messages	Comptage des paquets sur réseau adaptatif

Notons que NOR signifie une demande d'une unique interruption dès réception de toutes les pages composant le message, alors que NOS signifie une demande d'interruption dès la fin d'émission de la page contenant cet indicateur.

On constate à la lecture de ce tableau que toutes les combinaisons ne sont pas forcément exploitables. Par exemple, SM sans LMI est absurde car les seules données transportées par un message court (SM) sont justement déposées dans la LMI (il s'agit des champs PLSA et PRSA).

Autre exemple : un message constitué de deux pages, dont seule la première contient l'indicateur NOR. Suivant l'ordre d'arrivée, dans un réseau adaptatif, il y aura ou non interruption en réception, alors que dans le cas contraire on aura une interruption en réception. On perd donc le déterminisme de la production d'interruption en fin de transaction, comportement qui présente peu d'intérêt.

PUT va donc imposer des règles particulières de composition des combinaisons d'indicateurs dans les pages consécutives qui forment un message.

4.5.5 Signalisation : scrutation vs. événements

Nous avons défini, section 4.5.1 page 84, l'objectif de fournir un modèle de programmation par scrutation tout autant qu'un modèle de programmation par réaction aux interruptions.

Pourquoi cette nécessité de proposer une alternative aux interruptions ? D'une part car suivant le contexte matériel et logiciel, le temps de traitement d'une interruption peut être plus pénalisant qu'une boucle de scrutation. D'autre part car certains algorithmes sont plus facilement mis en œuvre suivant un modèle scrutateur ou au contraire suivant un modèle événementiel.

Définir pour PUT deux modèles d'accès disjoints consisterait à définir deux méthodes d'accès radicalement opposées, difficulté supplémentaire pour le programmeur : aborder deux interfaces distinctes pour une seule couche de programmation bas niveau, suivant les caractéristiques du type d'accès (interruption/scrutation).

Rappelons les fonctionnements des deux types d'interfaces traditionnels :

- ✓ *Signalisation par scrutation* : dans un tel modèle, la signalisation de fin d'opération est à la charge de l'application, qui vient d'elle même consulter, au moment où elle le désire, l'état de l'opération ;
- ✓ *Signalisation par événements* : dans un tel modèle, la fin d'opération se traduit par un déroutement de l'application.

L'interface PUT propose uniquement le modèle de signalisation par événements : lors de l'enregistrement d'un SAP, celui-ci fournit **deux fonctions de signalisation** (cf. prototype de `put_register_SAP()` section 4.5.2 page 86), qui doivent se trouver logées dans le module noyau du SAP et qui sont invoquées en mode noyau :

- ▷ La fonction de signalisation de fin d'émission est invoquée lorsqu'un message est complètement émis, c'est à dire lorsque les données de la dernière page qui le constitue sont entièrement injectées dans le réseau. Il signale donc que le tampon d'émission est alors à nouveau disponible. Pour pouvoir déterminer de quel tampon il s'agit, cette fonction reçoit en paramètre l'entrée de LPE correspondant à la dernière page du message. Ainsi, le SAP qui en extrait le MI peut connaître le numéro du message dont la fin d'émission lui est signalée, et par voie de conséquence déterminer le tampon associé.
- ▷ La fonction de signalisation de réception est invoquée lorsque la réception d'un message est achevée. Cette fonction reçoit en paramètre le numéro de MI pour pouvoir déterminer le tampon où les données viennent de se déposer. S'il s'agit d'un message court, elle reçoit en outre les données associées (rappelons qu'il s'agit des 8 octets composant les champs PRSA et PLSA de l'entrée de LPE de l'émetteur, que le récepteur retrouve dans l'entrée de LMI qui signale l'arrivée de ce message court). Les données d'un message court sont donc fournies directement à la fonction de signalisation du SAP auquel elles sont destinées.

PUT libère donc les entrées de LPE à l'appel des fonctions de signalisation de fin d'émission et les entrées de LMI à l'appel des fonctions de signalisation de réception.

C'est à travers les deux tables LPE et LMI que les tâches locales à un nœud peuvent communiquer avec l'extérieur. Le nombre d'entrées de ces tables étant limité, une famine d'entrées libres, c'est-à-dire une congestion de l'une ou l'autre de ces tables, peut mener à une impossibilité de communiquer, voire à un interblocage entre plusieurs tâches réparties.

Avec le modèle de signalisation par scrutation, la libération de la LPE ou de la LMI est liée à l'action des tâches communicantes, et l'on peut donc facilement aboutir à un remplissage complet. Avec la méthode de signalisation par événements sélectionnée pour PUT, à la condition qu'on suppose qu'à chaque événement est associée une interruption, on évite toute possibilité de famine car la signalisation, et donc la libération des listes LMI et LPE, est indépendante de la tâche communicante.

4.5.6 Signalisation vs. interruptions

Jusqu'à présent, on a identifié les notions de signalisation et d'interruption. Pour permettre tout autant la scrutation et la réaction aux interruptions avec l'unique modèle de programmation par fonctions de signalisation, on va dissocier les notions d'interruption et de signalisation.

Les fonctions de signalisation, points d'entrée des SAP, sont invoquées par PUT, soit en réaction à une interruption générée par PCI-DDC, pour introduire une réaction synchrone avec l'événement correspondant, soit à la suite d'une demande du SAP lui-même, en lieu et place d'une scrutation : un SAP peut en effet activer de son propre chef le sous-module de PUT, normalement invoqué par le gestionnaire d'interruption. Ce sous-module se charge de consulter les registres et les tables manipulées par PCI-DDC, et d'appeler, s'il y a lieu, les fonctions de signalisation des SAP (on le nommera par la suite sous-module de signalisation). On retrouve donc le comportement par scrutation tout en conservant le modèle de programmation par signalisation : plutôt que de scruter directement les registres et tables manipulées par PCI-DDC, le SAP invoque PUT, qui se charge, de manière synchrone, de consulter PCI-DDC et de rappeler les fonctions de signalisation si nécessaire.

Les deux points d'entrée utilisateur de ce sous-module sont les suivants (`minor` désigne le numéro de la carte FastHSL associée au PCI-DDC qui nous intéresse) :

`put_flush_lpe(minor)`	consultation de la LPE et appel de fonction de signalisation
`put_flush_lmi(minor)`	consultation de la LMI et appel de fonction de signalisation

Le programmeur se contente donc, dans tous les cas, d'écrire les fonctions de signalisation, et pour choisir la scrutation ou les interruptions, il se voit proposer un choix :

- ▷ réactions asynchrones (scrutation) : là où il aurait naturellement scruté PCI-DDC, le programmeur se contente d'invoquer le sous-module adéquat de PUT ;
- ▷ réactions synchrones (interruptions) : le programmeur se contente de positionner les drapeaux de demande d'interruption au moment de l'insertion dans la LPE des ordres d'écriture distante (fonction `put_add_entry()`).

Étudions à nouveau le problème de la famine de ressources sous l'angle des deux modèles de signalisation proposés par PUT et que nous venons de présenter. Pour les mêmes raisons que précédemment, si l'utilisation d'interruptions est systématique par tous les SAP, aucune application ne peut causer de famine. Par contre, si un SAP remplit la LPE avec des ordres de transfert sans interruption, la LPE ne se videra qu'à condition que le sous-module de PUT qui la gère soit invoqué, c'est à dire à condition que ce SAP, ou tout autre, tente d'activer une réaction par scrutation. Si aucun SAP ne le fait, la LPE restera pleine, ce qui se traduira par un refus à toute future tentative de demande d'émission. On aboutit donc à un risque d'interblocage.

Pour résoudre dans tous les cas de figure ce problème, la fonction de demande d'émission commence par invoquer le sous-module de signalisation, afin de purger la LPE.

Malheureusement, cette purge forcée à l'émission augmente de façon conséquente la durée du traitement de la fonction d'émission `put_add_entry()`. Si tous les SAP garantissent une signalisation par interruption ou un appel régulier du sous-module de signalisation, on pourra se passer de la purge forcée de la LPE à chaque émission. On indique alors au moment de la compilation de MPC-OS que cette opération n'est pas nécessaire.

4.5.7 Asymétrie dans la gestion des tables

Nous avons analysé le problème de famine sur la LPE.

La famine d'entrées libres de la LMI est un problème qui peut paraître symétrique de celui de la LPE : cette dernière est mise à jour par PUT et consultée par PCI-DDC alors que la LMI est mise à jour par PCI-DDC et consultée par PUT.

Mais, si on peut imposer d'exécuter des instructions au sein du code lié à l'émission sans imposer l'utilisation d'interruption (il s'agit de la purge de la LPE), on ne peut évidemment pas faire exécuter du code (purge de la LMI) au moment de la réception, sans utiliser de déroutement par l'action d'une interruption. Un SAP mal écrit peut donc produire un bourrage de la LMI, bloquant de ce fait toutes les réceptions.

Pour empêcher ce cas de se produire, on peut, à la compilation de MPC-OS, demander à ce qu'une interruption soit associée à chaque demande de signalisation dans la LMI ; c'est-à-dire que le drapeau NOR se trouve alors forcé.

4.5.8 Comportement bloquant vs. non bloquant

Jusqu'à maintenant, les fonctions à la disposition des applications que nous avons présentées étaient toutes non bloquantes.

Ces opérations correspondent à du code noyau, invoqué depuis le noyau lui-même : les SAP et le gestionnaire global d'interruptions sont dans le noyau. Le noyau Unix fournit une interface de gestion d'attente de ressource standardisée à travers les deux fonctions `tsleep()/wakeup()`. Un SAP peut en faire usage pour transformer en opérations bloquantes les opérations de communication à travers le réseau HSL.

Par exemple, pour transformer un appel à `put_add_entry()` en opération bloquante, il suffit de faire, de manière atomique, un appel standard à ce point d'entrée (comportement non bloquant) et une attente sur un événement particulier (`tsleep(&event)`), à la charge de la fonction de signalisation de réactiver le processus endormi sur l'événement en question (`wakeup(&event)`). L'atomicité sert à éviter que le gestionnaire d'interruption s'active entre l'appel à `put_add_entry()` et le début de l'attente, ce qui inverserait l'ordre d'invocation de `tsleep()` et `wakeup()` (le réveil serait alors demandé avant l'endormissement !).

Selon le même principe, on peut transformer l'opération de signalisation de fin d'opération (fin d'émission ou fin de réception) en opération bloquante. Il suffit de se mettre en attente avec `tsleep()` au moment désiré, et d'invoquer `wakeup()` dans la fonction de signalisation, appelée par le sous-module de signalisation. Là encore, une section critique est nécessaire, pour des raisons analogues.

> Nous conserverons dans la plupart des interfaces de programmation noyau construites au dessus de PUT un comportement non bloquant, facilement altérable en modèle bloquant, par le mécanisme que l'on vient de présenter ci-dessus. Le programmeur peut toujours transformer une API noyau non bloquante en bloquante, mais pas l'inverse. Il était donc judicieux de choisir un modèle non bloquant afin de pouvoir s'adapter à toutes les situations.

4.5.9 Structure interne de PUT

Aux vues des différentes caractéristiques du modèle de programmation de l'interface PUT que l'on vient d'examiner, on peut décrire son architecture interne de manière plus précise que la première approche tentée figure A.2 page 236.

Le service PUT se décompose fonctionnellement en cinq sous-modules, représentés figure 4.7 :

❶ un sous-module d'initialisation ;
❷ un sous-module d'écriture distante ;
❸ un sous-module de gestion de la signalisation ;
❹ un sous-module de gestion d'interruptions ;
❺ un sous-module d'accès PCI.

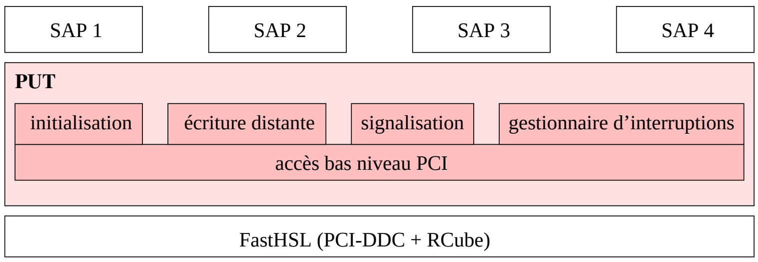


Fig. 4.7 *Composants de PUT*

Nous avons aussi vu que le code de ces modules pouvait différer en fonction de paramètres définis au moment de la compilation, afin d'affecter le traitement des opérations de PUT. Il s'agit de la purge systématique de la LPE rajoutée au sous-module d'émission, ainsi que de la purge automatique de la LMI rajoutée au module gestionnaire d'interruptions. Ces deux purges, naturellement présentes dans le sous-module de signalisation, peuvent donc se retrouver activées aussi dans le module d'écriture distante (purge LPE) et dans le module gestionnaire d'interruptions (purge LMI), comme on peut le constater figure 4.8.

En plus de ces deux paramètres de compilation, deux autres groupes de paramètres peuvent intervenir pour altérer le code généré pour PUT :

- ▷ *des paramètres d'optimisation* : en phase de déverminage ou de développement, le système doit effectuer des contrôles d'intégrité des données manipulées en vue de détecter des erreurs de programmation ou des erreurs algorithmiques éventuelles. Par exemple, PUT doit systématiquement vérifier si les paramètres fournis à ses points d'entrées figurent parmi les plages autorisées (on veut par exemple soulever une exception en cas de tentative d'écriture à destination d'un numéro de nœud invalide). Ces vérifications sont coûteuses et peuvent être supprimées quand le code est stable.
- ▷ *des paramètres d'activation de contournements logiciels* : il existe différentes versions de la carte FastHSL ainsi que des composants qui la peuplent. Notamment, des défauts de conception sont apparus dans les différentes versions de PCI-DDC, les caractéristiques réelles de ce circuit ne reflétant pas exactement les spécifications matérielles. D'autre part, tous les *bridges* PCI ne respectent pas toujours en détails la norme [Rowe, 1997]. C'est à PUT de s'adapter au matériel, et c'est au moment de la compilation que l'administrateur de la machine indique les contournements logiciels à inclure dans les différents sous-modules de PUT. L'annexe B page 237 analyse en détails la problématique des contournements logiciels.

La figure 4.8 présente ainsi une vue synthétique des différentes versions de code généré pour construire PUT en fonction des choix effectués au moment de la compilation.

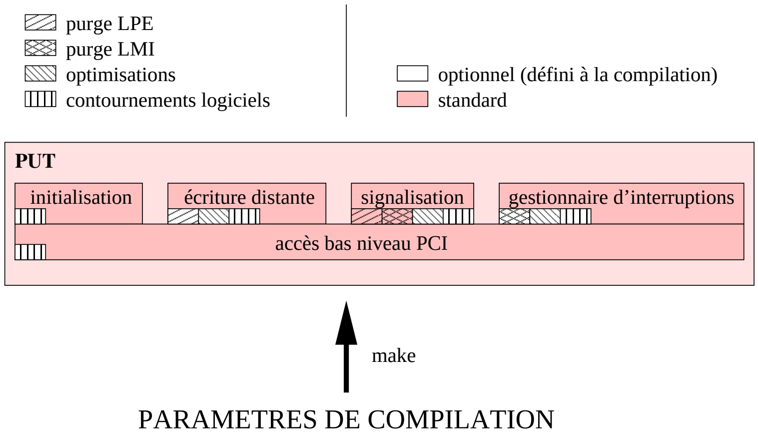


Fig. 4.8 *Influence des paramètres de compilation sur le code de PUT généré*

4.6 Conclusion

On vient de voir que quatre groupes de paramètres de compilation influencent le comportement de PUT, afin de fournir à l'utilisateur quatre groupes de services. Examinons-en les objectifs :

❶ purge de la LPE au moment de l'émission (par systématisation de la consultation de l'état de la LPE à chaque émission) : éviction de famine en cas d'applications simultanées non coopératives, mais en contrepartie, coût en performances ;

❷ purge de la LMI au moment de la réception (par systématisation de la génération d'interruption en réception) : éviction de famine en cas d'applications simultanées non coopératives, mais en contrepartie, coût en performances ;

❸ ajout/suppression de tests de cohérence : gain de performances ;

❹ ajout/suppression de contournements logiciels des défauts de conception matérielle : garantie de fonctionnement quels que soient les stimuli fournis par les applications, du moment qu'ils sont en phase avec les spécifications théoriques, mais en contrepartie, coût en performances.

On constate donc que, dans chacun de ces cas, il s'agit d'améliorer les performances par un transfert de responsabilité depuis la couche PUT vers les applications. Ces choix effectués au moment de la compilation ont donc une influence sur les contraintes de programmation.

C'est un coût à payer, incompressible pour accroître au mieux les performances. Le modèle de programmation n'en est néanmoins pas affecté : il s'agit ici seulement d'adapter PUT à son environnement, celui-ci étant défini par la réunion des applications et des composants matériels avec qui il doit interagir.

Pour ce qui est du modèle de programmation, l'originalité de PUT consiste à permettre toutes les options envisageables en demandant au programmeur un effort minimal : celui-ci doit écrire l'algorithme à activer en cas de fin de réception, et de fin d'émission, et fournir à PUT les deux fonctions de signalisation correspondant à chacun de ces deux algorithmes.

À partir de cette phase de conception unique, PUT autorise toutes les combinaisons en matière de signalisation (interruption/scrutation), ainsi qu'en matière de comportement de l'API (appels bloquants/non bloquants).

Le tableau suivant, aidé des figures 4.9 et 4.10, en explicite les différentes combinaisons :

		Signalisation	
		Interruption (fig. 4.9)	**Scrutation** (fig. 4.10)
Comportement API	Appels à PUT **bloquants**	L'application s'endort (`tsleep()`). Le gestionnaire d'interruptions appelle le sous-module de signalisation, qui appelle la fonction de signalisation fournie par l'application. Celle-ci réveille l'application (`wakeup()`).	L'application s'endort. Régulièrement, un autre *thread* ou processus de l'application appelle le sous-module de signalisation, qui appelle la fonction de signalisation fournie par l'application. Cette fonction réveille l'application.
	Appels à PUT **non bloquants**	Après appel à PUT, l'application passe à autre chose. Le gestionnaire d'interruptions appelle le sous-module de signalisation, qui appelle la fonction de signalisation de l'application. Celle-ci effectue les actions ad-hoc.	Après appel à PUT, l'application passe à autre chose. Régulièrement, celle-ci appelle le sous-module de signalisation, qui appelle la fonction de signalisation de l'application. Celle-ci effectue les actions ad-hoc.

Si toutes les combinaisons sont permises, elles ne sont pas toutes systématiquement opportunes, suivant les caractéristiques de l'algorithme à implémenter et le nombre de tâches communicantes mises en jeu dans l'application.

En consultant ce tableau, on constate que le choix de la ligne «API bloquante» ou «API non bloquante» est imposé par l'algorithme : certains algorithmes s'adaptent mieux à une programmation avec appels bloquants, d'autres nécessitent une API non bloquante.

Par contre, le choix de la colonne est conditionné par les caractéristiques matérielles de la machine utilisée ainsi que par les choix de découpage et de placement des tâches sur les différents nœuds de calcul, comme nous allons le montrer maintenant.

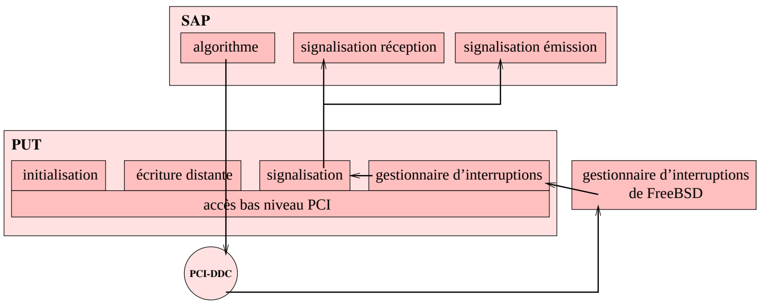


Fig. 4.9 *Signalisation par interruptions*

Étudions par exemple le comportement d'une application constituée de NT tâches, répartie sur N nœuds, vis-à-vis de la scrutation. Si toutes les tâches sont susceptibles de communiquer entre elles, chacune doit scruter $N\,T-1$ drapeaux signalant une réception en provenance d'une autre tâche. Soit δ_{mem} le temps d'un accès mémoire, et δ_{bcl} le temps cumulé d'un incrément de registre et d'un saut conditionnel. Le délai total maximal pour effectuer une scrutation a donc pour valeur $(N\,T-1)(\delta_{mem}+\delta_{bcl})$. En supposant que la distribution des destinataires de messages est équirépartie sur l'ensemble des émissions, le délai moyen de scrutation pour déterminer une réception vaut $\Delta(T,N) = \frac{(N\,T-1).(\delta_{mem}+\delta_{bcl})}{2}$.

Au contraire, le délai Δ' de réaction à une interruption pour détecter une réception est indépendant de N et varie peu avec T, Δ' est même totalement indépendant de T lorsque les ressources de chaque nœud ne sont pas sous dimensionnées vis-à-vis du nombre de tâches T qui y sont présentes.

La scrutation étant évidemment plus efficace lorsque $\Delta(T,N) < \Delta'$, on comprend donc que c'est suivant les valeurs des paramètres N et T et non suivant le type d'algorithme à implémenter qu'on va pouvoir décider de choisir un mode de réaction plutôt par interruption ou plutôt par scrutation.

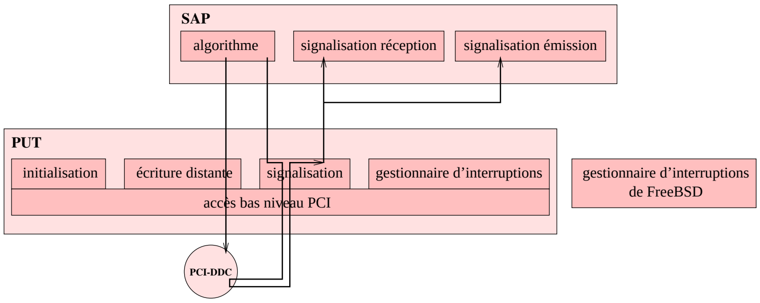


Fig. 4.10 *Signalisation par scrutation*

C'est ainsi qu'apparaît très clairement l'intérêt d'avoir choisi un modèle général de programmation unique (fourniture à PUT de fonctions de signalisation implémentées au sein de l'application), tout en autorisant l'ensemble des modes possibles de signalisation et de comportement de l'API : quand on consulte notre tableau, on s'aperçoit qu'un changement de colonne a peu d'implications sur la structure du programme. En effet, pour qu'un SAP s'adapte à l'une ou l'autre des colonnes de ce tableau, il suffit au programmeur d'apporter quelques modifications au sein du code exécuté par le *thread* principal de l'application ainsi que dans les fonctions de signalisation fournies à PUT, mais le canevas reste identique quelque soit la combinaison retenue : l'application se décompose dans tous les cas en trois modules, l'algorithme principal et les deux fonctions de signalisation.

Pour récapituler, on peut dire qu'au sein de notre tableau le choix de la ligne est imposé par l'algorithme et le choix de la colonne par les conditions d'exécution. Et justement, le modèle de programmation de PUT (fourniture de fonctions de signalisation) a été choisi pour être largement indépendant du choix de la colonne, permettant ainsi d'adapter facilement toute application aux conditions matérielles propre à son exécution, pour en tirer les meilleures performances.

Une fois PUT implémenté en suivant le modèle que l'on vient de décrire, il a fallu effectuer un travail minutieux d'optimisation. En effet, les performances offertes par le matériel étant extrèmes, il a été nécessaire d'analyser puis d'optimiser chaque ligne de code de l'implémentation de PUT, voire même dans certaines sections de passer directement à la programmation assembleur. Cette étape est aujourd'hui un passage obligé dans l'implémentation de tout protocole bas-niveau pour machines parallèles. Une étude similaire a été par exemple effectuée par les créateurs de l'interface BIP, ce qui leur a permis récemment d'en améliorer sensiblement les performances ([Tourancheau and Westrelin, 2000] et [Westrelin, 2001]).

Chapitre 5

Canaux de communication

Sommaire

La couche de communication SLR/P dont nous décrivons et justifions les choix architecturaux dans ce chapitre, s'appuie sur la primitive PUT, afin de proposer des services de communication entre les différentes tâches d'une application répartie, à travers des canaux virtuels. On passe donc d'un mode de communication inter-nœuds avec PUT, à un mode de communication intra-application avec SLR.

5.1 Couches de communication noyau

Au dessus de la couche logicielle de plus bas niveau, PUT, on a construit dans le noyau une succession de couches de protocoles qui fournissent des services de plus haut niveau et une plus grande abstraction. Nous avons déjà présenté une vue globale de l'architecture de MPC-OS sur la figure 4.4 page 83, au sein de laquelle l'empilement de protocoles noyaux était agrégé en une seule couche. La figure 5.1 ajoute à cette représentation le détail des couches de communication noyau.

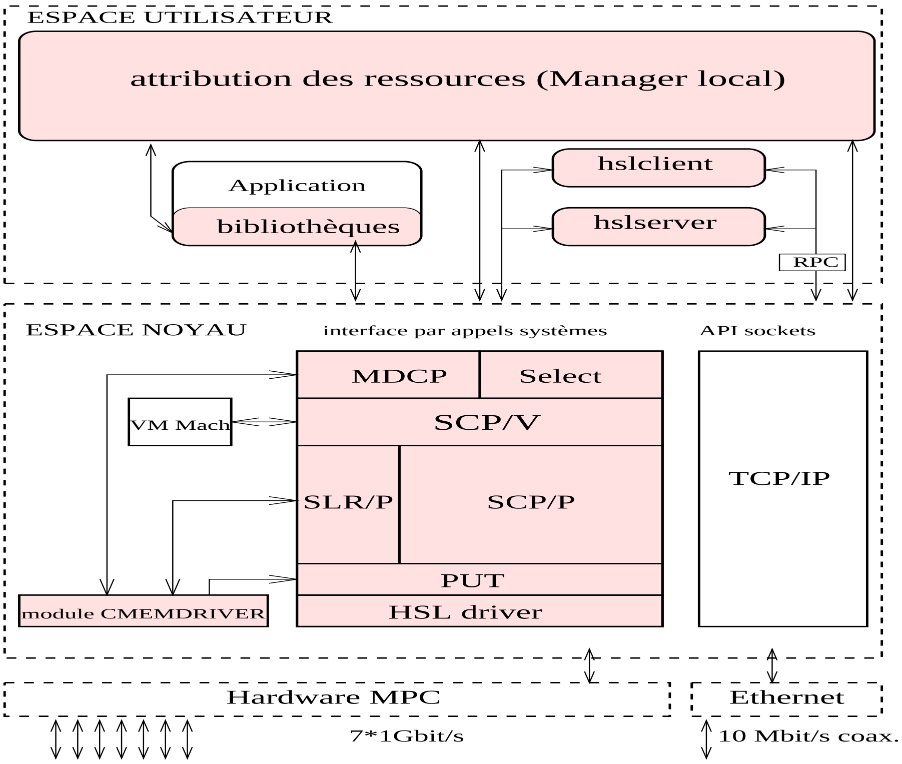


Fig. 5.1 *Empilement des couches noyau*

L'empilement se décompose ainsi :

- ✓ **pilote HSL** : interface avec le matériel ;
- ✓ **PUT** : interface bas niveau, permettant de faire des transferts DMA inter-nœuds ;
- ✓ **SLR/P** [1] : cette couche de communication s'appuie sur PUT pour implémenter un

1. SLR/P : StateLess Receiver protocol / using Physical addresses

protocole d'échanges sur canaux virtuels, avec adressage en mémoire physique, sans recopie des tampons de données[2] ;

- ✓ **SCP/P**[3] : cette couche de communication propose les mêmes services et présente la même interface que SLR/P, tout en garantissant, en plus, la reprise sur erreur. L'absence de copie des tampons est conservée ;
- ✓ **SCP/V**[4] : cette couche de communication s'appuie sur SCP/P ou SLR/P, au choix, pour proposer des services de communication sur canaux en adressage virtuel. Ils sont sécurisés ou non suivant la couche sous-jacente ;
- ✓ **MDCP**[5] : cette couche de communication, qui s'appuie sur SCP/V, propose des services de communication sur canaux en adressage virtuel, avec un modèle de programmation et une interface plus classiques et imposant moins de contraintes que SCP/V, au détriment de l'absence systématique de copies des tampons (il y aura copie ou non suivant l'ordre d'invocation des opérations d'émission et de réception) ;
- ✓ **Select** : cette couche de communication étend aux canaux de communication MDCP le service système `select()` du standard Unix.

On va présenter dans ce chapitre la couche de communication sur canaux virtuels avec adressage en mémoire physique : SLR/P.

5.2 Canaux avec adressage physique : SLR/P

5.2.1 Canaux virtuels

Les canaux virtuels, à travers lesquels les données transitent, sont désignés par des *numéros de canaux*. Chaque canal est bidirectionnel, et est complètement défini à partir de l'ensemble {{*numéro du nœud d'une extrémité, numéro du nœud de l'autre extrémité*}, *numéro de canal*}. Ainsi les canaux {{2, 5}, 3} et {{5, 2}, 3} sont identiques, alors que les canaux {{6, 7}, 4} et {{5, 7}, 4} portent un même numéro (4) mais sont distincts car les nœuds en jeu ne sont pas les mêmes.

5.2.2 Interface de programmation

SLR/P fournit deux primitives noyau, `slrpp_send()` et `slrpp_recv()`, permettant de communiquer à travers un canal. En voici les prototypes :

```
error = slrpp_send(dest, channel, pages, size, fct, param, proc);
error = slrpp_recv(dest, channel, pages, size, fct, param, proc);
```

2. La première version de la couche SLR/P a été définie avec la participation de F. Potter (LIP6) et P. David (PRiSM)
3. SCP/P : Secure Channelized Protocol / using Physical addresses
4. SCP/V : Secure Channelized Protocol / using Virtual addresses
5. MDCP : Multi-Deposit Channelized Protocol

Ces deux fonctions sont non bloquantes : elles n'attendent pas la fin du transfert demandé pour rendre la main. Néanmoins, pour faciliter leur utilisation, elles vont se bloquer lors d'incidents temporaires (par exemple lors d'une famine de ressource), afin d'éviter le plus souvent de renvoyer un message d'erreur à l'utilisateur. Celui-ci a la garantie qu'un message d'erreur provenant de `slrpp_send()` ou `slrpp_receive()` n'intervient qu'en cas d'erreur grave : paramètre incorrect, mauvais fonctionnement du matériel, etc. Si l'utilisateur désire un comportement bloquant, c'est à dire s'il désire attendre la fin de la transmission avant de continuer ses opérations, il peut toujours implémenter un mécanisme bloquant autour de ces appels non bloquants, la section 5.3.6 page 113 en présente le principe.

Notons qu'il s'agit ici d'échanges en adressage physique, le tampon d'émission étant représenté par un tableau de zones physiques contiguës, présent à l'adresse virtuelle `pages` et contenant `size` zones. Il peut s'agir de mémoire appartenant à un processus ou au noyau. Lorsque les tampons sont à nouveau accessibles, la fonction dite de *callback* pointée par `fct` est alors invoquée avec le paramètre `param`, afin de signaler la fin de l'opération (la fonction de *callback* d'émission reçoit en outre un paramètre indiquant la taille des données réellement transmises, on comprendra son utilité section 5.2.4). Le dernier paramètre, `proc`, désigne le processus qui a demandé le transfert (il est nul si l'invocation provient du noyau lui-même).

Un message RECEIVE, transportant les adresses des tampons de réception, est associé à chaque appel à `slrpp_receive()`, et un message SEND, transportant les données, est associé à chaque appel à `slrpp_send()`.

5.2.3 Influence de l'absence de copie sur le comportement de l'API

On s'est imposé, pour des raisons de performances, l'absence de copie de données au niveau SLR/P, cette couche doit donc connaître la localisation du tampon de réception pour pouvoir effectuer un transfert DMA à travers le réseau, en utilisant la primitive PUT, depuis le tampon d'émission et vers le tampon de réception. Ainsi, un appel à `slrpp_send()` n'est suivi d'effet qu'après que le nœud distant ait fait appel à `slrpp_recv()` : on n'envoie pas de données vers le destinataire tant qu'on ne sait pas où les déposer.

On peut traduire cela par la remarque suivante : **un tampon d'émission n'est libéré qu'après la demande de réception chez le distant**. Il s'agit là d'un comportement plus restrictif que ce que proposent les interfaces communes de communication par canaux, telles que les tubes, les sockets ou les pseudo-terminaux du monde Unix. **C'est à la contrainte d'absence de copie**, que l'on ne retrouve pas dans ces API, **qu'on doit cette particularité**.

5.2.4 Mise en concordance des tailles de tampons d'émission et réception

Imaginons que les tailles des tampons d'émission et de réception soient différentes et étudions le comportement de SLR/P.

Supposons tout d'abord que le tampon de réception soit de taille plus petite que celle du tampon d'émission. Il y a donc un surplus de données, pour lequel il n'y a pas de place allouée chez le récepteur. SLR/P va alors transférer uniquement le début du tampon d'émission. Il faut alors que l'émetteur effectue un nouvel appel à `slrpp_send()` et que le récepteur effectue un nouvel appel à `slrpp_receive()` pour envoyer spécifiquement la fin du tampon.

Cela permet de libérer le tampon d'émission le plus tôt possible : s'il avait fallu remettre en jeu automatiquement la fin du tampon d'émission, on aurait alors dû attendre une nouvelle demande de réception pour annoncer la libération du tampon d'émission.

Pour des raisons similaires dans la libération du tampon de réception, lorsque la taille de celui-ci est plus grande que celle du tampon d'émission, on libère le tampon de réception alors qu'il n'est pas entièrement rempli. On retrouve là, à la différence de la section précédente, un comportement typique des interfaces sur canaux traditionnelles.

Pour généraliser, dans tous les cas, la taille des données transmises est celle du plus petit des tampons d'émission et de réception.

Il faut cependant fournir aux deux tâches qui communiquent à travers un canal, un moyen de savoir quelle est la quantité de données réellement transférées.

Dans un souci d'efficacité et de simplicité, la fonction d'émission se contente d'émettre les données du tampon d'émission, et aucune autre information. En effet, envoyer ne serait-ce que la taille des données effectivement émises dans un tampon adéquat chez le récepteur nécessiterait une gestion de ce tampon distant pour éviter son écrasement entre ses différentes utilisations.

Dans une transaction SLR/P, il y a un message de contrôle du récepteur vers l'émetteur, en direction d'une boite aux lettres, suivi par un message de données dans le sens opposé. On peut donc se permettre de transporter la taille des données dans le message de contrôle, mais on ne peut pas le faire lors de l'émission des données proprement dites.

Du côté émetteur, SLR/P fournit, en paramètre de la fonction de *callback* d'émission, la taille des données réellement transmises, qui est la plus petite des deux valeurs suivantes : la taille du tampon de réception indiquée dans le message de contrôle qui provient du récepteur, et la taille du tampon de données chez l'émetteur.

Il nous reste à informer le récepteur de la taille des données effectivement transmises. Il suffit pour cela que la tâche émettrice indique dans le premier mot de son tampon d'émission la taille du tampon en question. Elle ne peut y mettre la taille des données réellement émises, car au moment de la demande d'émission, le message de contrôle de réception n'est pas forcément encore parvenu à destination. Par contre, une fois l'échange terminé, le récepteur pourra récupérer le premier mot du message de données, le comparer à la taille du tampon de réception, et conclure que la taille des données reçues est le plus petit de ces deux nombres.

Rappelons qu'un tampon d'émission ou de réception au niveau SLR/P est un tableau de zones physiques contiguës. Le premier mot d'un tampon n'est donc pas forcément contigu avec le reste du tampon. Et par suite, le premier mot des tampons d'émission et de réception, lors d'un envoi de taille inconnue, pourra être localisé à tout autre endroit hors de la zone proprement dite des données échangées par l'application.

Cette technique n'a donc pas d'implication pénalisante au niveau de l'organisation en mémoire des données échangées.

5.2.5 Numéros de séquence

Les messages SEND et RECEIVE correspondent respectivement aux diverses invocations de `slrpp_send()` et `slrpp_receive()`. Ces messages sont appariés deux-à-deux pour former des transactions. Quand plusieurs transactions sont en cours sur un même canal, il faut pouvoir associer chaque SEND au RECEIVE qui lui correspond. Pour cela, on utilise deux compteurs : le numéro de séquence en émission et le numéro de séquence en réception. À chaque `slrpp_send()`, on incrémente le numéro de séquence en émission du canal correspondant, et on stocke dans les structures de données représentant la transaction en cours cette valeur. Lors de chaque `slrpp_receive()`, on fait de même pour le numéro de séquence en réception, et on émet cette valeur à destination du nœud émetteur. Ce dernier peut donc apparier chaque `slrpp_send()` avec le `slrpp_receive()` qui lui correspond.

5.2.6 Choix d'un modèle de programmation et de signalisation

En étudiant les différents modèles de programmation section 4.6 page 94, on a remarqué que la signalisation par interruption était préférable à la scrutation, pour les applications nécessitant un grand nombre de tâches communicantes. Sachant que SLR/P s'adresse à tous types d'applications, et tout particulièrement à des applications complexes fonctionnant sur des machines constituées de plusieurs dizaines de nœuds multi-processeurs, on a donc choisi le modèle de signalisation par interruption pour SLR/P.

5.2.7 Zone de travail de SLR/P

SLR/P a besoin de transmettre des messages de contrôle (messages RECEIVE) vers des boites aux lettres pour indiquer la localisation d'un tampon de réception. Les données de contrôle font partie de l'espace mémoire réservé et géré par SLR/P. Elles sont manipulées par PUT, elles doivent donc figurer en mémoire physique. SLR/P doit donc allouer cet espace dans l'espace noyau, ce qui lui permet d'y avoir accès à tout moment, et en plus, caractéristique de nombreux noyaux Unix et de celui de FreeBSD en particulier, d'avoir l'assurance que cette zone est verrouillée. Si en sus, ces données sont contiguës, le travail de SLR/P sera simplifié. Il va donc de soi que SLR/P se fait allouer cette zone par l'intermédiaire des services mémoire implémentés dans CMEM.

5.2.8 Préallocation et gestion de l'état des canaux

Le développement d'un protocole de communication intégré au sein du noyau est une opération délicate, car la phase de mise au point d'un tel module est beaucoup plus complexe que le déverminage d'un processus en mode utilisateur.

On a donc préféré une allocation statique des structures de données de gestion des canaux, ce qui fixe le nombre maximal de canaux utilisables à un instant donné. Une gestion dynamique aurait de toute façon été incompatible avec le choix motivé de se servir d'une zone allouée par CMEM comme zone de travail de SLR/P.

Bien que le nombre maximal de canaux entre deux nœuds soit fixé, on offre la possibilité à une tâche de fermer un canal, ce qui permettra par la suite à cette tâche ou une autre de le réutiliser. Cette approche est compatible avec l'allocation statique des structures de données de gestion des canaux de SLR/P, mais cela nécessite la mise-en-œuvre d'un protocole évolué de terminaison de communication sur un canal. Il faut notamment s'assurer que plus aucun message concernant ce canal n'est en cours de transfert dans le réseau. Pour simplifier la conception d'un tel mécanisme, on a pris la décision de l'intégrer hors noyau, au sein d'un processus gestionnaire de ressources que nous avons précédemment nommé *Manager local*, présenté dans le chapitre 8 page 155.

Les numéros de canaux séparent ainsi les structures de contrôle en deux groupes :

- ✓ Les canaux dont le numéro est inférieur à une frontière, et qui sont préinitialisés et d'utilisation unique (on ne peut les réallouer). Ils ne nécessitent pas l'utilisation du processus gestionnaire de ressources ;
- ✓ Les canaux dont le numéro est supérieur à une frontière, et qui doivent être alloués avant utilisation, et libérés après utilisation. Ceux-ci nécessitent l'utilisation du processus gestionnaire de ressources.

La gestion dans le noyau de l'état d'un canal nécessite 50 octets. D'autre part, chaque échange en cours nécessite 168 octets (72 du côté de l'émetteur et 96 du côté du récepteur). Ainsi, dans une machine comportant N nœuds et C canaux entre chaque couple de nœuds, et disposant à un instant donné de M messages en transit sur chaque canal, la place mémoire nécessaire dans chaque noyau est de $NC(50 + 84M)$ octets.

5.3 Conception de SLR/P

5.3.1 Critères prépondérants

Les quatre critères fondamentaux qui ont présidé à la conception de la couche SLR/P consistent :

- ❶ d'une part à cacher à l'utilisateur des fonctions `send()` et `receive()` la connaissance de l'organisation mémoire des nœuds distants,
- ❷ d'autre part à optimiser les performances en termes de latence et de débit, même avec une machine constituée d'un grand nombre de nœuds,
- ❸ et enfin de supporter les configurations de réseaux adaptatifs,
- ❹ tout en se prémunissant de toute copie de tampon.

C'est pour satisfaire ces critères que l'on a choisi un protocole simple, mettant en jeu deux messages pour chaque échange :

- ✓ **un message de contrôle**, du récepteur vers une boite aux lettres chez l'émetteur, décrivant la localisation physique du tampon de réception,
- ✓ **un message de données**, dans le sens opposé, transportant les données utiles.

On a vu que ce protocole est source de limitations :

- ✓ Le tampon d'émission est bloqué jusqu'à ce qu'une demande de réception correspondante soit effectuée ;
- ✓ Les données n'ayant pas de place dans le tampon de réception ne sont pas retransmises par la suite, à moins que ce ne soit explicitement implémenté par l'utilisateur de SLR/P ;
- ✓ On manipule des adresses locales physiques, donc sans possibilité de protection mémoire.

Les couches de protocoles de plus haut niveau doivent pallier à ces limitations.

5.3.2 Typage des messages

SLR/P, comme tout SAP au dessus de PUT, se voit attribuer un certain nombre d'identificateurs de messages (MI) pour son usage propre. Ces identificateurs vont lui permettre de caractériser ses messages. Comme présenté à la section 4.5.2 page 85, la zone de MI attribuée peut être décomposée en deux champs de bits : une partie de poids fort fixe, et une partie de poids faible pouvant contenir n'importe qu'elle valeur. SLR/P va donc pouvoir redécouper la partie de poids faible en différents champs, afin de transporter une information dans le MI.

Lorsqu'un message de contrôle ou un message de données est déposé dans un nœud, SLR/P doit pouvoir distinguer le type de message, en découvrir la provenance (numéro de nœud source), ainsi que le canal correspondant. Il dispose pour cela uniquement du numéro de MI passé en paramètre à la fonction de signalisation de SLR/P.

On est tenté d'utiliser ces trois informations (type de message, nœud source, numéro de canal) pour former le MI associé à un message au niveau SLR/P. Ce serait oublier la condition sur les MI, imposée par le matériel : *il doit, à un instant donné, dans un réseau adaptatif, n'exister qu'un seul message d'un MI donné à destination d'un nœud de numéro donné*. Ainsi, effectuer deux opérations `slrpp_recv()` presque simultanément sur le même canal conduiraient à transgresser cette règle.

Pour pouvoir récupérer ces trois informations, tout en conservant la condition imposée par les réseaux adaptatifs, on a choisi une méthode de gestion des MI avec garantie d'intégrité à la charge des récepteurs. Pour cela, chaque récepteur, au moment d'émettre une demande d'envoi de données, choisit un indice destiné à garantir l'unicité du MI pendant la transaction. Plutôt que de tenter de garantir l'unicité du MI pendant l'émission d'un message de données *ou* d'un message de contrôle, on va garantir l'unicité du MI pendant toute la transaction constituée d'un message de contrôle *et* du message de données associé. En effet, comme on va le constater, il est plus simple de garantir l'unicité pendant une

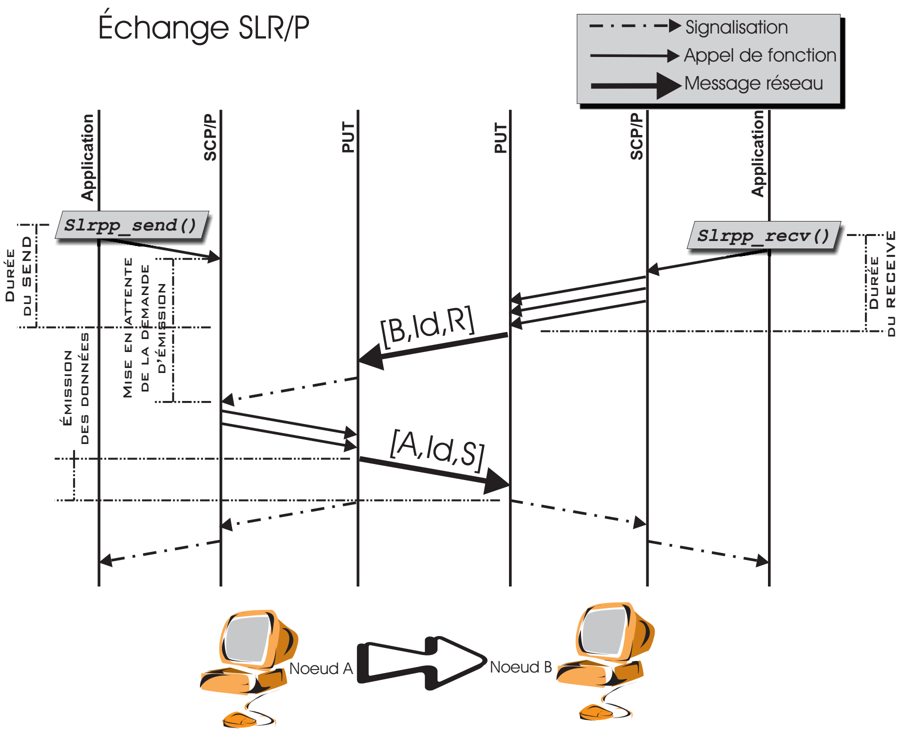


Fig. 5.2 *Échange SLR/P : slrpp_send() avant slrpp_recv()*

transaction complète que pendant le transfert d'un seul message, car une transaction est bornée dans le temps par les instants d'occurrence d'opérations **se produisant sur un même nœud** (récepteur dans le cas qui nous concerne).

L'indice est donc choisi parmi un *pool* d'indices géré par le récepteur. Quand il choisit un indice pour construire un MI destiné à envoyer un message RECEIVE de demande d'émission, le récepteur note que cet indice est en cours d'utilisation, et ne le libère que lorsque le message de données attendu en retour est finalement reçu.

Deux nœuds pouvant utiliser le même indice à un instant donné, on rajoute à côté de l'indice, en décomposant le MI en champs de bits, le numéro de nœud émetteur, pour garantir l'unicité.

Cela garantit l'unicité des messages de contrôle, mais pas celle des messages de données, puisque ces derniers sont envoyés par les nœuds émetteurs, et que seuls les récepteurs gèrent les MI. On pourrait imaginer que les émetteurs gèrent eux aussi les MI, mais cela nécessiterait un message d'acquittement de réception de données, pour libérer de tels MI. On veut réduire au possible le nombre de messages sur le réseau, on va donc éviter cette solution.

On utilise donc, dans un message de données, l'indice du message de contrôle qui décrivait le tampon prévu pour accueillir ces données.

Soient deux nœuds numérotés A et B. On désire transmettre des données de A vers B. Si le nœud récepteur B a choisi l'indice Id au moment de l'émission du message de contrôle suite à un appel à `slrpp_recv()`, le MI associé à ce message sera [B, Id], et le MI associé au message de données émis par le nœud A sera [A, Id].

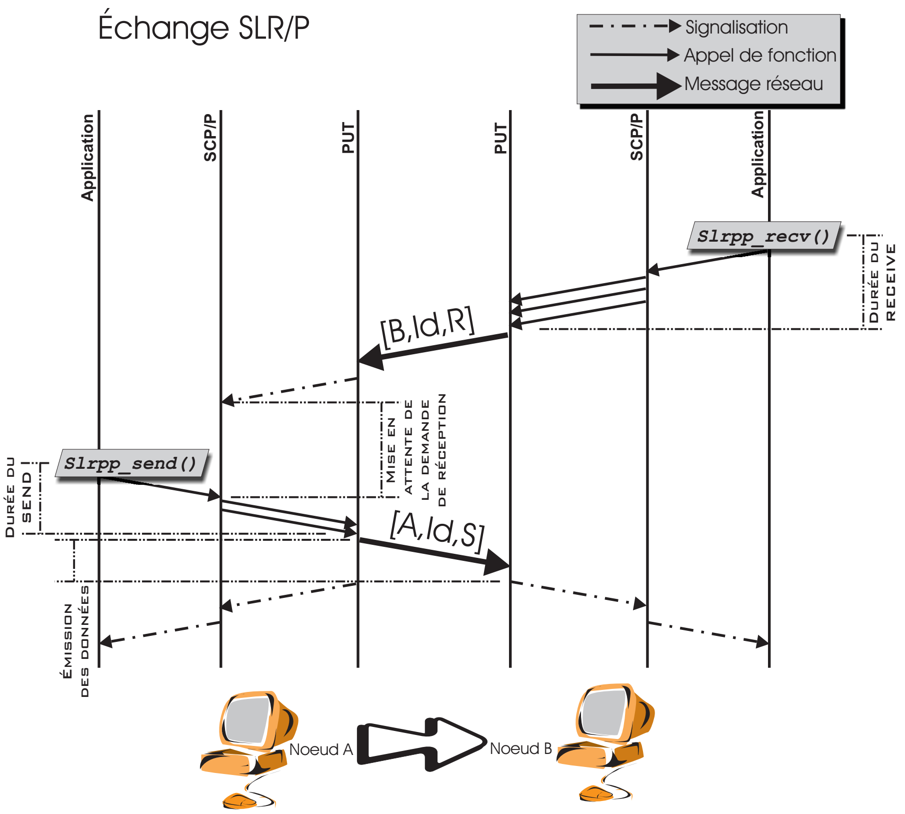


Fig. 5.3 *Échange SLR/P : slrpp_send() après slrpp_recv()*

Il reste cependant un problème : si le nœud A, agissant en récepteur, choisit dans son *pool* l'indice Id, il émettra un message de contrôle vers B de MI [A, Id]. C'est-à-dire que de A vers B, un message de données pourrait avoir le même MI qu'un message de contrôle.

Il suffit donc de rajouter un bit précisant le type de message pour garantir l'unicité dans tous les cas. On symbolisera ce bit par la lettre 'R' pour les messages de réception, et par la lettre 'S' les messages d'émission.

Pour résumer, on peut dire que le MI est constitué de trois champs de bits : un champ constitué du numéro du nœud émetteur, un autre contenant un indice choisi par le récepteur, et le dernier indiquant le type de message. Dans notre exemple, B émettra un message de contrôle de MI [B, Id, R] puis A émettra un message de données de MI [A, Id, S].

Le synopsis d'un tel échange, constitué d'un appel à `slrpp_send()` précédant l'appel à `slrpp_recv()`, est présenté figure 5.2 page précédente. À l'inverse, lorsque `slrpp_send()` succède à `slrpp_recv()`, on se trouve dans le cas présenté figure 5.3. Dans ces deux figures, on a choisi de représenter la situation où le tampon d'émission doit être découpé en deux zones discontiguës pour être copié vers le tampon de réception. Il y a donc deux appels à PUT pour signaler les deux entrées de LPE qui décrivent le message de données. Le réseau étant éventuellement adaptatif, on ne peut rien dire sur l'ordonnancement de ces pages réseau sur les liens HSL : il se peut que ce message constitué de deux pages réseau soit découpé en de multiples paquets qui vont se trouver déposés dans un ordre quelconque. Le même phénomène peut se produire pour le message de contrôle, décrit dans cet exemple par trois entrées de LPE. Le nombre de pages réseau constituant le message de données n'est pas borné, à l'inverse du nombre de pages réseau constituant le message de contrôle, qui ne peut dépasser le nombre 3, eu égard à la gestion particulière de la table *copy_phys_addr[]*, décrite ci-dessous. Notons enfin, sur ces figures, que la dernière flèche de signalisation vers l'application, sur chaque nœud, représente l'appel à la fonction de *callback*. Et rappelons que les appels aux primitives d'émission et de réception (fonctions `send()` et `receive()`) sont non bloquants. Elles rendent la main à l'application à la fin des délais respectifs indiqués DURÉE DU SEND et DURÉE DU RECEIVE.

5.3.3 Structures de données gérées par le récepteur

Les structures de données utilisées par SLR/P pour gérer les communications à travers les canaux virtuels peuvent être classées en deux groupes : d'une part les structures nécessaires pour jouer le rôle d'émetteur sur un canal, et d'autre part les structures nécessaires pour jouer le rôle de récepteur. Les canaux étant bidirectionnels, ces structures sont présentes toutes deux à chaque extrémité du canal.

Déroulons les étapes d'une transaction SLR/P, depuis le nœud A vers le nœud B, pour découvrir les structures de données mises en jeu, et rassemblées au sein de la figure 5.4. On n'a représenté sur cette figure que les structures pour l'émission du côté de A et celles pour la réception du côté de B. L'annexe D page 245 présente la même figure, enrichie des différents champs qui composent ces structures, notamment ceux permettant de mettre en place les références croisées entre les entrées.

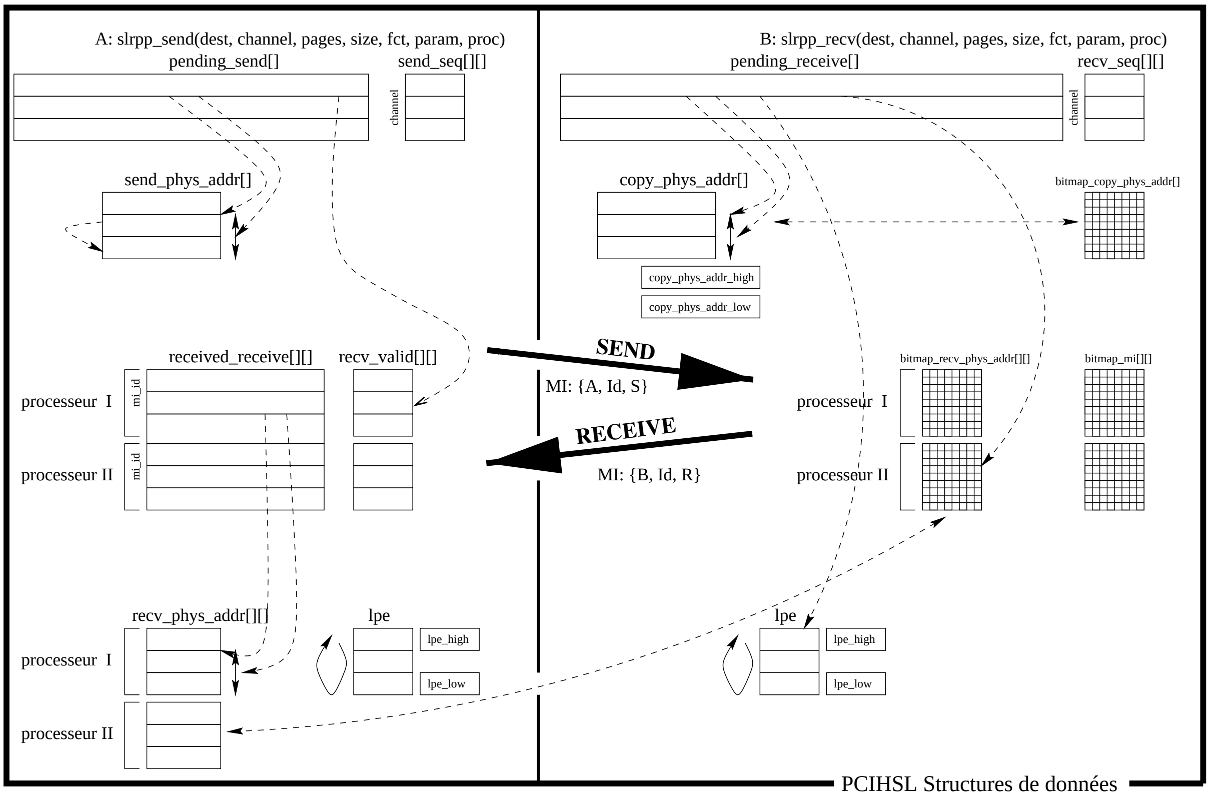


Fig. 5.4 *Structure de données en jeu dans SLR/P*

❶ Lorsque l'utilisateur invoque `slrpp_recv()`, la description de l'opération à effectuer est stockée dans la table *pending_receive[]*. Le numéro de séquence de cette demande de réception, sur le canal en question, est stocké dans la table *recv_seq[][]*.

❷ La description de la localisation physique du tampon de réception est placée dans une ou plusieurs entrées de la table *copy_phys_addr[]*, disposée dans un bloc fourni par CMEM. L'allocation de ces entrées est gérée par un champ de bits : la table *bitmap_copy_phys_addr[]*.

❸ Le nœud récepteur choisit alors un espace réservé dans le nœud émetteur pour y envoyer le contenu de *copy_phys_addr[]* qui vient d'être rempli. Comme on l'a montré précédemment, il est nécessaire que ce soit le récepteur qui gère cette zone de l'émetteur. Pour cela, la table *recv_phys_addr[][]* de l'émetteur est découpée en morceaux, chacun pour l'usage propre d'un processeur distant.

❹ La localisation en mémoire physique de A du morceau qui est attribué à B a été préalablement fournie par les daemons hslclient et hslserver, au moment de leur dialogue RPC sur le réseau de contrôle, dans la phase d'échange d'informations de configuration lors du *bootstrap* de la machine.

❺ Le récepteur effectue donc un DMA réseau, depuis la table *copy_phys_addr[]* vers la table *recv_phys_addr[][]*. Il coche alors, dans la table *bitmap_recv_phys_addr[][]*, les entrées qu'il a utilisées dans sa table *recv_phys_addr[][]*, elles seront libérées à la fin de l'échange.

❻ Le MI associé à cet envoi est [B, Id, R], la valeur Id correspondant au choix d'un bit libre de la table *bitmap_mi[][]*, qui constitue le *pool* d'indices que nous avons introduit précédemment. Le bit sera libéré à la fin de la transaction.

❼ Le problème de B est maintenant de pouvoir indiquer à A la localisation dans *recv_phys_addr[][]* de la description du tampon de réception. Le MI constitue la seule information provenant de B, qui est fournie à SLR/P au moment de la signalisation de l'arrivée de la description du tampon de réception. C'est donc à l'aide de la connaissance du numéro de nœud de B et de l'indice que A doit pouvoir retrouver la partie de *recv_phys_addr[][]* qui l'intéresse.

   Pour ce faire, A et B vont utiliser la table *received_receive[][]* de A, indexée à la fois sur les nœuds distants, et sur les indices. Ainsi, B va, au moment du DMA, transmettre en sus, dans l'entrée de cette table réservée au nœud B et à l'indice Id choisi précédemment, une référence vers la zone de *recv_phys_addr[][]* qui nous concerne (une partie de la table *pending_receive[]* est utilisée en tant que zone physique de B contenant ce qui doit être transmis dans cette entrée de table chez A).

❽ A ira donc, le moment venu, à partir du MI, examiner l'entrée de *received_receive[][]* correspondante, pour savoir où chercher dans *recv_phys_addr[][]* les informations qui l'intéressent.

❾ On va voir section suivante comment A est amené à envoyer un message de donnée à destination du tampon de réception. Une fois ce message signalé, toutes les entrées de tables de B en jeu dans la transaction sont effacées et la fonction de *callback* de l'utilisateur est invoquée par SLR/P.

Le message de contrôle envoyé par le destinataire est donc composé, chez B, d'une zone de *copy_phys_addr[]* ainsi que d'une zone de *pending_receive[]*. Ce message est à destination d'une zone de *received_receive[][]* et de *recv_phys_addr[][]*, ces zones ayant un rôle de boite aux lettres.

Certaines de ces zones chez A et chez B peuvent être discontiguës, le message de contrôle pourra donc être composé de plusieurs entrées de LPE.

5.3.4 Structures de données gérées par l'émetteur

.

❶ Lorsque l'utilisateur invoque `slrpp_send()`, le travail à effectuer est stocké dans la table *pending_send[]*, et la description des tampons d'émission dans *send_phys_addr[]*. Le numéro de séquence de cette demande d'émission, sur le canal en question, est stocké dans la table *recv_seq[][]*.

❷ Lorsque l'utilisateur invoque `slrpp_send()`, ou lorsque A vient de recevoir un message de demande d'émission, le nœud émetteur va parcourir les tables *pending_send[]* et *received_receive[][]* afin de pouvoir trouver deux entrées associées : le nœud distant, le numéro de canal et le numéro de séquence doivent être similaires.

❸ Si la recherche est fructueuse, toutes les informations pour effectuer un DMA entre les tampons d'émission et de réception sont réunies : l'émetteur invoque alors PUT.

❹ Une fois l'émission de ce message signalée, toutes les entrées des tables de A entrant en jeu dans cette transaction sont effacées et la fonction de *callback* de l'utilisateur est invoquée par SLR/P.

Nous avons indiqué un parcours de *received_receive[][]*. Cette table étant directement remplie par B, elle peut, à tout moment, perdre son intégrité, si le contrôleur réseau est en train d'y inscrire des données. Pour éviter de parcourir des entrées incomplètes, A utilise la table *recv_valid[][]* pour garder en mémoire les entrées qui ont été transmises en totalité. L'indication dans *recv_valid[][]* est mise en place lors de la signalisation de la réception d'un message de type 'R'.

5.3.5 Gestion des ressources : famine et interblocage

On vient de le voir, SLR/P a besoin d'une douzaine de tables pour gérer ses canaux virtuels.

Un certain nombre de ces tables sont allouées auprès de CMEM afin d'être contiguës en mémoire physique, pour minimiser le nombre de pages réseaux constituant le message de contrôle émis par le récepteur. Elles ont donc une taille fixe, définie à la compilation de MPC-OS.

SLR/P, au cours d'une opération d'émission ou de réception, peut donc se trouver face à une famine d'entrées libres d'une de ces tables. Unix étant un système d'exploitation multiprocessus, plusieurs émissions et réceptions vont pouvoir être effectuées *simultanément* sur un même nœud.

Pour éviter l'interblocage traditionnel de ce type de comportement, SLR/P est programmé en suivant la méthode classique dans les systèmes multi-processus. A chaque fois que SLR/P a besoin d'acquérir plusieurs ressources simultanément pour effectuer une opération particulière, il entre au préalable en section critique en interdisant toute interruption ou préemption, et tente d'acquérir chaque ressource séparément. Deux cas peuvent alors se produire :

✓ *Des ressources sont indisponibles* : SLR/P libère toutes les ressources disponibles qu'il vient de réquisitionner, puis sort de la section critique, et extrait le processus qui l'a invoqué de la file des processus éligibles par le système, pour le placer dans la file des processus en attente de ressource, en utilisant le service noyau standard Unix `tsleep()` ;

✓ *Toutes les ressources nécessaires sont disponibles* : SLR/P sort de la section critique, puis il effectue son opération. Lorsqu'elle se termine, il libère alors toutes les ressources, et déplace le processus de la file d'attente de ressource vers la file des processus éligibles, en utilisant le service noyau standard Unix `wakeup()`.

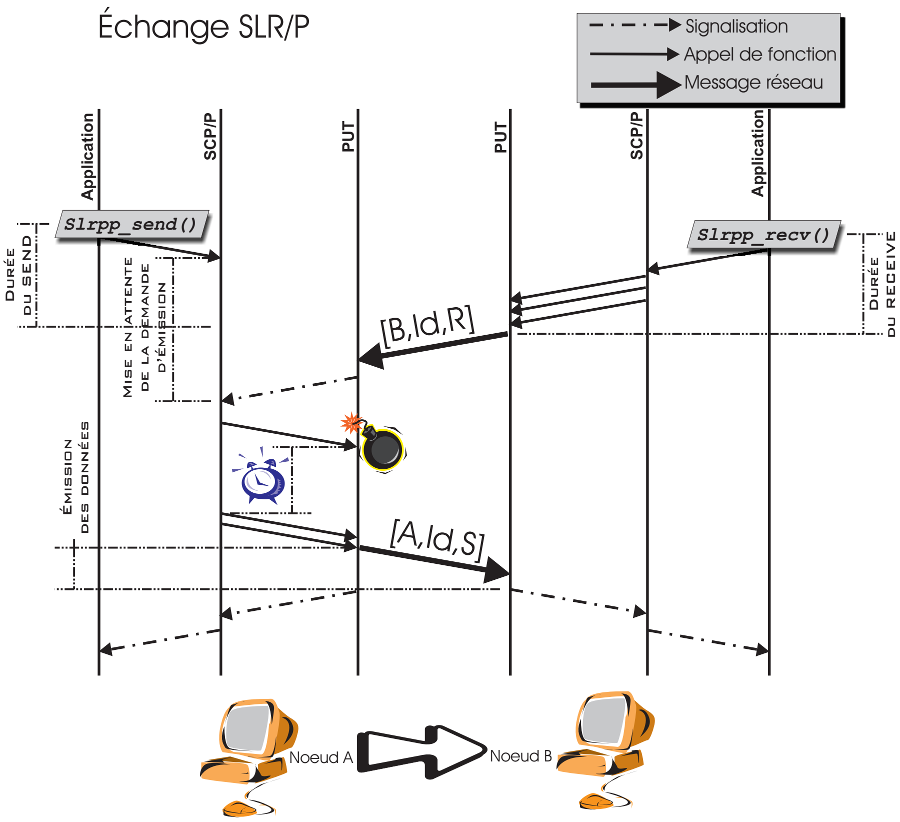


Fig. 5.5 *Échange SLR/P avec famine de la LPE*

Notons qu'on vient de citer « le processus qui vient d'invoquer SLR/P », dans notre algorithme pour éviter les famines de ressources (l'objectif étant de mettre ce processus en attente pour lui faire effectuer le travail plus tard). Malheureusement, il existe un cas pour lequel SLR/P doit effectuer une opération alors qu'il n'a été invoqué par aucun processus : il s'agit du cas où `slrpp_send()` est invoqué avant la réception du message de contrôle correspondant à l'exécution de la fonction `receive()`. Dans cette situation, SLR/P est invoqué en interruption, à la réception du message de contrôle.

Rappelons que l'opération à effectuer alors consiste à apparier deux entrées des tables *pending_send[]* et *received_receive[][]*. Il n'y a donc aucune ressource supplémentaire à acquérir, mais il faut invoquer PUT pour émettre le message de données. Dans le cas où la LPE est déjà pleine, PUT va refuser cet envoi. Il faut donc que SLR/P se mette en attente de libération d'entrées de la LPE. Il ne peut utiliser `tsleep()`, car il irait alors bloquer le processus qui se trouverait par hasard actif au moment où l'interruption s'est produite. Ce comportement menant à bloquer un processus quelconque est prohibé, car il peut aboutir à un blocage de la machine[6].

Ainsi, pour pallier à une famine de la LPE lors d'une opération d'émission de SLR/P se produisant lors d'une interruption matérielle, on met en place une alarme noyau produisant une interruption logicielle au bout d'un certain délai, et SLR/P tente alors à nouveau l'opération d'émission. On répète l'opération jusqu'à ce que la famine soit jugulée et que l'opération ait pu être effectuée, comme schématisé figure 5.5 page précédente.

Rappelons-nous que les points d'entrée de SLR/P sont spécifiés non bloquants et examinons maintenant les implications d'une famine de ressources de SLR/P ou de PUT :

- ▷ En cas de famine d'entrées dans les tables de SLR/P, les appels à `slrpp_send()` et `slrpp_recv()` sont bloqués jusqu'à ce que l'ensemble des ressources soit disponible ;
- ▷ En l'absence de famine de ressources, les points d'entrée de SLR/P ne sont pas bloquants, c'est-à-dire qu'ils rendent la main avant la fin de l'opération de transfert des données proprement dites. Il s'agit donc bien de points d'entrée de type non bloquant, et la signalisation de fin de transaction se fait sur interruption, par invocation asynchrone de fonctions de *callback*. Du côté réception, cette signalisation intervient quand toutes les données ont été déposées en mémoire du nœud récepteur. Du côté émission, la signalisation intervient quand toutes les données ont quitté le nœud émetteur.

5.3.6 Modèles de programmation avec SLR/P

Les fonctions `send()` et `receive()` de SLR/P sont par construction non bloquantes. Heureusement, il est possible d'utiliser un modèle de programmation avec appels bloquants (dans la plupart des cas pour se passer de fonction de *callback*). En effet, pour transformer un point d'entrée non bloquant de SLR/P en point d'entrée bloquant, il suffit de rajouter quelques opérations encadrées par une section critique après l'appel à la fonction `send()` ou `receive()`, et de définir une fonction de *callback* particulière. La figure 5.6 page suivante présente l'algorithme pour atteindre ce but.

6. Une interruption Unix est exécutée dans le contexte (pile système, registres, etc.) du processus qui se trouvait actif au moment de sa levée. On ne peut donc travailler avec les opérations de synchronisation du noyau `tsleep()` et `wakeup()`, qui vont agir sur ce contexte. Certains systèmes comme Solaris ou Windows NT sont capables d'instancier, et à faible coût, un nouveau contexte pour permettre l'utilisation de synchronisation en interruption ; ils utilisent pour cela un *pool* de contextes non affectés. Ni FreeBSD ni Linux ne le permettent.

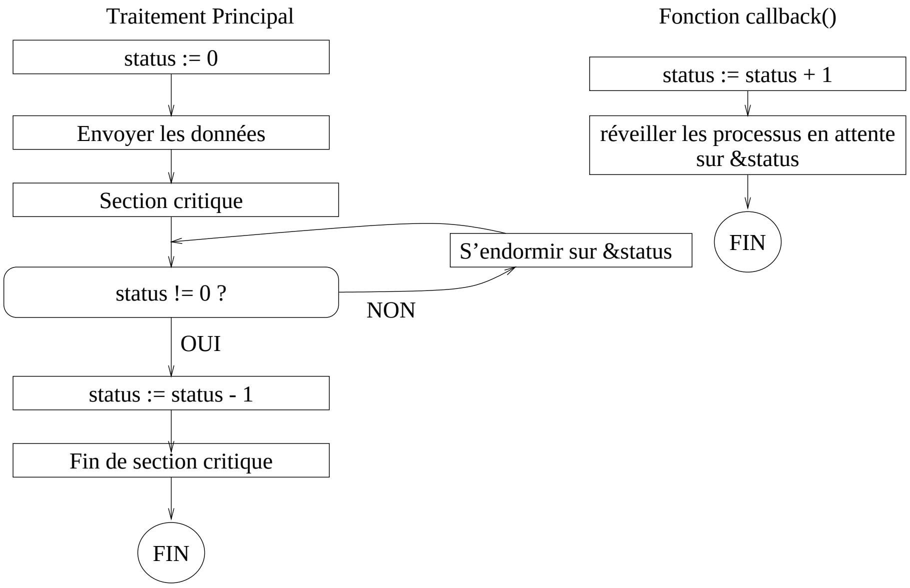


Fig. 5.6 *Exemple d'algorithme d'utilisation de SLR/P (ex.1)*

Entre l'instant où le point d'entrée non bloquant de SLR/P retourne la main à l'appelant, et le moment où la fonction de *callback* est invoquée, il y a un temps qui est perdu pour l'application quand on utilise un tel modèle bloquant. Le comportement non bloquant trouve son utilité quand l'application sait profiter de ce délai pour effectuer divers calculs. La figure 5.7 présente un algorithme implémentant ce mode de fonctionnement.

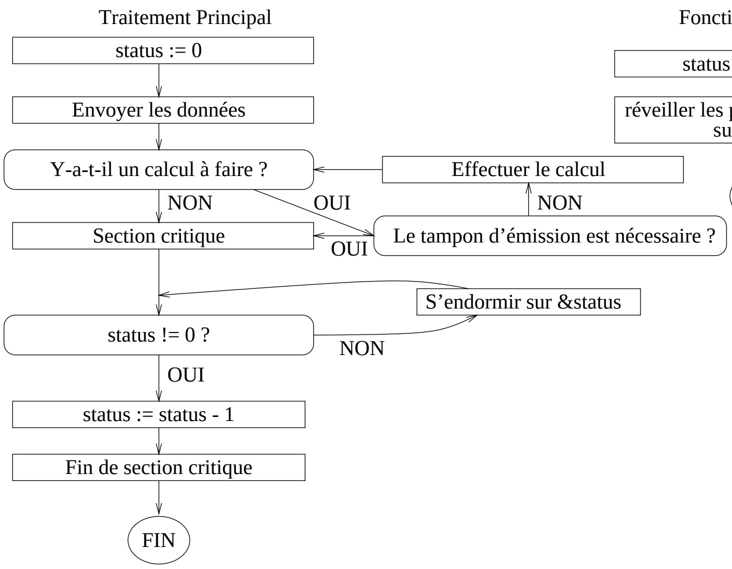


Fig. 5.7 *Exemple d'algorithme d'utilisation de SLR/P (ex.2)*

5.4 Conclusion

On vient d'étudier dans ce chapitre la construction d'un protocole de communication sur canaux virtuels, au dessus de la primitive d'écriture distante. On permet donc au programmeur d'effectuer des transferts sans connaître la localisation des tampons distants dans la mémoire du nœud récepteur. D'autre part, on a montré qu'on peut utiliser les primitives de SLR/P au choix en mode bloquant ou non bloquant.

Il reste néanmoins deux services fournis habituellement au programmeur, pour lesquels SLR/P ne propose pas de solution. D'une part, SLR/P n'est pas un protocole sécurisé, c'est-à-dire qu'il ne garantit pas la reprise sur erreur en cas de faute matérielle lors de l'acheminement d'un message, et d'autre part, il ne permet pas de manipuler des adresses virtuelles. Les deux chapitres qui suivent vont présenter comment on peut rajouter ces services au dessus de SLR/P.

Chapitre 6

Sécurisation de la communication

Sommaire

La couche de communication SCP/P, est l'homologue sécurisée de SLR/P. Elle offre la même interface et les mêmes services que SLR/P, tout en ajoutant la garantie d'intégrité des données reçues. Pour arriver à un tel résultat, on va étudier les causes d'erreur sur les liens et classifier leurs conséquences, afin de pouvoir faire des choix quant à l'architecture sous-jacente de SCP/P, qui reprend le noyau de SLR/P et y ajoute des sous-protocoles de traitement des erreurs.

6.1 Le problème de la tolérance aux fautes

La tolérance aux fautes dans les réseaux de communication pour machines parallèles consiste le plus souvent à garantir l'absence de fautes au niveau de la conception des constituants matériels du réseau. En effet, la prise en compte des fautes est habituellement incompatible avec les exigences de performances. C'est pourquoi la tolérance aux fautes a été le plus souvent étudiée dans des domaines où les exigences de performances sont moindres, tels que les réseaux sans fils [Gao, 2000] ou les réseaux WAN construits par exemple sur TCP/IP.

Avec la machine MPC, nous avons été confrontés au problème des fautes matérielles, et nous avons donc dû établir des protocoles tolérants à ces fautes, tout en gardant les meilleurs performances possibles, et donc notamment en préservant le caractère zéro-copie des transmissions. Pour des raisons de complexité et de faisabilité, il nous faut intégrer le support de tolérance aux fautes uniquement au sein du logiciel, donc de MPC-OS. On va ainsi agir de bout-en-bout, afin que deux extrémités communicantes ne soient pas tributaires des fautes sur un lien intermédiaire du réseau. On peut remarquer que cette approche de bout-en-bout est la plus traditionnelle. Elle est par exemple utilisée dans ATM, pour d'autres raisons. Notons cependant une approche au niveau du lien physique, assez rare, présentée dans [Lee *et al.*, 2000].

6.2 Classification des fautes du réseau

6.2.1 Comportement du matériel en cas de faute du réseau

Des sommes de contrôle et des bits de parité permettent aux circuits PCI-DDC et RCube de détecter les fautes sur les liens. Nous allons commencer par étudier ici leurs comportements respectifs en cas d'erreur détectée, afin de pouvoir déterminer dans la section suivante les conséquences d'une telle faute dans le nœud récepteur.

La figure 6.1 page ci-contre présente le format d'un paquet de données acheminé sur le réseau de routeurs RCube. Il est constitué de trois parties :

- ▷ Une en-tête pour le routage, indiquant le numéro de nœud destinataire ;
- ▷ Les données encapsulées dans ce paquet, correspondant au niveau de protocole géré par PCI-DDC ;

▷ Une fin de paquet, constituée d'un caractère réservé à cet effet.

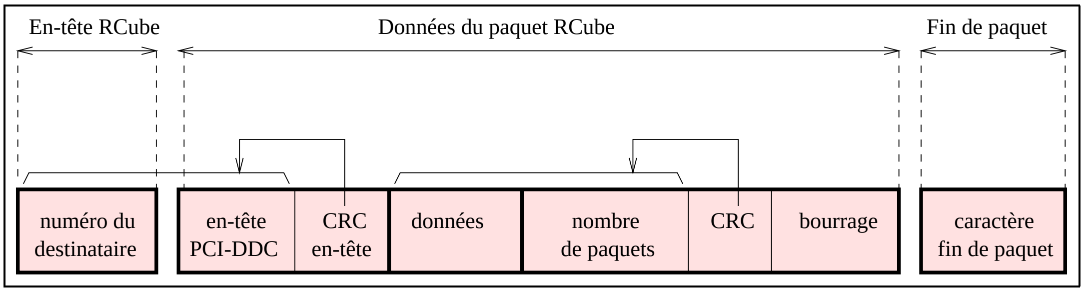


Fig. 6.1 *Format des paquets sur le réseau Gigabit*

Les données encapsulées par PCI-DDC dans ce paquet sont composées de plusieurs sections : une en-tête indiquant les adresses physiques destination et l'identificateur de message associé à ce paquet, les données proprement dites, le nombre total de paquets s'il s'agit du dernier paquet du message[1], et deux sommes de contrôle[2] sur 32 bits, une sur l'en-tête du paquet, une autre sur la partie données.

Lorsqu'aucun paquet ne transite sur un lien HSL, des caractères de bourrage nécessaires à synchronisation (calibration) sont échangés. Ceux-ci peuvent évidemment aussi se trouver altérés.

Un bit de parité est ajouté à chaque caractère transmis sur le lien (y compris les caractères de bourrage). Une faute réseau peut donc être détectée par ce biais, et dans un tel cas RCube clôt immédiatement le paquet en cours par un caractère EEP[3] (fin de paquet exceptionnelle), et ignore tous les caractères qui suivent, jusqu'à ce qu'il rencontre un caractère EP[4] (fin de paquet). En effet, il n'y a pas de caractère spécial de début de paquet, il faut donc attendre une fin de paquet pour se synchroniser sur le paquet suivant.

Si le bit de parité calculé et vérifié par RCube ne suffit pas, les sommes de contrôle calculées et vérifiées de bout en bout par PCI-DDC permettent de détecter les erreurs de lien.

Pour cela, il travaille en trois étapes :

✓ Première étape (réception de l'en-tête) :
il récupère les données de l'en-tête et teste la somme de contrôle associée. Si elle est correcte, il passe à l'étape suivante, sinon il indique une erreur dans un registre dédié et génère une interruption ;

✓ Deuxième étape (réception des données) :
il dépose les données directement en mémoire au fur et à mesure de leur arrivée, puis teste la somme de contrôle. Si celle-ci est correcte, il passe à l'étape suivante, sinon, il indique une erreur dans un registre dédié et génère une interruption ;

✓ Troisième étape (fin de paquet) :
il attend un caractère de fin de paquet. Il incrémente alors le nombre de paquets

1. Le nombre total de paquets est utilisé dans le mécanisme de comptage des paquets décrit précédemment section 2.1.9 page 43.
2. CRC
3. EEP : Exceptional End of Packet
4. EP : End of Packet

reçus dans la table LMR, et s'il s'agit du dernier paquet attendu, il signale la fin de message comme décrit section 2.1.6 page 41.

À tout moment, PCI-DDC est susceptible de recevoir un caractère EEP indiquant une fin de paquet exceptionnelle, généré par RCube. Dans un tel cas, il interrompt le traitement du paquet en cours : il positionne un drapeau dans un registre de statut pour indiquer la réception d'un caractère EEP, il lève une interruption, puis remet à zéro le drapeau BUS MASTER du registre de contrôle PCI. Ceci a pour effet de bloquer les modules d'émission et de réception de PCI-DDC. Quand, par la suite, le pilote de périphérique repositionne le BUS MASTER, PCI-DDC repart à l'étape d'attente de réception d'une en-tête, quelle que soit l'étape à laquelle il était lorsque le caractère EEP a été reçu.

6.2.2 Les conséquences d'une faute dans un nœud récepteur

Remarque préliminaire : on considère dans ce manuscrit que des erreurs sur les liens peuvent se produire, que des caractères peuvent donc se trouver altérés, éventuellement sans violation de parité, mais que les CRC discriminent, dans tous les cas, les données correctes des données altérées lors du transit à travers le réseau. On se permet cette supposition car les sommes de contrôle sont codées sur 32 bits.

Pour identifier les conséquences d'une faute du réseau, on va établir une classification des fautes en fonction de la section altérée, et déterminer leurs effets à partir des constatations de la section précédente.

Deux cas particuliers se présentent :

- ✓ Tout d'abord, envisageons le cas où le numéro de nœud destinataire, qui se trouve au début du paquet, est altéré. S'il s'agit d'une erreur de parité, le paquet est tout simplement perdu. Si la parité reste correcte, le paquet peut être acheminé vers un mauvais nœud (une erreur de CRC y sera générée). Mais si aucun nœud ne porte ce numéro erroné, le réseau peut se retrouver bloqué ou le paquet peut tourner indéfiniment dans le réseau, suivant l'état des tables de routage et la longueur du paquet.
- ✓ Imaginons maintenant qu'un caractère de bourrage ou bien un caractère de fin de paquet soit erroné. Si la parité permet de le détecter ou que ce caractère ne fait pas partie de ceux valides pour un début de paquet, RCube va alors éliminer les prochains caractères jusqu'à ce qu'il rencontre un nouveau caractère EP, et va donc ignorer le paquet suivant. On peut donc ainsi perdre un paquet qui n'est pas erroné. Si la parité reste correcte et que ce caractère de bourrage est valide pour commencer un nouveau paquet, on revient au premier cas particulier.

Si toute autre section d'un paquet se trouve endommagée, le traitement classique de RCube consiste à ignorer la suite du paquet, et celui de PCI-DDC à interrompre son travail sur le paquet en cours et à signaler par interruption la cause du problème.

Le tableau qui suit récapitule les effets d'une erreur de lien en fonction de la zone altérée :

Zone altérée	**Effets**
numéro de nœud	Interblocage ou paquet en boucle (baisse de performance du réseau et perte du paquet).
en-tête PCI-DDC ou CRC de l'en-tête	Aucune donnée n'est déposée. Le paquet est perdu.
données	Des données erronées sont déposées. Le paquet n'est pas compté en réception.
numéro de paquet, CRC ou bourrage	Les données sont correctement déposées. Le paquet n'est pas compté en réception.
caractère EP de fin de paquet	Les données sont correctement déposées. Le paquet n'est pas compté en réception. Le paquet suivant est perdu.
caractères de bourrage entre paquets	Le paquet suivant est perdu.

On constate ainsi que, du point du vue des nœuds de calcul, les erreurs sur les liens se traduisent de trois manières distinctes :

– Altération du fonctionnement du réseau (interblocage ou pertes de performances) ;
– Perte de paquets, et dans ce cas ils ne sont pas comptés ;
– Dépôt de données incorrectes, sur un nœud correct, à une adresse correcte.

Ainsi, seuls les paquets dont les données ont été déposées sans erreur sont comptés, et quand des données erronées sont déposées, elles le sont sur le nœud auquel elles étaient destinées et dans le tampon de réception qui leur était attribué.

Lorsque la table LMR est activée, la signalisation de la fin de réception d'un message utilise le système de comptage des paquets. PCI-DDC ne signale donc pas les messages dont un ou plusieurs paquets qui les composent ne se seraient pas correctement déposés. Il n'y a donc pas d'interruption ni d'entrée dans la LMI pour de tels messages.

6.3 Tolérance aux fautes matérielles du réseau : SCP/P

6.3.1 Intégration au niveau canaux

Nous avons pour l'instant présenté deux protocoles : le protocole d'écriture distante de l'interface PUT et le protocole de communication sur canaux de l'interface SLR/P. Où doit-on intégrer un mécanisme de reprise sur erreur ?

Le déroulement d'une transaction SLR/P démarre par une demande de transaction par le récepteur, et se termine par une signalisation de fin de transaction toujours chez le récepteur : la première et la dernière phase d'une transaction sont générées sur le même nœud récepteur.

Au contraire, avec PUT, la première phase se produit chez l'émetteur, tandis que la dernière phase se termine chez le récepteur.

On aura donc plus de facilités à implémenter la reprise sur erreur en travaillant au niveau du protocole de transmission sur canaux, car le nœud récepteur est tout à la fois initiateur de la transaction et destinataire de la signalisation de fin d'opération : on peut donc sur ce même nœud initier des transactions et attendre leurs effets, ce qui revient donc à détecter les transactions dont l'accomplissement n'est pas signalé, c'est-à-dire celles pour lesquelles des données ont été perdues ou erronées.

On va donc, pour construire notre protocole sécurisé SCP/P, réutiliser les deux opérations de base de SLR/P : la demande d'émission (opération RECEIVE), et l'émission effective de données (opération SEND). On va les regrouper au sein du sous-protocole de SCP/P nommé BSCP [5]. Il s'agit donc d'une version de SLR/P expurgée des mécanismes de contrôle de l'état du protocole et de génération d'interruption logicielle. Le contrôle de l'état du protocole va évidemment devoir être redéfini pour intégrer la reprise sur erreur.

6.3.2 Incompatibilité adaptativité/tolérance aux fautes

Le protocole SLR/P fonctionne sur réseau adaptatif ou non. En cas de réseau adaptatif, il utilise la table LMR pour que PCI-DDC puisse compter les paquets en réception afin de pouvoir émettre une signalisation quand tous les paquets composant un même message sont déposés.

Dans le cas d'un réseau adaptatif, un paquet peut potentiellement passer un temps très grand dans le réseau, car il peut se trouver bloqué dans une artère en contention, alors que d'autres voies en amont se sont libérées et laissent passer les paquets suivants du message.

Un message peut donc être déposé partiellement et voir une de ses parties se retrouver un long moment dans le réseau sans que l'on puisse distinguer s'il s'agit d'un paquet éliminé car corrompu, ou tout simplement en attente dans un routeur.

Si le réseau n'est pas adaptatif, il suffit d'émettre un *ping/pong* entre deux nœuds pour vérifier qu'un message censé transiter entre eux s'est perdu *et n'apparaîtra jamais plus* : en effet, le *ping/pong* fera office de voiture balai, et lorsqu'il sera revenu chez son initiateur, celui-ci aura la certitude qu'aucun paquet n'est en attente sur cette voie.

On constate donc que la tolérance aux fautes du réseau, et par voie de conséquence SCP/P, nécessite des tables de routage non adaptatives.

6.3.3 Gestion de la signalisation

Nous venons de voir qu'il est nécessaire de travailler avec un réseau non adaptatif. En principe, cela permet de se passer de la table LMR, qui sert au comptage des paquets en vue de la signalisation de fin de message reçu.

Nous allons néanmoins, dans le cadre de SCP/P, activer cette table en principe inutile dans un réseau non adaptatif, et l'utiliser de façon détournée. Quand un paquet est perdu ou

5. BSCP : Basic SCP

corrompu, il n'est évidemment pas compté dans la LMR. Activer cette table nous permet donc d'éviter de signaler les messages qui ne sont pas transmis en totalité : en effet, si le nombre de paquets attendus n'est pas égal au nombre de paquets reçus, le message n'est pas signalé.

Il faut néanmoins faire attention aux réémissions : le compteur de la LMR, qui est mis à zéro automatiquement lors de la signalisation par PCI-DDC, doit impérativement être remis à zéro 'manuellement' par le logiciel avant une réémission, sinon on risque des signalisations erronées ou de disposer d'un compteur n'atteignant jamais l'égalité entre le nombre de paquets attendus et le nombre de paquets reçus, comme on peut le constater sur l'exemple de la figure 6.2, qui représente l'état du compteur après deux émissions d'un même message, la première émission ayant été incomplète.

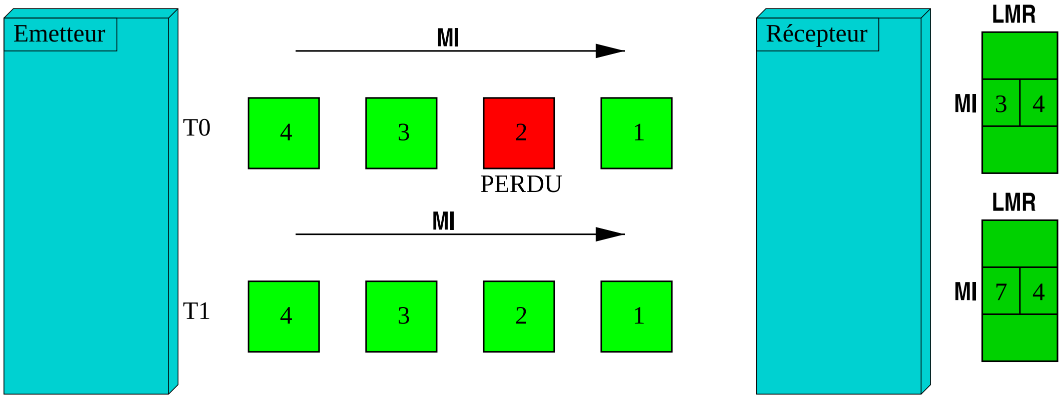


Fig. 6.2 *Comptage des paquets reçus sans remise à zéro du compteur*

6.3.4 Délai de garde

La solution naïve pour détecter le bon déroulement d'une transaction consiste à utiliser un délai de garde du côté récepteur. La figure 6.3 page suivante présente deux exemples de transactions et montre qu'il n'est malheureusement pas suffisant d'instaurer un délai de garde pour détecter une perte de données :

- ▷ Analysons la première transaction sur cette figure : le récepteur effectue un `receive()`, et l'émetteur a effectué un `send()` en attente sur le même canal, avec le même numéro de séquence. Deux cas peuvent se produire : le message RECEIVE n'arrive pas entier chez l'émetteur, ou bien au contraire il arrive à bon port, mais le message SEND qui en découle se trouve quant-à-lui corrompu. Dans les deux cas, il est nécessaire de relancer un message de type RECEIVE.
- ▷ La deuxième transaction est caractérisée par l'absence de `send()` en attente. Ainsi, quoi qu'il arrive sur le réseau, il n'y aura pas de message SEND parvenant au récepteur, même après l'expiration du délai de garde, mais il n'y a pas lieu de réémettre un message RECEIVE.

Cet exemple montre qu'un délai de garde n'est pas suffisant pour faire détecter au récepteur les mauvais fonctionnements du réseau. Il nous faut donc rajouter un sous-protocole qui

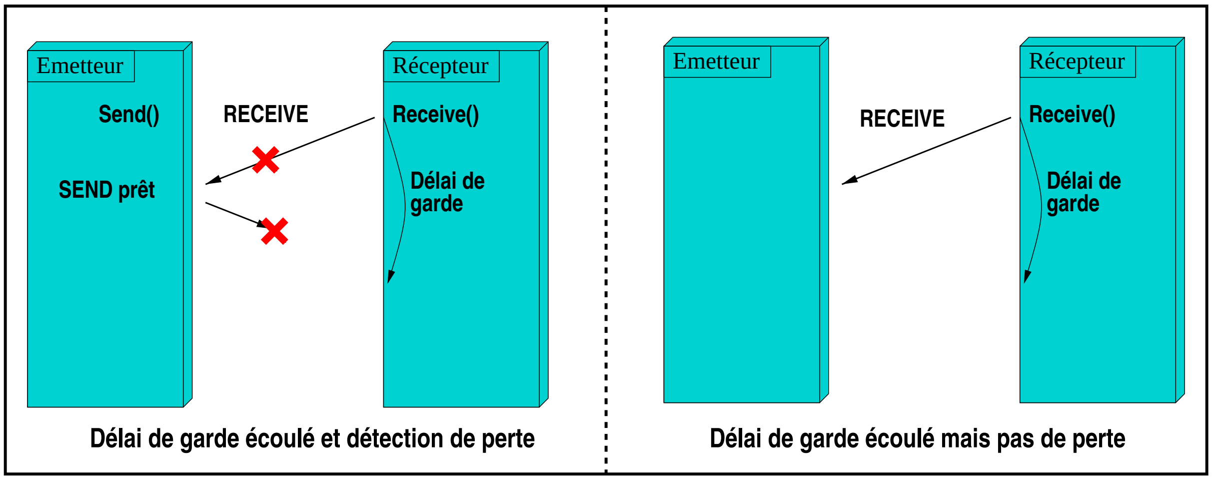


Fig. 6.3 *Délai de garde*

permette à l'émetteur d'indiquer au récepteur qu'un `send()` est en attente sur un canal donné. Nous nommerons ce sous-protocole SFCP [6].

Utilisé conjointement avec un délai de garde, SFCP va permettre au récepteur de décider à quel moment réitérer un message RECEIVE.

6.3.5 Données altérées en cas de réémission

Le mécanisme d'écriture distante peut créer des problèmes assez inhabituels dans le monde des protocoles réseau. Ceci est lié au fait que des données altérées peuvent être déposées, car les sommes de contrôles sur les données sont calculées au fur et à mesure du dépôt, la vérification n'intervenant qu'à la fin du dépôt.

On peut le constater sur l'exemple de la figure 6.4 page suivante : le délai de garde est trop court, on réactive alors un RECEIVE. Il s'en suit un deuxième SEND, les données sont donc transportées deux fois dans le réseau. Lors du premier dépôt, tout se passe correctement. L'application perçoit donc la signalisation de fin de réception. Elle peut alors immédiatement commencer à travailler sur les informations reçues. Pendant ce temps, les données en principe identiques du deuxième SEND sont déposées. Imaginons qu'une partie des données soit corrompue lors du transit à travers le réseau, sans erreur de parité. L'application va alors travailler sur des données qui étaient correctes au moment de la signalisation, et qui sont erronées au moment de leur utilisation.

6.3.6 Réutilisation des MI

Les MI associés aux opérations SEND et RECEIVE de BSCP sont évidemment ceux définis au sein de SLR/P, car BSCP en est issue. Nous avons justifié section 5.3.2 page 106 le choix des valeurs de ces MI, découpés en champs de bits. Nous ne pouvons donc pas ici décider d'un autre choix pour la valeur des MI.

6. SFCP : Sender Flow Control Protocol

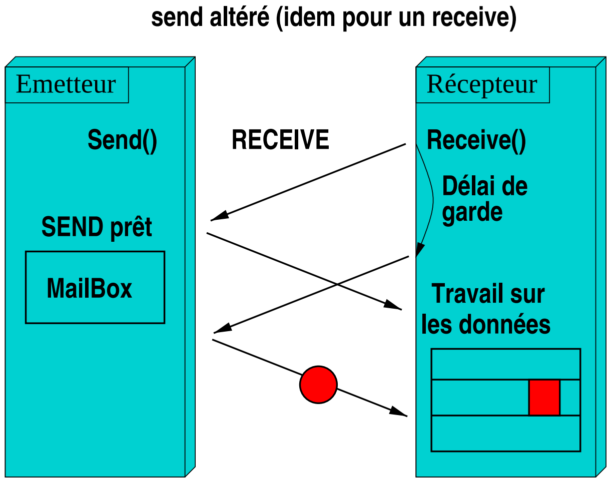


Fig. 6.4 *Travail sur des données altérées*

Aux différents messages RECEIVE ou SEND d'une transaction entre un émetteur et un récepteur correspondent bien sûr un même type de message, des mêmes numéros de nœud et un même numéro de canal. Or il s'agit des trois composantes du MI. En cas de réémission, les différents RECEIVE et SEND utiliseront donc le même MI.

Nous avons vu section 6.3.3 page 122 que la réutilisation d'un MI nécessitait une remise à zéro de son compteur dans la table LMR.

Il nous faut donc un mécanisme de remise à zéro des deux tables LMR du récepteur et de l'émetteur avant toute nouvelle tentative de réémission de RECEIVE ou de SEND.

Nous allons nommer MICP[7] ce sous-protocole de SCP/P.

Nous verrons qu'il nécessite deux messages, l'un du récepteur vers l'émetteur, et un autre en retour pour signaler la fin de la mise à zéro des tables LMR. Ce protocole est basé sur le principe du *ping/pong*, tout comme le protocole nécessaire pour résoudre les problèmes évoqués section 6.3.5 page ci-contre. MICP proposera donc une solution conjointe au problème de la réutilisation des MI et à celui des données altérées.

6.3.7 Libération du tampon d'émission

Dans le protocole SLR/P, le tampon d'émission est libéré dès que les données de l'émetteur viennent de rentrer en totalité dans le réseau. On ne peut plus faire de même dans SCP/P, puisque le récepteur peut être amené à redemander un émission, s'il détecte, par exemple à l'expiration de son délai de garde, que la transaction ne s'est pas accomplie correctement.

Il faut donc un moyen pour l'émetteur d'être prévenu que le récepteur a correctement reçu les données, et que la fin de transaction peut donc être signalée à l'application émettrice.

Ce sous-protocole qui transporte une information depuis le récepteur vers l'émetteur, est le dual du sous-protocole SFCP, qui, lui, transporte une information dans le sens opposé.

7. MICP : MI Cleaner Protocol

Ce protocole va donc être nommé RFCP [8].

6.4 Protocole SCP/P

6.4.1 Conception du protocole

Nous venons d'examiner différents problèmes spécifiques à l'apparition d'erreurs sur le réseau HSL. Nous avons été amenés à définir l'aspect fonctionnel de quatre sous-protocoles de SCP/P afin de contourner ces problèmes : BSCP, SFCP, RFCP et MICP.

La figure 6.5 page suivante présente de manière synthétique la réflexion que nous allons mener à la lumière des discussions qui ont précédé dans ce chapitre.

Tous les liens de causalité entre les boites dessinées sur cette figure sont explicités dans la discussion décomposée en huit étapes qui suit :

- ❶ Nous avons montré précédemment qu'il est plus ingénieux de baser notre protocole sur les opérations SEND et RECEIVE de SLR/P, que de recommencer à travailler sur PUT. Ainsi, deux types de messages vont transiter sur le réseau : les messages RECEIVE, à destination de boite aux lettres de l'émetteur, et les messages SEND transportant les données proprement dites.
- ❷ On peut donc avoir des erreurs sur les paquets de type SEND et ceux de type RECV. D'autre part, les paquets en question peuvent être corrompus (ceci menant à des pertes) ou tout simplement retardés (ceci introduisant une durée de transport plus longue que les éventuels délais de garde).
- ❸ On va donc étudier conjointement la perte d'un RECEIVE, la perte d'un SEND, les conséquences d'un paquet corrompu et les conséquences d'un paquet retardé.
- ❹ Les pertes d'un paquet RECEIVE ou d'un paquet SEND nécessitent toutes deux une réémission, donc un mécanisme de réutilisation des MI. Ce mécanisme sera implémenté par MICP.
- ❺ La perte d'un RECEIVE nécessite une réémission *uniquement* s'il y a un `send()` en attente. On va donc utiliser conjointement un délai de garde, et un mécanisme de détection de `send()` en attente : SFCP.
- ❻ Quand un RECEIVE est réémis, un nouveau SEND en découle, car le travail de l'émetteur consiste tout simplement à renvoyer un SEND pour chaque RECEIVE (c'est le récepteur qui gère et contrôle les opérations, l'émetteur se contente de répondre à quelques stimuli).
- ❼ Les réémissions de SEND ou de RECEIVE nécessitent l'accès à BSCP. La libération en fin d'opération est plus complexe pour l'émetteur, car il lui faut un protocole qui n'est pas présent dans SLR/P, donc pas présent dans BSCP, afin d'être informé de la réalisation complète d'une transaction. Ce protocole se nomme RFCP, et permet la libération définitive du tampon d'émission.
- ❽ Résoudre le problème des paquets corrompus consiste à éviter le travail sur des données altérées. Si un RECEIVE est altéré, il faut pouvoir le réémettre à l'identique,

8. RFCP : Receiver Flow Control Protocol

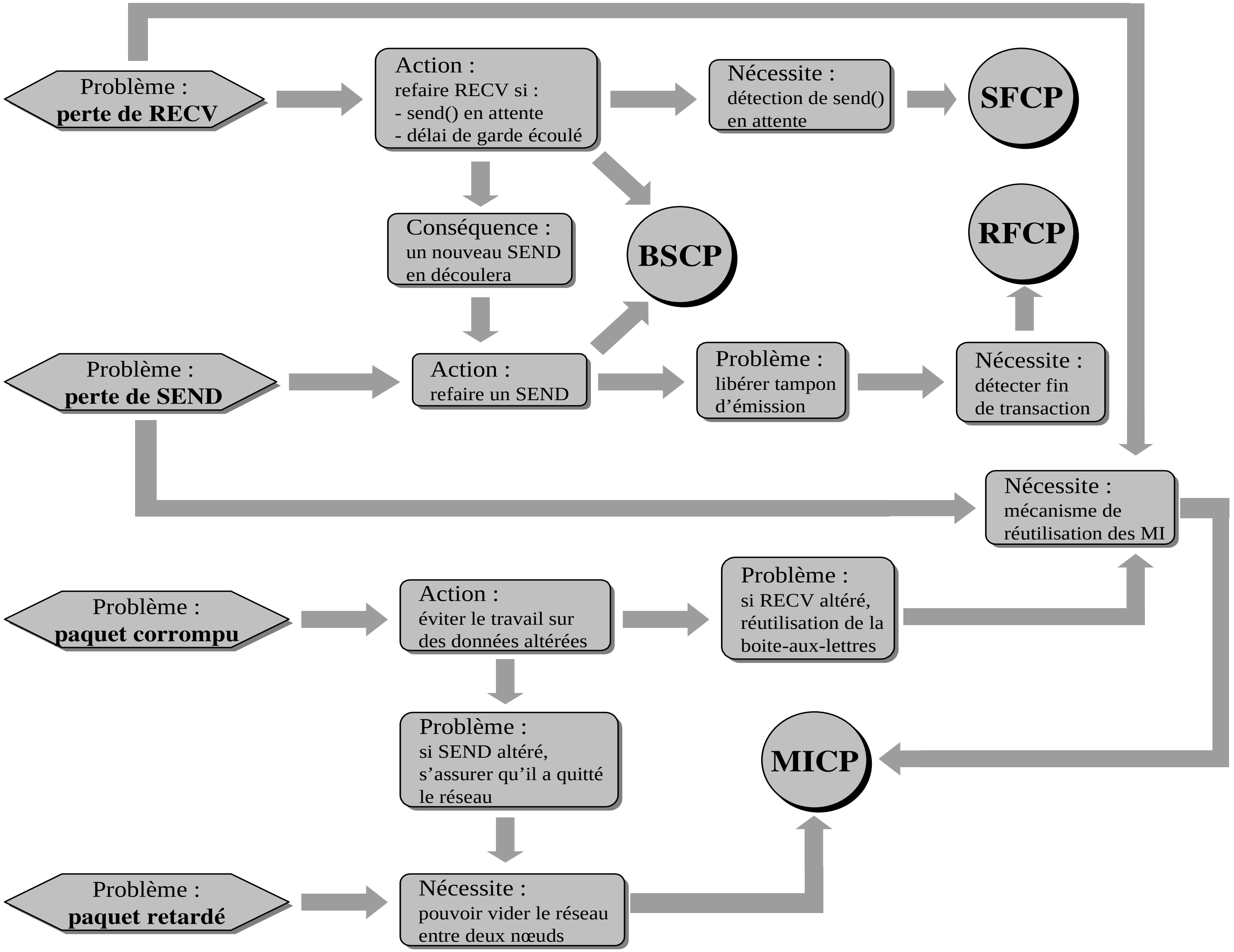


Fig. 6.5 *Conception du protocole*

donc pouvoir réutiliser la boite aux lettres qui lui est associée. Celle-ci étant indexée sur les MI, ce problème est équivalent à celui de la réutilisation des MI. Si un SEND est altéré, il faut s'assurer qu'il a quitté le réseau avant de retenter un échange. Une solution consiste à vider le réseau entre deux nœuds. Là encore, MICP va jouer ce rôle.

6.4.2 Sous-protocoles de SCP/P

BSCP est un sous-ensemble de SLR/P, rassemblant les deux opérations SEND et RECEIVE. Nous devons donc maintenant, pour finaliser notre solution au problème des fautes du réseau, montrer comment concevoir les trois protocoles SFCP, RFCP et MICP que l'on vient d'introduire.

Ils vont tous les trois utiliser des messages courts, afin d'éviter les écueils qu'ils sont censés contourner. En effet, les données d'un message court sont transportées au sein d'un message constitué d'un seul paquet, et sont directement déposées dans la LMI. Il n'y a donc pas d'adresse à fournir, donc pas de boite aux lettres à gérer, et d'autre part, les données étant transportées au sein de l'entête du paquet, on ne risque pas de dépôt de données erronées. Pour résumer, un message court est transmis de manière atomique, correctement ou définitivement perdu, et lorsqu'il est transmis correctement, les données sont tout simplement déposées dans une entrée vide de la LMI. Comme il est transmis de manière atomique, aucun problème de réutilisation de MI ne se pose. Chacun de ces sous-protocoles de SCP/P est un SAP au dessus de PUT qui demande dans ce cadre à disposer d'un seul MI.

Rappelons que la taille des données utiles transportées dans un message court est de 8 octets seulement.

6.4.3 Sous-protocoles SFCP et RFCP

Le protocole SFCP doit informer le récepteur de la présence d'un `send()` en attente. Il suffit pour ce faire d'informer le récepteur du numéro de séquence en cours dans le sens de l'émission, chez l'émetteur, sur le canal en question.

Le protocole RFCP doit informer l'émetteur de la fin d'une transaction, afin qu'il libère définitivement le tampon d'émission en signalant la fin de l'opération à l'application *via* un appel à sa fonction de *callback*. Il suffit à RFCP pour ce faire d'informer l'émetteur du numéro de séquence du plus grand RECEIVE complètement terminé chez le récepteur, sur le canal en question. Attention, il ne s'agit pas ici du numéro de séquence en réception sur ce canal : le numéro de séquence en réception sur ce canal est incrémenté à chaque `recv()` chez le récepteur, car plusieurs `recv()` peuvent avoir lieu sans qu'aucun n'ait été complètement satisfait (on peut effectuer plusieurs `recv()` sur un même canal, car cette opération, comme toutes les autres opérations des couches que l'on a étudiées jusqu'ici, sont non bloquantes). Il nous faut gérer une nouvelle table, indiquant non pas le plus haut numéro de séquence sur un canal donné, mais bien le plus haut numéro de séquence d'une opération terminée sur un canal. Pour introduire cette nouvelle table, il a fallu enrichir

BSCP.

Notons que les tables à transmettre contiennent des entrées uniquement croissantes dans le temps (à la remise à zéro près, en cas de dépassements de capacité du compteur : ce dépassement ne pose pas de problème car les numéros de séquence émetteur et récepteur sont dans des plages rapprochées par rapport à l'amplitude de ces compteurs ; la gestion des dépassement de capacité a donc pu être correctement gérée dans SCP/P).

SFCP et RFCP sont donc deux protocoles qui doivent permettre l'émission d'une table d'un nœud vers un autre. Pour ce faire, on va utiliser un ordonnanceur qui activera SFCP ou RFCP dès qu'une valeur d'une de ces tables aura changé. Le travail de SFCP ou de RFCP consiste donc, dès leur activation, à transporter une entrée (de taille un mot) d'une table, indexée sur les numéros de nœuds et les canaux. Une fois le travail effectué, SFCP ou RFCP informe l'ordonnanceur qu'il n'est plus besoin de l'invoquer.

On utilise un ordonnanceur afin de gérer les cas de pertes de messages courts constituant le protocole SFCP ou RFCP.

Les seules différences entre SFCP et RFCP sont situés au niveau du sens de transport des informations (il est inversé entre SFCP et RFCP) et dans la nature de la table à transporter (mais cette donnée étant opaque, cela ne change rien au protocole).

On va donc décrire un protocole générique pour transporter un mot, le lecteur pourra en extrapoler simplement SFCP et RFCP.

Imaginons que nous voulions transférer le contenu de l'entrée d'index `(noeud, canal)` de la table nommée `TBL_X` sur le nœud A. Il s'agit de la valeur de `TBL_X[noeud][canal]`, en adoptant la notation du langage `C`.

Ajoutons donc deux tables : toujours dans A, la table `TBL_Z`, et dans B la table `TBL_Y`, toutes deux de même format que `TBL_X`.

Notre but est de copier `TBL_X` dans `TBL_Y`, et `TBL_Y` dans `TBL_Z`.

La figure 6.6 présente les données de SFCP avec les tables réelles qui entrent en jeu (`TBL_X` = `send_seq`, `TBL_Y` = `SFCP_copy` et `TBL_Z` = `SFCP_image`).

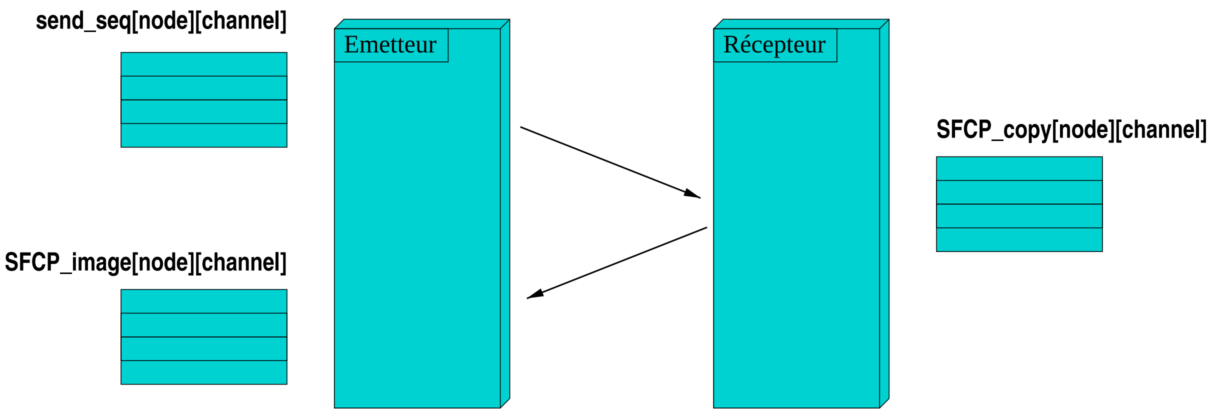


Fig. 6.6 *Protocole SFCP*

Supposons que la valeur de `TBL[noeud][canal]` change (elle augmente donc). L'ordonnanceur, qui utilise des interruptions logicielles, va alors activer régulièrement notre protocole. Celui-ci, à chaque invocation, parcours la liste des travaux à effectuer, c'est-à-dire la liste des index (noeud, canal) pour lesquels TBL_X[noeud][canal] $\neq$ TBL_Z[noeud][canal].

A va alors envoyer vers B, inscrits dans un message court, l'index ainsi que le contenu de TBL_X[noeud][canal]. Dès sa réception, B a la charge de noter la valeur reçue dans TBL_Y[noeud][canal], *uniquement si elle est plus grande que la valeur contenue actuellement*. On évite ainsi tout problème de réordonnancement des messages dans le réseau : TBL_Y[noeud][canal] ne peut que croître. Une fois la valeur notée, B la renvoie à A, et ce dernier la place dans TBL_Z[noeud][canal], toujours avec les mêmes conditions : il ne faut pas que la valeur soit décrémentée.

Vu le scénario qu'on vient de décrire, puisque l'information se propage depuis TBL_X vers TBL_Z en passant par TBL_Y, et puisque les entrées de ces trois tables sont croissantes, on a évidemment :

$$\texttt{TBL\_X[noeud][canal]} \leq \texttt{TBL\_Y[noeud][canal]} \leq \texttt{TBL\_Z[noeud][canal]} \tag{6.1}$$

A recommence ce scénario jusqu'à ce que :

$$\texttt{TBL\_X[noeud][canal]} = \texttt{TBL\_Z[noeud][canal]} \tag{6.2}$$

Une fois cette condition atteinte, A demande à l'ordonnanceur de ne plus être invoqué (à moins bien sûr qu'il y ait d'autres index pour lesquels les deux tables n'aient pas la même valeur).

Notre protocole va donc être actif jusqu'à ce que les deux équations 6.1 et 6.2 soient simultanément satisfaites, donc au moins jusqu'à ce que

$$\texttt{TBL\_X[noeud][canal]} = \texttt{TBL\_Y[noeud][canal]}$$

Notre protocole permet donc de transporter une table dont les valeurs des entrées sont croissantes, d'un nœud vers un autre nœud.

6.4.4 Sous-protocole MICP

Sur le schéma de la figure 6.5 page 127 qui représente les liens de causalité ayant permis d'établir la nécessité de fournir des sous-protocoles particuliers pour SCP/P, nous pouvons nous rendre compte que MICP est la conséquence de deux nécessités (deux flèches partant d'une case *Nécessité* pointent vers MICP sur le schéma) : pouvoir vider le réseau entre deux nœuds et fournir un mécanisme de réutilisation des MI. Nous avons précédemment évoqué la nécessité de fournir en plus un protocole de réutilisation des boites aux lettres, et nous avons remarqué à cette occasion que leur indexation sur les MI montre que ce problème est en pratique le même que celui de la réutilisation des MI. MICP est donc un protocole qui fournit trois des services dont on a besoin pour SCP/P.

Comme pour SFCP et RFCP, on va utiliser un mécanisme de *ping/pong*, qui conjointement à l'utilisation de tables de routage non adaptatives, va garantir, à la fin d'un échange, que le réseau est vide entre les deux nœuds en jeu, et dans les deux sens. Remarquons que la route suivie dans un sens peut être éventuellement distincte de la route de retour, mais cela n'a pas d'incidence sur le problème qui nous concerne.

La figure 6.7 présente le principe de fonctionnement de MICP.

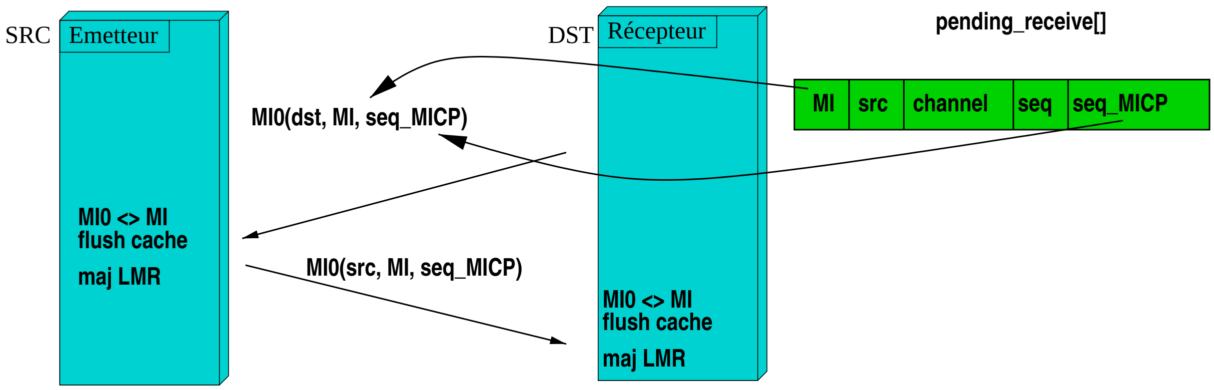


Fig. 6.7 *Protocole MICP*

Le contrôle de la machine à état constituant SCP/P est réalisé entièrement par le nœud destinataire, notamment grâce à son ordonnanceur qui donne régulièrement la main aux sous-protocoles quand ils en ont précédemment fait la requête. De la même façon, MICP est activé une première fois par un des autres sous-protocoles de SCP/P, en cas de besoin d'un des trois services explicités au début de cette section, puis à sa demande, par l'ordonnanceur, lorsqu'il a requis un délai de garde. Il utilise des messages courts avec un unique MI (MI0), et les données qu'il transporte sont décomposées en trois champs :

❶ Le premier champ contient le numéro de nœud qui émet ce message MICP ;

❷ Le deuxième champ contient le MI pour lequel MICP a été activé ;

❸ Le troisième champ contient un numéro de séquence pour cette transaction MICP.

Lorsqu'un message MICP est reçu par un nœud, ce dernier remet à zéro l'entrée de la LMR d'index le MI du deuxième champ, puis il renvoie une réponse du même type au nœud indiqué dans le premier champ, avec le même numéro de MI et le même numéro de transaction.

Lorsqu'une réponse est reçue, le nœud récepteur compare l'index reçu avec le dernier index de message MICP envoyé vers ce nœud pour ce MI. S'il s'agit du même, le récepteur a alors la garantie que le réseau est vide entre les deux nœuds. S'il ne s'agit pas du même, il s'agit donc d'un ancien message, le dernier pouvant être soi perdu, soit tout simplement retardé. Le réseau n'est donc pas forcément vide.

En cas d'expiration du délai de garde, MICP incrémente le numéro d'index et réémet une demande de réponse MICP. Ce procédé est répété jusqu'à ce que le numéro de séquence reçu ait la valeur du dernier numéro envoyé.

6.5 Optimisation des performances

Commençons par étudier le protocole SCP/P vis-à-vis de SLR/P, pendant les phases où le réseau est fiable.

On constate qu'il y a beaucoup plus de messages en transit dans le réseau avec SCP/P qu'avec SLR/P. Les messages supplémentaires sont des messages courts, ils prennent donc peu de bande passante supplémentaire sur le réseau. On pourrait imaginer qu'il en découle néanmoins une perte de performances importante, due principalement à la signalisation par interruptions qui leur est associée.

L'impact de la signalisation est réduit fortement car le plus souvent les messages courts de SFCP, RFCP et MICP suivent ou précèdent un message de données de BSCP. Par exemple, lors d'un appel à `send()`, postérieur à l'appel à `receive()` correspondant, l'émission d'un message de données de type SEND enclanche SFCP, puisque le numéro de séquence sur le canal en question est incrémenté à cette occasion. Et à chaque fois que deux messages se suivent, SCP/P ne génère qu'une seule demande de signalisation, pour le second, ce qui permet d'agréger le traitement des deux signalisations en une seule interruption.

D'autre part, lorsqu'un `send()` précède un `receive()`, SFCP est enclanché sans qu'il soit suivi d'un message de type SEND. On ne peut donc pas aggréger l'interruption du message court de SFCP avec un autre message. Néanmoins, il n'y a aucune perte de latence car les données proprement dites ne partiront de toute façon qu'après un appel à `send()`. SFCP est donc à cette occasion peu pénalisant.

La principale différence entre SCP/P et SLR/P réside dans un accroissement de latence de libération du tampon de réception. En effet, comme l'émetteur est susceptible de recevoir des demandes de retransmission de son tampon d'émission, celui-ci ne le libère qu'après que le récepteur l'ait informé de la fin de la transaction *via* le protocole RFCP. On a donc une latence de signalisation (invocation de la fonction de `callback`) chez l'émetteur qui est accrue du délai équivalent au transfert et à la signalisation d'un message-court sur le réseau.

Le réseau n'est pas toujours fiable, c'est tout l'intérêt de notre protocole, mais l'expérience nous a montré que le taux d'erreur est suffisamment faible pour que le surcoût important du protocole SCP/P en cas d'erreur (MICP rentre alors en action) soit négligeable.

6.6 Synopsis d'échanges SCP/P

La figure 6.8 page suivante présente l'enchaînement des opérations en cas d'erreur. L'action de BSCP est représentée par la case *Emission/Réception*. On constate que SFCP initie chaque transaction. Les échanges de données sont alors intercalés avec des phases MICP, jusqu'à ce que le tampon de réception accueille correctement les données qui lui sont destinées. Enfin, RFCP effectue la clôture de l'opération.

Les figures 6.9 page ci-contre et 6.10 page 134 présentent deux synopsis d'échange de données : le premier représente un échange sans faute réseau, le second représente un échange au cours duquel des données ainsi qu'un message MICP sont corrompus.

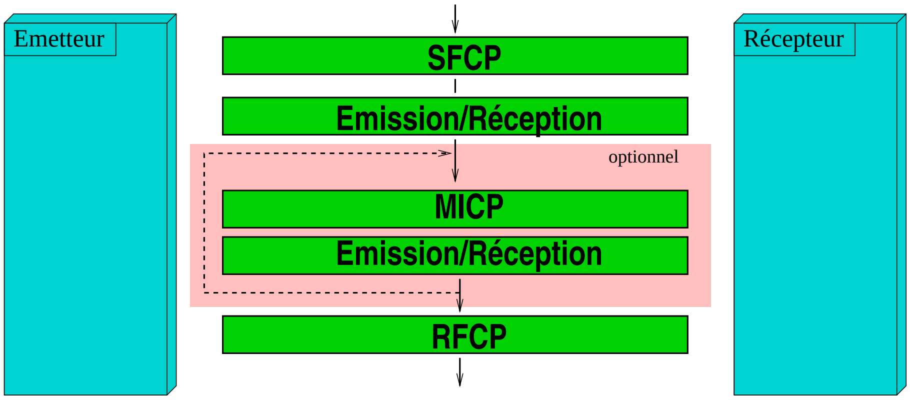


Fig. 6.8 *Enchaînement des opérations*

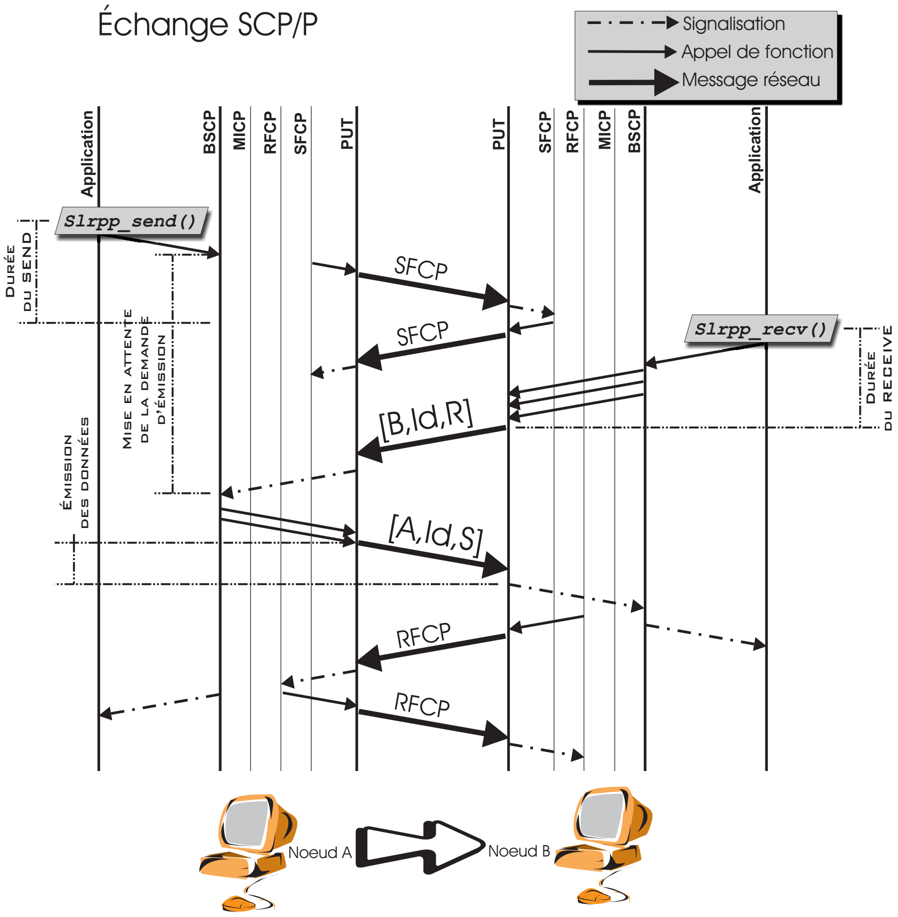


Fig. 6.9 *Échange SCP/P sans faute (et slrpp_send() avant slrpp_recv())*

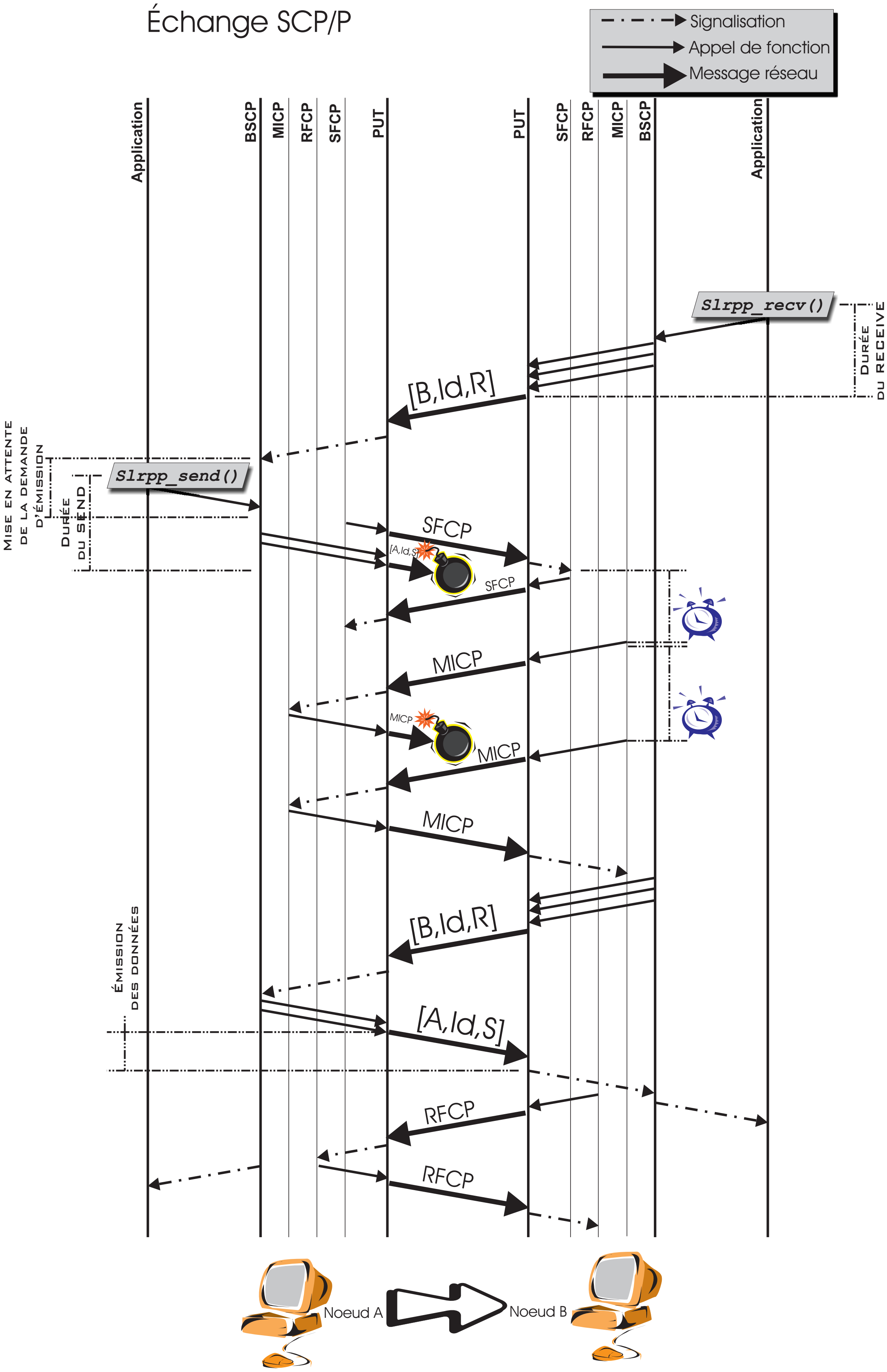


Fig. 6.10 *Échange SCP/P avec fautes (et slrpp_send() après slrpp_recv())*

6.7 Conclusion

Pour concevoir le protocole de tolérance aux fautes, on a d'abord décomposé le problème en une liste exhaustive de sous-problèmes à résoudre : perte de message RECEIVE, perte de message SEND, paquet corrompu et paquet retardé. On a alors cherché des solutions à ces sous-problèmes.

Plutôt que de trouver une solution par sous-problème, on a été amené à définir quatre protocoles distincts, qui, mis en commun, résolvent l'ensemble de nos sous-problèmes. Il s'agit des sous-protocoles de SCP/P : BSCP, MICP, RFCP et SFCP.

On peut en effet tirer de la figure 6.5 page 127 les relations entre sous-problèmes et sous-protocoles, exprimées dans le tableau suivant :

	perte de RECEIVE	perte de SEND	paquet corrompu	paquet retardé
BSCP	×	×		
RFCP		×		
SFCP	×			
MICP	×	×	×	×

Ces quatre sous-protocoles ont un fonctionnement complètement indépendant l'un de l'autre. Ils utilisent des structures de données distinctes, et le seul rapport qu'ils aient les uns avec les autres est concentré dans l'enchaînement de leurs activations respectives, présenté figure 6.8 page 133.

Nous n'avons pas effectué de preuve de protocole théorique proprement dite, mais la décomposition en sous-problèmes indépendants simples nous fournit une garantie sur l'analyse du fonctionnement de SCP/P : ayant confiance dans le bon fonctionnement de chacun des quatre sous-protocoles, on a alors une bonne confiance dans le fonctionnement du protocole dans son ensemble.

Chapitre 7

Mémoire virtuelle protégée et protocoles de haut niveau

Sommaire

La couche de communication SCP/V s'appuie sur SLR/P ou SCP/P, selon les options de compilation, pour fournir un service de communications sur canaux en adressage virtuel. Un processus, ou le noyau, peuvent alors échanger des données sans ne plus avoir à se soucier de leur localisation en mémoire physique. SCP/V permet en plus de décharger l'application de la gestion du verrouillage en RAM des espaces de mémoire virtuelle où se logent les tampons de communication. Là où SCP/P garantissait l'intégrité du contenu mémoire des différents nœuds même en cas de faute du réseau, SCP/V vient rajouter la même garantie en cas de faute de l'application. Sur SCP/V, des couches de communication haut-niveau ont pu être construites, facilitant la programmation en fournissant des points d'entrée aux comportements proches de ceux des API de communication classiques. Enfin, des bibliothèques de fonctions en espace utilisateur permettent d'utiliser le réseau HSL directement depuis un processus en mode utilisateur.

7.1 Le gestionnaire de mémoire virtuelle de MACH

7.1.1 Les structures de données de MACH

Le gestionnaire de mémoire virtuelle de FreeBSD est basé sur celui de BSD4.4, issu à son tour de celui du micro-noyau MACH [Tanenbaum, 1995].

Il utilise deux types d'objets pour maintenir l'état des différents espaces virtuels et zones physiques ou espace disque associées : des objets (structures de données et opérations associées) globaux, indépendants des processus et de leur espace mémoire, ainsi que des objets associés chacun à un seul processus, ou au noyau.

La figure 7.1 page ci-contre montre la structure de donnée associée à un processus, ainsi que les structures de données globales qu'elle référence.

7.1.2 Les objets mémoire propres aux processus

La structure de donnée fondamentale pour décrire un processus est la structure *proc*. Celle-ci pointe vers un objet de type *vmspace*, qui référence les structures de données de mémoire physique ou logique associées à un espace de mémoire virtuelle. Son champ *vm_pmap* référence des informations sur les pages physiques associées à cet espace (SCP/P ne travaille pas sur cette partie des structures de données). Le champ *vm_map* est la tête d'une liste doublement chaînée d'entrées *vm_map_entry*, chacune décrivant une zone contiguë allouée à cet espace. Évidemment, les différentes structures *vm_map_entry* référencent des zones non superposées, et leur conjonction forme la carte d'adressage de l'espace de mémoire virtuelle *vmspace*. Quand il s'agit d'un *vmspace* associé à un processus, on va trouver des instances *vm_map_entry* pour le tas, la pile, les données initialisées, le code, etc. Pour chacune de ces entités, on aura éventuellement plusieurs entrées *vm_map_entry*.

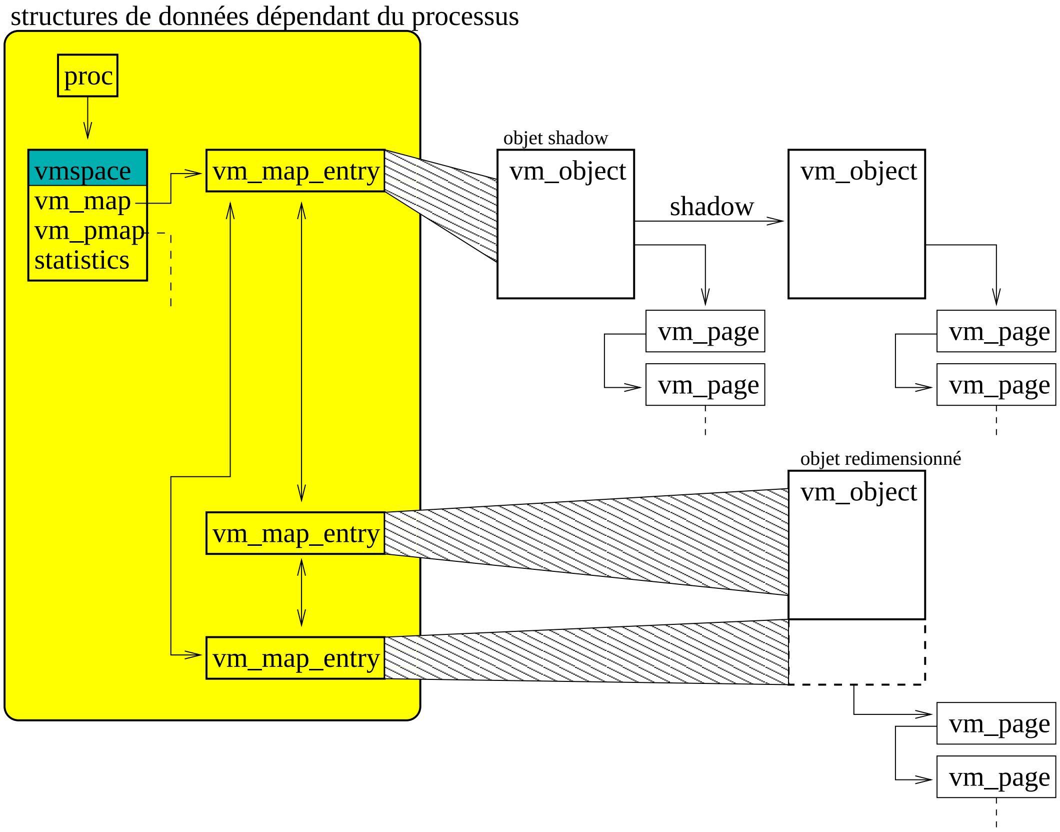


Fig. 7.1 *Exemple de structures de données MACH*

7.1.3 Les objets mémoire globaux

Chaque *vm_map_entry* appartient à un unique processus. Son rôle est d'associer une zone d'un espace virtuel (par exemple celui d'un processus, ou du noyau) à une *partie contiguë* d'un objet mémoire (*vm_object*). Plusieurs processus pouvant se partager une même zone de mémoire, l'objet mémoire est donc une entité qui a une existence globale, et qui peut ainsi être référencé par plusieurs processus par l'intermédiaire de plusieurs *vm_map_entry*.

Chaque objet mémoire représente des données présentes en mémoire physique, sur le disque (*swap*), ou bien même des données absentes : un objet peut représenter une zone de mémoire vide, contenant uniquement des zéro, et dans ce cas il ne sera associé à aucune page tant qu'aucun octet n'y sera modifié.

Lorsque les données d'un objet mémoire sont présentes en mémoire physique, elles sont localisées par des structures *vm_page*.

Il est important de noter que deux processus peuvent se partager un même objet mémoire en le référençant par des objets *vm_map_entry* de leurs carte de mémoire virtuelle. Ainsi, les mêmes données pourront être vues à des adresses distinctes par des processus distincts. A l'inverse, un objet *vm_page* n'est référencé que par un seul objet mémoire, ce qui traduit le fait qu'un objet mémoire constitue une entité de mémorisation unique.

Il existe plusieurs types d'objets mémoire, par exemple les objets de type *shadow* et les objets de type *copy*. Ils sont chaînés entre eux. Ce procédé permet de définir une gestion fine du partage des blocs mémoire entre tâches.

Par exemple, pour permettre à un processus X de visualiser une zone de données initialement créée par un autre processus Y, tout en préservant ce dernier des modifications du contenu de la zone par X, on va intercaler un objet mémoire de type *shadow* entre l'objet *vm_map_entry* de X et l'objet mémoire initial. A chaque page modifiée par le processus X, on commencera par en faire une copie dans l'objet *shadow* afin qu'elle soit modifiée dans cet objet et non pas dans l'objet initial. Le processus X voit alors sa mémoire comme un empilement d'objets. Pour une adresse virtuelle donnée, c'est la page de l'objet *shadow* qui sera accessible si elle existe (c'est-à-dire si X y a fait une modification), sinon c'est l'objet initial qui sera visé.

On peut construire selon ce principe des chaînes complexes afin d'obtenir de nombreuses sortes de partage. On en trouve quelques exemples dans [McKusick *et al.*, 1996].

Notons enfin qu'un objet est automatiquement détruit lorsqu'il n'est plus référencé par aucun autre objet.

7.2 La solution traditionnelle

Après une analyse du problème de conservation des tampons de données utilisateur en mémoire physique, section 2.9 page 54, on a classifié les conséquences éventuelles d'une faute à ce niveau en deux catégories : perte de confidentialité et perte d'intégrité.

Les interfaces de communication basées sur un matériel qui offre une primitive de type DMA utilisent le plus souvent une solution simple à implémenter mais peu satisfaisante. Elle consiste à invoquer les appels système `mlock()` et `munlock()` afin de rapatrier toutes les données des tampons de communication vers la mémoire physique, et d'en interdire la migration.

Le fonctionnement de `mlock()` est le suivant : il instancie des *vm_page* (donc des pages physiques), pour recouvrir les *vm_object* participant à la plage à verrouiller, puis il verrouille ces *vm_page*.

Mais ces tampons de mémoire physique peuvent être libérés alors qu'un transfert est encore en cours. En effet, un objet mémoire disparaît lorsque tous les processus qui le référencent *via* leurs structures *vm_map_entry* ont disparu.

Ainsi, que l'interface de programmation se charge d'invoquer `mlock()` ou qu'elle délègue cette tâche à l'application, on pourra néanmoins être confronté à des cas d'instabilité tels que ceux décrits section 2.9 page 54. Pour n'en citer qu'un, lorsqu'un processus en cours de réception quitte le système car il a effectué une instruction illégale, les données en cours de dépôt par PCI-DDC vont alors être écrites dans une zone physique potentiellement réassociée à un autre processus.

7.3 Garantie d'intégrité et de confidentialité

7.3.1 La carte de protection virtuelle

SCP/P propose aux applications le mécanisme traditionnel dont on vient de parler, ainsi qu'un mécanisme plus robuste qui protège la mémoire dans tous les cas de faute de l'application. Ce mécanisme est implémenté en tant que sous-module de SCP/V : le sous-module VMR[1].

SCP/V propose donc au programmeur deux mécanismes de communication à travers deux couples de fonctions : `send()` et `receive()` pour les communications avec mécanisme de protection traditionnel (`mlock/munlock`), et `send_prot()` et `recv_prot()` pour les communications avec prise en charge par SCP/V du mécanisme de garantie d'intégrité. Il n'y a donc pas besoin d'appel préalable pour spécifiquement demander à verrouiller les données, cette opération est implicite quand on utilise les deux appels mettant en œuvre les garanties supplémentaires.

On introduit pour cela une carte de protection virtuelle constituée d'un objet *vmspace* et de la liste chaînée des *vm_map_entry* associée. Jusqu'à présent, les objets *vmspace* étaient attribués au noyau et aux processus. Cette carte de protection ajoutée au noyau FreeBSD est allouée par et pour SCP/V, et son rôle est de référencer les objets dont une partie est en cours de transfert, afin de les protéger.

Au début d'une émission ou d'une réception, SCP/V commence par verrouiller les données, mais ne s'arrête pas là. Pour chaque objet mémoire dont une partie est concernée par le transfert demandé, SCP/V cherche une zone virtuelle libre dans la carte de protection et y insère un objet *vm_map_entry* pointant vers l'objet mémoire en question, comme on peut le constater dans l'exemple de la figure 7.2. A la fin du transfert, SCP/V demande à MACH de retirer le référencement en question.

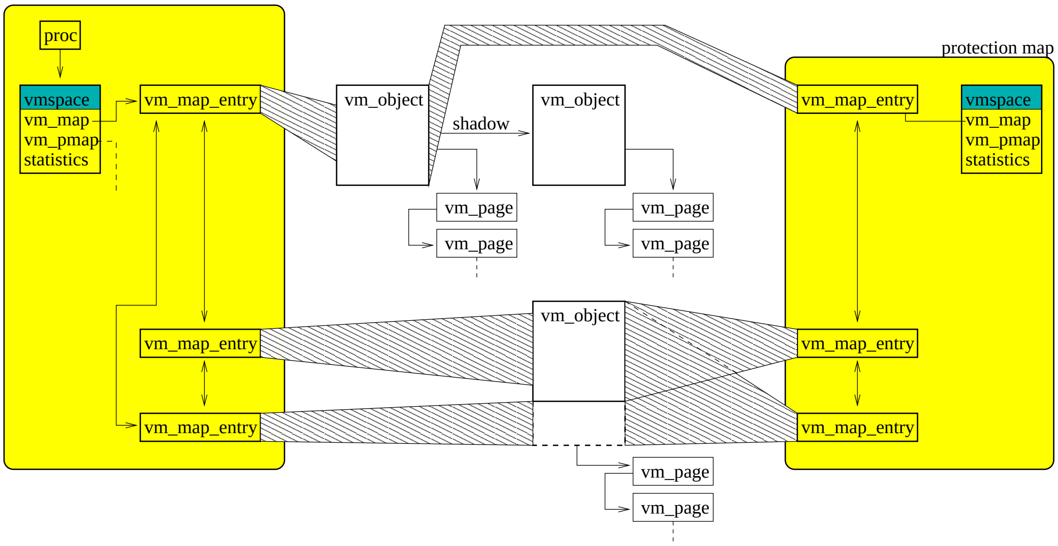


Fig. 7.2 *Référencement dans la carte de protection*

1. VMR : Virtual Memory Referencer

Ainsi, un objet mémoire reste verrouillé tant qu'un transfert est en cours vers certaines de ses pages physiques, quoiqu'il arrive aux processus à l'origine du transfert.

7.3.2 Optimisation des performances : ramasse-miette

L'opération de référencement est intrinsèquement lourde, car il y a potentiellement des pages évincées sur disque à rapatrier en mémoire centrale, ce qui se fera à la vitesse du disque, c'est-à-dire pour un temps d'accès grand par rapport aux performances des couches de communication MPC. On veut donc éviter ce surcoût à tout prix. Pour cela, on utilise un mécanisme de ramasse-miette : les objets ne sont pas verrouillés et référencés à chaque appel à `send()` ou `receive()`.

On utilise au contraire le mécanisme de ramasse-miette qui suit : à chaque demande d'émission ou de réception, on vérifie avant tout verrouillage ou référence que l'objet en question n'en a pas déjà profité. On n'effectue ces deux opérations de protection qu'au cas ou elles n'ont pas déjà eu lieu.

C'est aussi pour une raison de performance qu'on verrouille et référence la globalité d'un objet mémoire quand une partie seulement entre en jeu dans un échange. On suppose en effet que si une partie d'un objet mémoire entre en jeu dans un échange, il y a de fortes chances qu'un autre zone de cet objet soit aussi dédiée à l'échange de données.

Régulièrement, SCP/V libère les objets qui ne sont plus eu jeu dans aucun transfert. Il détruit pour l'occasion les objets *vm_map_entry* associés.

7.3.3 Nouvelles structures de données

Pour gérer le mécanisme de ramasse-miette que nous venons de décrire, SCP/V a besoin de déterminer, pour chaque objet *vm_map_entry* de la carte de protection mémoire, si l'échange SEND/RECEIVE pour lequel il a été créé est terminé. On va devoir pour cela compléter légèrement les structures de données de SCP/P.

On a vu, lors de la description des structures de données de SLR/P, que pour toute transaction en cours, il y a une entrée qui la représente dans les tables *pending_send[]* et *pending_receive[]*.

On a donc ajouté à chacune des entrées de ces deux tables une structure permettant de stocker l'ensemble des objets *vm_map_entry* qui ont été créés pour protéger les données associées à cette transaction sur un canal.

Pour implémenter le mécanisme de ramasse-miette, on se contente donc de parcourir tous les objets *vm_map_entry* de la carte de protection, et pour chacun d'eux on détermine s'il est référencé par une entrée de *pending_send[]* ou *pending_receive[]*. Si ce n'est pas le cas, on détruit alors l'objet *vm_map_entry*, ce qui a pour effet de décrémenter le compte-référence de verrouillage de l'objet mémoire vers lequel il pointe.

La figure 7.3 page ci-contre présente les structures de données permettant de mettre en jeu le mécanisme de ramasse-miette.

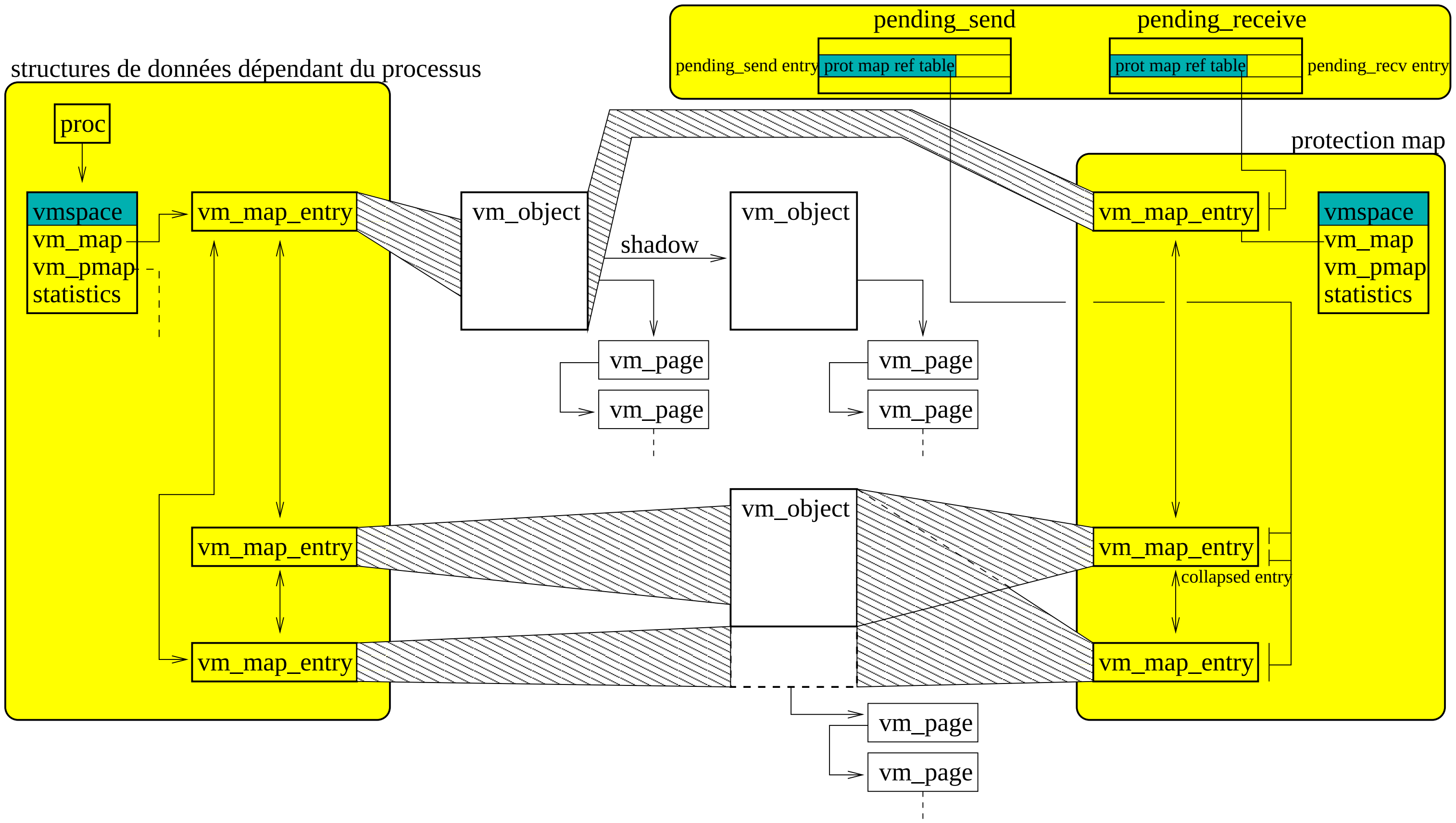


Fig. 7.3 *Mécanisme de ramasse-miette*

7.4 Permissions

L'interface SCP est accessible depuis le mode noyau. Elle ne peut donc être invoquée par un processus utilisateur qu'à travers un appel système, ou directement par le noyau dans le cadre d'une utilisation, par exemple, par un processus système.

Dans le cas d'un appel système, les tampons de communication sont décrits par leurs adresses virtuelles, passées en paramètres de l'appel système. Celui-ci les fournit à SCP/V, qui en déduit les adresses physiques équivalentes. SCP/V les fournit donc à PUT, et PCI-DDC va alors directement y chercher ou y déposer des données.

Le gestionnaire matériel de la pagination (MMU), dans un PC, fait partie intégrante du processeur. Lorsque PCI-DDC dialogue avec la mémoire locale, il n'y a donc pas de vérification matérielle des droits d'accès aux données, puisque la MMU n'est pas sur le chemin de données entre PCI-DDC et la mémoire.

Il nous faut donc rajouter un support logiciel pour cette vérification. Pour cela, SCP/V vérifie avant tout transfert que les droits du processus demandeur (droit d'écriture pour un tampon de réception, droit de lecture pour un tampon d'émission) sont compatibles avec l'opération demandée. Pour effectuer cette vérification rapidement, SCP/V n'utilise pas la méthode traditionnelle et portable qui consiste à examiner la structure *pmap* de MACH associée au processus concerné, mais il va directement analyser le contenu des tables de pages de la MMU.

7.5 Déréférencement virtuel/physique

L'opération de déréférencement virtuel/physique est effectuée par le sous-module V2P (Virtual To Physical Addresses), qui doit être extrêmement rapide, pour ne pas perdre le bénéfice des efforts d'optimisation de SCP/P et de PUT. V2P va donc lui-aussi directement consulter les entrées des tables de MMU.

7.6 Protocole SCP/V

La figure 7.4 page suivante représente le déroulement d'une opération d'émission ou de réception au niveau SCP/V. Il y a cinq phases successives lors d'un appel à la fonction d'émission ou de réception de SCP/V :

❶ À l'aide de la carte d'adressage physique (tables de la MMU) et de la carte d'adressage virtuel de MACH, le sous-module V2P détermine les différents morceaux contigus de la mémoire physique constituant le tampon de communication ;

❷ A partir de la carte d'adressage virtuel de MACH, le sous-module VMR force le rapatriement en mémoire physique des objets mémoire concernés par le transfert, puis il les référence dans la carte de protection ; il conserve une liste des adresses des nouvelles structures *vm_map_entry* venant d'être créées, en vue de son stockage à l'étape ❹ page ci-contre ;

❸ Connaissant les emplacements physiques, SCP/V appelle la fonction d'émission ou de réception de SCP/P. Plus précisément, c'est une version légèrement modifiée de la fonction d'émission ou de réception de SCP/P, qui à la différence de celle qu'on a étudiée précédemment, fournit en plus, en sortie, l'adresse de l'entrée de la table `pending_send[]` ou `pending_receive[]` qui vient d'être allouée à l'opération. De plus, l'entrée de cette table est verrouillée, ce qui permet de garantir qu'elle ne sera pas effacée avant la fin de l'étape qui suit. Si le transfert se termine avant la fin de l'étape en question, la libération de l'entrée verrouillée sera tout simplement retardée jusqu'au déverrouillage, c'est-à-dire jusqu'à la cinquième opération de SCP/V ;

❹ L'adresse qui vient d'être fournie permet de copier, dans l'entrée de table correspondante, la liste des adresses des structures *vm_map_entry* créées à l'étape ❷ page précédente. Cette copie dans la table `pending_send[]` (ou `pending_receive[]` selon l'opération effectuée) sera utilisée au moment de l'opération de ramasse-miette ;

❺ L'entrée de la table `pending_send[]` ou `pending_receive[]` qui avait été verrouillée en ❸ est déverrouillée.

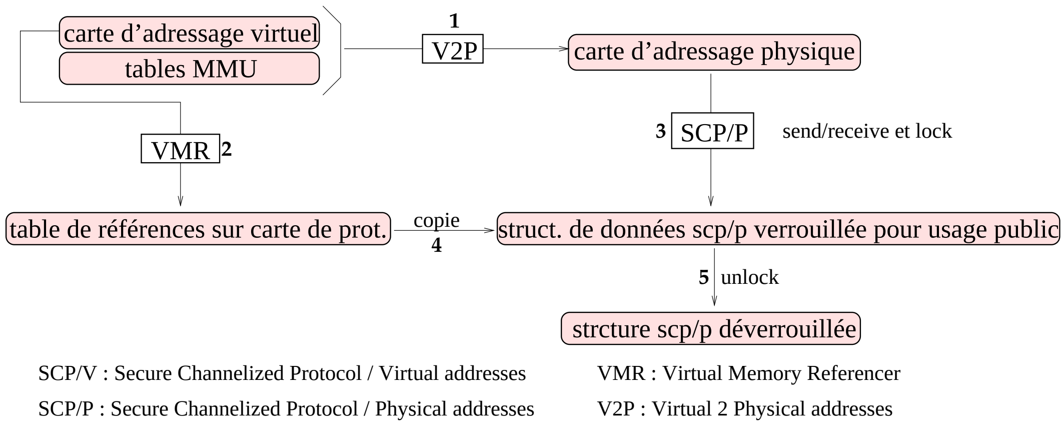


Fig. 7.4 *Protocole SCP/V*

7.7 Récupération de la taille des données tronquées

On a présenté section 5.2.4 page 102 un moyen de récupérer la taille des données tronquées dans une transaction SLR/P. On s'appuyait sur la possibilité de fournir un tampon d'émission constitué de deux zones discontiguës. Pour faire de même avec SCP/V, les fonctions d'émission et de réception acceptent en paramètres deux zones virtuelles discontiguës.

7.8 Les couches noyau supérieures

7.8.1 Empilement

La figure 7.5 présente l'empilement des couches noyau propres à MPC-OS, dans leur totalité. On trouve, au dessus de SCP/V, les couches MDCP et SELECT dont le but est de fournir des services de communication à travers des API proposant un comportement Unix traditionnel.

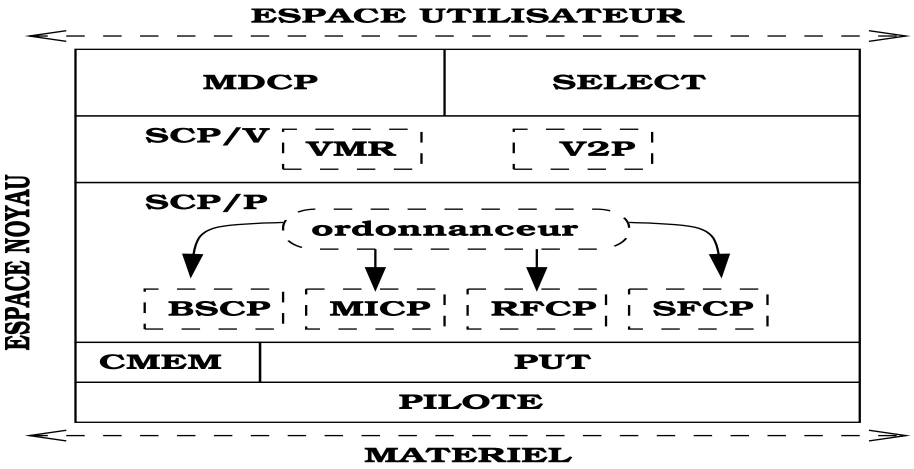


Fig. 7.5 *Protocoles noyau*

Notons que l'ensemble des couches de communication de MPC-OS implantées dans le noyau FreeBSD sont accessibles depuis le mode utilisateur. Les processus ne sont pas obligés de faire appel à l'API de plus haut niveau de cet empilement, ils peuvent accéder à n'importe quelle API intermédiaire. Pour cela, il faut construire un appel système permettant de rentrer dans le noyau pour y effectuer l'invocation des points d'entrée désirés. Un exemple de module à chargement dynamique est fourni dans ce but dans la distribution source de MPC-OS. Il suffit donc au programmeur d'utiliser ce canevas pour construire un nouveau module noyau fournissant aux programmes en mode utilisateur l'accès aux fonctions désirées de MPC-OS.

7.8.2 Les canaux MDCP

La couche SCP/V fournit des communications sur canaux en adressage virtuel, sans copie de tampons. Elle possède néanmoins trois caractéristiques non conventionnelles :

- ✓ Les données sont transportées sur des canaux virtuels, mais à chaque transaction les couples `send()/receive()` sont appariés : les données non transmises au cours d'une transaction, pour cause de tampons d'émission et de réception de tailles distinctes, ne sont pas remises en jeu dans la transaction suivante ;
- ✓ Les points d'entrée sont non bloquants. On peut les transformer en appels bloquants comme on l'a montré, mais il n'y a toujours pas de copie locale des tampons de communication, et en conséquence, un `send()` bloquant n'est libéré qu'à la fin de la

transaction. Le tampon d'émission n'est donc libéré qu'une fois que le récepteur a effectué un `receive()`. Cela crée une influence inhabituelle du processus récepteur sur le comportement du processus émetteur, et peut aboutir à un interblocage qui n'existerait pas avec un protocole traditionnel. En effet, un `send()` bloquant, avec un protocole traditionnel, rend la main avant la demande de réception, au prix d'une copie locale des données à émettre vers l'espace du noyau ;

✓ L'utilisation de fonctions de *callback* invoquées dans un contexte asynchrone est nécessaire pour gérer la signalisation.

Pour proposer une interface sur canaux ayant un comportement plus familier, on a choisi de construire, au dessus de SCP/V, la couche de communication MDCP [2]. Ses points d'entrée seront notés `read()/write()`.

Le principe sous-jacent à MDCP tient dans la remarque suivante : imaginons qu'on dispose d'un tampon d'émission prêt à être transmis, mais qu'on ne connaisse pas la localisation du tampon de réception prêt à l'accueillir (cas d'un `send()` sans `receive()`). Plutôt que d'attendre le `receive()`, on peut rapprocher les données de leur destinataire pour gagner de la latence. Pour cela, on envoie les données chez le récepteur (connu car il est fixé une fois pour toute pour un canal donné), dans un tampon de transit. Au moment où le `receive()` est enfin effectué, on ne fait pas de communication sur le réseau HSL, mais on va directement copier les données depuis leur tampon de transit vers leur emplacement définitif. Évidemment, cette étape intermédiaire n'a lieu que si le `receive()` se produit après le `send()`.

Un problème se pose immédiatement quand on essaie de proposer un protocole de communication appliquant ce principe : lorsque la tâche émettrice appelle `send()`, et que MDCP constate qu'aucun `receive()` précisant la position du tampon de réception n'est parvenu, il va décider d'émettre les données vers un tampon de transit distant. Mais il se peut que le message RECEIVE ait été déjà envoyé. Il va alors croiser le message SEND à destination du tampon de transit.

Pour fournir des tampons de transit, il faut réserver une partie de la mémoire, et informer les nœuds distants de leurs adresses. Pour cela, chaque canal MDCP va utiliser un canal SCP/V réservé à cet effet, et un ensemble de tampons de transit pour lesquels on effectue des `receive()`. Pour pouvoir transmettre des données sans passer par les tampons de transit (cas où le `receive()` est produit avant le `send()`), un canal MDCP est aussi constitué d'un second canal SCP/V dédié à cet effet.

Chaque canal MDCP est donc constitué de deux canaux SCP/V, le canal principal et le canal de transit ou canal tampon, représentés sur la figure 7.6 page suivante, ainsi que d'une plage de mémoire contiguë réservée auprès de CMEM pour gérer les tampons de transit.

Mais émettre des données sur deux canaux pose le problème de leur ordonnancement en réception : il faut que l'émetteur fournisse au récepteur un moyen de remettre toutes ces données en ordre, pour constituer un flux unique. Pour ce faire, l'émetteur insère un

2. MDCP : Multi Deposit Channelized Protocol

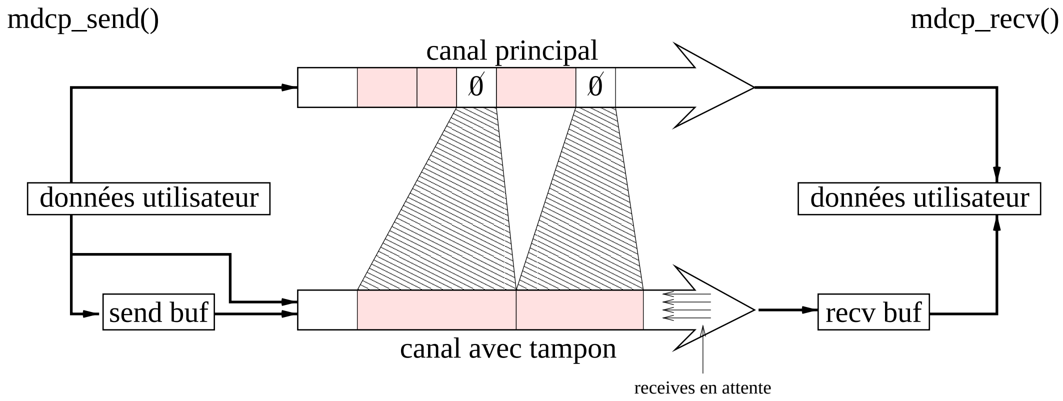


Fig. 7.6 *Protocole MDCP*

marqueur dans le flux du canal principal chaque fois qu'il émet des données dans le canal tampon, comme on peut le constater sur la figure 7.6.

Les opérations sur un canal MDCP sont alors définies de la sorte :

- *émission* : lorsque l'émetteur veut envoyer des données sur un canal MDCP, il examine le canal principal associé, et si celui-ci ne dispose pas de RECEIVE en attente, il envoie les données sur le canal de transit, et émet un message de taille nulle sur le canal principal. Pour émettre un message de taille nulle, il suffit d'utiliser le mécanisme proposé section 7.7 page 145. Le message sous-jacent contiendra deux sections, la première de 1 mot transportant la taille des données, la seconde vide. La couche SCP/P ne sera donc pas confrontée à un message de taille nulle (ce qui n'est pas prévu par son protocole), mais à un message de taille 1 mot;
- *réception* : à chaque fois que le récepteur tente de recevoir des données, il effectue une demande de réception sur le canal principal vers le tampon de réception fourni par l'application. Si les données sont de taille nulle, il sait qu'un message a été envoyé sur le canal de transit. Si aucun tampon de transit ne contient d'information, c'est qu'un message qui leur est destiné est actuellement dans le réseau, on attend alors son dépôt complet. Il suffit, une fois le tampon de transit rempli, de recopier les informations dans le tampon de l'application, puis d'émettre une nouvelle demande de réception sur le canal de transit pour le tampon de transit qu'on vient de libérer. On utilise plusieurs tampons de transit pour obtenir un mécanisme de fenêtrage, en cas d'émissions successives et rapprochées dans le temps sur un canal MDCP.

Rappelons les trois caractéristiques non conventionnelles du protocole SCP/V dont on veut s'affranchir : d'une part les données non transmises dans une transaction `send()/receive()` ne sont pas transmises dans la transaction suivante, d'autre part un `send()` bloquant n'est pas libéré tant que le récepteur n'a pas invoqué `receive()`, et enfin l'utilisation de fonctions de *callback* est impératif pour contrôler la signalisation des opérations.

Pour résoudre le premier de ces problèmes, MDCP va implémenter le mécanisme de récupération de la taille des données tronquées de la section 7.7 page 145. Cela lui permet de noter l'état d'avancement de la taille des données transmises sur un canal, et donc de pouvoir remettre en jeu les données non transmises.

Les tampons de transit sont indiqués *recv buf* sur la figure 7.6. Ils permettent de résoudre le deuxième problème : un `send()` bloquant est libéré même si le récepteur n'a pas encore invoqué de `receive()` pour ces données.

On trouve aussi des tampons en tête du canal de transit, indiqués *send buf* sur la même figure, localisés dans une plage de mémoire contiguë réservée auprès de CMEM. Leur utilisation est optionnelle et au choix de l'émetteur. Ils permettent d'éviter de de bloquer la fonction `send()` pendant le transfert à travers le réseau sur le canal transit. Les performances en sont accrues pour les petits messages : sans passer par ces tampons locaux, il faut attendre l'interruption de fin d'émission dans le réseau avant de rendre la main à l'application. Ce délai est largement plus important que le délai de copie locale d'un petit paquet de données. On bloque donc l'application beaucoup moins longtemps en passant par les tampons locaux pour les petits paquets.

Enfin, on résout le troisième problème en s'affranchissant des fonction de *callback* : les opérations `read()/write()` sur les canaux MDCP garantissent qu'à leur retour, les données auront quitté le tampon utilisateur pour se retrouver, soit dans le réseau, soit dans un tampon local, soit dans un tampon de transit distant, soit dans le tampon final de l'application réceptrice. Il n'est donc plus utile de signaler à l'application qu'elle peut réutiliser tel ou tel tampon, comme c'est le cas avec les autres protocoles noyau de MPC-OS.

En conclusion, on peut dire que ce mécanisme, conforme au comportement standard dit `read()/write()` sur canal, est réalisé de manière plus efficace que les implémentations classiques. En effet, il n'y a pas de copie systématique des données, mais au contraire, ce sont les conditions dans lesquelles se déroulent la communication qui influencent le choix de MDCP de faire ou non des copies pour coller au comportement naturel auquel s'attend un programmeur d'applications. Notamment, si les `read()` précèdent les `write()`, les données sont transportées de l'espace utilisateur de l'émetteur à celui du récepteur sans aucune copie. Et lorsqu'il y a copie, c'est un moyen de rapprocher les données de leur destination finale, en les amenant en transit dans leur nœud destinataire. La lecture, lorsque l'application l'invoquera, sera plus rapide.

L'annexe E page 247 présente les structures de données utilisées par MDCP pour gérer les états de son protocole.

7.8.3 Le service SELECT

À l'aide de MDCP, on obtient des canaux de communication possédant le comportement Unix standard. Il est temps d'aborder le problème de la synchronisation des applications sur les données.

Le modèle de synchronisation des flux de données entrant dans un processus avec les opérations qu'il effectue est basé, dans le monde Unix, principalement sur la primitive `select()`.

Toutes les sources de flux de données sont représentées à travers une structure de donnée opaque standard, désignée par un *numéro de descripteur de fichier ouvert*, et l'application peut effectuer des opérations standard ((read()/write()) s'appliquant à n'importe quel descripteur, quelque soit l'origine du flux de données (connexion réseau, système de fichiers, terminal d'entrées sorties, etc.).

La signalisation des flux de données auprès du processus communiquant est effectuée au travers de l'appel système `select()`, qui prend en paramètres un ensemble de numéros de descripteurs de fichiers ouverts, se bloque jusqu'à ce qu'une opération puisse être effectuée sur au moins l'un de ces descripteurs, et rend alors la main à l'application tout en lui permettant de connaître quels descripteurs ont été cause de cette activation du processus, et quelles sont les opérations qu'ils sont prêts à honorer. Le cas le plus courant consiste à utiliser `select()` pour effectuer des lectures sur plusieurs sources de données simultanément.

Malheureusement, aucun descripteur de fichier n'est associé aux canaux MDCP. Pour pouvoir néanmoins offrir le modèle standard de synchronisation des flux de données standard Unix et des flux de données MDCP, la couche SELECT a été développée. Elle implémente une version enrichie du classique point d'entrée `select()` d'Unix, prenant en paramètres une liste de numéros descripteurs de fichiers ouverts tout à fait standard, et une liste de canaux de communication MDCP. Le fonctionnement de SELECT est complètement standard, il permet donc de faire cohabiter au sein d'une même application des canaux de communication sur le réseau MPC avec l'ensemble des canaux de données standard du monde Unix.

7.9 Les couches en mode utilisateur

7.9.1 La bibliothèque LIBMPC

Pour tous les protocoles noyau autres que MDCP, un contrôle par l'application de l'état des tampons de communication est nécessaire, *via* des fonctions de *callback* fournies par celle-ci. Ces fonctions étant éventuellement invoquées en interruption, elles doivent impérativement se trouver implémentées au sein du noyau. C'est pour cela qu'on a indiqué précédemment qu'un module noyau d'exemple est disponible dans la distribution de MPC-OS. Même les programmeurs désireux d'utiliser ces protocoles hors noyau *via* des appels système doivent écrire la gestion des fonctions de *callback* au sein du noyau.

Pour MDCP, le problème du programmeur est beaucoup plus simple car il n'a pas à fournir de fonction de *callback*. Tout son travail de programmation peut donc se situer en mode utilisateur. Pour cela, MPC-OS fournir une bibliothèque de fonctions nommée LIBMPC[3]. Elle permet d'utiliser les canaux MDCP et les fonctionnalités des services de synchronisation SOCKET depuis le mode utilisateur.

3. La bibliothèque LIBMPC est fournie, dans la distribution de MPC-OS, sous la forme d'un fichier à *linker* avec l'application : `libmpc.a`.

7.9.2 La bibliothèque d'accès transparent SOCKETWRAP

La bibliothèque LIBMPC fournit des services standards de communication sur canaux, à l'aide d'une API propriétaire. La bibliothèque SOCKETWRAP a été développée pour proposer des services standards de communication sur canaux, à l'aide d'une API standard : le sous-ensemble des communications en mode connecté de l'API des *sockets*.

En s'appuyant sur LIBMPC, elle traduit les appels *sockets* en appels à la bibliothèque LIBMPC, uniquement quand ils concernent des communications ne faisant pas intervenir de nœud extérieur à la machine MPC. Cette bibliothèque crée pour cela des numéro de descripteur de fichiers virtuels, qu'elle associe à des canaux de communication MDCP. Elle intercepte alors tous les appels systèmes standards de l'API *socket*, et les redirige, s'ils concernent un descripteur de fichier virtuel qu'elle a attribué, vers des appels à la bibliothèque MPC avec laquelle elle a subi une édition de lien.

Pour encore plus de transparence, SOCKETWRAP est fournie dans la distribution de MPC-OS sous forme d'une librairie partagée, `libsocketwrap.so.1.0`, qu'on peut charger dynamiquement dans n'importe qu'elle application, sans aucune recompilation ni édition de lien, du moment que l'application cible a subi une édition de lien dynamique avec les librairies standard du langage C.

L'éditeur de lien dynamique fournit en effet la possibilité de charger une librairie non prévue au moment de l'édition de lien, par passage de son nom dans une variable d'environnement.

Une application standard communiquant en mode connecté à travers l'interface *sockets* voit donc son code modifié juste au moment de son chargement dans le système, de manière totalement transparente et sans qu'elle puisse l'en empêcher. La librairie SOCKETWRAP vient s'intercaler entre le code initial de l'application et les librairies permettant les appels systèmes de communication.

Ce mécanisme a été expérimenté avec succès avec une application construire sur l'interface *socket* de l'implémentation MPICH de la bibliothèque de communication parallèle MPI.

MPICH a pour cela été compilée de manière traditionnelle, avec les options de compilation générant du code invoquant TCP/IP pour les communications. L'application a été compilée elle-aussi de manière traditionnelle, sans modification, et liée à la librairie fournie par MPICH. L'éditeur de lien dynamique a été informé, par positionnement d'une variable d'environnement, de notre désir de charger SOCKETWRAP en plus des autres librairies standards, au moment du démarrage de nos tâches. Et on a pu constater que l'application distribuée utilisait effectivement le réseau HSL à travers des canaux MDCP.

Une démarche similaire récente est présentée dans [Itoh *et al.*, 2000], où les auteurs indiquent comment ils ont implémenté l'interface *socket* au dessus de VIA, et comment ils l'ont validée sur l'architecture matérielle Synfinity-2 [Ito, 2000].

7.10 Portabilité de MPC-OS

7.10.1 Choix de FreeBSD

FreeBSD a été choisi en 1995 pour supporter le développement de MPC-OS. Il nous fallait choisir un système d'exploitation dont les sources du noyau étaient libres et bien documentées. On pouvait alors trouver de nombreux écrits sur FreeBSD car il héritait de la lignée du système Unix qui avait débuté quinze ans auparavant.

À cette époque, le système Linux, autre alternative possible, existait depuis moins longtemps, et on trouvait peu d'informations sur la structure et le fonctionnement de son noyau.

Le matériel spécifique à MPC permet des échanges DMA entre nœuds, décrits en adressage physique. Les processus communiquant qui doivent pouvoir bénéficier de ces échanges travaillent quant-à-eux dans un espace protégé de mémoire virtuelle. MPC-OS doit donc pouvoir utiliser les services du système d'exploitation afin d'acquérir des informations sur l'état des espaces mémoire des processus, éventuellement manipuler leurs structures, et se voir proposer un accès simple aux tables de la MMU. Là aussi, le choix de FreeBSD a été motivé par sa capacité à mettre en œuvre de tels mécanismes : il utilise pour cela le gestionnaire de mémoire virtuelle du micro-noyau MACH.

7.10.2 Contraintes de portabilité logicielles

Le succès du système Linux depuis quelques années nous a incités à étudier le problème de la portabilité de MPC-OS.

De ce point de vue, on peut distinguer dans MPC-OS trois types de composants :

- ✓ Les processus et bibliothèques en mode utilisateur (le manager local, hslclient, hslserver, LIBMPC et LIBSOCKETWRAP) : ces composants sont facilement portables vers n'importe quel système Unix, car ils utilisent principalement les services systèmes à travers l'interface standard POSIX (notamment au niveau de l'implémentation *multi-thread* du manager local) ;
- ✓ Le pilote de périphérique et la couche PUT : la réécriture du pilote de périphérique est propre à l'OS sur lequel on destine le portage. Il s'agit d'une opération classique et sans grande difficulté, de même que pour le portage de la couche PUT, qui nécessite uniquement de réécrire l'interface avec l'API d'accès PCI ;
- ✓ Les couches de communication sur canaux virtuels et le module CMEM : CMEM se substitue au système d'exploitation pour allouer aux applications une partie de la mémoire physique. Ces zones doivent néanmoins pouvoir se retrouver dans les espaces d'adressage virtuel gérés par le système. Il y a donc ici un problème non trivial à résoudre, complètement dépendant du système vers lequel on souhaite porter FreeBSD. Sachant que l'ensemble des couches de communication noyau utilise CMEM, on ne peut en outre pas se passer du portage de ce composant.
  Un problème analogue se pose pour le portage des couches de communication sur canaux virtuels : le verrouillage automatique des tampons en cours de transfert est très

fortement lié aux structures de données représentant les espaces virtuels, et à l'API permettant de les manipuler. Pour cette raison, le portage nécessite une réécriture totale d'une partie assez complexe du code de ces couches de communication.

7.10.3 Contraintes de portabilité matérielles

L'architecture matérielle de gestion mémoire des processeurs Intel est basée sur une double translation : segmentation, puis pagination. Quand la segmentation est activée, le contenu d'un registre d'adresse doit être associé à un numéro de segment au travers d'un registre de segment, afin de désigner de manière unique une adresse physique. Deux registres d'adresse peuvent donc contenir une même adresse virtuelle, mais être associés à des segments distincts (segment de code, segment de données, etc.) et donc désigner des emplacements différents.

En principe, seule la partie matérielle de la MMU effectue cette double translation, et il n'y a alors pas d'ambiguïté possible, puisque la désignation du registre de segment associé est incluse dans le code opérande de chaque instruction. Dans le cadre de MPC-OS, les couches de protocoles sur canaux virtuels doivent aussi effectuer cette translation, car pour effectuer une émission ou une réception, seules des adresses virtuelles sont fournies à leurs points d'entrée. Ces couches doivent alors effectuer les transferts en invoquant PUT, qui accepte uniquement des adresses physique.

On constate donc que si le système d'exploitation sur lequel on veut porter MPC-OS utilise le mécanisme de segmentation, on doit alors redéfinir l'API des couches de communication sur canaux virtuels, pour inclure la désignation d'un segment associé à chaque adresse virtuelle passée en paramètre.

On a rencontré cette difficulté avec Linux, dont certaines versions du noyau utilisent la segmentation de la façon suivante : lorsque le système s'exécute en mode noyau, deux segments sont utilisés, l'un pour désigner l'espace d'adressage virtuel du processus actif, et l'autre pour désigner l'espace du noyau. Quand par exemple `slrpp_send()` est invoqué avec une adresse virtuelle de tampon local en paramètre, SLR/P ne sait pas si cette adresse désigne des données du noyau ou des données du processus actif. Si ce dernier est à l'origine de cette invocation de SLR/P, il faut utiliser le segment utilisateur, mais si elle provient par exemple du système de pagination distribué MAÎS [Cadinot *et al.*, 1997], il faut au contraire utilise le segment noyau.

Pour cette raison, on a attendu la réalisation récente d'une version du noyau Linux pour processeur Intel, expurgée du mécanisme de segmentation, pour porter MPC-OS vers Linux.

7.10.4 Portage vers Linux

MPC-OS a été porté en partie sous Linux, par un stagiaire du DEA ASIME[4]. Seuls CMEM, PUT et les daemons hslclient/hslserver ont été portés sous une version du noyau

4. DEA ASIME : Architecture des systèmes intégrés et micro-électronique

linux n'utilisant pas la segmentation. La difficulté majeure a consisté à réécrire le cœur de CMEM afin qu'il cohabite avec le gestionnaire de mémoire de Linux.

Un stage ingénieur de l'INT d'Evry a permis de concrétiser ce portage et d'effectuer des tests de performances comparés entre FreeBSD et Linux, qui ont abouti à des résultats analogues.

7.11 Conclusion

Nous avons au cours des chapitres précédents étudié les protocoles noyau qui composent MPC-OS. Dans ce chapitre, on s'est attaché à décrire les protocoles noyau de plus haut niveau, et leur interface depuis le mode utilisateur, au travers de bibliothèques de programmation.

Au fur et à mesure de l'étude de ces protocoles, on a pris le parti de supposer que les ressources utilisées (canaux de communication, tampons de transit, etc.) étaient toujours allouées de manière statique.

Cette démarche était justifiée car nous voulions aboutir à des protocoles simples, à la fois pour obtenir les meilleures performances, et car ces protocoles étaient destinés à être implémentés au sein d'un noyau Unix, opération toujours délicate.

Maintenant que l'on a terminé l'étude des protocoles noyau, on va s'attacher dans le chapitre qui suit à étudier comment, depuis le mode utilisateur, on a intégré dans MPC-OS une gestion dynamique des ressources.

Chapitre 8

Gestion dynamique des ressources

Sommaire

Nous avons travaillé jusqu'à maintenant avec des canaux de communication alloués de manière statique. Dans les protocoles que l'on a décrits, aucun mécanisme de création dynamique, ni de fermeture contrôlée des canaux n'est prévue. Ces opérations sont complexes et se produisent peu souvent. C'est pourquoi elles sont intégrées au sein d'une application distribuée de gestion dynamique des ressources, dont un représentant est présent sur chaque nœud.

8.1 Le Manager local

8.1.1 Le Manager local et les autres entités de MPC-OS

Sur la figure 4.4 page 83 que l'on a déjà commentée, on distingue le processus gestionnaire des ressources présent sur chaque nœud de calcul, appelé Manager local.

Sa conception a été guidée par trois objectifs :

❶ Le Manager local est responsable de l'activation des tâches sur les différents nœuds de calcul ;

❷ Il doit gérer le protocole de rendez-vous permettant à deux tâches de se synchroniser lors de la création d'un nouveau canal de communication SLR/P, SCP/P, SCP/V ou MDCP ;

❸ Il doit implémenter en plus, à cette occasion, les deux protocoles respectifs d'attribution et de terminaison d'un canal.

Dans le cadre de ces trois objectifs, le Manager local noue des liens privilégiés avec d'autres entités locales ou distantes. Ils sont représentés sur la figure 4.4 et concernent :

✓ les liens avec l'application : au travers une *socket* Unix, `/tmp/mpc`, pour laquelle il joue le rôle de serveur, les applications communiquent avec le Manager local pour invoquer ses services. La bibliothèque LIBMPC gère le protocole de communication sur cette *socket* ;

✓ les liens avec le noyau : le Manager local est chargé, à la demande des applications, de créer ou clore dynamiquement des canaux de communication. Dans ce cadre, pour manipuler les ressources et tables noyau qui maintiennent l'état des protocoles SLR/P, SCP/P, SCP/V et MDCP, il doit disposer d'un accès à des point d'entrée de ces protocoles. Il utilise pour cela des appels systèmes dont l'utilisation lui est réservée ;

✓ les communications avec les autres Manager locaux : l'application Manager est une application distribuée, organisée en représentants locaux, un par nœud de calcul, qui coopèrent entre eux en mettant en œuvre des algorithmes distribués. Pour ce faire, ils dialoguent tous les uns avec les autres.

8.1.2 Les relations inter-Manager

L'architecture distribuée de MPC-OS est constituée de quatre niveaux de domaines imbriqués les uns dans les autres :

❶ Le processeur sur lequel une tâche s'exécute de manière séquentielle constitue le plus bas niveau d'imbrication ;

❷ Le nœud de calcul forme le deuxième niveau : il rassemble plusieurs processeurs, une mémoire et des périphériques locaux sous une architecture SMP ;

❸ Le cluster MPC, rassemblement de nœuds de calcul connectés *via* un réseau de contrôle et un réseau HSL, constitue le troisième niveau d'imbrication. Un modèle de communication hybride est mis en place au sein du cluster de SMP : il réunit des communications par passage de messages entre les différents nœuds, et des communications par mémoire partagée au sein d'un même nœud (les performances de MPI dans une telle configuration hybride ont été examinées dans [Cappello *et al.*, 2001]) ;

❹ La Machine MPC, organisée en configuration multi-sites, constitue le plus haut niveau d'imbrication. Elle rassemble des clusters de SMP situés sur des sites distincts et communiquant avec les protocoles de la famille TCP/IP sur un réseau WAN.

La figure 8.1 page suivante présente les communications entre Manager locaux sur une configuration multi-sites.

On peut y constater que tous les Manager locaux ne sont pas reliés directement entre eux. Le graphe qui les relie est néanmoins connexe, étudions donc sa structure.

Les Manager d'un même cluster mettent en place des canaux de communication MDCP suivant un graphe complet[1], ce qui leur permet de disposer de moyens de communication directs. Il n'est pas envisageable, pour des raisons techniques évidentes, d'avoir une configuration multi-sites suivant un graphe complet. Pour permettre néanmoins les communications distantes, un nœud, nommé *nœud inter-cluster*, est élu dans chaque cluster pour constituer la passerelle avec le monde extérieur : le Manager présent sur ce nœud est chargé d'acheminer les messages qui ne lui sont pas destinés vers son homologue du cluster destination, ce dernier acheminant le message vers le Manager du nœud destinataire. Les Manager des nœuds inter-cluster se connectent pour cela suivant un graphe complet à travers le réseau WAN.

Les Manager effectuent donc un travail de routage permettant d'acheminer une requête de n'importe quel nœud à n'importe quel autre nœud en effectuant au plus trois sauts.

Cette conception multi-sites a pour ambition de permettre l'utilisation d'un cluster sur site distant sous-utilisé, depuis une machine d'un site local. Les communications inter-cluster sont lentes, et on ne cherche pas à répartir une application entre plusieurs clusters, pour des questions de performance. Par contre, pouvoir activer les tâches d'une application sur

1. Les canaux de communication entre les Manager d'un même cluster peuvent être de type MDCP, ou de type TCP. Sachant que les performances ne sont pas requises pour ces communications, on a préféré dans l'implémentation actuelle choisir les communications TCP, car le Manager est pour l'instant à l'état de prototype.

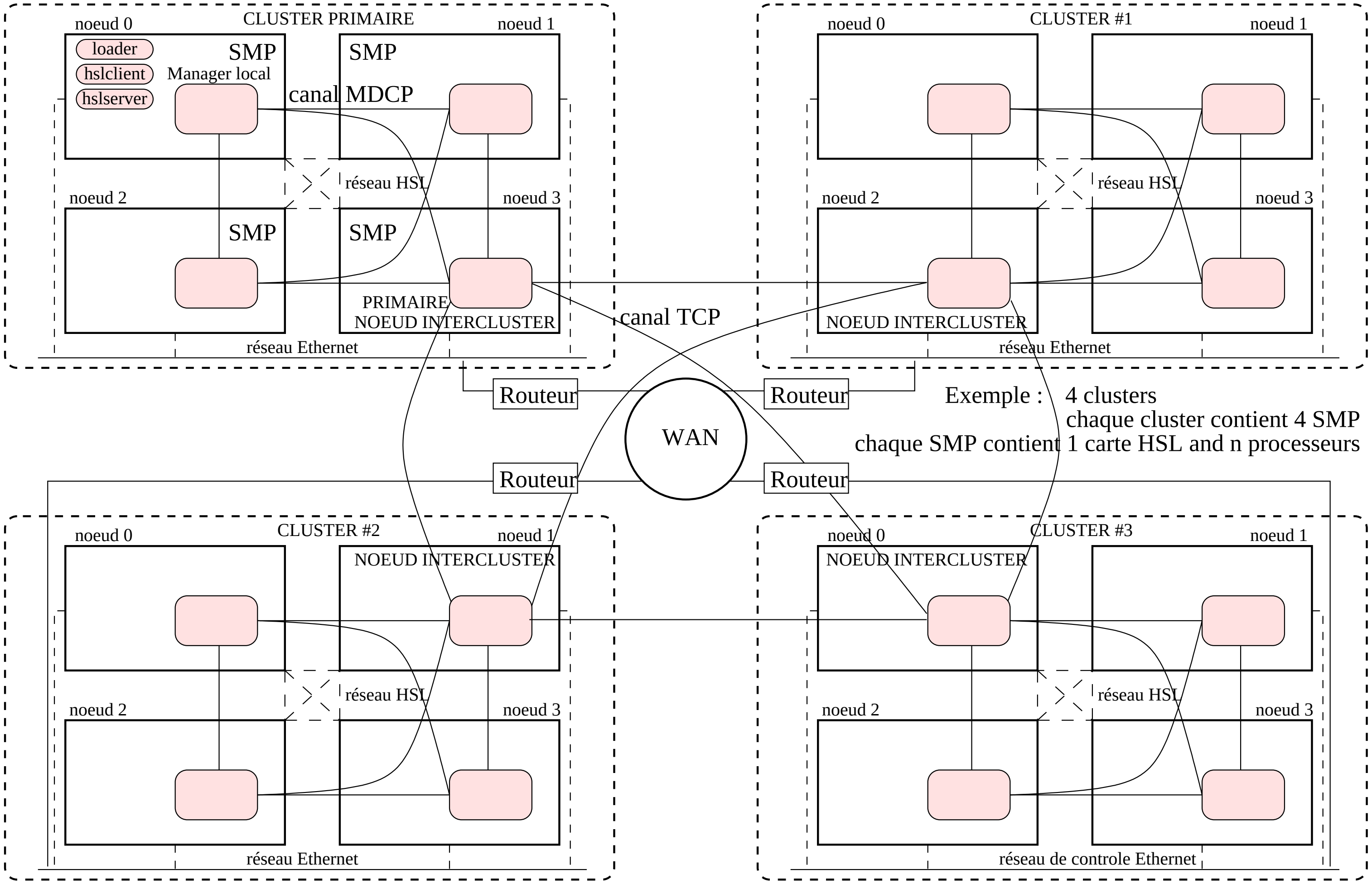


Fig. 8.1 *Une machine MPC en configuration multi-sites*

un site distant, depuis une tâche sur le site local permet d'automatiser l'utilisation de sites distants peu chargés.

Signalons que la bibliothèque de communication Madeleine II [Aumage *et al.*, 2000] propose un modèle permettant d'interconnecter des grappes avec de bonnes performances au niveau des communications inter-grappes, à l'aide de technologie performantes d'interconnexion des grappes telles que SCI.

Pour conclure sur la problématique multi-sites, on peut signaler que si les Manager ont été dotés de la possibilité de dialoguer entre différents sites, les tâches constituant une application parallèle dialoguent entre elles *via* le réseau HSL, et sont donc en principe incapables d'utiliser les couches de communication PUT, SLR/P, SCP/P, SCP/V ou MDCP entre différents sites. C'est pourtant possible, car la primitive d'écriture distante fournie par PUT peut fonctionner dans un mode d'émulation, utilisant le réseau TCP/IP pour échanger les données. Ce mode d'émulation est activable au cas par cas en fonction de la localisation du nœud destinataire des données.

La primitive d'écriture distante, ainsi que tous les protocoles qui en font usage, peuvent donc être disponibles entre **tous** les nœuds d'une machine MPC en configuration multi-sites. Cela a pour unique intérêt d'unifier l'interface de communication entre les tâches, car les performances sont très pénalisées lors des communications inter-sites.

8.1.3 Organisation interne du Manager

Le rôle du Manager est d'allouer les ressources et de gérer l'ensemble des tâches de la machine MPC.

Il doit donc gérer des ressources de différents types, pour différentes tâches, et tout ceci de manière distribuée. Son organisation et les structures de données qu'il va manipuler sont donc vouées à être imbriquées et probablement inter-dépendantes.

On a donc choisi, à la fois pour faciliter le développement, et aussi pour pouvoir organiser puis présenter de manière satisfaisante son architecture interne, de faire un développement avec **une approche orientée objets**. On a choisi pour cela le langage C++, qui propose tous les principaux concepts de l'approche objet tout en restant proche du langage C, le langage de prédilection des applications système. Cela nous facilitera la tâche lors des communications entre le Manager et les couches noyau. Il propose en plus une bibliothèque de charpentes standardisée, la STL[2], qui permet de créer de nombreuses structures de données pour manipuler n'importe quel type d'objet. On dispose donc, sans à avoir à les reprogrammer, d'arbres binaires, de tables associatives, d'algorithmes de tri et de parcours de structures de données, qui vont être utilisés par le Manager pour gérer les ressources.

Il y a un seul manager, et probablement plusieurs tâches (surtout dans un contexte SMP), sur chaque nœud. Lorsque le manager est amené à dérouler un algorithme pour servir une tâche, il serait fâcheux qu'il soit indisponible pour toutes les autres tâches jusqu'à la fin de l'opération en cours, car celle-ci peut aussi dépendre de l'exécution d'un Manager distant. Son temps de traitement est donc potentiellement grand, et non borné si ce Manager

2. STL : Standard Template Library

distant attend une réponse lors d'une opération avec une de ses tâches locales. On a donc choisi, pour résoudre de tels conflits, **une approche *multi-threads***.

Nous avons besoin d'implémenter diverses opérations distribuées, par exemple lors de la négociation de l'attribution d'un nouveau canal entre deux tâches distantes. En effet, ce sont les Manager locaux des nœuds où se situent les tâches en question qui vont devoir orchestrer les opérations effectuées par leurs noyaux respectifs, pour identifier et mettre en place un nouveau canal de communication, puis le signaler aux deux tâches concernées. La démarche traditionnelle pour autoriser des opérations distribuées dans un environnement orienté objet consiste à effectuer les opérations à l'aide d'un ORB[3]. On a donc décidé de construire un ORB pour faciliter la programmation des algorithmes distribués du Manager. **L'approche ORB** est à ce propos bien adaptée aux environnements *multi-threads*.

Pour récapituler, on a choisi une approche orientée objet, suivant une organisation *multi-threads*, avec des communications de type ORB.

Les protocoles implémentés à l'aide de cet ORB ne nécessiteront pas des performances importantes, notre approche ORB n'est pas basée sur cette contrainte. Il existe néanmoins des ORB conçus en vue de proposer des performances importantes sur grappes de PCs. CORBA a par exemple été implémenté sur l'intertace VIA [Kishimoto *et al.*, 2000].

L'architecture du Manager peut alors se définir comme une encapsulation de poupées russes, présentée par la figure 8.2. Il y a, tout d'abord, au plus haut niveau, un ensemble d'algorithmes permettant d'effectuer les travaux requis par les tâches auprès du Manager. Ces algorithmes sont implémentés en faisant usage d'une boite à outils construite dans le but de faciliter les opérations distribuées, et construite autour d'un ensemble d'objets contenant à la fois les structures de données et le code pour effectuer toutes les opérations de base traditionnelles. Cette boite à outils, la MOODT[4], est constituée de la réunion de deux ensembles de services de base. Le premier, *l'arbre de classes du noyau du système* ou CSCT[5], offre l'encapsulation des *threads* et des communications au sein d'un arbre d'héritage d'objets. Le second, *l'arbre de charpentes de distribution* ou DTCT[6], offre une encapsulation dans des charpentes (*template class*) des primitives d'invocation de méthode distante constituant un ORB, et des algorithmes distribués standards.

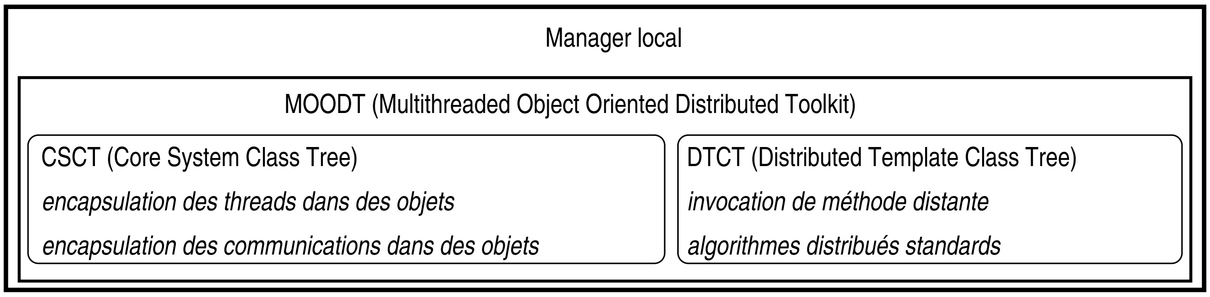


Fig. 8.2 *Structure interne du Manager local*

3. ORB : Object Request Broker
4. MOODT : Multi-threaded Object Oriented Distributed Toolkit
5. CSCT : Core System Class Tree
6. DTCT : Distributed Template Class Tree

8.2 Le CSCT (Core System Class Tree)

8.2.1 Arbre d'héritage

La figure 8.3 présente l'arbre d'héritage des classes du noyau du système (CSCT). Les classes dont le nom surmonte l'indicatif *virtual* constituent des classes de base virtuelles. On va, au cours de cette section présenter brièvement l'ensemble de ces classes qui forment les outils de base du Manager.

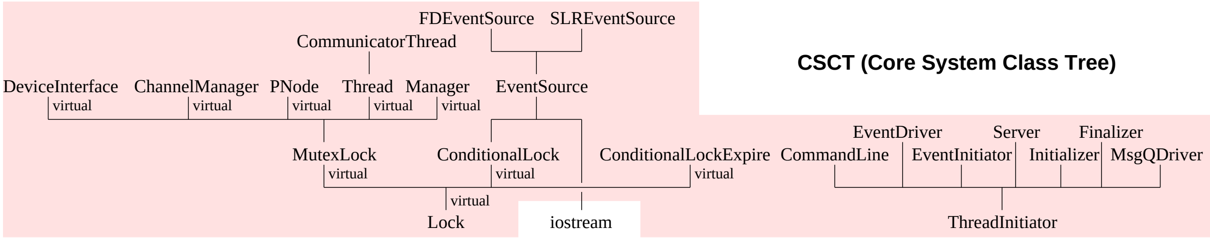


Fig. 8.3 *Arbre de classes du CSCT*

8.2.2 Les classes de verrous

Le Manager est une application constituée de multiples *threads* travaillant simultanément sur une structure de données constituée d'un ensemble d'objets. Il nous faut donc disposer d'un puissant mécanisme de gestion de verrou afin de garantir l'intégrité du fonctionnement du système. Pour cela, le CSCT propose trois classes de verrous (MutexLock, ConditionalLock et ConditionalLockExpire) offrant des services de verrou plus ou moins évolués, et nécessitant pour cela plus ou moins de ressources, dérivées d'une classe de base virtuelle pure.

Le tableau 8.4 énumère les propriétés des différentes classes de verrous.

Classe	Méthodes	Un autre thread peut libérer le verrou	Un autre thread peut réveiller un thread en attente	L'implementation nécessite	Notes
Lock	Acquire() = 0 Release() = 0				La classe Lock est une classe de base virtuelle pure
MutexLock	Acquire() Release()	NON	NON	1 mutex	
ConditionalLock	Acquire() Release()	OUI	NON	1 mutex 1 variable de condition 1 booléen	
ConditionalLockExpire	Acquire() Release() Expire()	OUI	OUI	1 mutex 1 variable de condition une énumération à 3 états 1 compteur	Ne doit pas être invoquée par un thread possédant le verrou

Fig. 8.4 *Propriétés des classes de verrous*

La classe ConditionalLockExpire est celle proposant le plus de possibilités. Les autres classes constituent des verrous très classiques, mais celle-ci fournit une propriété qui lui est propre : une méthode, `Expire()`, permet à un *thread* quelconque de réveiller tous les *threads* en attente sur le verrou. Cette propriété fait de cette classe une parfaite candidate pour des verrous pouvant bloquer un *thread*, et dont le relâchement est conditionné à l'action d'un stimulus extérieur au Manager. En effet, un tel verrou peu bloquer indéfiniment un *thread*, et par là même empêcher la terminaison propre du Manager en cas de nécessité. Le *thread* dont le rôle est de faire le ménage lors d'une terminaison, peut, grâce à la méthode ConditionalLockExpire réveiller tous les *threads* bloqués, afin qu'ils se terminent et libèrent par là même proprement les ressources qu'ils détiennent.

Notons que l'énumération à trois états dont il est fait référence dans la figure 8.4 contient les états suivants : verrouillé, déverrouillé, en cours de terminaison. D'autre part, le compteur dont il est question sert à comptabiliser le nombre de *threads* en attente sur le verrou concerné.

La figure 8.5 page ci-contre présente les trois organigrammes représentant les fonctionnements respectif des trois méthodes de ConditionalLockExpire (`Acquire()`, `Release()`, `Expire()`).

Les classes de verrous son utilisées principalement en tant que classes de base de nombreuses autres classes. Quand un *thread* parcourt le code d'une méthode d'un objet quelconque, afin que cette objet effectue une opération particulière, il faut toujours envisager le cas ou un autre *thread* pourrait lui-même aussi parcourir une autre méthode de cet objet. Les données privées de l'objet sont accessibles par les différentes méthodes qui le composent, ces dernières doivent donc mettre en place un mécanisme d'accès verrouillé pour que les données gardent leur intégrité. C'est pourquoi de nombreux objets sont dérivés d'une classe de verrou.

8.2.3 Les classes dérivées de ThreadInitiator et de Thread

La classe ThreadInitiator constitue la classe de base que l'on dérive pour créer les classes qui contiendront le code des *threads*.

Il y a 7 types de *threads* dans chaque Manager, représentés par 7 classes qui héritent de ThreadInitiator :

- ✓ La classe Initializer : il n'y en a qu'une instance, dont le rôle est d'initialiser le processus Manager. Elle instancie notamment tous les objets initiaux.
- ✓ La classe Finalizer : une instance de cette classe est créée lors de la phase de terminaison du processus Manager, par exemple quand la remontée d'une exception n'a pu être interceptée par aucun bloc *catch*. Son rôle est de débloquer puis libérer tous les *threads* et de détruire les derniers objets en mémoire, pour que l'application quitte correctement le système.
- ✓ La classe CommandLine : une instance de cette classe est créée au démarrage du processus dans le but d'interpréter des commandes issues de l'entrée standard, afin de permettre à l'utilisateur d'analyser au moment de l'exécution l'état des différentes structures de données internes du Manager (par exemple la liste des canaux négociés

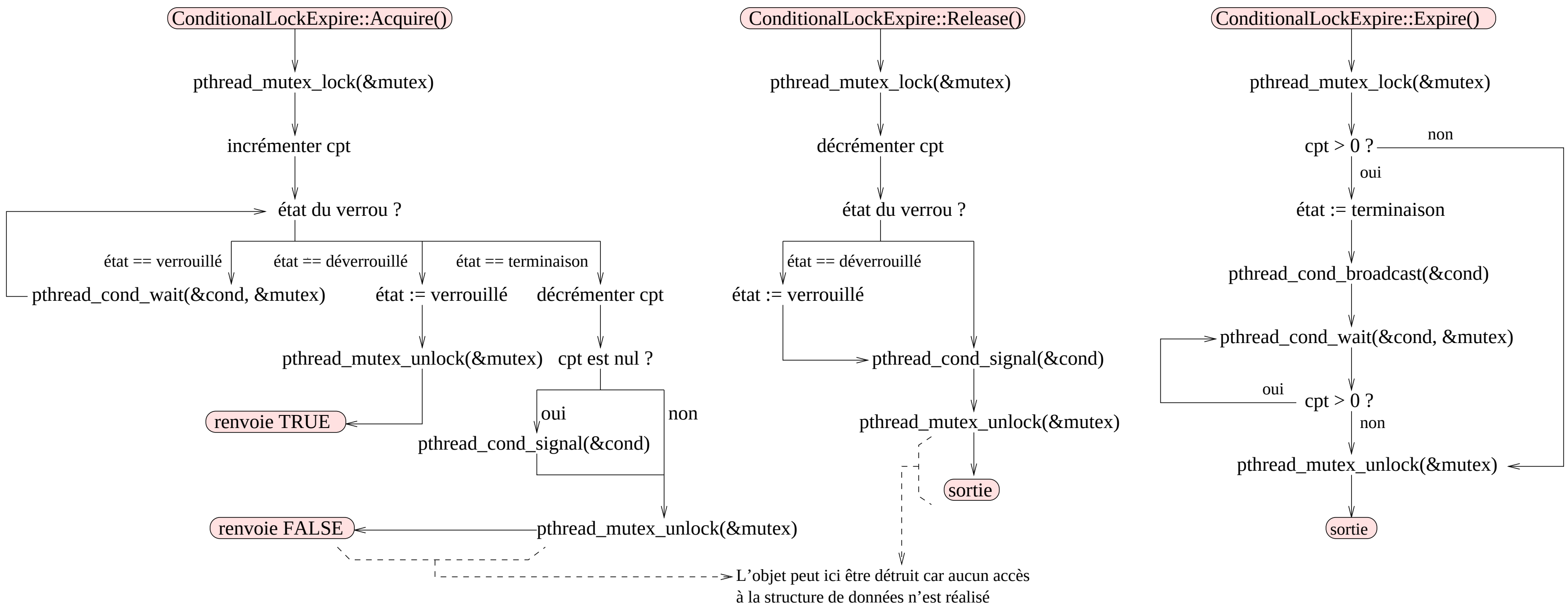


Fig. 8.5 *Organigramme de l'algorithme de verrouillage*

dynamiquement), dans un but de *monitoring*.

- ✓ La classe MsgQDriver : une instance de cette classe sert des requêtes sur une file de messages Unix. Son rôle est de permettre d'étendre le contrôle d'applications externes sur le fonctionnement du Manager. Avec une telle instance, on dispose d'un moyen de communication simple pour envoyer des stimuli au Manager depuis une application utilisateur. Cela peut par exemple permettre d'implémenter des outils externes de *monitoring* en mode batch (le *monitoring* en mode interactif est permis par la classe CommandLine).
- ✓ La classe Server : chaque instance de cette classe écoute sur un port de communication et y attend des connexions depuis des Manager distants. Pour chaque connexion reçue, cette classe instancie les objets représentant le Manager distant et le canal de communication qui permet de dialoguer avec. Il y a donc deux instances de Server dans les Manager de nœuds inter-clusters (une pour les connexions internes, et une autre pour les connexions WAN). Dans tous les autres nœuds, il n'y a qu'une instance de Server.
- ✓ La classe EventInitiator : cette classe constitue l'interface entre les applications distribuées et le Manager. Chaque tâche d'une application distribuée est représentée dans le Manager par une instance de cette classe. La tâche communique avec son représentant de type EventInitiator à l'aide d'une *socket* Unix (`/tmp/mpc`) et du protocole T2MP [7] qu'on présentera dans la suite de ce chapitre. Elle y émet des requêtes et en reçoit des acquittements. L'objet de type EventInitiator dialogue donc d'une part avec la tâche qu'il représente, et transmet les demandes aux autres objets du Manager. Il achemine alors en retour les résultats d'opérations.
- ✓ La classe EventDriver : les instances de cette classe implémentent le protocole de bas niveau permettant aux Manager de dialoguer entre eux. Son rôle consiste donc à analyser les requêtes provenant d'un autre Manager et à éventuellement les rerouter vers un autre Manager, ou bien à les acheminer en interne vers les objets auxquels elles sont destinées. C'est donc toujours dans le contexte d'un *thread* EventDriver que sont exécutées les méthodes des objets locaux invoquées par un Manager distant. Le rôle de EventDriver est donc double : fournir un contexte d'exécution de requêtes distantes, et rerouter des requêtes qui ne sont pas destinées à un objet local.

Les *threads* de type EventInitiator et EventDriver sont organisés en *pool* : ils sont créés au démarrage du Manager et restent sans affectation jusqu'à ce qu'ils soient associés temporairement à un travail particulier, avant de retourner dans le *pool* des *threads* inactifs.

A chaque nouvelle connexion sur la *socket* Unix `/tmp/mpc`, un *thread* libre du *pool* EventInitiator est choisi pour représenter la tâche qui vient de se connecter. A chaque requête sur le canal de communication inter-Manager, un *thread* libre du *pool* EventDriver est choisi pour traiter la requête. Lorsqu'il n'y a aucun *thread* disponible dans un *pool*, la requête de l'objet qui tente de s'en procurer un est placée en attente jusqu'à ce qu'elle puisse être satisfaite.

La figure 8.6 page suivante présente les différentes activités au sein du Manager.

Les classes dérivées de ThreadInitiator contiennent le code initial de chaque *thread*. Elles implémentent une méthode `ThreadInitiator::StartThread()` qui est invoquée une fois

7. T2MP : Task To Manager Protocol

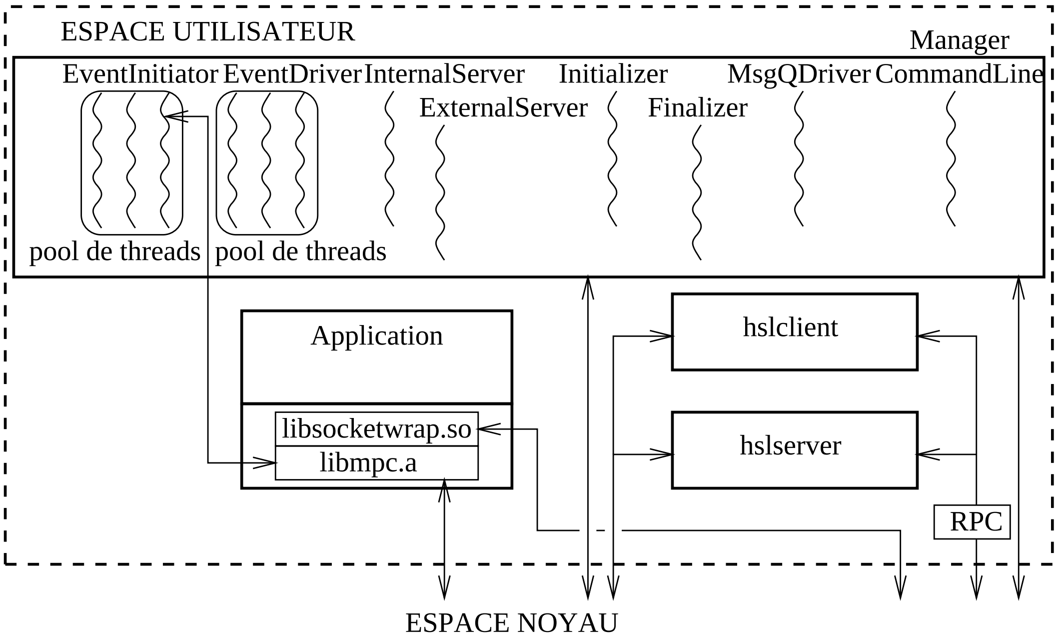


Fig. 8.6 *Les différentes activités au sein du Manager*

le *thread* créé. A chaque instance de ThreadInitiator est associé un objet de type Thread qui joue le rôle d'interface avec la bibliothèque Unix implémentant les *threads*. L'objet Thread a à sa charge la gestion (création/suppression) d'un *thread* Unix dont le point de départ est le code de la méthode `StartThread()` de l'objet ThreadInitiator.

La classe CommunicatorThread, dérivée de Thread, contient les données nécessaires à la synchronisation d'un *thread* quelconque avec les *threads* EventDriver qui gèrent les communications. Par exemple, comme on le verra plus loin, un *thread* qui émet une requête vers un autre Manager doit être mis en attente, *via* un verrou, jusqu'à ce qu'une réponse revienne. Ce verrou appartient à CommunicatorThread.

8.2.4 Les autres classes du CSCT

Les autres classes du CSCT, présentes sur la figure 8.4 page 161 et dont nous n'avons pas encore parlé, seront étudiées plus en profondeur par la suite. On peut néanmoins d'ores et déjà préciser leur rôle général.

Les instances des classes dérivées de EventSource sont les objets qui gèrent toutes les communications avec les autres Manager.

Les objets de type PNode[8] sont les représentants des nœuds distants de la machine MPC. Chaque objet PNode référence donc notamment un objet EventSource par lequel le Manager distant associé peut être joint.

Un des principaux rôles du Manager est de gérer les attributions dynamiques de canaux. La

8. Nous avons choisi de nommer cet objet PNode et non Node pour différencier l'objet du numéro de nœud lui-même. Si le numéro de nœud change au cours du temps, une référence sur un objet de type PNode restera néanmoins toujours valide.

classe ChannelManager permet de tenir ce rôle : un canal entre deux nœuds est représenté au sein d'une instance de ChannelManager dans l'un de ces nœuds.

La classe DeviceInterface n'est instanciée qu'une seule fois, pour fournir un objet permettant de dialoguer avec les couches de communication noyau, à travers des appels systèmes. Quand un objet quelconque désire dialoguer avec une couche noyau, il passe donc par l'intermédiaire d'une méthode de DeviceInterface.

La classe Manager fournit le premier objet créé juste après l'activation du processus Manager. Cet objet a un rôle important dans la création des autres objets, dans le maintien de tables de références sur de nombreuses ressources (objets, *threads*) du Manager. Il a donc une vision d'ensemble du système et propose donc des services d'aide à la communication intra-Manager aux différents objets.

8.3 Routage statique dans le réseau virtuel des Manager : les classes EventSource et PNode

La figure 8.7 présente le schéma d'héritage[9] des classes FDEventSource et SLREventSource, toutes deux dérivées de EventSource, et dont le rôle est de gérer les communications avec les autres Manager. Les traits pleins représentent les dérivations de classes, tandis que les flèches en pointillés représentent les références sur des structures de données.

La classe FDEventSource utilise un objet référencé par un descripteur de fichier Unix pour jouer son rôle, alors que la classe SLREventSource utilise un canal MDCP.

Les communications à travers un objet EventSource se font par l'intermédiaire de deux méthodes : `EventSource::read()` et `EventSource::write()`.

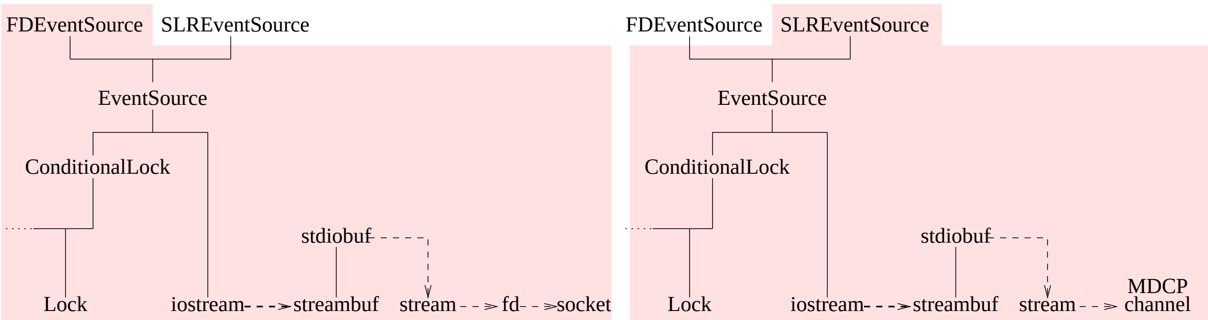


Fig. 8.7 *Les dérivées de la classe EventSource*

Il y a une instance de EventSource pour chaque Manager distant auquel on est directement relié. Il y a donc, dans un Manager donné, un objet EventSource représentant les canaux MDCP le reliant avec les Manager distants du même cluster. Il y a en plus, sur le Manager inter-cluster un objet EventSource pour représenter chacun des autres Manager inter-cluster.

9. Les classes iostream, streambuf, stdiobuf et stream sont des classes standard de la norme C++, on ne va donc pas les décrire dans ce manuscrit.

Sur la figure 8.8, les objets EventSource et PNode d'une machine MPC constituée de 2 clusters de 2 nœuds chacun sont représentés.

Les objets EventSource sont associés deux à deux puisqu'ils représentent un canal reliant deux Manager sur deux nœuds distincts. Sur la figure 8.8, on peut distinguer les *threads* à l'origine de la création des PNodes et EventSource : il s'agit des *threads* Server. Sur les nœuds intercluster, il y en a deux : InternalServer sert les requêtes de connexions des Managers du même cluster, et ExternalServeur les requêtes provenant des autres clusters. Il y a donc un ExternalServeur uniquement sur les nœuds inter-clusters.

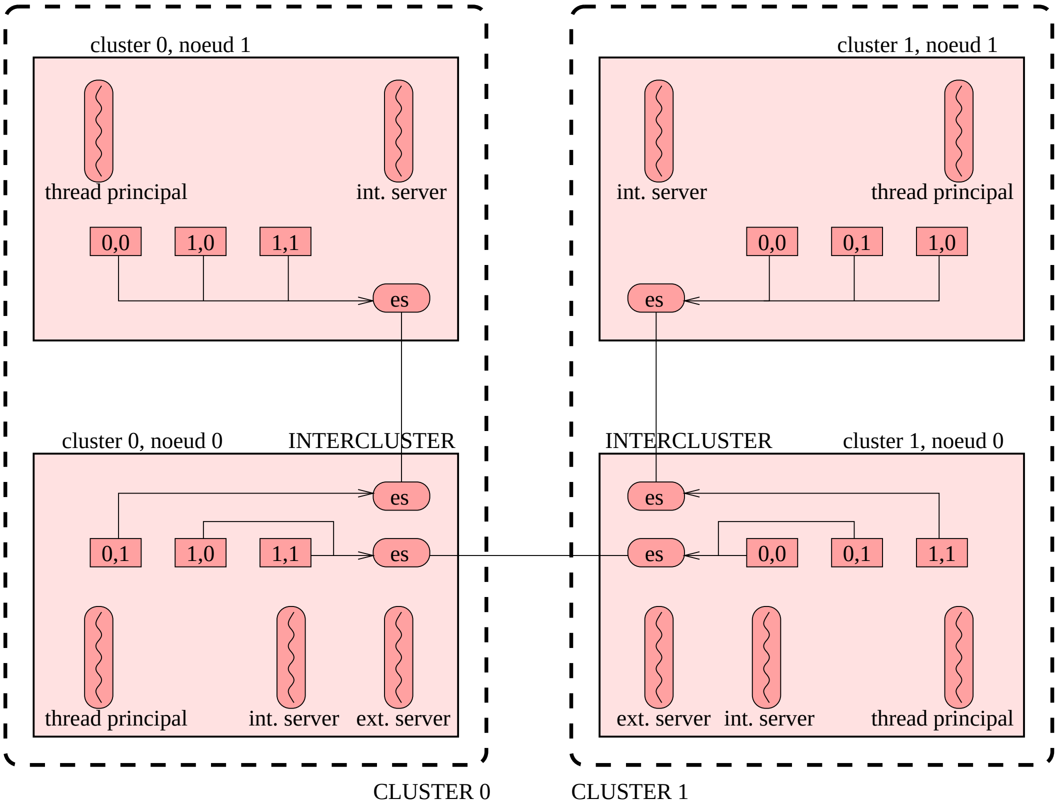


Fig. 8.8 *Routage statique entre Manager*

Les classes PNodes, qui représentent l'ensemble des nœuds distants, sont créées au même moment que les EventSource par les *threads* Server. Mais s'il n'y a qu'une EventSource par Manager directement connecté, il y a un objet PNode pour chaque nœud de la machine, qu'il soit directement connecté et donc directement accessible par l'intermédiaire d'un EventSource, ou que ce ne soit pas le cas.

Le rôle d'un objet PNode est de maintenir une référence sur l'objet EventSource qui représente le Manager par lequel il faut passer pour se rapprocher du destinataire, sur le réseau virtuel constitué par l'ensemble des Manager de la machine MPC.

Quand un *thread* (par exemple le *thread* noté *thread principal* sur la figure 8.8) exécute le code d'un objet désireux de communiquer avec un nœud particulier, la méthode PNode::GetEventSource() est alors invoquée, et la communication est émise à travers l'objet EventSource qui est alors fourni.

On a donc affaire à un routage statique entre les Manager.

8.4 Le DTCT (Distributed Template Class Tree) : un cœur d'ORB

8.4.1 Structure d'objet distribué

Le DTCT (Distributed Template Class Tree) fournit un modèle d'objets distribués formant un cœur d'ORB dont le rôle est d'implémenter l'invocation de méthode distante de manière transparente pour le programmeur.

Un objet distribué dispose d'un représentant dans le Manager de chacun des nœuds sur lesquels il est distribué. Un objet distribué n'est donc schématiquement rien d'autre qu'un ensemble d'objets présents sur différents nœuds et instanciés à partir d'une même classe.

L'objet distribué dispose donc des mêmes méthodes sur tous les nœuds. Dans notre modèle, plusieurs instances d'un même objet distribué peuvent coexister sur le même nœud. D'autre part, plusieurs objets distribués du même type peuvent coexister, en ayant éventuellement un ou plusieurs représentants sur les mêmes nœuds. La figure 8.9 montre un exemple comportant trois objets distribués, certains disposant de plusieurs représentants sur un même nœud. Sur les nœuds 0 et 1, on y trouve des représentants multiples de plusieurs objets distincts mais de même type.

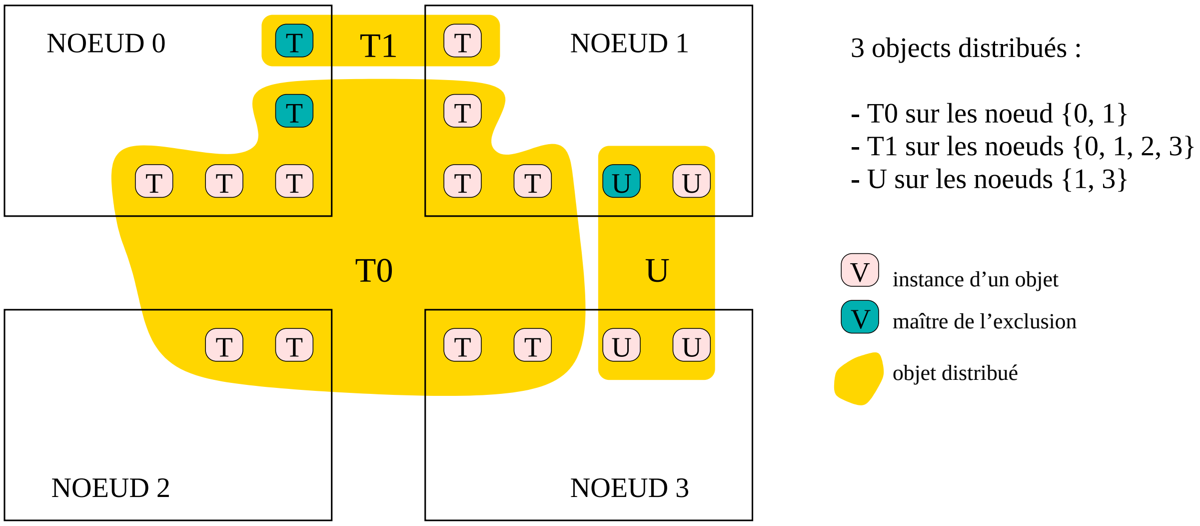


Fig. 8.9 *Exemple d'objets distribués*

Bien sûr, chaque instance doit pouvoir invoquer une méthode d'un représentant situé sur un autre nœud, ou sur le même nœud, et ceci de manière totalement transparente.

8.4.2 La charpente TDistObject<> : gestion de l'invocation de méthode distante et de la topologie

La charpente (*template class*) TDistObject<> constitue le noyau central de notre ORB. Elle intègre le mécanisme d'invocation de méthode distante, met en œuvre le protocole associé et gère la topologie de l'objet distribué.

La charpente TDistObject<> forme une classe en prenant en paramètre une autre classe. Par exemple, supposons que nous ayons une classe X que nous désirons distribuer, c'est-à-dire que nous désirons fournir la possibilité à un groupe distribué de ses instances d'invoquer mutuellement leurs méthodes. La classe TDistObject<X> maintient dans des structures de données privées la localisation de l'ensemble des représentants de cet objet distribué. Elle inclut aussi des méthodes privées implémentant un protocole de dialogue sur le réseau (à travers les objets EventSource).

Ce protocole permet d'effectuer des invocations de méthodes distantes, en transportant sur le réseau les adresses des méthodes à invoquer. Ainsi, un objet de classe TDistObject<X> (ou bien sûr dérivant de cette classe) peut dialoguer avec un autre objet de la même classe TDistObject<X> afin d'invoquer une de ses méthodes. Plus précisément, un objet TDistObject<X> peut invoquer des méthodes d'un TDistObject<X> distant, ou bien d'un X distant. En effet, le but final reste quand même de permettre à l'objet X de dialoguer avec ses pairs. On verra plus loin dans cette section que l'architecture choisie permet de garantir qu'à partir du moment où TDistObject<X> sait invoquer un X distant, il sait alors invoquer un TDistObject<X> distant.

Cela permet à TDistObject<X> d'implémenter simplement une distribution de ses données privées. Sachant que celles-ci consistent principalement en la localisation des différents représentants de l'objet distribué, elles vont être sujettes à peu de modifications. Ainsi, à chaque modification, c'est à dire quand il va falloir mettre à jour ces données dans tous les représentants, une méthode centralisée va être utilisée.

Le principe de cette méthode consiste à élire un représentant particulier. Celui-ci est le seul à disposer d'un verrou destiné à mettre en œuvre cette méthode. On nomme ce représentant *maître de l'exclusion* (*exclusion owner* - un exemple est présenté figure 8.9). Pour chaque opération nécessitant une mise à jour des données de tous les représentants, celui à la source de l'opération invoque un méthode chez le *maître de l'exclusion*. Cette méthode est naturellement invoquée dans le cadre d'un *thread* EventDriver, vu le système conjoint de communications et de gestion de *threads* qu'on a explicité précédemment. L'objet *maître de l'exclusion* verrouille le verrou, ce qui permet d'éviter que deux requêtes simultanées se produisent. Ainsi, deux *threads* EventDriver ne peuvent simultanément effectuer l'opération.

Le *maître de l'exclusion* modifie ses données puis invoque une méthode chez tous ses homologues pour distribuer la mise à jour. Il termine alors son travail en libérant le verrou.

Un procédé de migration de la fonction *maître de l'exclusion* d'un représentant vers un autre est fourni. Il est par exemple utile quand l'opération demandée consiste à supprimer un représentant, et qu'il s'agit justement du *maître de l'exclusion*.

Un procédé de gestion de la topologie est ainsi fourni par TDistObject<X>.

8.4.3 Dérivation d'objets distribués

Rappelons maintenant notre objectif de base : disposer, à partir d'un objet X, d'un objet distribué *proposant les mêmes services*. Pour cela, il suffit tout simplement d'ajouter TDistObject<X> comme classe de base de X.

Il suffit donc à TDistObject<X> de savoir invoquer une méthode d'un X distant pour savoir invoquer une méthode d'un TDistObject<X> distant, comme nous l'avions admis précédemment. Une classe distribuée a donc pour classe de base la classe formée en instanciant TDistObject<> sur la classe distribuée en question.

La classe distribuée la plus simple est nommée DistObject. Elle ne permet aucune opération particulière : elle hérite tout simpement de TDistObject<DistObject>. La première partie de la figure 8.10 montre sa définition.

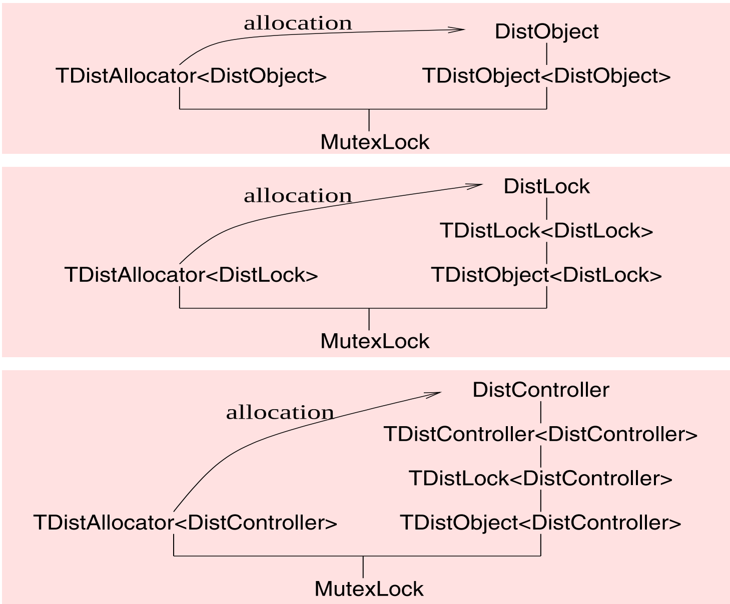


Fig. 8.10 *Objets distribués et allocateurs*

Essayons maintenant de construire une classe DistLock distribuée proposant un mécanisme de verrou distribué. L'idée première est de fournir DistObject comme classe de base de DistLock. Ce serait une erreur : DistObject ne sait invoquer que des méthodes de TDistObject<DistObject> et DistObject. DistLock serait donc dans l'impossibilité d'invoquer des méthodes d'un DistLock distant.

Il faut utiliser un schéma un peu plus subtil pour arriver à nos fins. Créons une charpente TDistLock<> définie de manière à ce que la classe TDistLock<X> hérite de TDistObject<X>, comme schématisé sur la figure 8.10. Créons alors l'objet DistLock dérivant de TDistLock<DistLock>. Ainsi, DistLock dispose de l'ensemble des méthodes de TDistObject<DistLock>. **Mais ça ne prouve pas qu'il peut invoquer les méthodes distantes d'un représentant de TDistLock<DistLock> ou de DistLock**.

C'est néanmoins possible, et il nous faut le montrer rigoureusement. DistLock dispose des méthodes de TDistObject<DistLock>, il peut donc invoquer les méthodes distantes de TDistObject<DistLock> et de DistLock, par définition des propriétés fournies par TDistObject<> que nous avons étudiées précédemment.

DistLock peut donc invoquer des méthodes distantes de DistLock, donc des méthodes distantes des classes dont il dérive, donc des méthode distantes de TDistLock<DistLock>.

Maintenant, il nous faut examiner où placer les méthodes implémentant le verrou distribué. On pourrait naïvement penser que c'est dans l'objet DistLock qu'il faut les implémenter. Ce serait une erreur, car ces méthodes ne seraient alors pas accessibles à des objets distribués dérivant de DistLock (par exemple DistController dont on parlera plus loin).

En effet, on vient de montrer que pour dériver DistObject en DistLock, il faut faire hériter DistLock de TDistLock<DistLock>, donc de TDistObject<DistLock>, et non pas faire dériver directement DistLock de DistObject. Il en est évidemment de même pour faire dériver DistController de DistLock. C'est à dire que DistController héritera de TDistLock<DistController> et non pas de DistLock. Il faut donc implémenter l'algorithme de verrou dans TDistLock<> plutôt que dans DistLock si on veut que DistController puisse en bénéficier.

8.4.4 Allocation d'objets distribués

Dans le cadre de la gestion de topologie d'un objet distribué, TDistObject<X> doit pouvoir créer un nouveau représentant d'un objet distribué X sur un nœud n'en disposant d'aucun. Il doit donc pouvoir invoquer un objet présent sur chaque nœud et se chargeant de cette opération. Pour cela, la charpente TDistAllocator<> a été créée conjointement à TDistObject<>, et cette dernière a été conçue pour pouvoir invoquer des méthodes distantes de TDistAllocator<>. Précisément, TDistObject<X> sait invoquer des méthodes distantes de TDistAllocator<X>. De plus, au démarrage d'un Manager, pour tout X distribué, un objet TDistAllocator<X> est créé. La figure 8.10 page précédente montre les allocateurs des objets distribués standards.

La figure 8.11 rassemble les 12 classes de DTCT. On pourra noter qu'on n'a développé spécifiquement que 4 charpentes (TDistAllocator<>, TDistObject<>, TDistLock<> et TDistController<>) et aucune classe, pour obtenir ces douze classes.

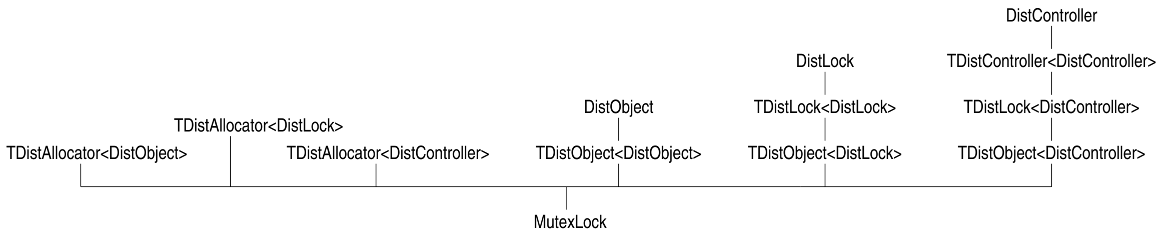


Fig. 8.11 *Arbre d'héritage des objets distribués*

8.4.5 La charpente TDistLock<> : un verrou distribué

La classe DistLock implémente par l'intermédiaire de la charpente TDistLock<> un verrou distribué. L'algorithme choisi est basé sur celui de Chandy et Misra [Raynal, 1992]. Ce dernier suppose que l'ensemble des sites pouvant acquérir le verrou est fixé une fois pour toutes. Or, notre classe DistLock dérive de TDistObject<DistLock> et peut donc voir son nombre de représentants évoluer dans le temps.

Les deux propriétés fondamentales de l'algorithme de verrou distribué de Chandy et Misra sont *la sûreté* [10] et *la vivacité* [11].

Il est facile de se rendre compte que la sûreté n'est pas compromise par l'ajout d'un site au cours du temps. Par contre, la vivacité, basée sur l'étude d'un graphe qui reste acyclique au cours du temps, n'est plus vérifiée dans le cas où de nouveaux sites peuvent apparaître. En effet, on peu facilement créer des objets DistLock dont l'état interne garantit que le graphe reste acyclique, mais en examinant de près la démonstration de Chandy et Misra, on se rend compte qu'il faut en plus utiliser une majoration de la distance avec un nœud particulier de l'arbre, et cette condition n'est plus vraie quand le graphe contient de nouveaux nœuds (en effet, le nombre d'arcs augmente).

Par contre, si le nombre total de changements de topologie au cours du temps, dans un objet dérivant de TDistLock<>, est borné, alors on retrouve la vivacité.

On ne détaillera pas plus ces problèmes algorithmiques, sachant que dans MPC-OS, le nombre de sites des objets distribués utilisés est fixe et correspond au nombre de nœuds de la machine.

8.4.6 La classe DistController

L'approche objet, et *a fortiori* le choix de concevoir un ORB, ont pour but de rendre réalisable la mise en œuvre de protocoles complexes de gestion des ressources. Le Manager propose actuellement uniquement l'allocation de canaux et la gestion des tâches. Pour cela, il utilise un unique objet distribué, contenant un seul représentant par Manager et constitué d'une instance de la classe DistController. On étudiera plus loin son fonctionnement.

L'allocation de cet objet distribué est effectuée au démarrage de la machine, par le Manager du nœud inter-cluster du cluster de plus bas numéro (en pratique celui de numéro 0). Ce Manager instancie un objet DistController. Ce dernier est un objet distribué ayant un représentant unique, il est donc *maître de l'exclusion*. Il va donc pouvoir de lui-même créer des instances dans tous les nœuds, en invoquant la méthode `New()` des objets TDistAllocator<DistController> distants.

Cet objet distribué reste alors présent sans modification de sa topologie jusqu'à la fin du fonctionnement de la machine.

10. Montrer la sûreté d'un algorithme consiste à prouver qu'il est toujours conforme à son objectif.

11. Prouver la vivacité d'un algorithme de verrou consiste à montrer que quel que soit le comportement de l'ensemble des sites au cours du temps, une demande d'acquisition du verrou sera toujours satisfaite au bout d'un temps fini.

Nous avons vu que notre ORB est capable de gérer simultanément de multiples objets distribués, ainsi que les migrations des représentants de ces objets au cours du temps. Nous n'utilisons pas ces fonctionnalités dans le cadre de la machine MPC. Cela ne veut pas dire qu'il n'est pas utile d'en disposer, car le Manager a justement été construit dans une optique évolutive : il forme lui-même une charpente permettant d'ajouter à volonté de nouvelles fonctionnalités, donc de nouveaux services pour les applications, sans interférer avec les protocoles déjà existants (cela justifie notamment le choix de l'environnement *multi-threads*).

8.5 Invocation de méthode distante

8.5.1 Les primitives de base

Supposons dans cette section que nous disposons d'une classe distribuée T, qui dérive donc de TDistObject<T>. Nous allons étudier comment se construit le mécanisme d'invocation des méthodes distantes de T.

La charpente TDistObject<>, une fois instanciée en classe à l'aide de T, propose deux fonctions pour effectuer des invocations distantes de méthodes de T : `remoteMSG()` qui rend la main immédiatement, et `remoteRPC()` qui fournit en retour la valeur retournée par la méthode distante. Pour des raisons d'implémentation, cette valeur ne peut prendre qu'un type de taille celle d'un entier. Par contre, la signature[12] de la méthode distante est virtuellement quelconque. Il y a donc deux types d'invocation : les RPC et les messages sans retour.

T hérite des méthodes `remoteMSG()` et `remoteRPC()` car il hérite de toutes les méthodes de TDistObject<T>, étant dérivé de cette classe.

Voici donc un exemple d'invocation de méthode distante.

```
class PNode *p_node; // pointeur sur un objet PNode
class T     *p_t;    // pointeur sur une instance de la classe T du noeud
                     // distant, classe disposant d'une méthode T::method()

      remoteMSG(p_node, p_t, &T::method, param1, param2, ...);
ret = remoteRPC(p_node, p_t, &T::method, param1, param2, ...);
```

8.5.2 Le protocole inter-Manager

Comme nous l'avons déjà signalé, les Manager utilisent MDCP pour communiquer entre eux. La figure 8.12 page suivante présente schématiquement le format de paquet de données qui transite lors d'une invocation de méthode distante et dans l'éventuel retour de la valeur de statut renvoyée par l'objet distant.

12. Dans un langage objet, on appelle signature d'une méthode la conjonction des types de chacun de ses arguments.

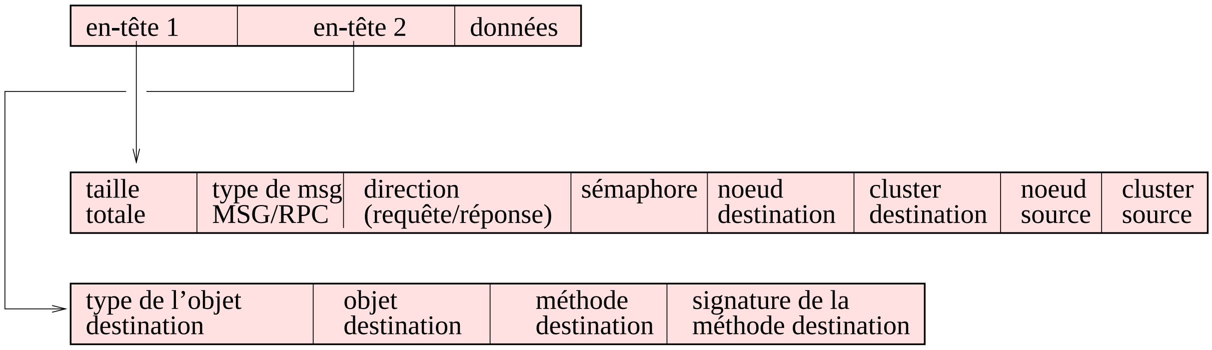


Fig. 8.12 *Protocole d'invocation de méthode distante*

Il y a successivement deux en-têtes puis les données proprement dites, c'est-à-dire les paramètres destinés à la méthode distante.

La première en-tête sert au routage, elle est donc toujours décodée par les Manager qui la reçoivent. Elle contient la taille totale des données, le sens du message (invocation distante ou retour de statut), la localisation du destinataire (numéros de nœud et cluster).

Dans le cas d'une invocation de type RPC, on trouve aussi la localisation du Manager initial, afin de pouvoir acheminer en retour la valeur de statut. Le *thread* initiateur de l'invocation est donc bloqué jusqu'au retour de ce statut, il faut donc prévoir un mécanisme pour à la fois lui signaler l'arrivée du statut et pour lui indiquer sa valeur.

Pour cela, on dispose dans la première en-tête d'un emplacement pour indiquer l'adresse d'une sémaphore sur laquelle le *thread* qui est à l'origine de l'invocation distante se bloque. Il sera débloqué au retour du statut.

La seconde en-tête contient tout ce que le Manager destinataire doit connaître pour invoquer la méthode distante : type de l'objet, adresse de l'objet, adresse de la méthode et signature de la méthode.

8.5.3 Synchronisation entre Manager

La figure 8.13 page ci-contre présente un exemple d'invocation de méthode distante avec `remoteRPC()`.

On y trouve deux types de *threads* : d'une part les instances de EventInitiator, qui, rappelons-le, sont les représentants des tâches locales ; dans ce contexte ils répondent aux requêtes des tâches en mettant en œuvre des opérations sur les ressources. L'objet EventInitiator contient les ressources nécessaires à l'émission d'une requête inter-Manager, notamment la sémaphore qui va permette la synchronisation pour l'obtention du statut. On trouve d'autre part les *threads* EventDriver dont le rôle est de lire des données à travers les objets EventSource qui en reçoivent ; les EventDriver interprètent ces données suivant le format de paquet décrit précédemment, et les acheminent alors, en local (invocation de méthode), ou à distance (retransmission des données sur un autre objet EventSource, plus proche de la cible).

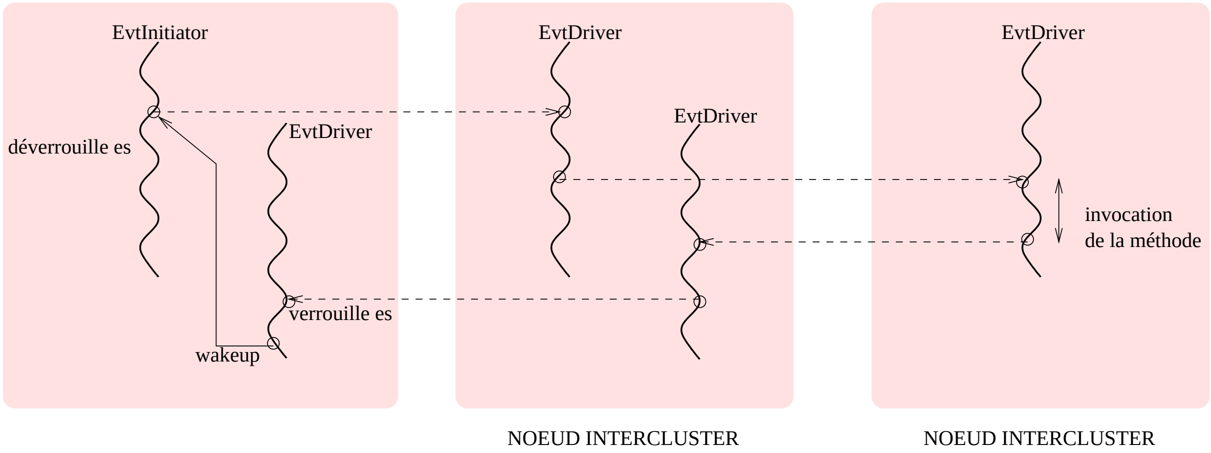


Fig. 8.13 *Exemple d'invocation de méthode distante*

Examinons les différentes étapes dans l'échange de type RPC mettant en jeu trois nœuds de la figure 8.13 :

- ▷ Un objet distribué quelconque invoque dans le contexte d'un *thread* EventInitiator une méthode distante d'un de ses représentants sur un autre nœud ;
- ▷ La méthode `remoteRPC()` demande à l'objet PNode représentant le destinataire, quel est l'objet EventSource sur lequel il pointe (c'est celui qui permet de s'en rapprocher) ;
- ▷ La méthode `remoteRPC()` verrouille l'objet EventSource, y écrit les données au format de paquet décrit précédemment, et déverrouille l'EventSource ;
- ▷ La méthode `remoteRPC()` se met en attente sur la sémaphore appartenant aux données de l'objet EventInitiator représentant le *thread* qui lui permet de travailler ; c'est l'adresse de cette sémaphore qui a été placée dans la première en-tête du message ;
- ▷ Dans le second Manager, un *thread* inactif du *pool* des *threads* EventDriver est réveillé, car l'objet EventSource connecté avec le premier Manager reçoit des données. Cet EventSource est verrouillé, les données de la première en-tête en sont extraitres, l'objet PNode de numéro celui trouvé dans l'en-tête est invoqué pour fournir l'EventSource qui permet de se rapprocher de la destination. Les données sont alors retransmises sur cet EventSource (il y a évidemment une phase de verrouillage/déverrouillage pour cette opération, puis le déverrouillage de l'EventSource verrouillée au début de cette étape) ;
- ▷ Un *thread* EventDriver se retrouve donc avec les données, c'est dans son contexte qu'est effectuée l'invocation de méthode, puis le statut est renvoyé dans l'autre sens ;
- ▷ Le statut traverse le deuxième nœud ;
- ▷ Dans le Manager initial, un *thread* EventDriver est réveillé suite à l'activité sur l'EventSource initial. Notons que cet objet avait été libéré juste après qu'on y ait envoyé les données initiales, il a donc pu être utilisé par d'autres *threads* pour envoyer ou recevoir d'autres messages sur le canal MDCP qui relie les deux premiers Manager ;
- ▷ Nous allons maintenant aborder la technique d'échange des données sans copie entre le *thread* EventDriver et le *thread* EventInitiator :

L'EventSource est à nouveau verrouillé. L'EventDriver lit la première en-tête et s'apperçoit qu'il s'agit d'un message de réponse, contenant donc un statut dans la partie données, et à destination du nœud local. Il lit alors la deuxième en-tête, puis effectue une opération de libération des *threads* en attente sur la sémaphore dont l'adresse est dans la première en-tête.

- ▷ Il retourne alors dans son *pool* pour redevenir inactif, **sans avoir préalablement libéré le verrou de l'EventSource** ;
- ▷ Le *thread* EventInitiator se retrouve alors libéré de son attente sur sa sémaphore, quelques instants plus tard. L'EventSource est toujours verrouillée, aucun *thread* ne peut s'en servir pour y lire des données ;
- ▷ Le *thread* EventInitiator lit la fin du message, c'est-à-dire le mot de statut, et déverrouille enfin l'EventSource. La méthode `remoteRPC()` peut alors rendre la main à l'objet appelant, en lui renvoyant par la même occasion le mot de statut.

8.5.4 Invocation des allocateurs

Nous avons indiqué précédemment que pour disposer de l'adresse d'un représentant distant d'un objet distribué X, il faut invoquer la méthode `New()` sur l'unique instance de son allocateur sur ce nœud distant, de type TDistAllocator<X>. Il faudrait donc pour cela disposer, en principe, de l'adresse de cette instance. Or cette information n'est disponible que sur le nœud distant.

Pour résoudre ce problème, *sachant qu'il n'y a qu'une instance de chaque allocateur*, et que le type de l'objet destination se trouve inclus dans la seconde en-tête des messages inter-Manager, le *thread* EventDriver du nœud récepteur n'utilise pas l'adresse incluse dans le message mais détermine l'adresse de l'allocateur à partir de son type. On peut donc toujours invoquer les méthodes d'un allocateur distant sans connaître sa localisation précise.

8.5.5 Généralisation de la syntaxe d'invocation de méthode du C++

Plutôt que d'utiliser des méthodes locales particulières pour invoquer des méthodes distantes d'un objet (en l'occurrence les méthodes locales `remoteRPC()` et `remoteMSG()` fournies par DistObject), on a préféré rendre ce procédé transparent en étendant la syntaxe C++ habituelle des invocations par un objet de ses propres méthodes.

Il a donc fallu tout d'abord définir un objet répertoriant un représentant distant d'un objet distribué. On a défini pour cela la charpente dist_obj<> dont le constructeur nécessite une paire[13] indiquant l'adresse de l'objet PNode représentant le nœud distant ainsi que l'adresse de l'objet distant dans l'espace virtuel du Manager distant.

Une classe T représentant un objet distribué étant fixée, un représentant de T à l'adresse `p_obj` sur le nœud distant `node` est ainsi défini de la manière suivant, sous l'identificateur

13. La charpente pair<X,Y> est définie dans la norme STL.

d_obj :

```
pair<class PNode *, class T *> d(node, p_obj);
dist_obj<class T> d_obj(d);
```

A l'aide de la charpente less<> qui implémente un comparateur lexicographique, on peut créer automatiquement *via* STL des structures de données complexes organisant des objets de type dist_obj<T>. Pour définir le comparateur lexicographique sur un objet distribué, il suffit d'utiliser la syntaxe suivante :

```
less<dist_obj<class T> >
```

On peut ainsi facilement créer des listes ou des tables associatives à partir de références sur des représentants d'un objet distant. Par exemple, dans le cadre de l'implémentation de l'algorithme de verrou distribué de Chandy et Misra, on a dû définir la table associative des permissions, mettant en relation un booléen avec chaque site. Cette table associative est simplement définie de la sorte[14] :

```
map<dist_obj<class T>, bool, less<dist_obj<class T> > > permission_map;
```

Lorsqu'un objet traditionnel invoque une méthode qui lui est propre (ou appartient à une de ses classes de base), il se contente de préciser l'identificateur de la méthode. Par exemple, si la méthode `method1()` de DistController (codée dans TDistController<DistController> plutôt que directement dans DistController, pour les raisons que l'on a vues précédemment) invoque la méthode `method2()`, le code est le suivant :

```
template <class T>
void TDistController::method1()
{
  result = method2();
};
```

Imaginons maintenant que nous voulions invoquer la même méthode, mais cette fois-ci sur un représentant distant `d_obj`. La syntaxe du C++ étendue par notre ORB est alors la suivante :

```
@{référence_sur_l_objet_distant},{reply|async},<classe>:méthode(paramètres)
```

On constate qu'on a ajouté trois paramètres devant le nom de la méthode, encadrés par un couple (`@`,`:`).

Le paramètre `reply` indique que l'on veut être bloqué jusqu'à la réception du code de retour, et dans le cas contraire on utilisera `async`.

14. Les charpentes de fonction less<> et d'objet map<> sont définies dans la norme STL.

Ainsi, le code d'invocation de `method2()` devient, sous sa forme générale distribuée :

```
template <class T>
void TDistController::method1()
{
  result = @d_obj,reply,<T>:method2();
};
```

Si `d_obj` référence un représentant local (par exemple le représentant qui fait cet appel), l'invocation provoquera un simple appel de méthode locale.

Les deux derniers des trois paramètres supplémentaires dans notre syntaxe ont des valeurs par défaut : s'ils ne sont pas précisés, l'invocation est de type `reply` et le paramètre de la charpente en cours est T.

On peut donc réécrire plus simplement le code précédent comme suit :

```
template <class T>
void TDistController::method1()
{
  result = @d_obj:method2();
};
```

Pour intégrer cette nouvelle syntaxe, un préprocesseur particulier, se rajoutant derrière le préprocesseur standard, a été construit. Il se contente d'effectuer les transformations suivantes (en cas de conflit dans les noms d'identificateurs rajoutés par le préprocesseur, celui-ci les règle en utilisant des noms uniques) :

```
@d_obj,async,<T>:method(params);
             // sera transformé en :
dist_obj<T> d(d_obj);
remoteMSG(d.pnode, d.obj, &T::method, params);

result = @d_obj,reply,<T>:method(params);
             // sera transformé en :
dist_obj<T> d(d_obj);
result = remoteRPC(d.pnode, d.obj, &T::method, params);

result = @d_obj:method(params);
             // sera transformé en :
dist_obj<T> d(d_obj);
result = remoteRPC(d.pnode, d.obj, &T::method, params);
```

Dans ce qui précède, le code de retour d'une invocation est de taille limitée : celle d'un entier. On fournit un mécanisme pour pouvoir renvoyer n'importe quel type d'objet de manière transparente. Il consiste à provoquer, dans la méthode distante invoquée, une nouvelle invocation vers une méthode de l'objet initiateur, transportant en paramètre l'objet à retourner. Le paramètre de retour est donc placé dans un paramètre d'appel de méthode, il peut donc être de taille quelconque. Une fois cette invocation terminée, l'objet distant rend la main. On a donc deux invocations croisées entre deux instances d'un même objet distribué, donc quatre messages (deux de plus que dans le cas où on renvoie uniquement un entier).

8.6 Flux de données et protocoles

8.6.1 Structure générale

La figure 8.14 présente la structure globale de communication dans le cadre des opérations distribuées effectuées par les Manager. Ceux-ci disposent d'un seul objet distribué, DistController, et d'objets locaux, notamment ChannelManager dont le rôle est de négocier des canaux avec un nœud distant (il y a un ChannelManager par couple de nœuds), et DeviceInterface qui permet de dialoguer avec les couches noyau.

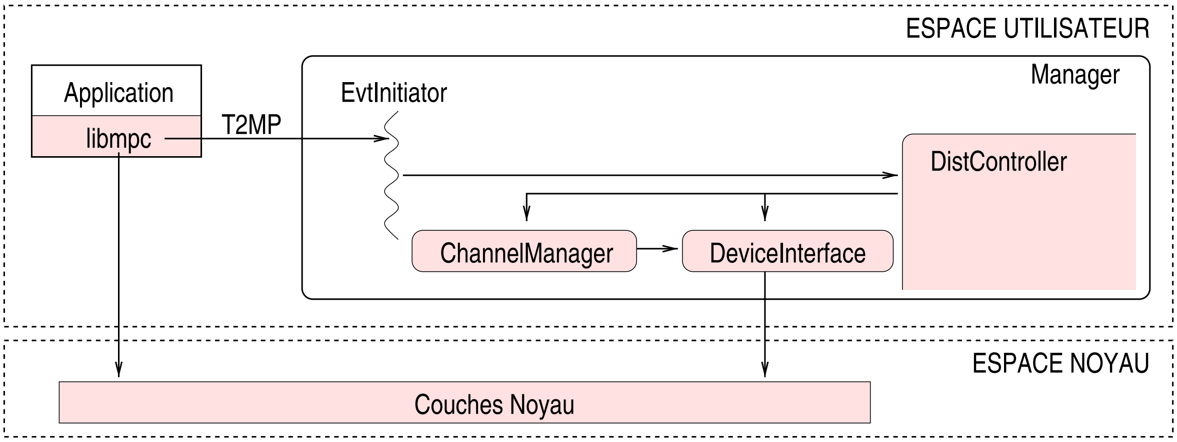


Fig. 8.14 *Flux de données sur un nœud*

Une opération distribuée correspond donc, grosso-modo, à une suite d'étapes mettant en jeu des communications entre les différentes entités présentes sur la figure 8.14.

Une opération commence habituellement par une demande de l'application. Elle effectue un appel à la bibliothèque LIBMPC. Cette dernière reporte l'ordre au *thread* qui la représente dans le Manager, à travers une *socket* Unix, *via* le protocole T2MP[15]. L'acquittement ou la valeur de retour revient par ce même canal, s'il y a lieu.

Dans le contexte du *thread* EventInitiator, une opération distribuée, codée dans une méthode de DistController, est invoquée. Cette méthode peut implémenter n'importe quelle opération distribuée, car elle a accès non seulement aux méthodes des objets locaux, mais aussi aux méthodes des objets DistController des nœuds distants, et par leur intermédiaire, elle peut donc invoquer facilement les méthodes de tous les objets distants.

15. T2MP : Task To Manager Protocol

L'algorithme distribué prend alors la forme d'une série d'invocations de méthodes distantes entre les objets DistController, et à chaque étape, l'invocation d'objets locaux, comme le ChannelManager par exemple.

Nous allons dans cette section étudier successivement les trois principaux protocoles distribués fournis par le Manager : la création de tâche distante, la création de canaux, et la destruction de canaux.

8.6.2 Création de tâche distante

La figure 8.15 présente les flux de données dans le cadre de la création d'une tâche distante.

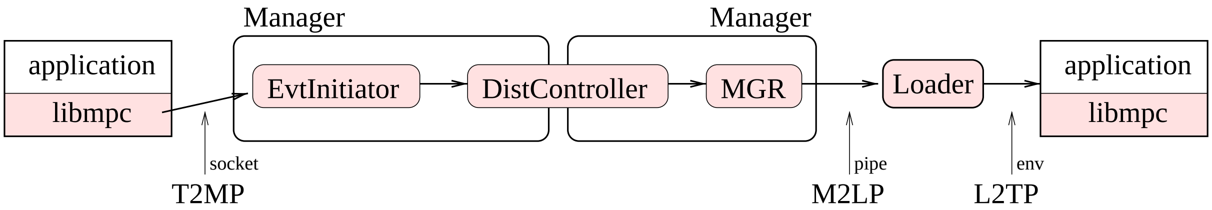


Fig. 8.15 *Création de tâche*

L'application invoque pour cela la bibliothèque LIBMPC en lui fournissant la ligne de commande et le nœud sur lequel lancer la nouvelle tâche.

LIBMPC propage cette demande à son représentant (le *thread* EventInitiator) *via* le protocole T2MP. La méthode de DistController destinée à l'invocation de tâche distante est alors invoquée dans le contexte du *thread* EventInitiator. DistController est un objet distribué, il peut donc invoquer une de ses méthodes sur un de ses représentants distant. Il effectue donc un tel appel de méthode distante dans le Manager du nœud destinataire. Le représentant distant de DistController invoque alors l'objet MGR dont le rôle est, entre autres, de dialoguer avec le processus d'activation et d'inactivation de tâches, le LOADER.

Cette communication s'effectue *via* le protocole M2LP[16] à travers un tube.

Enfin, le LOADER crée l'application, et communique à la bibliothèque LIBMPC des informations telles que la localisation de la tâche à l'initiative de sa création, *via* le protocole L2TP[17], qui propage de l'information en l'encapsulant dans des variables d'environnement.

8.6.3 La classe ChannelManager

Pour que deux tâches puissent effectuer une négociation de canal, il faut qu'elles invoquent toutes deux simultanément un point d'entrée de LIBMPC (`mpc_get_channel(...)`) en précisant le nœud distant où se situe la tâche avec laquelle elles veulent créer un canal, le type de protocole sous-jacent (SLR/P, SCP/P, SCP/V ou MDCP), et une donnée opaque identique dont le rôle est d'apparier les deux requêtes. Il n'est donc pas utile de connaître

16. M2LP : Manager to Loader Protocol
17. L2TP : Loader to Task Protocol

précisément la tâche distante avec laquelle on veut négocier un canal, seul le nœud doit être fourni et un rendez-vous est organisé par les deux Manager des nœuds en jeu dans cette négociation.

Le rendez-vous est précisément effectué au sein de l'objet ChannelManager qui s'occupe des canaux entre ces deux nœuds. Cet objet est présent uniquement dans l'un de ces deux nœuds, et chacun des Manager sait dans lequel il se trouve : c'est le Manager sur le nœud de plus bas numéro qui dispose du ChannelManager en question. La figure 8.16 présente la localisation des ChannelManager dans le cadre d'une machine MPC constituée de deux clusters de quatre nœuds chacun.

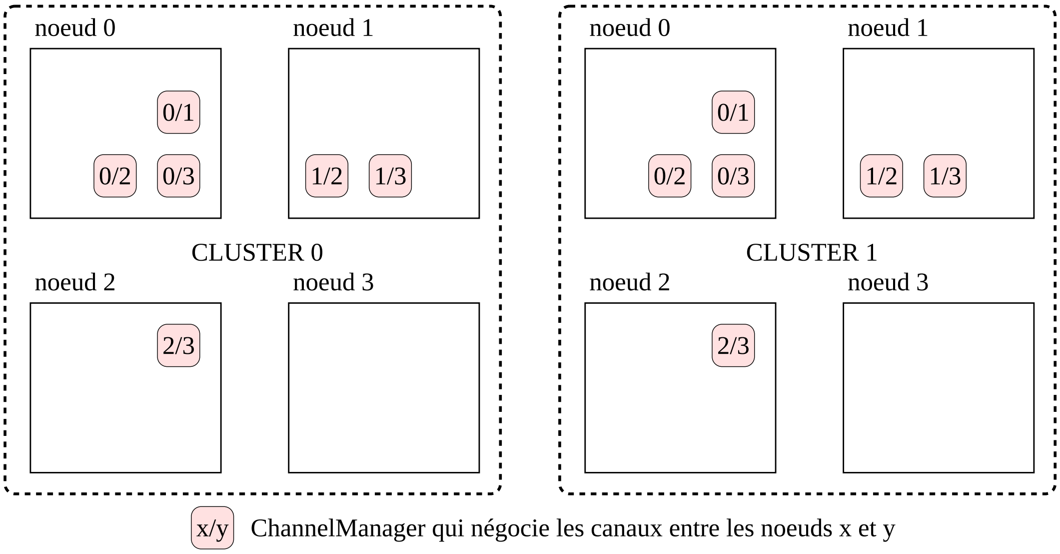


Fig. 8.16 *Localisation des ChannelManager*

Une version nettement plus évoluée dans le mode de négociation est spécifiée et partiellement implémentée. L'idée est de permettre de négocier un canal entre deux tâches sans qu'aucune ne connaisse la localisation du nœud de l'autre. Pour cela, le procédé qu'on vient de décrire est précédé d'une phase préliminaire permettant aux deux tâches de connaître leurs localisations respectives, *via* un rendez-vous sur le nœud inter-cluster de leur cluster. Une méthode de l'objet MGR de ce nœud est donc invoquée par les deux tâches, toujours en précisant une donnée opaque identique dont le rôle est d'apparier les deux requêtes. MGR fait alors la jointure de ces deux requêtes et fournit en retour les nœuds respectifs de ces deux tâches.

Avec cette méthode de négociation, deux tâches peuvent se faire attribuer un canal sans même disposer de leurs localisations respectives.

8.6.4 Création d'un canal

La figure 8.17 page suivante présente les opérations effectuées sur les deux nœuds concernés lorsqu'une tâche demande une négociation de canal.

Examinons un exemple de cette procédure, et plus précisément les opérations qui découlent

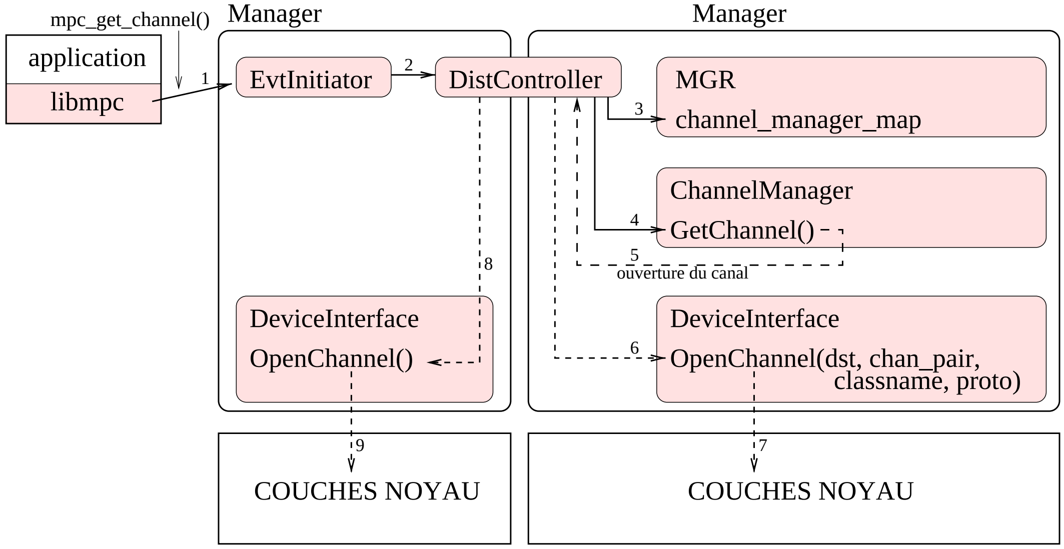


Fig. 8.17 *Création d'un canal*

de l'appel de la tâche qui ne se trouve pas sur le nœud de plus haut numéro (donc pas sur le nœud contenant le ChannelManager qui va gérer la négociation) :

1. L'application invoque LIBMPC, qui indique la demande de négociation au *thread* EventInitiator ;
2. Celui-ci invoque une méthode du DistController local destinée à gérer l'ensemble des opérations qui vont suivre ;
3. Le DistController local s'aperçoit que le numéro du nœud distant est plus petit que le numéro du nœud local. Le ChannelManager est donc situé sur le nœud distant. Il invoque donc le DistController distant. Celui-ci demande à MGR de consulter sa table `channel_manager_map` afin de retrouver l'adresse du ChannelManager chargé des relations entre nos deux nœuds ;
4. Le DistController invoque alors `GetChannel()` sur le ChannelManager. Deux cas peuvent maintenant se produire : soit une autre application a déjà demandé un canal entre ces deux nœuds en précisant la même donnée opaque permettant d'apparier les deux requêtes, et alors le numéro de canal est renvoyé et la procédure se termine, soit c'est la première requête avec cette donnée opaque, et la procédure passe à l'étape suivante. Dans les deux cas le compteur de références sur le canal en question est incrémenté ;
5. Le ChannelManager invoque DistController pour qu'il indique aux deux DeviceInterface des nœuds en question que le canal choisi est désormais ouvert ;
6. DistController invoque donc `OpenChannel()` de DeviceInterface, afin de signaler l'attribution du canal à la couche noyau correspondant au protocole indiqué par la tâche ;
7. La couche en question stocke le numéro d'UID[18] afin de n'autoriser les opérations sur ce canal que par les processus possédant cet UID ;
8. DistController invoque alors, via son représentant dans le nœud initial, la méthode OpenChannel() du DeviceInterface sur le nœud initial.

18. UID : User ID ; dans le monde Unix, l'UID est un entier qui représente un utilisateur donné.

9. Le noyau est alors informé de l'ouverture du canal.

Le numéro de canal est finalement renvoyé à la tâche, qui va maintenant utiliser les opérations d'émission/réception de la couche noyau pour laquelle ce canal a été attribué, sans plus passer par le Manager.

8.6.5 Suppression d'un canal

La figure 8.18 présente les différentes étapes qui se produisent quand on veut déréférencer un canal, en vue de le fermer (si on dispose de la dernière référence). L'objectif est qu'à la fin de l'opération de fermeture, aucune donnée concernant ce canal ne se situe encore dans le réseau ou en partance, et que les tables représentant l'état de ce canal dans les deux nœuds qu'il associe soient réinitialisées.

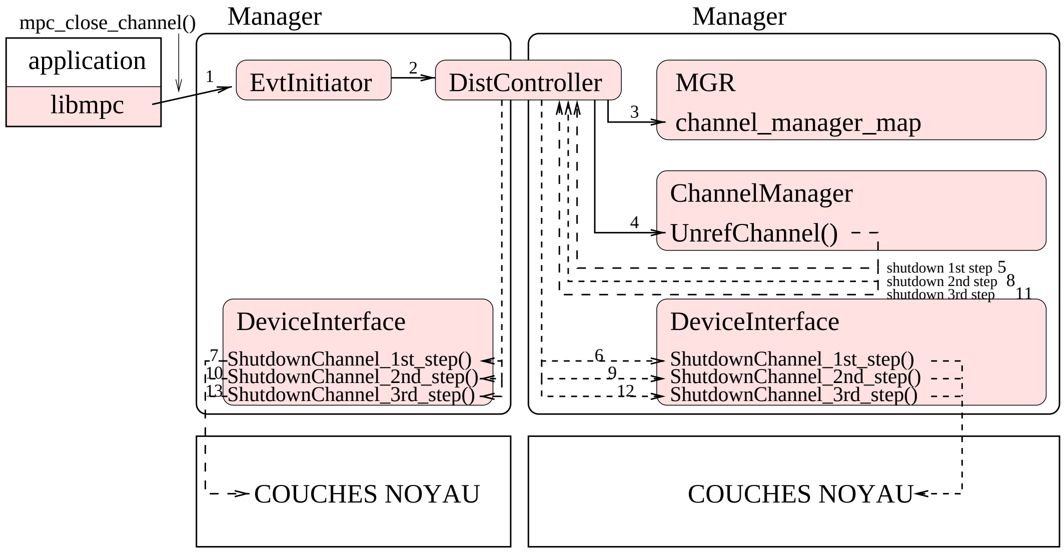


Fig. 8.18 *Suppression d'un canal*

La difficulté particulière de cette procédure consiste à garantir que le réseau est vide de paquet concernant le canal à fermer, sachant qu'une tâche peut invoquer la suppression d'un canal à tout moment.

Le DistController va réaliser cette opération en invoquant les objets DeviceInterface des deux nœuds en trois étapes successives :

❶ La première étape consiste à invoquer les deux objets DeviceInterface pour qu'ils ordonnent à la couche noyau en jeu d'interdire dorénavant toute opération d'un utilisateur sur le canal à fermer. Notons qu'il reste éventuellement des données, pour ce canal, en transit dans le réseau, et des entrées dans différentes tables gérant le protocole sur ce canal, correspondant à des émissions ou réceptions non appariées (cas où le nombre d'émissions n'est pas égal au nombre de réception). Les deux DeviceInterface récupèrent en même temps le numéro de séquence en émission et réception sur le canal en question, et renvoient cette information au DistController.

Ce dernier est donc informé des quatre numéros de séquence caractérisant le canal (numéros en émission et en réception, dans un sens comme dans l'autre) ;

❷ Le DistController invoque alors à nouveau les deux DeviceInterface pour qu'ils effectuent des émissions ou réceptions éventuellement multiples sur le canal en question, afin que le nombre de demandes d'émission sur chacun des nœud soit égal au nombre de demandes de réception sur l'autre nœud. Le DistController complète donc comme cela les émissions et réceptions non appariées ;

❸ Vu que les émissions et réceptions sont appariées, les tables noyau concernant le canal en question vont donc se vider, et à partir de cet instant, les numéros de séquences vont devenir identiques ; le DistController attend donc cet instant en venant consulter régulièrement les DeviceInterface des deux nœuds, et quand il a lieu, il demande aux DeviceInterface de remettre à zéro les compteurs de séquence.

Décortiquons donc maintenant l'ensemble des étapes de la procédure entière :

1. L'application invoque la LIBMPC, qui indique la demande de négociation au *thread* EventInitiator ;
2. Celui-ci invoque une méthode du DistController local destinée à gérer l'ensemble des opérations qui vont suivre ;
3. Le DistController local s'aperçoit que le numéro du nœud distant est plus petit que le numéro du nœud local. Le ChannelManager est donc situé sur le nœud distant. Il invoque donc le DistController distant. Celui-ci demande à MGR de consulter sa table `channel_manager_map` afin de trouver l'adresse du ChannelManager chargé des relations entre nos deux nœuds ;
4. Le DistController invoque alors `UnrefChannel()` sur le ChannelManager. Le compteur de références sur le canal est décrémenté et deux cas peuvent maintenant se produire : soit le compteur est non nul et la procédure se termine, soit le compteur est nul et la procédure se poursuit. Dans ce cas, le ChannelManager va alors invoquer successivement les trois procédures suivantes `ShutDownChannel_1st_step()`, `ShutDownChannel_2nd_step()`, et `ShutDownChannel_3rdt_step()` du DistController pour lui faire effectuer les trois étapes de la fermeture d'un canal évoquées précédemment ;
5. Le ChannelManager invoque donc `DistController::ShutDownChannel_1st_step()` ;
6. Le DistController invoque alors `DeviceInterface::ShutDownChannel_1st_step()`, sur le DeviceInterface du second nœud ;
7. Le DistController invoque ensuite cette même méthode sur le DeviceInterface du premier nœud ;
8. Le ChannelManager invoque donc `DistController::ShutDownChannel_2nd_step()` ;
9. Le DistController invoque alors `DeviceInterface::ShutDownChannel_2nd_step()`, sur le DeviceInterface du second nœud ;
10. Le DistController invoque ensuite cette même méthode sur le DeviceInterface du premier nœud ;
11. Le ChannelManager invoque donc `DistController::ShutDownChannel_3rd_step()` ;
12. Le DistController invoque alors `DeviceInterface::ShutDownChannel_3rd_step()`, sur le DeviceInterface du second nœud ;

13. Le DistController invoque ensuite cette même méthode sur le DeviceInterface du premier nœud.

8.7 Conclusion

Nous venons de voir des protocoles relativement complexes et qui mettent en jeu des opérations distribuées avec des appels simultanément imbriqués et croisés. Néanmoins, aucun interblocage n'est à craindre, car aux étapes de ces protocoles correspondent des opérations effectuées dans le contexte de *threads* soit de type EventInitiator, dont le blocage ne peut handicaper que les tâches qu'ils représentent, soit de type EventDriver, c'est-à-dire des *threads* membres d'un *pool* destiné à fournir des contextes d'utilisation en grand nombre. Tant que l'opération n'est pas terminée, la tâche qui l'a demandée est de toute façon bloquée, donc dans tous les cas, l'interblocage, au sens large, n'est pas à craindre.

Par interblocage au sens large, il faut comprendre interblocage entre opérations distinctes. C'est-à-dire que notre Manager est capable de réaliser les opérations complexes qu'on vient de décrire, et ceci de multiples fois simultanément pour de multiples couples de tâches.

Le choix d'un modèle *multi-thread* est donc, par ces opérations de création/destruction de canaux, largement justifié.

D'autre part, ces mêmes opérations peuvent nécessiter de nombreuses étapes ayant lieu alternativement dans un nœud puis dans un autre. L'opération de suppression d'un canal est par exemple composée de 13 étapes ayant lieu alternativement dans les deux nœuds présents aux extrémités du canal à détruire.

Le choix d'un modèle de communication par RPC a donc été prépondérant dans la faisabilité de l'implémentation de telles opérations. Il aurait été beaucoup plus difficile d'implémenter de tels protocoles avec une simple bibliothèque de passage de messages.

Cela justifie dans le même temps le choix d'implémenter ces opérations hors noyau.

Chapitre 9

Mesures et évaluation de performances

Sommaire

La machine MPC est constituée de composants matériels et logiciels, et ces deux classes de composants, indissociables lors de notre campagne de mesures, ont un impact sur les performances. Notre objectif consiste dans un premier temps à présenter les expériences et résultats expérimentaux, puis à mener une analyse en détails de nos mesures afin de comprendre les phénomènes sous-jacents. On tâchera alors de modéliser ces phénomènes, tout en quantifiant les paramètres utilisés à l'aide des mesures effectuées.

9.1 Mesures et modélisation

Dans le cadre de l'analyse des performances de l'informatique parallèle, le modèle LogP ([Mei-E, 2000] et [Touyama and Horiguchi, 2001]) a été développé afin de permettre, à partir de la mesure de quelques paramètres caractéristiques de l'implémentation d'un protocole particulier, de pouvoir déduire les performances que l'on peut en attendre.

Pour étudier en détails les performances de la machine MPC et de ses couches de communication, plutôt que d'utiliser ce modèle général, nous allons, plus loin dans ce chapitre, établir un modèle propre au fonctionnement du matériel et de la couche de communication de bas-niveau PUT. Il sera validé en le comparant aux mesures de performances brutes que nous allons présenter maintenant.

9.2 Mesures de latence

Afin de mesurer la latence matérielle et logicielle bas niveau de la machine MPC, on a mis en place un banc d'essai basé sur une machine MPC constituée de deux nœuds et reliés par un unique lien HSL.

Chaque nœud contient l'équipement suivant :

- ✓ Une carte FastHSL disposant d'un oscillateur cadencé à 66 MHz ;
- ✓ Un processeur Intel Pentium II 350 MHz ;
- ✓ Un chipset PCI Intel 440BX ;
- ✓ 128 Mo de SDRAM 100 MHz.

Sur chacun de ces nœuds, un processus en mode noyau effectue un ping-pong avec son homologue. Les messages échangés sont constitués chacun d'un seul mot de 32 bits, il y a donc un seul paquet échangé à chaque phase.

Pour effectuer une phase du protocole, le processus invoque la fonction d'émission de PUT puis se place alors en attente de réception active sur le tampon de réception (polling). Aucune interruption n'entre donc en jeu dans ce scénario.

On effectue alors quelques dizaines de milliers d'allers-retours de ce type, et à l'aide d'une horloge externe à notre banc de mesure, on déduit la latence de bout en bout d'un transfert, depuis le moment où l'émetteur en mode noyau décide d'effectuer une émission, jusqu'au moment où le recepteur en mode noyau est informé de la fin du transfert. **On mesure alors 4,0** μs.

On peut décomposer cette mesure en différentes phases :

▷ Temps de traitement de la fonction d'émission de PUT :
on a pu mesurer ce délai sur un nœud de calcul de notre banc de test, **la fonction d'émission de PUT coûte 1,9** μ**s** ;

▷ Latence matérielle :
la latence minimale théorique mesurée dans [Wajsbürt *et al.*, 1997] est de 1,7 μ**s** ;

▷ Délai de signalisation (polling) :
le temps d'accès mémoire est un minorant de la latence de scrutation, **la phase de scrutation coûte donc au moins 10 ns**.

On dispose donc de durée exacte de la première phase, de minorants des durées des deuxièmes et troisième phases, ainsi que de la totalité de la durée d'un transfert. On peut donc déterminer un encadrement des durées de latence matérielle et de signalisation, comme indiqué sur la figure 9.1.

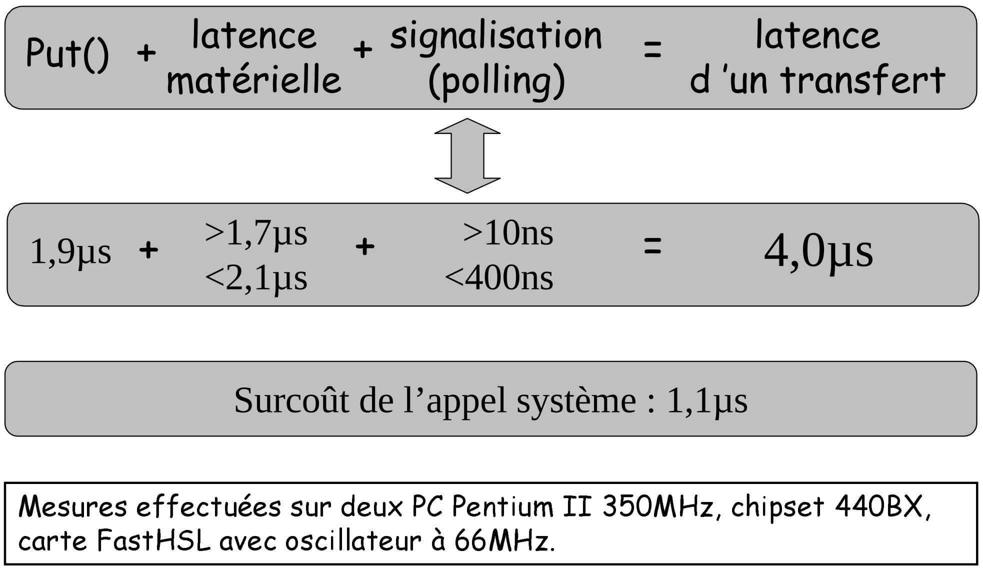


Fig. 9.1 *Latence de transfert au niveau PUT*

On a enfin mesuré **la durée depuis l'entrée jusqu'à la sortie d'un appel système vide : 1,1** μs.

Cela nous permet d'extrapoler nos résultats à des applications fonctionnant en mode utilisateur. On peut notamment indiquer que la durée d'un ordre d'émission, c'est-à-dire le temps consommé sur le processeur émetteur depuis le mode utilisateur est de $1{,}1\mu s + 1{,}9\mu s = 3\mu s$. La latence d'un transfert de mode utilisateur à mode utilisateur est majorée par $1{,}1\mu s + 4{,}0\mu s = 5{,}1\mu s$: il s'agit d'une majoration car la phase de retour d'appel système, comptabilisée dans les $1{,}1\mu s$, n'entre pas en jeu dans la latence. On notera que la latence de signalisation par scrutation est la même que le processus récepteur soit en mode noyau ou en mode utilisateur, car il s'agit simplement du coût d'une boucle d'accès mémoire.

9.3 Mesures de débit

9.3.1 Classification

À partir de la machine MPC du banc d'analyse de performances précédent, on a mesuré le débit entre deux nœuds : pour cela, sur l'un des deux nœuds, on a placé un processus effectuant des appels à PUT afin de générer des transferts de données unidirectionnels. Ce processus est constitué d'une boucle d'appels à la primitive d'émission de PUT, depuis le mode noyau.

La taille du tampon d'émission pouvant être paramétrée dans l'application, on a ainsi pu récolter la valeur du débit généré en fonction de la taille des pages émises. La figure 9.2 page suivante représente ces résultats bruts sur un graphe logarithmique.

On distingue cinq zones d'intérêt sur cette courbe.

- Zone A : pour les petites pages, l'allure exponentielle de la courbe tracée sur un graphe à échelle logarithmique nous fait supposer une certaine linéarité du débit en fonction de la taille des données.
- Zone B : dans cette zone, le débit atteint la moitié de la bande passante maximale. On l'appellera zone de demi-bande.
- Zone C : on découvre au centre de la zone C une rupture de pente assez inattendue de la courbe de débit.
- Zone D : encore plus inattendu, le débit est discontinu au centre de la zone D. De nombreuses mesures ont été prises autour de ce point pour confirmer qu'il ne s'agit pas d'une erreur de mesure, et faire apparaître nettement la discontinuité.
- Zone E : pour les pages les plus grandes (proches de 65535 octets), la courbe de débit approche son asymptote horizontale, limite supérieure des débits que l'on pourra atteindre entre deux nœuds.

9.3.2 Instants de discontinuité

La zone D présente une discontinuité du graphe de débit, que l'on va tenter d'expliquer.

> Rappelons qu'une *page réseau* est une zone de données contiguës en mémoire physique. Cette *page* peut être décomposée en un ou plusieurs paquets sur le réseau. Les données traversent successivement, à l'émission, le bus PCI 32 bits/33 MHz (donc 132 Mo/s), l'interface 8 bits à 66 MHz entre PCI-DDC et RCube (66 Mo/s), et enfin le lien HSL (1 Gb/s). **Le goulot d'étranglement se situe donc au niveau du bus 8 bits raccordant PCI-DDC et RCube.**

Si la page est suffisamment petite, elle est constituée d'un seul paquet. Si on augmente progressivement la taille des données, on arrivera à un moment où PCI-DDC prendra la

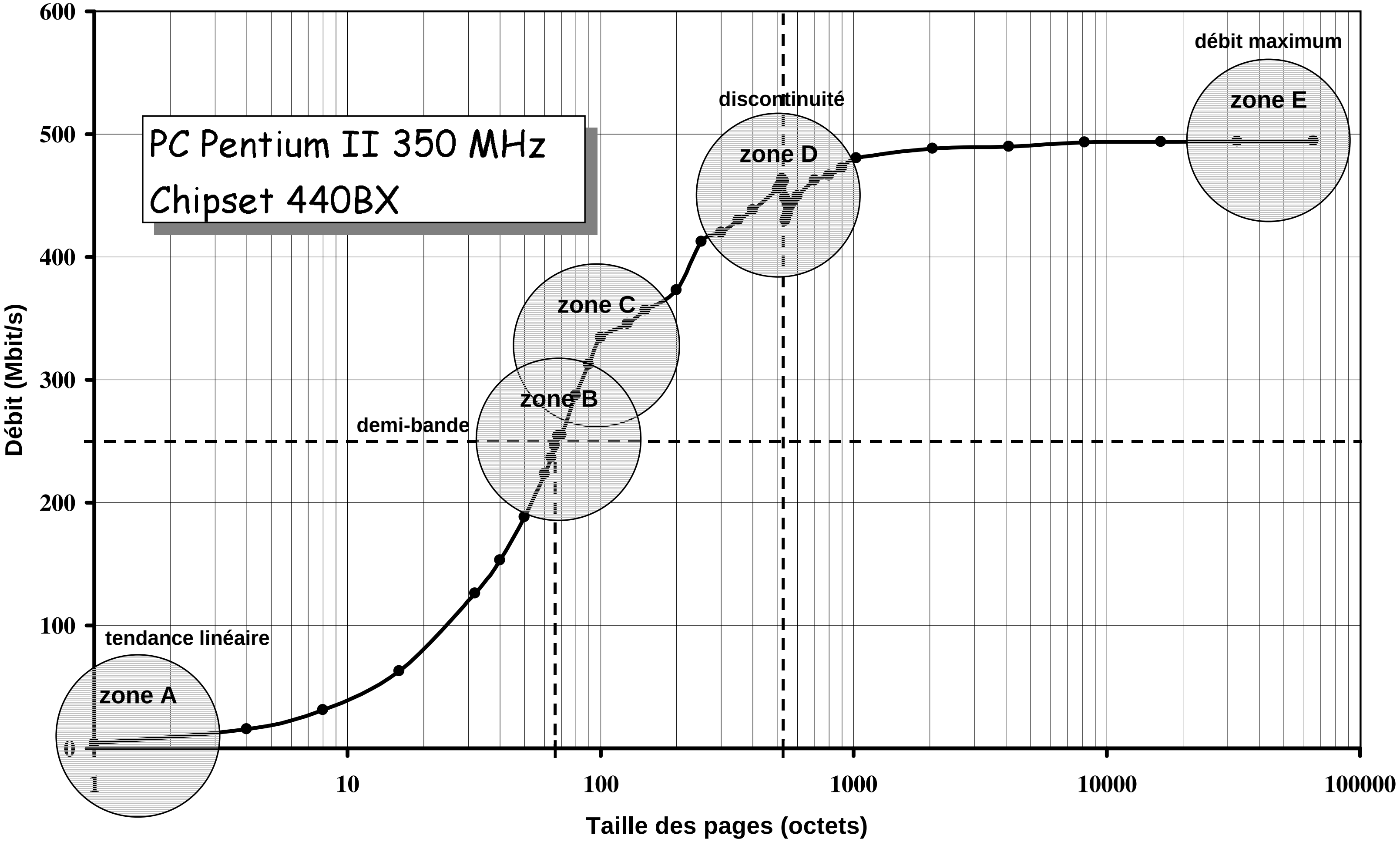


Fig. 9.2 – *Cinq zones singulières*

décision de clore le paquet en cours, et d'en initier un second pour terminer la page. Ainsi, pour un mot de données utiles supplémentaire, on aura une en-tête et une fin de paquet supplémentaire. Plusieurs causes peuvent conduire à une telle décision de PCI-DDC : par exemple, le protocole PCI impose à tous les maîtres de relâcher le bus régulièrement, afin de partager la bande passante équitablement entre les différents maîtres. PCI-DDC peut aussi décider de lui-même de relâcher le bus lors d'un engorgement du réseau HSL, afin de ne pas monopoliser inutilement la bande-passante PCI. Dans tous les cas, lorsque PCI-DDC relâche le bus, il clôt le paquet en cours de transmission, afin de ne pas bloquer inutilement un chemin sur le réseau HSL.

Rappelons qu'un paquet est constitué de 4 mots d'en-tête, des données utiles, d'un bourrage pour atteindre un nombre de mots entier au cas où les octets de données utiles n'auraient pas une taille multiple de 4, et enfin de 3 mots de fin de paquet (End of Data, Data CRC et End of Paquet). **On a donc 7 mots de contrôle en plus des mots de données utiles.**

Ainsi, lorsqu'on passe de un à deux paquets, on a 8 mots de plus à transmettre sur le bus à 66 Mo/s. On va donc faire chuter le débit, ce qui explique la discontinuité au centre de la zone D.

La figure 9.3 présente ce phénomène et permet de déterminer la taille des données utiles au moment du saut de débit : **540 octets utiles pour le paquet de données**.

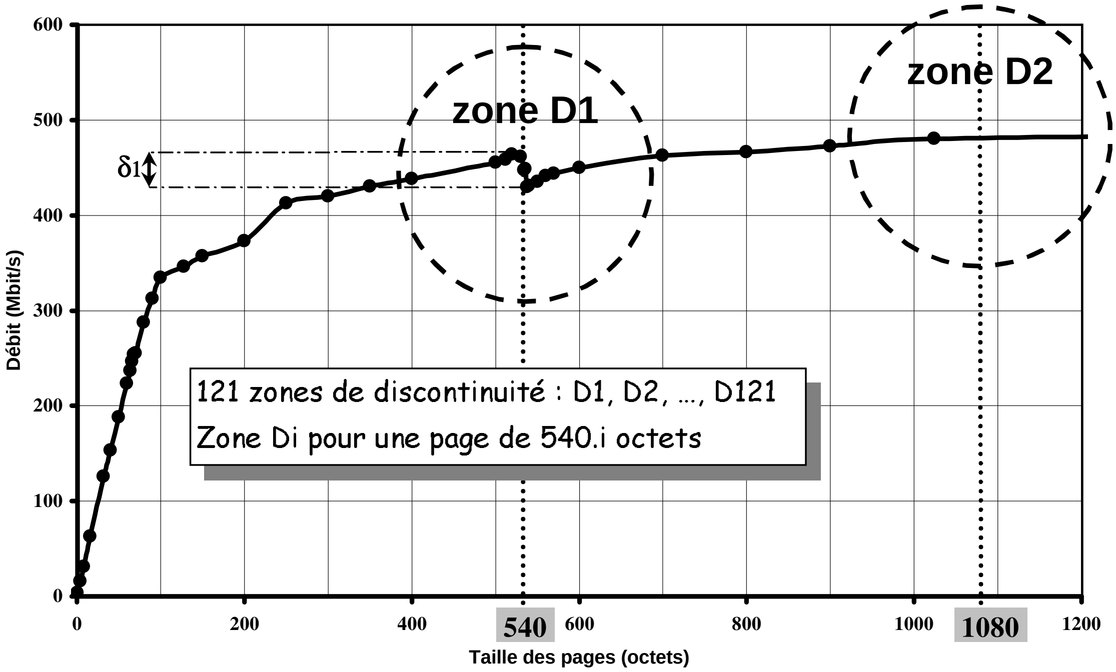


Fig. 9.3 *Zones de discontinuité*

Évidemment, ce phénomène se répète pour tous les paquets de 540 octets qui constituent une même page. Pour une page de taille maximale, c'est-à-dire de 65535 octets de données, on obtiendra 122 paquets, donc 121 zones de discontinuité.

Avec $\mathbf{S_{max} = 540}$ **octets**, $\mathbf{D_{DDC/R^3} = 66}$ **Mo/s** et $\mathbf{S_h = 7 \times 4}$ **octets**, et à condition que

PCI-DDC ait toujours au moins une page à traiter, le débit utile est donné par :

$$\mathcal{D}_{utile}(s) = \frac{s\, D_{DDC/R^3}}{[E(\frac{s-1}{S_{max}}) + 1]S_h + s} \tag{9.1}$$

On ne distingue sur le graphe que la première discontinuité, car les chutes de débit sont de plus en plus faibles (la première chute, pour 541 octets utiles, vaut $\mathcal{D}_{utile}(540) - \mathcal{D}_{utile}(541)$, soit 23,5 Mb/s ; la deuxième, pour 1081 octets, vaut 12,0 Mb/s).

9.3.3 Débit maximum et demi-bande

Sur la figure 9.4, on peut constater que le débit maximum est atteint dans la zone E, pour une valeur de 494 Mb/s. La demi-bande est donc atteinte pour 247 Mb/s. On lit sur le graphe que ce débit est atteint pour une page de 66 octets de données utiles. Il s'agit là d'une excellente performance.

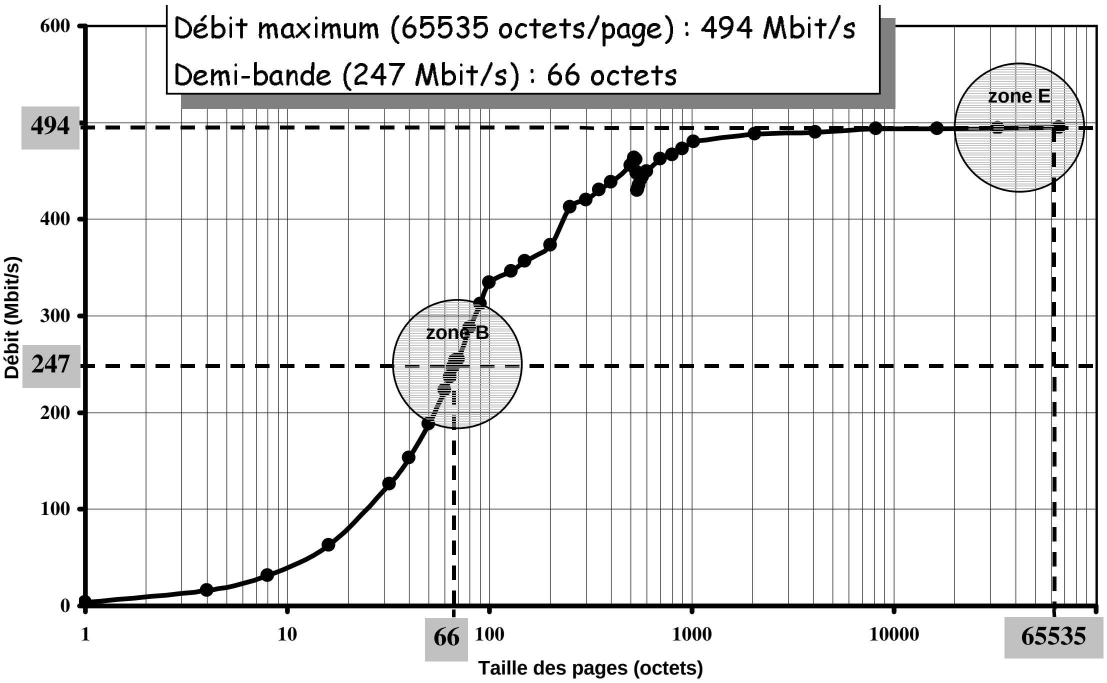


Fig. 9.4 *Débit maximum et demi-bande*

Le débit maximum théorique, atteint à chaque discontinuité, est donné par :

$$D_{max} = \frac{S_{max}\, D_{DDC/R^3}}{S_h + S_{max}} = 502\, Mb/s \tag{9.2}$$

Le taux d'erreur avec la valeur mesurée n'est que de 1,6 %. Notre modèle nous fournit donc une très bonne approximation de cette grandeur.

9.3.4 Couplage matériel/logiciel

Sur le graphe 9.5, tracé avec une échelle linéaire, on obtient confirmation de la linéarité du débit pour de petits paquets, comme on l'avait envisagé précédemment. La zone de linéarité A s'étend de 1 octet utile par paquet à 98 octets utiles par paquet. S'en suit alors une rupture de pente qui débute une zone non linéaire.

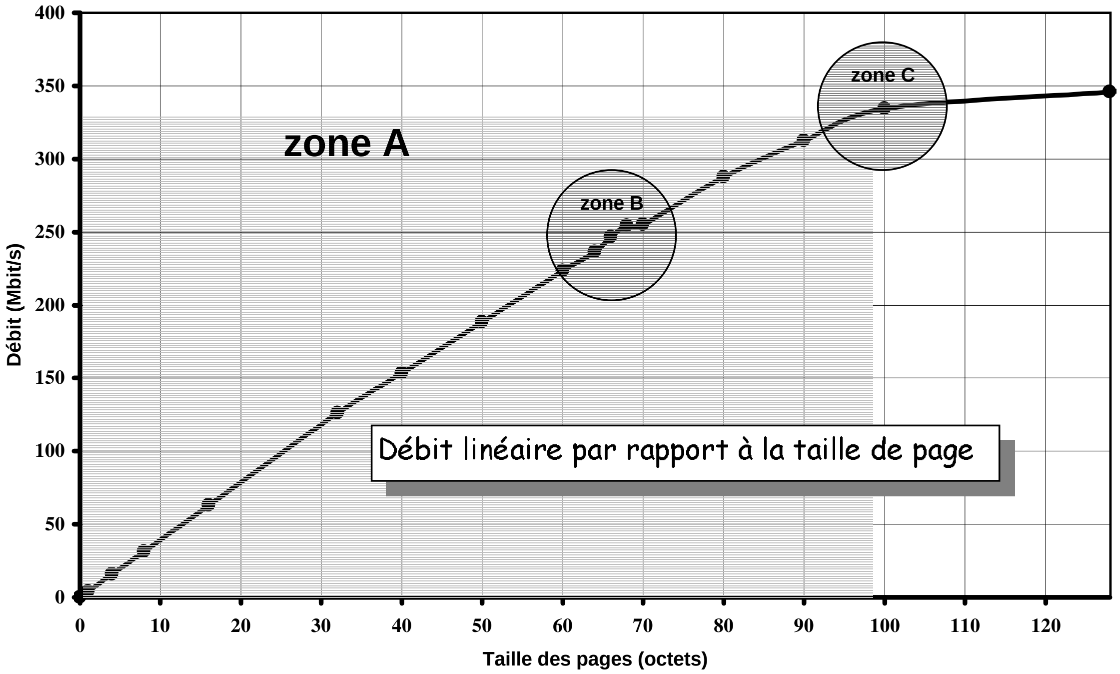


Fig. 9.5 *Zone A : débit linéaire par rapport à la taille de page*

Pour expliquer ce phénomène, on va considérer deux modes de fonctionnement d'un nœud de la machine :

❶ Dans le premier mode, la LPE n'est jamais vide. Il suffit pour cela que la tâche demande très souvent des émissions, ou bien que les tailles de pages à émettre soient suffisamment grandes pour que la tâche ajoute au moins un nouvelle page dans la LPE avant l'expiration du délai de transmission. Rappelons que dans le cadre des mesures de débit que nous sommes en train d'analyser, la tâche émettrice est constituée d'une simple boucle en mode noyau, qui se contente à chaque étape d'invoquer la fonction d'émission de PUT afin d'envoyer une page, c'est-à-dire un tampon de données de taille s. Pour être dans le premier mode, il faut et il suffit que le délai pendant lequel PCI-DDC émet une page de taille s soit plus grand que le délai d'une itération de la boucle, c'est-à-dire plus grand que le délai cumulé d'un appel à la fonction d'émission de PUT (1,9 μs) et du surcoût lié à l'implémentation de la boucle : gestion d'un compteur, test de rebouclage (délai estimé : 100 ns).
Comme la LPE n'est jamais vide, le débit sur le réseau ne dépend que de PCI-DDC. Le temps pour dérouler l'appel à PUT n'influence pas le résultat. Nous dirons donc que ce mode de fonctionnement correspond, dans le cadre des performances, à un Découplage Logiciel/Matériel.

❷ Dans le deuxième mode, PCI-DDC a toujours fini sa transmission avant qu'un nouvelle demande d'émission lui soit signalée. Il y a donc, à un instant donné, une entrée au plus dans la LPE. Avec la tâche qui effectue des émissions en boucle, cela correspond à une durée de boucle supérieure à la durée d'émission d'une page de taille s, donc à des pages très petites. On enverra donc s octets pendant la durée, constante, d'exécution de la fonction `put_add_entry()`, tant qu'on sera dans ce deuxième mode. **Le débit est donc proportionnel à la taille s de la page. Il s'agit donc d'un mode dans lequel les performances de la machine dépendent non seulement du matériel, mais aussi du logiciel. Nous dirons donc que ce mode de fonctionnement correspond, dans le cadre des performances, à un** COUPLAGE LOGICIEL/MATÉRIEL.

9.3.5 Modèle du débit

La valeur du débit dans l'expression 9.1 page 193 est valide lorsque PCI-DDC a toujours au moins une page à traiter, c'est à dire en phase de découplage.

Notons $\tau_{put} = 1{,}9\ \mu s$, la durée d'exécution de la fonction `put_add_entry()`. En phase de couplage, pour un tampon d'émission de s octets, il faut un délai τ_{put} afin d'émettre s octets. Le débit est donc s/τ_{put}. Cette formule est bien compatible avec la linéarité constatée sur notre courbe de mesures expérimentales dans la zone A.

Ainsi, quelle que soit la phase, la valeur du débit est la suivante :

$$\mathcal{D}_{\mathbf{utile}}(\mathbf{s}) = \min\{\frac{\mathbf{s}\ \mathbf{D_{DDC/R^3}}}{[\mathbf{E}(\frac{\mathbf{s-1}}{\mathbf{S_{max}}}) + \mathbf{1}]\mathbf{S_h} + \mathbf{s}}, \frac{\mathbf{s}}{\tau_{\mathbf{put}}}\} \tag{9.3}$$

Le changement de pente constaté sur la courbe expérimentale s'explique donc par la forme de cette formule : quand s s'accroît depuis la valeur 1, le débit va commencer par prendre la valeur du deuxième terme, puis dès l'instant de découplage, il va prendre la valeur du premier terme, on va donc distinguer sur la courbe un brusque changement de pente.

On remarque d'après cette formule que le débit utile n'est pas influencé par le débit sur le bus PCI. Cela s'explique par le goulot d'étranglement qui se situe au niveau du lien reliant RCube à PCI-DDC, et non au niveau du bus PCI.

Le point de découplage correspond aux conditions où il faut exactement un délai τ_{put} pour émettre une page, c'est-à-dire pour la valeur $s_{decoupl}$ vérifiant l'équation suivante :

$$\frac{s_{decoupl}\ D_{DDC/R^3}}{[E(\frac{s_{decoupl}-1}{S_{max}}) + 1]S_h + s_{decoupl}} = \frac{s_{decoupl}}{\tau_{put}} \tag{9.4}$$

Sur la courbe expérimentale, $s_{decoupl}$ vaut 98 octets au point de découplage. Il est donc inférieur à S_{max}. Calculons dans ces conditions la valeur théorique de $s_{decoupl}$. L'équation 9.4 se réécrit comme suit :

$$\frac{s_{decoupl}\, D_{DDC/R^3}}{S_h + s_{decoupl}} = \frac{s_{decoupl}}{\tau_{put}} \tag{9.5}$$

La solution de cette équation est la suivante :

$$s_{decoupl} = \tau_{put}\, D_{DDC/R^3} - S_h = 104\, octets \tag{9.6}$$

Le taux d'erreur avec la valeur mesurée est de 5,7 %. Notre modèle nous fournit donc une approximation cohérente de cette grandeur.

Pour comparer nos mesures aux résultats théoriques, trois courbes ont été tracées sur la figure 9.6 page suivante :

- ▷ La courbe indiquée *Mesures* dans la légende reprend les résultats expérimentaux.
- ▷ La courbe indiquée *Modèle PUT* représente le tracé de la fonction $\mathcal{D}_{utile}$ tel qu'il est exprimé dans la formule générale 9.3 page précédente. L'allure de cette courbe est identique à celle de la courbe des mesures réelles. La seule différence observable est la suivante : vers $s = 70$ octets, c'est-à-dire pour un débit déjà relativement important d'environ 250 Mb/s, la courbe de débit mesuré se dissocie de la courbe théorique en suivant dès lors une ascension plus faible, mais les formes de ces deux courbes sont identiques. Pour comprendre ce phénomène, il faut se rappeler que notre modèle théorique a été construit en tenant uniquement compte des comportements logiciels et matériels au sein du nœud émetteur. Il faut tenir compte du composant RCube récepteur, de PCI-DDC récepteur, et du bus PCI à travers lequel les données vont transiter en direction des tampons de réception. Des phénomènes de rétropropagation d'engorgement vont se produire, si le récepteur n'est pas capable *à tout moment* d'absorber le débit qui est alors de 250 Mb/s et qui s'achemine vers le débit maximum de 497 Mb/s.
- ▷ La courbe indiquée *Modèle limite* représente le tracé de la fonction $\mathcal{D}_{utile}$ tel qu'il est exprimé dans la formule 9.1 page 193. Lors de l'établissement de cette formule, nous avions indiqué qu'elle n'était valable qu'en phase de découplage. Or il se trouve qu'en phase de découplage, c'est-à-dire pour $s > 104$ octets, elle est confondue avec la courbe précédente. Mais en examinant la formule générale 9.3 page précédente, on s'aperçoit que dans le cas où τ_{put} tend vers 0, les valeurs de $\mathcal{D}_{utile}$ explicitées dans les équations 9.1 et 9.3 tendent l'une vers l'autre (en convergence simple de fonctions ; on peut même montrer, mais cela sort du cadre de cette étude, qu'il y a convergence uniforme). Notre courbe *Modèle limite* représente donc, sur toute la plage de taille de pages possibles, la valeur théorique du débit maximal atteignable, et que l'on ne pourra jamais dépasser, même en optimisant au mieux le code de PUT ou en accélérant à l'infini les processeurs des nœuds de calculs.

Pour clore cette analyse, on va maintenant étudier à l'aide de la figure 9.7 page 198 le comportement du débit utile en fonction du paramètre τ_{put}. Plusieurs courbes de débit y sont représentées pour des valeurs de ce paramètre de 2, 4, 25, 50 et 100 μs.

Le délai τ_{put} traduit à la fois l'optimisation du code de PUT et les performances du processeur qui équipe le nœud de calcul. Pour un nœud de calcul tel que ceux de la

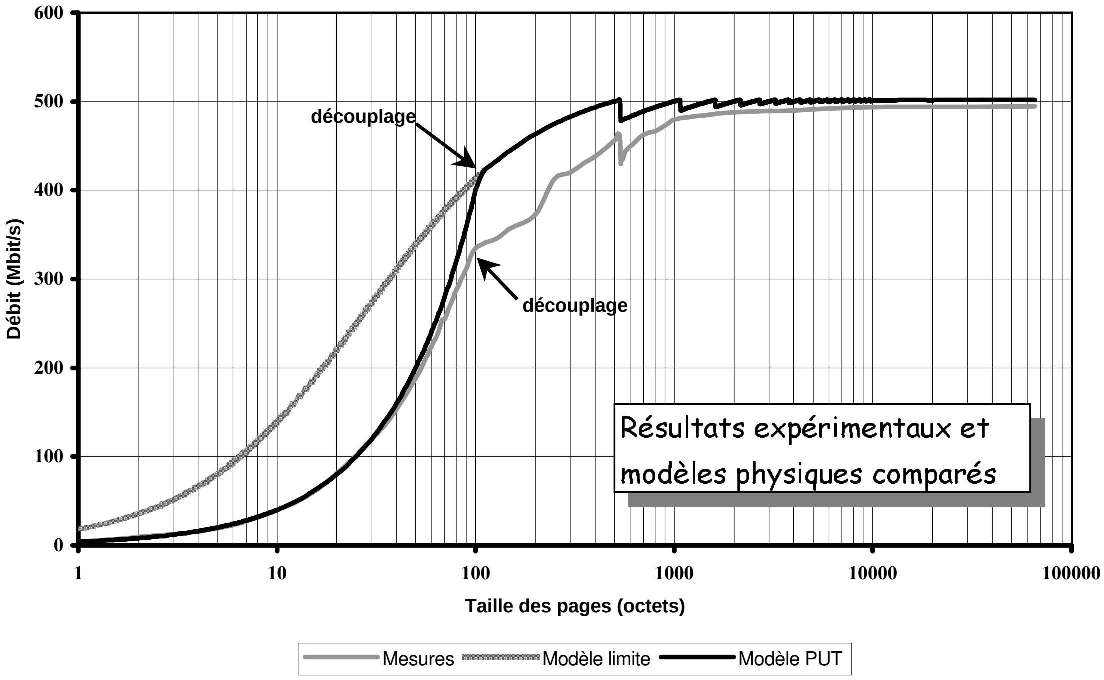


Fig. 9.6 *Mesures et modèle physique*

machine utilisée pour effectuer nos mesures, τ_{put} vaut environ 2 μs. Imaginons que, sur un tel nœud de calcul, on dispose d'une tâche dont l'algorithme particulier de calcul l'entraîne à effectuer une demande d'émission à PUT environ toutes les 23 μs. La courbe de débit mesurée sur le nœud de calcul correspondra alors à la courbe tracée pour une valeur de τ_{put} de 25 μs, alors que τ_{put} vaut toujours 2 μs. On comprend donc l'intérêt d'une telle abaque : elle peut aussi nous informer des performances comparées de diverses applications.

Pour conclure, on peut remarquer que c'est l'introduction de la notion d'**instant de découplage** qui nous a permis de définir la valeur théorique du débit d'émission en fonction de la taille de page, sur toute la plage de taille de page possible (1 à 65535 octets). Cette notion nous a aussi permis d'expliquer le phénomène physique constaté expérimentalement consistant en un changement brutal et inattendu de pente de la courbe de débit.

9.4 Modélisation du matériel

Afin de modéliser les interactions entre le matériel et la couche logicielle bas niveau PUT, nous avons, au sein des sections précédentes de ce chapitre, déterminé des variables caractéristiques des performances ou du fonctionnement de la machine MPC lors des communications. Ces résultats sont paramétrés à l'aide des débits aux interfaces entre les composants, du temps CPU consommé par la couche logicielle PUT, et de la valeur mesurée $S_{max} = 540$ octets, qui exprime la taille maximale d'un paquet construit par PCI-DDC.

Pour comprendre les phénomènes qui jouent un rôle dans le positionnement de cette constante, on a proposé un modèle du matériel. On présente, en annexe F page 249, cette

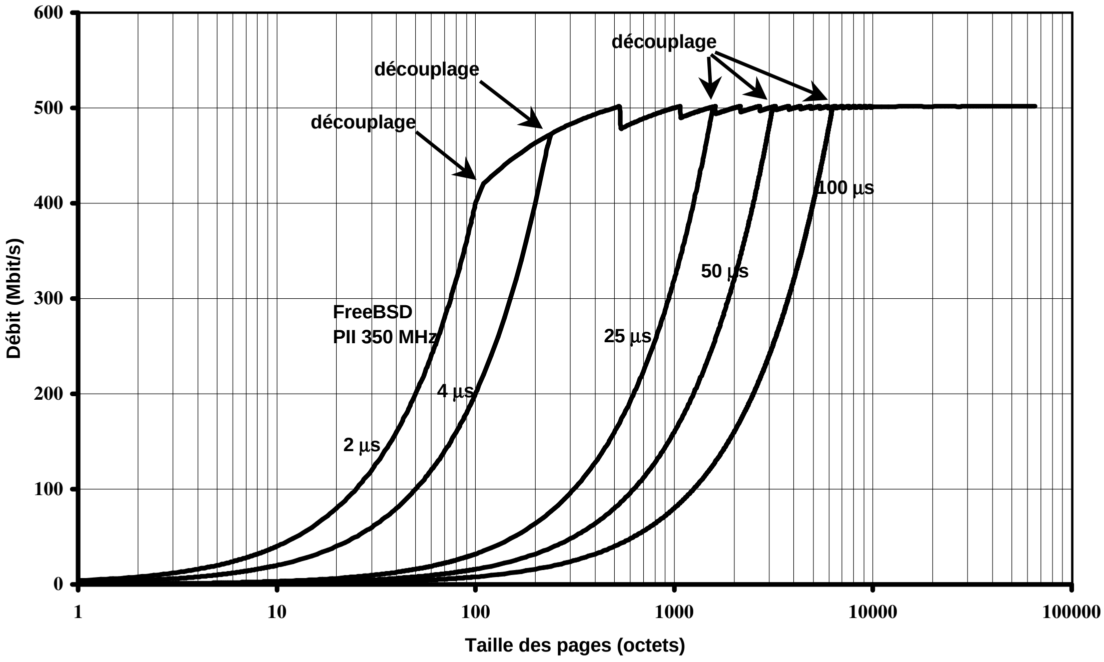


Fig. 9.7 *Débit théorique pour différentes valeurs du délai de boucle*

modélisation qui nous permet de retrouver une valeur approchée de la valeur mesurée de S_{max}.

9.5 Performances des couches sécurisées haut-niveau

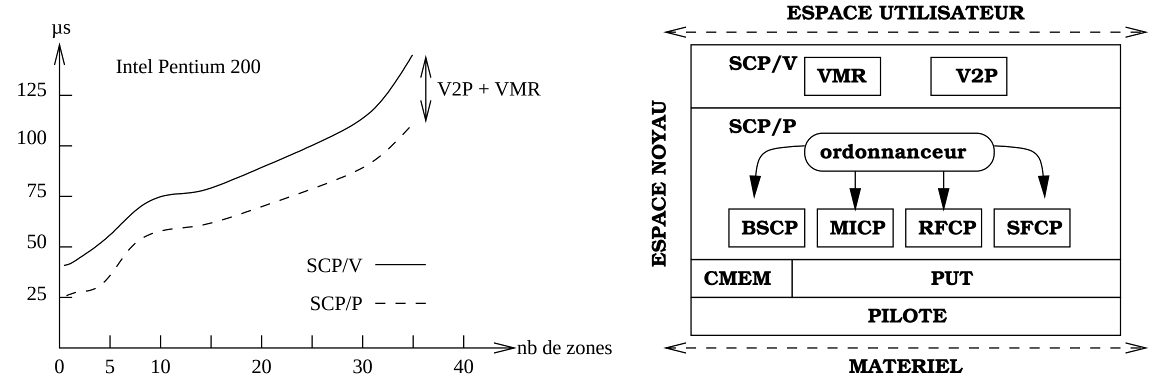
Jusqu'à maintenant, on a étudié en détails les performances de la couche PUT, en analysant notamment le fonctionnement interne du matériel. On va maintenant remonter dans l'empilement des protocoles, pour étudier les performances au niveau de l'utilisateur.

Les protocoles de plus haut niveau sont empilés sur la couche logicielle PUT. La figure 9.8 page suivante rappelle d'une part l'empilement des couches et modules implémentant les protocoles sécurisés noyau SCP/P et SCP/V, et présente d'autre part les mesures de performances effectuées sur ces couches : il s'agit du délai d'invocation de la fonction d'émission au niveau SCP/P et au niveau SCP/V, au sein d'un nœud de calcul.

Cette campagne de mesures a été effectuée sur un banc d'analyse de performances constitué de deux PC chacun équipé d'une carte FastHSL, d'un chipset Intel 440LX, de 64 Mo de mémoire vive ainsi que d'un processeur Intel Pentium cadencé à 200 MHz.

Ces protocoles ont été conçus pour éviter les copies des données des tampons d'émission ou de réception. La taille des données influe donc sur les performances uniquement à travers l'appel à la fonction d'émission de PUT. Celle-ci étant non bloquante, son temps d'exécution τ_{put} ne dépend pas de la taille des données, seul le débit utile généré sur le réseau HSL y est directement lié.

On comprend donc que la taille des données fournies à SCP/P et SCP/V influence le débit



délai de traversée des couches — empilement des couches SCP

Fig. 9.8 *Performances des couches sécurisées haut-niveau*

généré mais que le temps d'exécution $\tau_{scp/p}$ d'un appel à SCP/P ou $\tau_{scp/v}$ d'un appel à SCP/V est indépendant de la taille des données.

La fonction d'émission de SCP/P, couche de communication travaillant en adressage physique, reçoit en paramètre la cartographie de la mémoire physique constituant le tampon d'émission local, et connaît d'autre part la cartographie de la mémoire physique constituant le tampon de réception du nœud distant.

Elle va donc devoir analyser les deux cartes mémoire afin de distinguer les zones à transférer qui sont contiguës en émission comme en réception. Pour chacune de ces zones, SCP/P devra effectuer une invocation de PUT, et gérer des tables d'état indexées sur les numéros de ces zones. C'est pour cela que l'échelle du graphe 9.8 est graduée selon le nombre de zones contiguës à transférer. La longueur de chacune de ces zones ne joue pas dans le délai pour invoquer la fonction d'émission PUT, seul le nombre de ces zones y participe.

Les fonctions d'émission de SLR/P et SLR/V étant non bloquantes, à leur sortie, PUT a été invoquée, éventuellement plusieurs fois, donc la LPE n'est plus vide, et PCI-DDC a du travail. On peut donc se reporter à l'étude sur la mesure de débit de la couche de plus bas niveau, section 9.3 page 190 afin d'obtenir des informations sur le comportement de la machine à partir de cet instant là. C'est la raison pour laquelle on ne s'occupe ici que des délais de traversée des couches.

Comme on peut le constater sur le schéma d'empilement des couches de protocoles, SCP/V est constitué de deux sous-modules qui vont être invoqués avant l'appel à SCP/P :

✓ Le sous-module VMR[1] : son rôle est de garantir que les tampons d'émission et de réception sont toujours présents au sein de la mémoire physique, et notamment jamais paginés, lors d'un transfert. Pour cela il les référence au sein d'une carte mémoire de MACH particulière.
  Plutôt que de libérer la carte mémoire à chaque fin de transaction, une opération de ramasse-miette se charge de la mettre à jour régulièrement. Le coût de la couche VMR en est largement réduit : il faut 6 μs sur un Pentium 200 MHz pour vérifier qu'une zone en mémoire de 4 Ko est déjà référencée dans la carte mémoire de VMR,

1. Virtual Memory Referencer

alors qu'il faut plus de 200 μs pour l'y insérer.

✓ Le sous-module V2P[2] : son rôle est de décortiquer la carte mémoire d'un processus afin de fournir à SCP/P l'ensemble des zones physiques qui forment le tampon dont la localisation virtuelle a été passée en paramètre à SCP/V.

Il faut noter que si les zones de mémoire utilisées pour contenir les tampons d'émission et de réception ont été attribuées par CMEM, les performances, quelles que soient la longueur des données, correspondront à la valeur de l'ordonnée à l'origine sur notre graphe de délai.

9.6 Conclusion

On constate sur la courbe de délai de traversée des couches que les performances d'une application utilisant SCP/P ou SCP/V seront largement moins intéressantes qu'une application utilisant directement PUT. Mais les services rendus par ces différentes couches n'ont évidemment pas la même valeur. Leur facilité de mise-en-œuvre non plus.

Les premières applications qui ont vu le jour sur la machine MPC, ont été développées, pour ces raisons de facilité de mise-en-œuvre, à l'aide des canaux virtuels. Il s'agit par exemple du système de pagination distribué MAÎS [Cadinot *et al.*, 1997], développé sur SLR/P, ou du portage optimisé de PVM sur la machine MPC, développé sur SLR/V.

Les performances mesurées de cette implémentation de PVM ont été plutôt décevantes, et ont conduit à remettre en cause le choix de la simplicité de programmation. Des tentatives de développement directement sur PUT ont alors été tentés, en vue d'améliorer les performances, et ont aboutit au récent portage de MPI sur PUT, par un doctorant du LIP6.

2. Virtual address to Physical address

Chapitre 10

Analyse stochastique du couplage de pannes

Sommaire

Jusqu'à présent, on a cherché à corriger tous les types de fautes, ce qui impose un protocole sécurisé relativement complexe. On va maintenant aborder une approche profondément différente, pour montrer qu'un protocole de correction d'erreur capable de gérer uniquement les cas les plus simples peut être satisfaisant à condition que les cas complexes soient suffisamment rares. On va pour cela s'intéresser au domaine de la fiabilité des systèmes redondants, et plus précisément à l'analyse stochastique du couplage des pannes. En appliquant des méthodes usuelles du domaine des probabilités, on va quantifier, par des calculs simples, l'apparition des différents types de fautes. Une application numérique nous montrera enfin que dans le cas de la machine MPC, les fautes complexes sont tout à fait improbables.

10.1 Caractérisation des fautes fréquentes

Les expérimentations menées sur la machine MPC du LIP6 ont montré que les erreurs se manifestent toujours par une perte de calibration d'un lien HSL. Les autres types d'erreurs, théoriquement possibles, sont suffisamment rares pour n'avoir à ce jour jamais été observés.

Une perte de calibration peut se produire alors qu'aucune donnée ne traverse le réseau, et dans un tel cas les nœuds de calcul ne sont pas informés de l'incident.

Dans le cas contraire, un paquet en transit va se trouver partiellement ou totalement détruit et RCube va ajouter un marqueur *fin de paquet exceptionnelle*. Une interruption se produit alors dans le nœud destinataire et PCI-DDC gèle volontairement toutes ses activités.

Une fois relancé par le logiciel, PCI-DDC va reprendre les opérations normalement, sauf pour ce qui concerne la mise à jour de la LMI. En effet, un paquet manquant à l'appel, le MI du message associé ne sera donc jamais déposé. De plus, si le paquet touché par la perte de calibration était le dernier du message en cours de transfert, le MI du message suivant pourra alors lui aussi se trouver absent de la LMI (il s'agit du cas de perte d'un paquet correct, que nous avons déjà étudié au cours de la description du comportement du matériel en cas de faute quelconque du réseau).

Pour résumer le comportement du matériel en cas de perte de calibration, on peut distinguer trois étapes successives :

❶ PCI-DDC se fige, la LMI contient le MI du dernier message complètement reçu ;

❷ PCI-DDC est réactivé mais omet de signaler dans la LMI un ou deux messages successifs ;

❸ PCI-DDC retrouve un fonctionnement normal et insère un numéro de MI dans la LMI.

10.2 Protocole de correction simpliste

Pour une plus simple compréhension du protocole que nous allons maintenant décrire, faisons tout d'abord abstraction des multiples émetteurs potentiels vers un récepteur donné, et imaginons pour cela que la machine MPC est composée seulement de deux nœuds.

On peut facilement imaginer un moyen pour détecter les données perdues qui font suite au déroulement du scénario qu'on vient de décrire : quand PCI-DDC indique par une interruption qu'il vient de se figer, PUT va consulter la LMI et note le dernier MI reçu. Il réactive alors PCI-DDC, puis dès l'interruption suivante indiquant une écriture dans la LMI, PUT consulte la nouvelle entrée de cette table. Il dispose donc alors des deux MI encadrant le ou les messages perdus, et peut en informer l'émetteur.

Celui-ci doit pouvoir retrouver dans sa LPE la trace des messages à envoyer de nouveau, à l'aide uniquement des deux MI qu'il vient de recevoir. Il peut pour cela se réserver quelques bits dans chaque MI pour numéroter les messages dans l'ordre croissant d'émission (les autres bits constituant le MI doivent rester à la disposition de l'utilisateur de PUT). Ainsi, l'encadrement suffit pour retrouver tous les messages perdus, et donc les réémettre.

Ce protocole simpliste que l'on vient de décrire succinctement pose trois problèmes :

- ▷ On a considéré une machine MPC à deux nœuds. À partir de trois nœuds, des réceptions simultanées de messages différents peuvent se produire. Un émetteur, qui reçoit une demande de réémission d'une plage de MI, doit donc rechercher dans sa LPE uniquement les messages à destination du nœud signalant la perte ;
- ▷ D'autre part, le récepteur doit pouvoir lui aussi, au moment de la panne, détecter pour chaque entrée de LMI le nœud émetteur associé. Pour cela, il faut réquisitionner encore quelques bits dans chaque MI pour indiquer le numéro de nœud émetteur ;
- ▷ Comme chaque message est potentiellement destiné à être réémis, il faut prévoir un mécanisme pour informer un émetteur de la bonne réception d'un message et introduire un retard avant de signaler les fins d'émission. On peut par exemple pour cela demander à chaque récepteur d'envoyer régulièrement un message court indiquant le numéro du dernier MI de sa LMI.

Il est important de constater que le protocole simpliste qu'on esquisse ici n'est correct que si aucune faute ne se produit lors de la réparation d'une autre faute. Dans le cas contraire, le protocole ne saura pas corriger les deux fautes rapprochées, et le comportement de la machine deviendra alors imprédictible.

On pourrait être tenté d'imaginer un protocole tolérant aux fautes multiples et simultanées : cela nécessiterait alors d'y inclure des mécanismes de détection des pertes des paquets de contrôle, de réémission de ces paquets, de la détection des doublons, etc. Le protocole deviendrait alors très complexe.

On a observé expérimentalement le caractère transitoire des fautes du réseau : elles sont exceptionnelles et l'indisponibilité est de très courte durée. On peut donc espérer que la double-faute se produit rarement, et dans un tel cas le protocole à implémenter reste simpliste. **C'est cette hypothèse qu'on va s'employer à vérifier formellement dans ce chapitre**, en utilisant des techniques simples de calcul des probabilités.

10.3 Enjeux

La quantification du phénomène de double-faute, au delà de son application à notre protocole simpliste de contournement des fautes du réseau HSL, peut s'appliquer à toutes sortes de protocoles de communication réseau.

Le contournement des fautes de transmission est bien souvent un problème suffisamment complexe pour qu'il soit écarté d'emblée dans le cadre des réseaux de communication des machines hautes performances. En effet, intégrer dans le matériel un algorithme de contournement des fautes est dans la plupart des cas irréalisable, et le contournement logiciel entraîne une baisse de performance.

Mais à partir du moment où l'on peut montrer que les conditions sont réunies pour que la double-faute soit suffisamment rare, on peut implanter dans le matériel un protocole de correction simpliste des fautes, ou proposer un tel protocole logiciel à faible coût.

10.4 Notions de base en fiabilité

10.4.1 Choix de l'approche probabiliste

L'étude poussée et systématique des problèmes de fiabilité a réellement commencé au cours des années 1960, grâce aux résultats de la Recherche Opérationnelle sur les processus stochastiques. Les premières études autour du problème particulier de la double-faute ont été réalisées par les mathématiciens russes, les apports les plus importants ayant été fournis par Soloviev et Gnédenko.

Au début des années 1990, une approche théorique non probabiliste du problème du couplage des fautes s'est développée dans le cadre de l'étude du test logiciel (cf. les récents travaux de K.S. How Tai Wah [Wah, 2000]).

Bien que le protocole de réparation esquissé précédemment soit suffisamment simple pour qu'on puisse facilement en décrire le comportement de manière quasi-systématique (il est par exemple facile de calculer une valeur approchée du surplus de débit ou de latence imposé par ce protocole), le comportement du réseau est plus complexe à analyser car nous ne pouvons que donner des hypothèses pour expliquer l'apparition des fautes. Ne connaissant pas exactement les causes de l'apparition d'une panne du réseau, on va se contenter d'une description probabiliste : on peut représenter les fautes du réseau par un processus stochastique, qu'on ne connaît pas *a priori*, mais dont on peut facilement estimer les caractéristiques comme par exemple la moyenne et la variance, par une simple observation de la machine en cours de fonctionnement.

Pour cette raison, on a choisi d'aborder le problème sous l'angle des probabilités.

10.4.2 Analogie avec la méthode de la réservation

Nous allons maintenant introduire le vocabulaire destiné à décrire notre problème. Remarquons pour cela que la double-faute est un problème analogue à celui de la réservation, qui constitue l'une des principales méthodes d'élévation de la fiabilité.

La réservation consiste à adjoindre à un élément des éléments de réserve destinés à le remplacer en cas de panne. On nomme groupe de réserve l'ensemble constitué de l'élément principal et des éléments de réserve.

Il y a trois types de réservation :

- ✓ la réservation chargée, pour laquelle les éléments de réserve se trouvent au même régime que l'élément principal (ils s'usent donc même au repos) ;
- ✓ la réservation non chargée, pour laquelle les éléments en réserve ne s'usent pas ;
- ✓ la réservation allégée, pour laquelle les éléments en réserve se trouvent à un régime allégé.

Pour chacun de ces types de réservation, on distingue le mode avec renouvellement pour lequel les éléments en panne sont réparés, et le mode sans renouvellement.

Il va de soi que le problème de double-faute du réseau est analogue au problème de la réservation non chargée avec renouvellement, et groupe de réserve se composant uniquement de deux éléments identiques. On nomme le groupe de réserve un *couple* et on parle alors de *doublage avec renouvellement.*

Deux processus stochastiques vont alors être nécessaires pour étudier la double-faute : le premier représente les instants d'arrivée de fautes du matériel, et le second les durées de réparation. La durée d'une réparation est le délai pendant lequel le protocole de contournement de faute a dû fonctionner suite à une erreur matérielle sur un lien HSL. Le chronomètre se déclenche au moment où la faute se produit, et s'arrête lorsque les réémissions ont été effectuées. Les pannes et réparations se succèdent jusqu'au moment où une panne apparaît alors que la réparation précédente n'est pas terminée. Le couple a alors généré une double-faute, et le temps pour l'atteindre est représenté par une variable aléatoire.

On utilise donc deux processus stochastiques, décrivant les instants de pannes et les durées de réparation, pour définir une variable aléatoire représentant les doubles-fautes. **Les variables aléatoires qui composent ces deux processus sont supposées mutuellement indépendantes.**

10.4.3 Les résultats de Gnédenko

Les *processus de naissance et de mort* constituent un outil privilégié pour décrire des mécanismes analogues aux fautes multiples. Des résultats généraux peuvent ainsi être fournis dans le cadre du doublage sans renouvellement.

Néanmoins, dans le cas qui nous intéresse, c'est-à-dire dans le cadre du doublage avec renouvellement, il nous faut absolument considérer le cas particulier du couple car c'est le seul moyen pour établir des résultats littéraux, impossibles à obtenir avec un groupe de réserve de cardinal quelconque.

Dans un article publié en 1972, Gnédenko [Gnédenko *et al.*, 1972] étudie par récurrence la loi de la variable aléatoire de double-faute, ce qui lui permet d'établir une équation-intégrale. Pour la résoudre, il utilise naturellement la transformation de Laplace, ce qui lui fournit une expression littérale de la transformée de Laplace de la densité de cette variable aléatoire, quelles que soient les lois des instants d'arrivées et celles des durées de réparation.

Il est rarement facile de retrouver une expression littérale de la distribution à partir de sa transformée de Laplace (souvent moins facile que pour d'autres transformations intégrales). L'expression obtenue par Gnédenko n'est donc pas le meilleur outil pour découvrir la distribution de double-faute qui se cache derrière. Néanmoins, cette expression possède d'autres intérêts : on peut notamment en déduire facilement l'espérance de la variable aléatoire de double-faute, c'est-à-dire le temps moyen avant double-faute, que nous appellerons **MTBF du couple**.

Quelques années après l'établissement de ce premier résultat, Gnédenko et Soloviev s'attaquèrent à la résolution de l'équation intégrale caractérisant la loi de la double-faute, et aboutirent à un théorème limite qu'on utilisera dans l'application pratique de la fin de ce chapitre pour fournir une approximation de la fonction de répartition de la double-faute.

Une synthèse des principaux résultats issus des travaux de Gnédenko et Soloviev sur la fiabilité est présentée dans un article récent de Korolyuk [Korolyuk, 1997], ancien élève de Gnédenko.

10.5 Résultats

Voici l'énumération des résultats qu'on va déduire de calculs simples dans la suite de ce chapitre. Les calculs fastidieux sont rejetés en annexe G page 255, tandis que les résultats établis rapidement sont présentés dans le corps du chapitre, en souhaitant intéresser ceux des lecteurs informaticiens qui ne manipulent pas régulièrement les probabilités.

Nous avons indiqué précédemment que pour calculer l'expression du MTBF du couple, Gnédenko a dû introduire la transformée de Laplace de la densité de la variable aléatoire décrivant les inter-arrivées, avec laquelle il définit une équation intégrale récurrente.

Nous commencerons par retrouver, par un calcul direct simple, l'expression du MTBF du couple.

Le quotient du délai moyen avant apparition d'une double-faute par le délai moyen d'inter-arrivées des fautes simples représente le *gain en sécurité* : c'est le facteur multiplicatif qui relie le MTBF du couple au MTBF d'un élément. Nous expliciterons le comportement asymptotique du gain en sécurité quand le processus des fautes suit une loi sans mémoire, tout ceci sans restriction sur le processus de réparation, afin d'en tirer une valeur approchée simple du MTBF du couple.

On calculera alors, toujours par une méthode directe, la variance de la double-faute. Ce résultat, utilisé conjointement à la valeur du MTBF du couple calculée précédemment, nous permettra, lors des applications numériques à la fin de ce chapitre, de calculer des intervalles de confiance.

Enfin, avant d'effectuer une application numérique à la machine MPC, on montrera la continuité de la double-faute, ce qui justifiera l'utilisation de mesures approchées pour obtenir une estimation de la valeur numérique de la double-faute.

10.6 Calculs d'erreurs

10.6.1 Références

Le lecteur qui désire une présentation des concepts de base utilisés par la suite pourra se référer aux ouvrages suivants :

- ✓ les notions primaires en probabilités sont fournies par [Foata and Fuchs, 1998], et [Revuz, 1997] ; pour les aborder dans de bonnes conditions, quelques notions de topologie présentées dans [Wagschal, 1998] peuvent être utiles ;
- ✓ Dans [Cocozza-Thivent, 1997], de très nombreux résultats sur la fiabilité des systèmes sont obtenus en manipulant des processus stochastiques. Les prérequis pour aborder cet ouvrage sont par exemple introduits dans [Kleinrock, 1975] ;
- ✓ Les aspects fondamentaux des probabilités sont passés en revue dans [Shiryaev, 1989], et l'on y trouve notamment une présentation des notions de convergence et des théorèmes limites qu'on utilisera plus loins dans ce chapitre ;
- ✓ Un état de l'art des probabilités modernes est présenté dans [Kallenberg, 1997], et [Ionescu *et al.*, 1999] regroupe de nombreux travaux récents en fiabilité.

10.6.2 Modèle mathématique

On se propose de définir un modèle mathématique représentant l'apparition de fautes matérielles, et d'en déduire un certain nombre d'informations sur la distribution des doubles-fautes en fonction des caractéristiques du protocole de correction d'erreur mis en œuvre.

Pour ce modèle, un formalisme plus rigoureux est présenté en annexe G.1 page 255.

Imaginons que nous puissions être confrontés à un ensemble Ω d'expériences aléatoires. Chaque expérience, qui commence au temps $t = 0$, consiste à observer le comportement d'une machine MPC constituée de deux nœuds reliés par un unique câble HSL. Ω constitue donc l'ensemble des épreuves.

Pour décrire les instants de faute matérielle, on introduit la suite de variables aléatoires A, constituée de la famille $A = (A_i, i \in \mathbb{N})$ de variables aléatoires. Pour abréger, on écrira dans la suite *processus* A, plutôt que *suite de variables aléatoires* A, ces suites constituant un cas particulier de *processus stochastique*.

La suite (A_i) est strictement croissante car elle décrit les instants successifs de panne.

On peut en déduire le processus X des délais d'inter-arrivées des pannes comme suit : $X = (X_i, i \in \mathbb{N}) = (A_i - A_{i-1}, i \in \mathbb{N})$ avec la notation A_{-1} pour désigner une variable aléatoire nulle.

Nous disposons d'un protocole qui permet, en cas de faute simple, de recouvrer le bon fonctionnement de la machine. Définissons les délais de réparation à chacun des instants de panne par le processus $Y = (Y_i, i \in \mathbb{N}^*)$. Y_i désigne le délai de réparation de la faute qui s'est produite à l'instant A_{i-1}.

Avec les cartes FastHSL de troisième génération, l'expérience nous a montré que le temps moyen entre deux pannes est inférieur à l'heure et est caractérisé par une variance finie. Ces quantités dépendent néanmoins de l'application en cours, ou plus précisément de la charge réseau imposée par cette application : une erreur de lien qui se produit alors qu'aucun paquet ne transite sur le réseau est sans conséquence, et passe donc inaperçue.

Le processus X va représenter les délais d'inter-arrivées de pannes avec une des cartes de troisième génération, dont on vient de signaler qu'elles sont caractérisées par des inter-arrivées possédant une espérance et une variance finies. On va cependant travailler dans un cadre général, sans rien supposer sur les moments des variables en jeu.

L'expérience montre que les erreurs sur les liens sont suffisamment séparées dans le temps pour pouvoir supposer que les variables aléatoires qui composent X sont indépendantes et identiquement distribuées.

Par essence, le fonctionnement du protocole de compensation des fautes matérielles nous permet d'affirmer que les variables aléatoires qui composent Y sont elles-aussi indépendantes et identiquement distribuées.

Enfin, l'apparition d'une faute n'a aucune raison d'être liée de quelque manière que ce soit au protocole de correction d'erreur. On va donc supposer que les variables aléatoires composant X et Y sont mutuellement indépendantes.

10.6.3 Construction du processus de panne

Une double-faute se produit quand une panne survient alors que le protocole de correction d'erreur est en pleine activité, afin de compenser la faute précédente.

Les processus A et Y étant fixés, il nous faut construire la variable aléatoire définissant l'instant de première double-faute.

Définition 1 Définissons l'application η de Ω dans $\overline{\mathbb{N}^*}$, qui a toute épreuve ω associe l'indice de la première panne double :

$$\forall\, \omega \in \Omega, \eta(\omega) = \begin{cases} +\infty & \text{si } \forall\, n \in \mathbb{N}^*, X_n(\omega) > Y_n(\omega), \\ min\{n \in \mathbb{N}^*, X_n(\omega) \leqslant Y_n(\omega)\} & \text{dans le cas contraire.} \end{cases}$$

Définissons à l'aide de η l'application Z de Ω dans $\overline{\mathbb{R}}$, qui à toute épreuve ω associe le premier délai de panne double :

$$\forall\,\omega \in \Omega,\, Z(\omega) = \begin{cases} A_{\eta(\omega)}(\omega) & \text{si } \ \eta(\omega) < +\infty, \\ +\infty & \text{dans le cas contraire.} \end{cases}$$

On a construit Z à partir de A, X et Y. Sachant que A peut s'exprimer à partir de X, on en déduit que Z dépend uniquement de X et Y.

Définition 2 Notons Ψ l'application qui à tout couple (X, Y) de processus stochastiques associe $\Psi_{X,Y} = Z$ selon la construction précédente.

Construite de cette manière, Z est une variable aléatoire réelle. Notons que toute construction d'une fonction opérant sur un ensemble d'événements n'aboutit pas forcément à une variable aléatoire. En effet, des propriétés de mesurabilité particulières sont nécessaires. La démonstration est évidente pour le probabiliste, mais on la présente néanmoins en annexe G.2 page 256 pour l'informaticien curieux.

La figure 10.1 présente, pour un événement donné, le calcul de la double-faute.

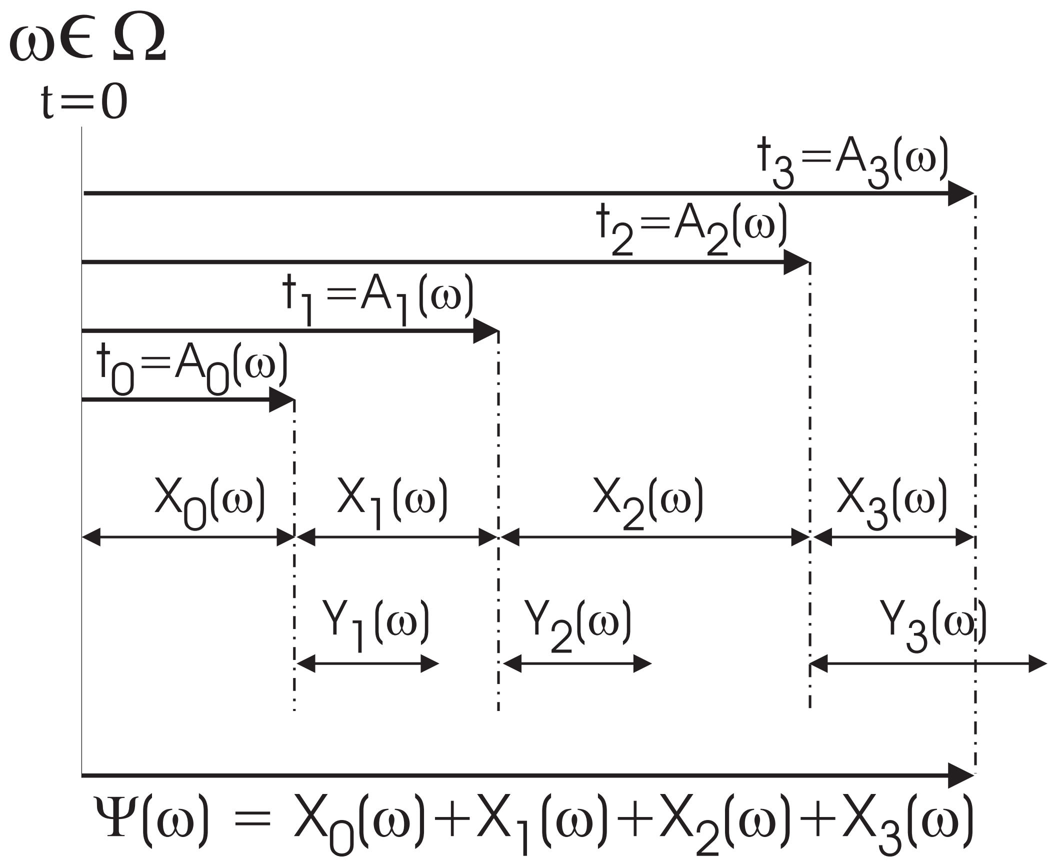


Fig. 10.1 *Exemple de double-faute*

10.6.4 Calcul du temps moyen avant double-faute

Définissons $(\Delta_n, n \in \mathbb{N}^*)$, $(\Lambda_n, n \in \overline{\mathbb{N}^*})$ les parties de Ω suivantes :

$$\forall\, n \in \mathbb{N}^*,\ \Delta_n = \{\omega \in \Omega \,/\, X_n > Y_n\}$$
$$\forall\, n \in \mathbb{N}^*,\ \Lambda_n = \{\omega \in \Omega \,/\, \forall i \in \{1,\cdots,n-1\}, X_i > Y_i \text{ et } X_n \leqslant Y_n\}$$
$$\Lambda_\infty = \{\omega \in \Omega \,/\, \forall i \in \mathbb{N}^*, X_i > Y_i\}$$

On va maintenant étudier l'espérance mathématique de la double-faute, soit $E(\Psi_{X,Y})$.

Remarquons tout d'abord que si $P(X_1 > Y_1) < 1$, alors $P(\Lambda_\infty) = 0$.
En effet, pour tout entier n non nul, $\Lambda_\infty \subset \{\omega \in \Omega \,/\, \forall i \in \{1,\cdots,n\}, X_i > Y_i\}$, donc $P(\Lambda_\infty) \leqslant P(X_1 > Y_1)^n$. Ceci étant vérifié pour tout n, on a donc $P(\Lambda_\infty) = 0$.

D'autre part, si $P(X_1 > Y_1) = 1$, alors $\Psi_{X,Y} = +\infty$ p.s.[1]
On supposera donc dans la suite que $P(X_1 > Y_1) < 1$, seul cas intéressant.

On peut aussi remarquer qu'il découle des définitions que :
$\forall\, n \in \mathbb{N}, n \geqslant 2 \Rightarrow \Lambda_\infty \uplus (\uplus_{i \geqslant n}\Lambda_i) = \cap_{1 \leqslant i < n}\Delta_i$, et que $\Lambda_\infty \uplus (\uplus_{i \in \mathbb{N}^*}\Lambda_i) = \Omega$.

Pour s'en persuader, on peut consulter l'annexe G.3 page 259, qui propose une démonstration rigoureuse de ces deux propriétés.

Nous utiliserons dans les calculs qui vont suivre dans ce chapitre un résultat de base en probabilités :

Soit $d \in \mathbb{N}^*$, U et V deux variables aléatoires indépendantes, et B un borélien de $\mathbb{R}^d$ (par exemple un ouvert, ou un fermé). Alors, $\int_{U^{-1}(B)} V\, dP = E(V)\, P(U \in B)$.

On fournit en annexe G.3 page 259 une démonstration de cette propriété.

On va maintenant effectuer le calcul de $\mathbf{E}(\Psi_{\mathbf{X},\mathbf{Y}})$.

Sachant que $\Lambda_\infty \uplus (\uplus_{i \in \mathbb{N}^*}\Lambda_i) = \Omega$, on peut écrire :

$$\begin{aligned} E(\Psi_{X,Y}) &= \int_\Omega \Psi_{X,Y}\, dP = \int_{\Lambda_\infty \uplus (\uplus_{i\in\mathbb{N}^*}\Lambda_i)} \Psi_{X,Y}\, dP = \int_{\Lambda_\infty} \Psi_{X,Y}\, dP + \int_{\uplus_{i\in\mathbb{N}^*}\Lambda_i} \Psi_{X,Y}\, dP \\ &= \int_{\Lambda_\infty} \Psi_{X,Y}\, dP + \sum_{i=1}^{\infty} \int_{\Lambda_i} A_i\, dP = \sum_{i=1}^{\infty} \int_{\Lambda_i} A_i\, dP \end{aligned}$$

Calculons les sommes partielles des variables aléatoires X_i :

$$\forall\, n \in \mathbb{N},\ \sum_{i=0}^{n} X_i = \sum_{i=0}^{n} (A_i - A_{i-1}) = A_n$$

En remplaçant A_i par cette expression dans notre calcul de l'espérance de $\Psi_{X,Y}$, on obtient :

$$E(\Psi_{X,Y}) = \sum_{i=1}^{\infty} \int_{\Lambda_i} \left[\sum_{j=0}^{i} X_j\right] dP = \sum_{i=1}^{\infty} \int_{\Lambda_i} \left[\sum_{j=1}^{i} X_j\right] dP + \sum_{i=1}^{\infty} \int_{\Lambda_i} X_0\, dP$$

1. p.s.: presque sûrement

Calculons le dernier terme, encore à l'aide de la remarque que $\Lambda_\infty \uplus (\uplus_{i \in \mathbb{N}^*} \Lambda_i) = \Omega$:

$$\sum_{i=1}^{\infty} \int_{\Lambda_i} X_0 \, dP = \int_{\uplus_{i \geqslant 1} \Lambda_i} X_0 \, dP = \int_{\Omega \setminus \Lambda_\infty} X_0 \, dP = E(X_0) - \int_{\Lambda_\infty} X_0 \, dP = E(X_0)$$

Réintroduisons la valeur évaluée du dernier terme de $E(\Psi_{X,Y})$, et faisons sortir la somme la plus imbriquée du premier terme (les X_j sont positifs ou nuls). On obtient alors :

$$E(\Psi_{X,Y}) = \sum_{i=1}^{\infty} \sum_{j=1}^{i} \int_{\Lambda_i} X_j \, dP + E(X_0)$$

Pour traiter le premier terme, on applique le théorème de Fubini, qui s'exprime ainsi : pour toute suite double de réels $(u_{i,j})_{(i,j) \in \mathbb{N}^{*2}}$, on peut écrire :

$$\forall (i,j) \in \mathbb{N}^{*2}, u_{i,j} \in \overline{\mathbb{R}^+} \Rightarrow \sum_{i=1}^{\infty} \sum_{j=1}^{i} u_{i,j} = \sum_{j=1}^{\infty} \sum_{i=j}^{\infty} u_{i,j}$$

Donc on peut écrire :

$$\begin{aligned} E(\Psi_{X,Y}) &= \sum_{j=1}^{\infty} \sum_{i=j}^{\infty} \int_{\Lambda_i} X_j \, dP + E(X_0) = \sum_{j=1}^{\infty} \int_{\uplus_{i \geqslant j} \Lambda_i} X_j \, dP + E(X_0) \\ &= \sum_{j=2}^{\infty} \int_{\uplus_{i \geqslant j} \Lambda_i} X_j \, dP + 2E(X_0) \end{aligned}$$

Sachant que $\forall\, n \in \mathbb{N}, n \geqslant 2 \Rightarrow \Lambda_\infty \uplus (\uplus_{i \geqslant n} \Lambda_i) = \cap_{1 \leqslant i < n} \Delta_i$, on obtient :

$$\begin{aligned} E(\Psi_{X,Y}) &= \sum_{j=2}^{\infty} \int_{(\cap_{1 \leqslant i < j} \Delta_i) \setminus \Lambda_\infty} X_j \, dP + 2E(X_0) \\ &= \sum_{j=2}^{\infty} \left[\int_{\cap_{1 \leqslant i < j} \Delta_i} X_j \, dP - \int_{\Lambda_\infty} X_j \, dP \right] + 2E(X_0) \\ &= \sum_{j=2}^{\infty} \int_{\cap_{1 \leqslant i < j} \Delta_i} X_j \, dP + 2E(X_0) \end{aligned}$$

Sachant que $\Delta_i = \{\omega \in \Omega \,/\, X_n(\omega) > Y_n(\omega)\}$ et que les X_n et Y_n sont mutuellement indépendants, on peut écrire :

$$\int_{\cap_{1 \leqslant i < j} \Delta_i} X_j \, dP = E(X_j)\, P(\cap_{1 \leqslant i < j} \Delta_i) = E(X_j) \prod_{i=1}^{j-1} P(\Delta_i) = E(X_1)\, P(\Delta_1)^{(j-1)}$$

On peut donc terminer le calcul de $E(\Psi_{X,Y})$:

$$E(\Psi_{X,Y}) = \sum_{j=2}^{\infty} E(X_1)\, P(\Delta_1)^{(j-1)} + 2E(X_0) = E(X_1) \left[1 + \frac{1}{1 - P(X_1 > Y_1)} \right]$$

On résume ce résultat par le théorème suivant :

Théorème 1 ESPÉRANCE D'UNE VARIABLE ALÉATOIRE DE DOUBLE-FAUTE

Hypothèses :
Soit $(\Omega,\mathcal{A},P)$ un espace de probabilité.
Soient $X = (X_i, i \in \mathbb{N})$ et $Y = (Y_i, i \in \mathbb{N}^*)$ deux processus stochastiques à valeur dans $(\overline{\mathbb{R}},\mathcal{B}(\overline{\mathbb{R}}))$, vérifiant :

- les variables aléatoires X_i et Y_i sont positives et mutuellement indépendantes,
- les variables aléatoires X_i sont identiquement distribuées, tout comme les variables aléatoires Y_i,
- $0 < E(X_0) < +\infty$.

Soit $\Psi_{X,Y}$, variable aléatoire définie sur l'espace probabilisé $(\Omega,\mathcal{A},P)$ et à valeur dans l'espace mesurable $(\overline{\mathbb{R}},\mathcal{B}(\overline{\mathbb{R}}))$, dont la construction a fait l'objet de la section 10.6.3 page 208.

Énoncé :
. Si $\mathbf{P(X_1 > Y_1) = 1}$, alors $\Psi_{X,Y} = +\infty$ p.s.
. Si $\mathbf{P(X_1 > Y_1) < 1}$, alors $\Psi_{X,Y}$ admet un moment d'ordre 1, qui vaut :

$$\mathbf{E}(\Psi_{\mathbf{X},\mathbf{Y}}) = \mathbf{E}(\mathbf{X_1}) \left[\mathbf{1} + \frac{\mathbf{1}}{\mathbf{1} - \mathbf{P}(\mathbf{X_1} > \mathbf{Y_1})}\right]$$

Pour comprendre le raisonnement mis en œuvre dans cette démonstration, on résume son principe comme ceci : pour calculer le délai moyen avant double-faute, on a effectué une classification des délais d'inter-arrivée de faute simple par un découpage hiérarchique à deux niveaux :

✓ on a constitué les groupes des délais d'inter-arrivée de panne simple qui interviennent dans les pannes doubles d'un indice donné ;

✓ on a sommé tous ces groupes sur l'ensemble de tous les indices d'arrivée de panne double.

On a alors appliqué le théorème de Fubini, ce qui a consisté simplement à inverser la hiérarchie de la classification :

✓ les groupes de plus bas niveau ont alors été constitués des délais d'inter-arrivée de panne simple d'un indice d'arrivée donné (par exemple, le *premier* groupe est constitué de l'ensemble des délais d'inter-arrivée de panne simple qui représentent la *première* panne simple d'une série de pannes aboutissant à une quelconque panne double) ;

✓ on a sommé tous ces groupes sur l'ensemble de tous les indices d'arrivée de panne simple.

On a alors montré que le calcul des sous-groupes de plus bas niveau disposait d'une expression simple, et que la somme de ces expressions simples pouvait se condenser encore en une expression simple.

10.6.5 Comportement asymptotique

On désire exprimer la qualité d'un protocole de correction d'erreur conjointement au processus d'apparition des fautes. On cherche donc un indice caractéristique de $E(\Psi_{X,Y})/E(X_1)$ et simple à exprimer, sachant que $E(Y_1) \ll E(X_1)$ (il s'agit du seul cas aboutissant à une application pratique utile).

$E(\Psi_{X,Y})/E(X_1)$ représente le *gain en sécurité* apporté par le protocole de correction de faute. Observons que ce quotient risque de prendre rapidement des valeurs très grandes suite à la condition $E(Y_1) \ll E(X_1)$.

Intuitivement, on peut facilement imaginer que le gain en sécurité soit proportionnel au rapport du temps moyen entre deux fautes consécutives par le temps moyen de correction d'une faute : en effet, plus vite on corrige les fautes simples et moins on a de chance de voir une faute double se produire.

On va montrer que pour un processus de correction d'erreur donné quelconque, alors, lorsque les fautes forment un processus sans mémoire, on a effectivement $E(\Psi_{X,Y})/E(X_1) \simeq E(X_1)/E(Y_1)$ quand $E(Y_1) \ll E(X_1)$. Notons que cette propriété peut être mise en défaut avec d'autres types de processus décrivant les fautes.

Cela va nous permettre de définir le *facteur de qualité* du protocole de correction d'erreur par $\mathcal{Q} = ln_{10}(E(X_1)/E(Y_1))$, valeur approchée du log du gain en sécurité, donc valeur approchée du nombre d'ordres de grandeur gagnés grâce au protocole de correction d'erreur.

Pour établir ce résultat, commençons par fixer le processus stochastique (Y_n) constitué de variables aléatoires mutuellement indépendantes de même densité f_Y.

Considérons la suite de processus $(X^{(m)}, m \in \mathbb{N}) = ((X_n, n \in \mathbb{N}^*)^{(m)}, m \in \mathbb{N})$. Supposons de plus que les variables $X_n^{(m)}$ et Y_n sont mutuellement indépendantes, et que la loi de $X_n^{(m)}$ ne dépend que de m.

On traduit la condition *sans mémoire* par l'existence d'une suite de réels positifs (λ_m) telle que $f_{X_1^{(m)}}(t) = \lambda_m \, e^{-\lambda_m t}, \forall t \in \mathbb{R}^+$.

La condition $E(Y_1) \ll E(X_1^{(m)})$ est alors équivalente à $E(Y_1) \ll 1/\lambda_m$. Sachant que (Y_n) est fixé, $E(Y_1)$ est donc aussi fixé. Notre condition est finalement équivalente à $\lambda_m \to 0$.

On veut montrer que $E(\Psi_{X^{(m)},Y})/E(X_1^{(m)}) \simeq E(X_1^{(m)})/E(Y_1)$, or (Y_n) est fixé, il faut donc montrer que $\lim_m E(X_1^{(m)})^2/E(\Psi_{X^{(m)},Y}) = E(Y_1)$. On effectue donc le calcul suivant :

$$\frac{E(X_1^{(m)})^2}{E(\Psi_{X^{(m)},Y})} = \frac{E(X_1^{(m)})}{1 + \frac{1}{1-P(X_1^{(m)}>Y_1)}} = \frac{1}{\lambda_m(1 + \frac{1}{1-P(X_1^{(m)}>Y_1)})}$$

On va évaluer le dénominateur à partir de la fonction génératrice des moments de Y :

$$P(X_1^{(m)} > Y_1) = \int_0^{+\infty} \int_y^{+\infty} \lambda_m \, e^{-\lambda_m x} \, f_Y(y) \, dx \, dy = \int_0^{+\infty} e^{-\lambda_m y} f_Y(y) \, dy = \mathcal{L} f_Y(\lambda_m)$$

On peut donc continuer le calcul comme suit :

$$\frac{E(X_1^{(m)})^2}{E(\Psi_{X^{(m)},Y})} = \frac{1}{\lambda_m(1+\frac{1}{1-\mathcal{L}f_Y(\lambda_m)})} = \frac{1}{\lambda_m(1+\frac{1}{\mathcal{L}f_Y(0)-\mathcal{L}f_Y(\lambda_m)})}$$

$\mathcal{L}f_Y$ est continue au voisinage de 0 donc $1 \ll 1/(\mathcal{L}f_Y(0) - \mathcal{L}f_Y(\lambda_m))$, $\lambda_m \to 0$. Ainsi, quand λ_m est au voisinage de 0, on peut écrire :

$$\frac{E(X_1^{(m)})^2}{E(\Psi_{X^{(m)},Y})} \sim \frac{\mathcal{L}f_Y(0) - \mathcal{L}f_Y(\lambda_m)}{0-\lambda_m} \sim (\mathcal{L}f_Y)'(0) = E(Y_1)$$

On obtient donc le résultat attendu :

$$\lim_{m\to+\infty} \frac{E(X_1^{(m)})^2}{E(\Psi_{X^{(m)},Y})} = E(Y_1)$$

Remarquons que le facteur de qualité s'exprime sous la forme d'une fonction aux variables séparées, on peut donc simplement prévoir son évolution en fixant l'un ou l'autre de ses deux paramètres :

$$\mathcal{Q}(X,Y) = \mathcal{Q}_1(X) - \mathcal{Q}_1(Y)$$

On verra un exemple d'application du facteur de qualité à la fin de ce chapitre.

10.6.6 Calcul de la variance

On va maintenant montrer qu'il est possible de calculer la variance de la variable aléatoire de double-faute dans un cadre général, sans présumer des distributions des variables aléatoires en jeu.

Pour cela on va naturellement commencer par le calcul du moment d'ordre 2 de $\Psi_{X,Y}$.

Nous avons démontré que si $P(X_1 > Y_1) = 1$, alors $\Psi_{X,Y} = +\infty$ p.s., donc on ne peut pas calculer de moment. Supposons donc que $P(X_1 > Y_1) < 1$.

Sachant que $P(\Lambda_\infty) = 0$, on peut écrire :

$$\begin{aligned}
\int_\Omega \Psi^2_{X,Y}\, dP &= \int_{\Lambda_\infty} \Psi^2_{X,Y}\, dP + \sum_{i=1}^{\infty} \int_{\Lambda_i} \left[\sum_{j=0}^{i} X_j\right]^2 dP = \sum_{i=1}^{\infty} \int_{\Lambda_i} \sum_{j=0}^{i}\sum_{k=0}^{i} X_j X_k\, dP \\
&= \sum_{i=1}^{\infty}\sum_{j=0}^{i} \int_{\Lambda_i} X_j^2\, dP + \sum_{i=1}^{\infty} \int_{\Lambda_i} \sum_{\substack{j=0\\ j\neq k}}^{i}\sum_{k=0}^{i} X_j X_k\, dP \\
&= \sum_{i=1}^{\infty}\sum_{j=1}^{i} \int_{\Lambda_i} X_j^2\, dP + \sum_{i=1}^{\infty} \int_{\Lambda_i} X_0^2\, dP + \sum_{i=1}^{\infty}\sum_{\substack{j=0\\ j\neq k}}^{i}\sum_{k=0}^{i} \int_{\Lambda_i} X_j X_k\, dP \\
&= \sum_{j=1}^{\infty}\sum_{i=j}^{\infty} \int_{\Lambda_i} X_j^2\, dP + \int_{\Omega} X_0^2\, dP + \sum_{i=1}^{\infty}\sum_{\substack{j=0\\ j\neq k}}^{i}\sum_{k=0}^{i} \int_{\Lambda_i} X_j X_k\, dP \\
&= S_1 + S_2 + S_3
\end{aligned}$$

Calculons séparément chacun des trois termes S_1, S_2 et S_3 qui composent la somme précédente. Par définition, $S_2 = E(X_0^2)$. Pour calculer S_1, on va utiliser le fait que $\forall\, n \in \mathbb{N}, n \geqslant 2 \Rightarrow \Lambda_\infty \uplus (\uplus_{i\geqslant n}\Lambda_i) = \cap_{1\leqslant i<n}\Delta_i$:

$$\begin{aligned}
S_1 &= \sum_{j=1}^{\infty} \int_{\uplus_{i\geqslant j}\Lambda_i} X_j^2\, dP = \int_{\uplus_{i\geqslant 1}\Lambda_i} X_1^2\, dP + \sum_{j=2}^{\infty} \int_{\uplus_{i\geqslant j}\Lambda_i} X_j^2\, dP \\
&= E(X_1^2) + \sum_{j=2}^{\infty} \int_{(\cap_{1\leqslant i<j}\Delta_i)\setminus\Lambda_\infty} X_j^2\, dP = E(X_1^2) + \sum_{j=2}^{\infty} E(X_1^2)\, P(\Delta_1)^{j-1} \\
&= \sum_{j=0}^{\infty} E(X_1^2)\, P(\Delta_1)^j = \frac{E(X_1^2)}{1 - P(\Delta_1)}
\end{aligned}$$

Pour calculer S_3, on va permuter à plusieurs reprises les indices des séries :

$$\begin{aligned}
S_3 &= \sum_{i=1}^{\infty}\sum_{\substack{j=0\\ j\neq k}}^{i}\sum_{k=0}^{i} \int_{\Lambda_i} X_j X_k\, dP = 2\sum_{i=1}^{\infty}\sum_{j=1}^{i}\sum_{k=0}^{j-1} \int_{\Lambda_i} X_j X_k\, dP \\
&= 2\sum_{j=1}^{\infty}\sum_{i=j}^{\infty}\sum_{k=0}^{j-1} \int_{\Lambda_i} X_j X_k\, dP = 2\sum_{j=1}^{\infty}\sum_{k=0}^{j-1}\sum_{i=j}^{\infty} \int_{\Lambda_i} X_j X_k\, dP \\
&= 2\sum_{j=2}^{\infty}\sum_{k=0}^{j-1} \int_{\uplus_{i\geqslant j}\Lambda_i} X_j X_k\, dP + 2\sum_{i=1}^{\infty} \int_{\Lambda_i} X_1 X_0\, dP \\
&= 2\sum_{j=2}^{\infty}\sum_{k=0}^{j-1} \int_{(\cap_{1\leqslant i<j}\Delta_i)\setminus\Lambda_\infty} X_j X_k\, dP + 2E(X_1)^2
\end{aligned}$$

On extrait alors de la somme le terme en $k = 0$:

$$\begin{aligned}
S_3 &= 2\sum_{j=2}^{\infty}\sum_{k=1}^{j-1}\int_{(\cap_{1\leqslant i<j}\Delta_i)\setminus\Lambda_\infty} X_j X_k \, dP + 2\sum_{j=2}^{\infty} E(X_1)^2 P(\Delta_1)^{j-1} + 2E(X_1)^2 \\
&= 2\sum_{j=2}^{\infty}\sum_{k=1}^{j-1}\int_{(\cap_{1\leqslant i<j}\Delta_i)\setminus\Lambda_\infty} X_j X_k \, dP + \frac{2E(X_1)^2}{1-P(\Delta_1)}
\end{aligned}$$

L'indépendance des variables aléatoires X_i nous permet de simplifier une partie de l'expression :

$$\begin{aligned}
S_3 &= 2\sum_{j=2}^{\infty}\sum_{k=1}^{j-1} P(\cap_{1\leqslant i<j,\, i\neq k}\Delta_i)\int_{\Delta_k} X_j \, X_k \, dP + \frac{2E(X_1)^2}{1-P(\Delta_1)} \\
&= 2\sum_{j=2}^{\infty}\sum_{k=1}^{j-1} P(\Delta_1)^{j-2} E(X_1)\int_{\Delta_k} X_k \, dP + \frac{2E(X_1)^2}{1-P(\Delta_1)} \\
&= 2E(X_1)\int_{\Delta_1} X_1 \, dP \sum_{j=2}^{\infty}(j-1)P(\Delta_1)^{j-2} + \frac{2E(X_1)^2}{1-P(\Delta_1)} \\
&= \frac{2E(X_1)}{(1-P(\Delta_1))^2}\int_{\Delta_1} X_1 \, dP + \frac{2E(X_1)^2}{1-P(\Delta_1)}
\end{aligned}$$

En rassemblant les résultats intermédiaires, on obtient :

$$\int_{\Omega} \Psi_{X,Y}^2 \, dP = E(X_1^2) + \frac{E(X_1^2) + 2\,E(X_1)^2}{1-P(\Delta_1)} + \frac{2\,E(X_1)}{(1-P(\Delta_1))^2}\int_{\Delta_1} X_1 \, dP$$

On exprime la variance de $\Psi_{X,Y}$ à l'aide de la relation suivante :

$$\mathrm{Var}(\Psi_{X,Y}) = E(\Psi_{X,Y}^2) - E(\Psi_{X,Y})^2$$

On peut donc énoncer un nouveau théorème qui résume ce résultat :

Théorème 2 VARIANCE D'UNE VARIABLE ALÉATOIRE DE DOUBLE-FAUTE

Hypothèses :
On se place sous les hypothèses du théorème 1 page 212, auxquelles on ajoute les deux conditions suivantes :

- les variables aléatoires réelles X_i sont de carré intégrable,
- les lois de X_i et Y_i sont telles que $P(\Delta_1) = P(X_1 > Y_1) < 1$.

Énoncé :
$\Psi_{X,Y}$ est de carré intégrable et sa variance a pour expression :

$$\mathrm{V}(\Psi_{X,Y}) = \mathrm{V}(X_1) + \frac{\mathrm{V}(X_1) + E(X_1)^2}{1-P(\Delta_1)} - \frac{E(X_1)^2}{(1-P(\Delta_1))^2} + \frac{2E(X_1)}{(1-P(\Delta_1))^2}\int_{\Delta_1} X_1 \, dP$$

démonstration :

La variable aléatoire de double-faute est de carré intégrable si tous les termes qui composent l'expression de son moment d'ordre 2 sont finis. Vues les hypothèses, il suffit de montrer que $\int_{\Delta_1} X_1 \, dP$ est fini. Sachant que X_1 est une variable aléatoire réelle positive, on peut écrire :

$$0 \leqslant \int_{\Delta_1} X_1 \, dP \leqslant \int_{\Omega} X_1 \, dP = E(X_1) < +\infty$$

Donc $\Psi_{X,Y}$ est de carré intégrable. □

Notons enfin que le calcul de $\int_{\Delta_1} X_1 \, dP$ à partir des densités de probabilité se fait simplement à l'aide d'une intégrale double :

$$\int_{\Delta_1} X_1 \, dP = \int_{X_1 > Y_1} X_1 \, dP = \int_{-\infty}^{+\infty} \int_{y}^{+\infty} x f_X(x) f_Y(y) \, dx dy$$

Le calcul que l'on vient d'effectuer, plus délicat que celui de l'espérance, utilise néanmoins sensiblement les mêmes méthodes. Sa démonstration permet de se convaincre de la possibilité de développer de manière formelle n'importe quel moment de la variable de double-faute.

Maintenant que l'on connaît la variance et l'espérance de $\Psi_{X,Y}$, on peut l'encadrer autour de sa moyenne, en appliquant l'inégalité de Tchebychev. Ainsi, si $\Psi_{X,Y}$ est de carré intégrable, c'est-à-dire si X est de carré intégrable, alors :

$$\forall \Theta \in \mathbb{R}^{+*}, \, P(|\Psi_{X,Y} - E(\Psi_{X,Y})| \geqslant \Theta) \leqslant \frac{\mathrm{Var}(\Psi)}{\Theta^2}$$

En pratique, ce résultat permet de déterminer la plage de valeurs du MTBF englobant par exemple au moins 95% des cas possibles. Cela donne une idée de la dispersion des doubles-fautes.

10.6.7 Critère d'existence des moments

Plaçons nous sous les hypothèses du théorème 1 page 212, auxquelles on ajoute la condition $P(X_1 > Y_1) < 1$.
On peut alors affirmer que pour tout n, X_1 admet un moment d'ordre n si et seulement si $\Psi_{X,Y}$ admet un moment d'ordre n.

Notre démonstration de cette proposition, disponible en annexe G.4 page 261, consiste principalement à effectuer une majoration de $E(X_1^n)$ assez fastidieuse.

10.7 Continuité de la double-faute

Pour toute variable aléatoire réelle, on notera Φ_X la fonction caractéristique de X, définie comme suit :

$$\begin{aligned} \Phi_X : \mathbb{R} &\longrightarrow \mathbb{C} \\ u &\mapsto E(e^{iuX}) \end{aligned}$$

Nous allons maintenant présenter le calcul de l'expression de la fonction caractéristique de la variable aléatoire représentant les doubles-fautes, et en déduire la continuité de cette variable.

La connaissance de la fonction caractéristique d'une variable aléatoire permet d'obtenir des informations d'intérêt. On peut par exemple en extraire les différents moments, par la relation $E(X^n) = (-i)^n \, \Phi_X^{(n)}(0)$.

Il semble difficile d'établir une expression littérale simple de la fonction caractéristique, c'est-à-dire de la transformée de Fourier de la variable aléatoire de double-faute, et il n'est pas évident qu'une telle expression existe. On pourra néanmoins, comme on le verra dans la section suivante, déduire de l'expression que l'on va calculer ici une propriété de convergence fondamentale pour les applications de nos résultats à un modèle physique.

On va supposer dans ce qui suit que $P(X > Y) < 1$, seul cas intéressant. Pour effectuer le calcul de $\Phi_{\Psi_{X,Y}}$, on va décomposer l'espérance de e^{iuX} en la sommant successivement sur tous les ensembles Λ_i :

$$\begin{aligned} \forall u \in \mathbb{R},\ \Phi_{\Psi_{X,Y}}(u) = E(e^{iu\Psi_{X,Y}}) &= \int_\Omega e^{iu\Psi_{X,Y}} dP = \sum_{i=1}^{+\infty} \int_{\Lambda_i} e^{iu\Psi_{X,Y}} dP = \sum_{i=1}^{+\infty} \int_{\Lambda_i} e^{iu\sum_{j=0}^{i} X_j} dP \\ &= \sum_{i=1}^{+\infty} \int_{\Lambda_i} \prod_{j=0}^{i} e^{iuX_j} dP = \sum_{i=1}^{+\infty} \prod_{j=0}^{i} \int_{\Lambda_i} e^{iuX_j} dP = \sum_{i=1}^{+\infty} \prod_{j=0}^{i} \int_{\cap_{k=1}^{i-1} \Delta_k \cap \Delta_i^c} e^{iuX_j} dP \\ &= \sum_{i=2}^{+\infty} \prod_{j=0}^{i} \int_{\cap_{k=1}^{i-1} \Delta_k \cap \Delta_i^c} e^{iuX_j} dP + P(\Delta_1^c) \int_\Omega e^{iuX_0} dP \int_{\Delta_1^c} e^{iuX_1} dP \end{aligned}$$

On peut calculer chaque terme du produit comme suit :

$$i \geq 2 \Rightarrow \int_{\cap_{k=1}^{i-1} \Delta_k \cap \Delta_i^c} e^{iuX_j} dP = \begin{cases} \int_\Omega e^{iuX_0} dP \ P(\Delta_1)^{i-1} \ P(\Delta_1^c) & \text{si } j = 0, \\ \int_{\Delta_1} e^{iuX_0} dP \ P(\Delta_1)^{i-2} \ P(\Delta_1^c) & \text{si } j \in \{1, \cdots, i-1\}, \\ \int_{\Delta_1^c} e^{iuX_0} dP \ P(\Delta_1)^{i-1} & \text{si } j = i. \end{cases}$$

On injecte alors ce résultat dans l'expression de $\Phi_{\Psi_{X,Y}}(u)$:

$$\forall u \in \mathbb{R},\ \Phi_{\Psi_{X,Y}}(u) = \int_\Omega e^{iuX_0} dP \int_{\Delta_1^c} e^{iuX_0} dP \sum_{i=1}^{+\infty} \left[\int_{\Delta_1} e^{iuX_0} dP \right]^{i-1} P(\Delta_1)^{i(i-1)} P(\Delta_1^c)^i$$

Notons $X_{|X>Y}$ la restriction de X aux événements tels que $X > Y$. La fonction caractéristique de $X_{|X>Y}$ dans l'espace sur lequel cette variable aléatoire opère sera donc notée $\Phi_{X_{|X>Y}}$.

Soit h la fonction de deux variables définie comme suit :

$$\begin{aligned} h : \mathbb{R} \times [0,1[&\longrightarrow \mathbb{R} \\ (x,y) &\mapsto \sum_{i=1}^{+\infty} x^{i-1} y^{i(i-1)} (1-y)^i \end{aligned}$$

La fonction caractéristique de la variable aléatoire de double-faute peut alors s'exprimer ainsi :

$$\forall u \in \mathbb{R},\ \Phi_{\Psi_{X,Y}}(u) = \Phi_X(u)\, \Phi_{X_{|X<Y}}(u)\, h(\Phi_{X_{|X>Y}}(u), P(X > Y))$$

Sachant que $\Phi_{X_{|X<Y}}(u) = \Phi_X(u) - \Phi_{X_{|X>Y}}(u)$, on peut aussi écrire :

$$\Phi_{\Psi_{X,Y}} = \Phi_X\ (\Phi_X - \Phi_{X_{|X>Y}})\, h(\Phi_{X_{|X>Y}}, P(\Delta_1))$$

Le problème consiste maintenant à déterminer une expression plus simple de h, s'il en existe une. Pour cela, on va effectuer le changement de variable suivant : $\xi = \frac{x(1-y)}{y}$ et $\nu = y$. On va alors définir la fonction de deux variables z comme suit :

$$z(\xi,\nu) = x\ h(x,y) - x\ (1-y) = \sum_{i=2}^{+\infty} \xi^i \nu^{i^2}$$

On constate alors que z vérifie l'équation aux dérivées partielles suivante :

$$\frac{\partial^2 z}{\partial \xi^2} = \frac{\nu}{\xi^2} \frac{\partial z}{\partial \nu} - \frac{1}{\xi} \frac{\partial z}{\partial \xi}$$

Il s'agit de la forme canonique d'une équation aux dérivées partielles de type parabolique (voir [Evans *et al.*, 2000] pour la classification de ce type d'équation). Malheureusement, la méthode de résolution par séparation des variables aboutit à une équation différentielle d'une variable dont les solutions font intervenir des valeurs particulières des fonctions de Bessel. On se retrouve alors avec une combinaison linéaire de ces solutions. Pour trouver *la* combinaison solution de notre problème, on fait intervenir les contraintes dues aux conditions aux frontières, et notre solution est finalement définie sous la forme d'une série. Il semble donc bien que la forme la plus simple de h soit celle sous-laquelle elle a été initialement définie.

La fonction h est continue, donc la fonction caractéristique l'est aussi. Un résultat classique de la convergence faible de suites de variables aléatoires permet de déduire de la continuité de $\Phi_{\Psi_{X,Y}}$ que si $X_n \xrightarrow{d} U$ et $Y_n \xrightarrow{d} V$, alors $\Psi_{X_n,Y_n} \xrightarrow{d} \Psi_{U,V}$, à condition que U et V soient indépendantes, et que $P(U = V) = 0$.

Or, dans un espace métrique, pour montrer la continuité d'une fonction en un point, il suffit de montrer que pour toute suite convergeant vers ce point, l'image de cette suite converge vers l'image de ce point (propriété qui n'est pas généralisable aux espaces topologiques non métrisables). Or la convergence faible (à l'inverse de la convergence P-presque sûre) est justement métrisable : la distance de Lévy-Prokhorov en est un exemple classique. Ainsi, Ψ est continue en (U,V) du moment que U et V sont indépendantes et telles que $P(U = V) = 0$.

La continuité de Ψ permet de s'assurer qu'un modèle physique *approché* des processus de panne et de réparation est suffisant pour obtenir une estimation du comportement de la double-faute.

10.8 Application à un modèle physique

10.8.1 Choix des lois représentatives

On se propose maintenant d'appliquer à un modèle physique, représentant le comportement du matériel et du logiciel de la machine MPC, la théorie mathématique que l'on vient de développer, dans le but de caractériser l'apparition des doubles-fautes sur un câble HSL.

Une étude, menée conjointement par le CERN et la société Tachys au premier semestre 2000, a permis de déterminer la cause des fautes sur le réseau HSL : selon cette analyse, la perte de calibration est liée à une erreur sur la récupération de l'horloge au début de la réception d'un mot. L'erreur est alors amplifiée au cours de la réception des différents bits composant le mot, et mène à une perte de calibration avant la resynchronisation par le mot suivant.

La récupération de l'horloge est effectuée à chaque nouveau mot reçu, il n'y a donc pas d'effet mémoire au delà du délai de transmission d'un mot. Un tel phénomène peut donc être considéré sans mémoire, car le délai de récupération d'un mot est de l'ordre de la dizaine de nanosecondes. On peut donc modéliser l'apparition de fautes par un processus stochastique sans mémoire, donc de loi exponentielle et de paramètre λ, inverse de sa moyenne. La densité de probabilité des X_i est donc définie comme suit :

$$\begin{aligned} f_X : \mathbb{R} &\longrightarrow \mathbb{R} \\ t &\mapsto \begin{cases} \lambda\, e^{-\lambda t} & \text{si } t \geqslant 0 \\ 0 & \text{si } t < 0 \end{cases} \end{aligned}$$

Le délai de réparation dépend du protocole mis en jeu en cas de faute. Désignons par τ sa moyenne. Nous allons traiter le pire cas, c'est-à-dire lorsque chaque réparation dure τ, ce qui va nous permettre de considérer X comme une variable aléatoire constante. Sa densité de probabilité prend donc la forme de la distribution de Dirac :

$$f_Y = \delta_\tau : t \mapsto \delta(t - \tau)$$

10.8.2 Résultats littéraux

Les mesures effectuées sur la machine MPC montrent que la valeur du paramètre $1/\lambda$ observé en utilisation courante s'approche de l'heure, mais il s'agit là uniquement d'une valeur *apparente*, car seule une charge totale du lien HSL peut permettre de connaître la valeur réelle. Celle-ci a été mesurée à l'aide d'un banc de test logiciel spécifiquement conçu pour charger les liens HSL au plus haut débit possible. Ainsi, dans ces conditions, $1/\lambda$ vaut approximativement 120 secondes.

Des calculs simples fournissent les résultats intermédiaires suivants :

$$E(X_1) = \frac{1}{\lambda} \qquad E(X_1^2) = \frac{2}{\lambda^2} \qquad P(\Delta_1) = e^{-\lambda\tau} \qquad \int_{\Delta_1} X_1 \, dP = (\frac{1}{\lambda} + \tau)\, e^{-\lambda\tau}$$

A l'aide de ces résultats, le théorème 1 page 212 nous permet maintenant de calculer l'espérance de $\Psi_{X,Y}$:

$$E(\Psi_{X,Y}) = \frac{1}{\lambda}\left[1 + \frac{1}{1 - e^{-\lambda\tau}}\right]$$

En pratique, le protocole de contournement de faute est simple. Son délai de réalisation est donc particulièrement faible devant les délais d'inter-arrivées de fautes dont on vient d'estimer un ordre de grandeur de plusieurs dizaines de secondes. On peut donc légitimement se permettre d'approcher le résultat par un équivalent lorsque τ se trouve très petit devant $1/\lambda$, ce qui simplifie le résultat :

$$E(\Psi_{X,Y}) \underset{\tau\to 0}{\sim} \frac{1}{\lambda^2\,\tau}$$

On peut donc exprimer la relation entre les temps moyens d'attente entre fautes (MTBF(hsl)) et les temps moyens d'attente entre double-faute (MTBF(put)) :

$$MTBF(put) = MTBF(hsl)\left[1 + \frac{1}{1 - e^{-\frac{\tau}{MTBF(hsl)}}}\right] \underset{\tau\to 0}{\sim} \frac{MTBF(hsl)^2}{\tau}$$

Calculons maintenant le MTBF de la machine MPC complète. Les pertes de calibration lorsqu'aucune donnée utile ne transite dans le réseau sont sans conséquence. Désignons par D_{utile} le débit moyen utile sur un lien HSL, par $D_{\max}$ le débit en pleine charge, par N le nombre de nœuds et par v la valence de chaque nœud, c'est à dire le nombre de partenaires avec lesquels il est directement relié. Il y a donc $\frac{Nv}{2}$ liens HSL dans la machine, chacun étant bi-directionnel et composé de deux câbles coaxiaux. La machine met donc en œuvre Nv câbles coaxiaux. Le MTBF de la machine dans sa totalité s'exprime donc ainsi :

$$MTBF(mpc) = \frac{D_{max}\,MTBF(hsl)}{N\;v\;D_{utile}}\left[1 + \frac{1}{1 - e^{-\frac{\tau}{MTBF(hsl)}}}\right] \underset{\tau\to 0}{\sim} \frac{D_{max}\,MTBF(hsl)^2}{N\;v\;D_{utile}\;\tau}$$

À l'inverse, on peut se demander quelles performances doit accomplir le protocole de correction d'erreur simple pour obtenir un MTBF choisi arbitrairement. Il suffit pour répondre à cette question d'inverser par rapport à τ la formule du MTBF, ce qui donne la borne supérieure du délai de reprise sur faute que le protocole de correction ne doit pas dépasser :

$$\tau = -MTBF(hsl)\ \ ln\left(1 + \frac{1}{1 - \frac{MTBF_{\text{désiré}}}{MTBF(hsl)}}\right) \underset{MTBF(hsl) \ll MTBF_{\text{désiré}}}{\approx} \frac{MTBF(hsl)^2}{MTBF_{\text{désiré}}}$$

La variance de $\Psi_{X,Y}$ se calcule à l'aide du théorème 2 page 216, et on peut en trouver un équivalent lorsque τ se trouve très petit devant $1/\lambda$:

$$\text{Var}(\Psi_{X,Y}) = \frac{1}{\lambda^2}\left[2 + \frac{3}{1-e^{-\lambda\tau}} + \frac{2e^{-\lambda\tau}(1+\lambda\tau)}{(1-e^{-\lambda\tau})^2} - \left(\frac{2-e^{-\lambda\tau}}{1-e^{-\lambda\tau}}\right)^2\right] \underset{\tau\to 0}{\sim} \frac{1}{\lambda^4\,\tau^2}$$

Notons $\sigma(\Psi_{X,Y})$ l'écart-type de $\Psi_{X,Y}$. On peut remarquer que l'écart-type et l'espérance de $\Psi_{X,Y}$ sont équivalents quand τ se trouve au voisinage de 0 :

$$\sigma(\Psi_{X,Y}) \underset{\tau\to 0}{\sim} \frac{1}{\lambda^2\,\tau}$$
$$E(\Psi_{X,Y}) \underset{\tau\to 0}{\sim} \frac{1}{\lambda^2\,\tau}$$

Calculons maintenant le facteur de qualité qui, rappelons-le, représente une valeur approchée du nombre d'ordres de grandeur gagnés grâce au protocole de correction d'erreur :

$$\mathcal{Q}(X,Y) = ln_{10}\frac{E(X_1)}{E(Y_1)} = -ln_{10}(\lambda\,\tau)$$

On peut donc exprimer les équivalents de l'espérance et de l'écart-type à partir du facteur de qualité :

$$\frac{\sigma(\Psi_{X,Y})}{\sigma(X_1)} \underset{\tau\to 0}{\sim} 10^{\mathcal{Q}}$$
$$\frac{E(\Psi_{X,Y})}{E(X_1)} \underset{\tau\to 0}{\sim} 10^{\mathcal{Q}}$$

10.8.3 Application numérique

La figure 10.2 page ci-contre présente le MTBF résultant de la correction de faute simple d'un lien dont le MTBF est de 120 secondes, en fonction du délai de correction. On constate qu'en dessous de 10 μs (i.e. $\mathcal{Q} > 7{,}1$), la courbe tend rapidement vers son asymptote.

Il est tout à fait envisageable d'implémenter par logiciel le protocole de correction d'erreur décrit au début de ce chapitre, en faisant en sorte que le délai de correction nécessite 20 ou 30 μs ($\mathcal{Q}$ vaut alors respectivement 6,7 et 6,6). On lit sur la courbe que le taux de panne résultant serait alors de l'ordre de **5 millions de fois moins important** que le taux de panne simple du lien.

Si le protocole était implémenté en matériel, on pourrait imaginer un délai de correction d'erreur de l'ordre de la dizaine de μs, ce qui permettrait d'atteindre un taux de panne plus de deux fois moindre.

On constate enfin qu'au dessus de 50 μs ($\mathcal{Q} < 6{,}4$) de délai de correction de faute, les micro-secondes gagnées influent peu sur le MTBF résultant. Il est assez remarquable que le domaine où le gain de quelques micro-secondes de délai permet d'accroître de façon importante le MTBF résultant correspond justement au domaine limite, où l'implémentation logicielle doit faire place à une implémentation matérielle pour pouvoir augmenter les performances.

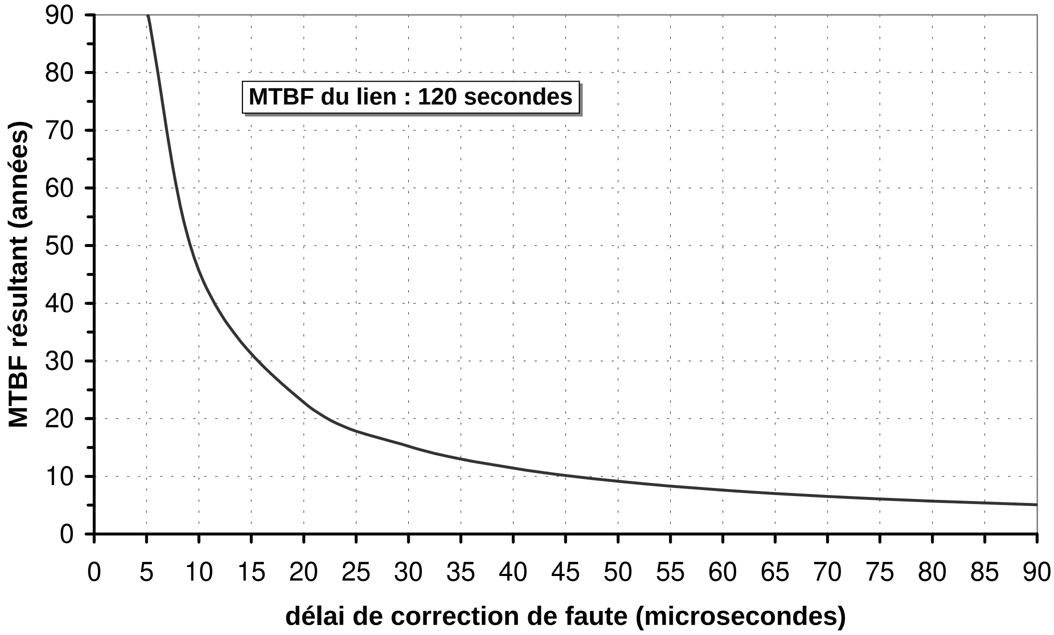


Fig. 10.2 *MTBF après correction de faute simple pour un lien de MTBF 120 secondes*

Pour avoir une idée de la distribution de la double faute, pour le cas particulier où la simple faute suit une distribution exponentielle et où le délai de correction est constant, le comportement du système a été formalisé à l'aide du langage de description du logiciel Scilab. Scilab est un logiciel conçu par l'INRIA pour le traitement du signal et le contrôle des systèmes. Il permet de générer des événements aléatoires à partir des lois simples, et de les composer pour obtenir toutes sortes de comportements. On a ainsi pu simuler les erreurs de liens et le protocole de correction d'erreur pour obtenir l'histogramme de la figure 10.3 page suivante, qui représente la distribution de la double faute pour $1/\lambda = 120s$ et $\tau = 10^{-3}s$.

Pour avoir une idée de la dispersion des occurrences de double-faute, on va utiliser l'inégalité de Tchebychev. Rappelons que si $\Psi_{X,Y}$ est de carré intégrable, c'est-à-dire si X_1 est de carré intégrable, alors la relation suivante est établie (inégalité de Tchebychev) :

$$\forall \Theta \in \mathbb{R}^{+*}, P(|\Psi - E(\Psi)| \geqslant \Theta) \leqslant \frac{\mathrm{Var}(\Psi)}{\Theta^2}$$

Le théorème 2 page 216 nous a permis de calculer la variance de Ψ. Donc à l'aide de cette inégalité on peut déterminer l'écart à la moyenne, une fois fixée la probabilité d'englober

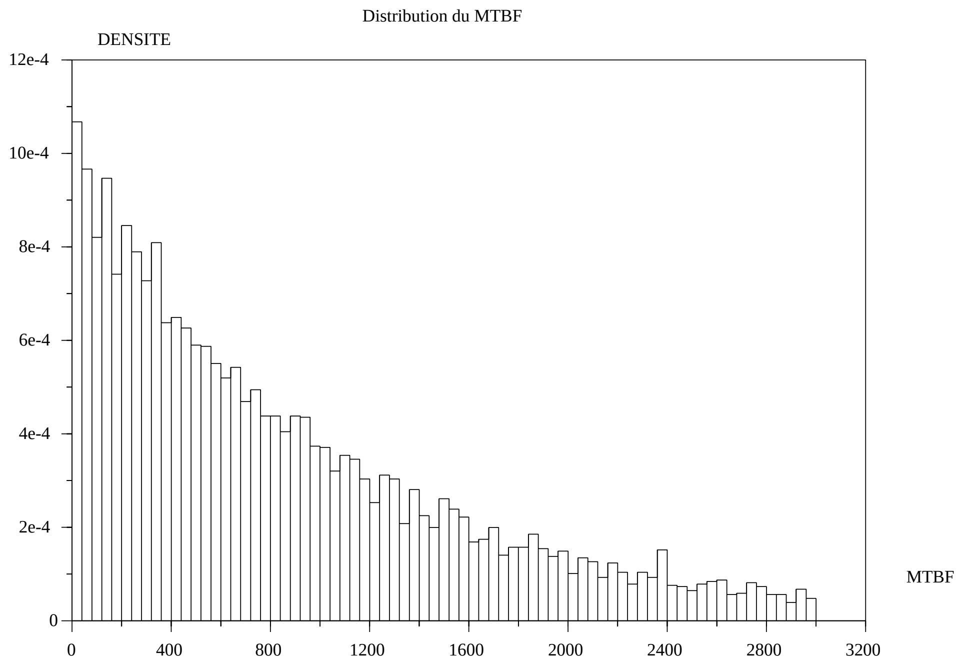


Fig. 10.3 *Histogramme de la densité de la double-faute (en secondes)*

au moins un certain pourcentage des événements. On a donc choisi $1/\lambda = 120s$ pour représenter sur la figure 10.4 page suivante la courbe représentative de la double-faute en fonction du délai de réparation, et y superposer les deux surfaces suivantes :

- ✓ En gris foncé, on a représenté la zone englobant au moins 95% des événements possibles. Pour ce faire, on a donc calculé $\Theta(\lambda,\tau) = \sqrt{\frac{\mathrm{Var}(\Psi_{X(\lambda),Y(\tau)})}{0{,}05}}$ et alors représenté $E(\Psi_{X(\lambda),Y(\tau)})) \pm \Theta(\lambda,\tau)$.
- ✓ En gris clair, on a représenté la zone englobant au moins 5% des événements possibles, en calculant $\Theta(\lambda, \tau) = \sqrt{\frac{\mathrm{Var}(\Psi_{X(\lambda),Y(\tau)})}{0{,}95}}$ et en représentant $E(\Psi_{X(\lambda),Y(\tau)}) \pm \Theta(\lambda,\tau)$.

On constate que dans les deux cas, $E(\Psi_{X,Y}) - \Theta < 0$, donc que les deux surfaces sont délimitées uniquement par leur borne supérieure. On aurait préféré que $E(\Psi_{X,Y}) - \Theta > 0$ afin de pouvoir garantir que les doubles-fautes ont peu de chance de se produire dans un délai court. Ce n'est pas le cas (même si *en moyenne* elles se produisent après un très grand délai), car cette dispersion importante de la double-faute est due à la nature du phénomène de faute qu'on étudie ici : les fautes sont modélisées par un processus stochastique sans mémoire, donc exponentiel, donc de variance importante.

10.8.4 Étude de la répartition

Le calcul de l'espérance de la double-faute nous indique que la durée moyenne de bon fonctionnement d'une machine MPC utilisant le protocole de correction d'erreur simpliste proposé est très largement suffisant (plusieurs années).

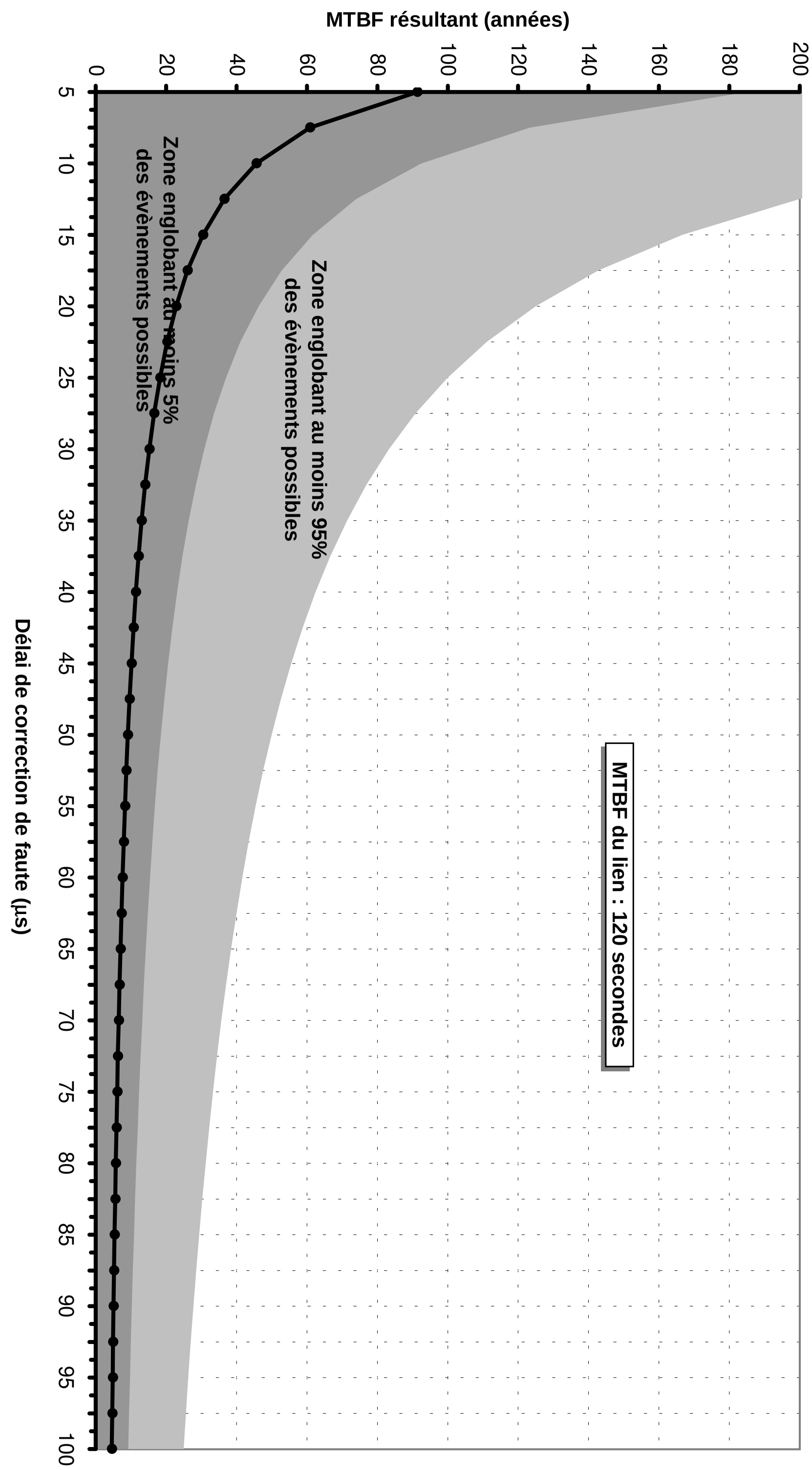


Fig. 10.4 *Dispersion des événements de double-faute*

Par contre, l'espérance ne nous donne aucune information sur un éventuel seuil au dessous duquel le délai de bon fonctionnement ne pourrait pas descendre.

Pour tenter d'obtenir une telle information, on a donc calculé la variance, ce qui nous a permis d'appliquer l'inégalité de Tchebychev. Celle-ci nous a alors indiqué que la dispersion des événements est importante : on aurait bien sûr préféré qu'elle nous indique au contraire qu'une part importante des délais de double-faute se trouve très proche de la moyenne, ce qui aurait permis de calculer un seuil au dessous duquel une double-faute aurait été fortement improbable.

On cherche donc un seuil T, représentant une grandeur temporelle, tel que la condition $\Psi < T$ soit en pratique irréalisable.

Pour quantifier le sens du terme *irréalisable*, on peut par exemple dire qu'un événement qui se produit tous les 10^9 événements est irréalisable. En informatique, on prend parfois la valeur 10^{18} pour signifier qu'on accepte qu'un composant commette une erreur tous les 10^{18} cycles, car il s'agit du taux d'erreur observé d'un fond de panier, il est donc inutile d'espérer mieux des composants constituant l'ordinateur. En télécom, on se contente le plus souvent d'une erreur tous les 10^9 cycles. Par exemple, dans [Nishimura *et al.*, 2000], un commutateur à ports optiques de débit proche de MPC (800 Mb/s) et de BER (Bit Error Rate) de 10^{-11} est considéré comme très fiable.

Notons au passage que le temps de cycle en télécom n'est pas forcément identique au temps de cycle d'un ordinateur. Dans notre cas, on peut par exemple décider de se baser sur le temps d'exécution d'une application, et donc décider d'accepter un plantage toutes les 1000 applications. Certes, 1000 est bien inférieur à 10^{18}, mais le temps de vie d'une application est très largement supérieur au temps de cycle d'un Pentium, cette valeur peut donc par exemple être considérée comme satisfaisante.

Représentons par S le taux de panne acceptable choisi par l'utilisateur de la machine ; on posera par exemple $S = \frac{1}{1000}$.

On cherche donc à évaluer T tel que $P(\Psi < T) = S$.

On va pour cela poser $q = P(X_1 < Y_1)$ et $a = E(X_1)$, et utiliser le théorème limite général établi par Gnédenko et Soloviev, présenté par Korolyuk [Korolyuk, 1997] :

$$\lim_{q \to 0} P(q\Psi > t) = e^{-t/a}$$

Dans le cas particulier des processus X et Y choisis ici, ceux-ci sont complètement caractérisés par les valeurs des paramètres λ et τ. On va donc utiliser la notation $\Psi_{\lambda,\tau}$ pour désigner la double-faute, et réécrire le résultat précédent comme suit :

$$\lim_{\tau \to 0} P((1 - e^{-\lambda\tau})\, \Psi_{\lambda,\tau} > t) = e^{-\lambda t}$$

On va donc approximer $P((1 - e^{-\lambda\tau})\, \Psi_{\lambda,\tau} > T)$ par $e^{-\lambda T}$. La construction de $\Psi_{\lambda,\tau}$ la rend diffuse, on peut alors approximer $P(\lambda\, \tau\, \Psi_{\lambda,\tau} > T)$ par $e^{-\lambda T}$. Et par suite, $P(\Psi_{\lambda,\tau} > T)$ est proche de $e^{-\lambda^2\, \tau\, T}$. Finalement, $P(\Psi > T)$ est proche de $e^{-T/E(\Psi)}$, Ψ est donc localement proche d'une variable aléatoire exponentielle.

On cherche donc à évaluer T tel que $1 - S \approx e^{-\lambda^2 \tau T}$:

$$T \approx \frac{S}{\lambda^2 \tau}$$

Avec $S = 10^{-3}$, $\lambda = 1/120\,\mathrm{Hz}$ et $\tau = 30\,\mu\mathrm{s}$, on obtient $T \approx 133\,\mathrm{h}$. Donc **seulement un millième des doubles-fautes se produisent dans les 5 premiers jours**, donc une application qui dure moins de 5 jours a moins d'une chance sur 1000 d'être confrontée à une panne.

Sans notre protocole de correction des fautes, seules les applications durant moins de 120 ms auraient moins d'une chance sur 1000 d'être confrontées à une panne.

10.9 Conclusion

Les résultats numériques auxquels on a abouti montrent que le protocole de correction d'erreur, proposé au début de ce chapitre, est certes simpliste (il ne corrige pas les doubles-fautes), mais qu'il offre une garantie de sécurité néanmoins suffisante pour les applications que l'on désire faire tourner sur la machine MPC. Nos calculs ne sont pas propres à la machine MPC, mais s'appliquent à un cadre plus large (voir les hypothèses des différentes proposition), et nous espérons donc que ces résultats puissent intéresser les concepteurs de protocoles réseau en général.

Chapitre 11

Conclusion et perspectives

Notre objectif consistait à bâtir, à l'aide de la primitive d'écriture distante, un noyau de communication pour machines parallèles de type «grappe de PCs» : MPC-OS. La machine MPC, composée d'un ensemble de nœuds standards de type PC/Intel, raccordés par un réseau d'interconnexion Gigabit de type HSL, a constitué un environnement adéquat pour développer et tester les protocoles et modules auxquels nos réflexions ont abouti.

Pour atteindre notre objectif, on a commencé par étudier en détails les caractéristiques de la primitive d'écriture distante de type zéro-copie fournie par le composant matériel PCI-DDC, et les contraintes auxquelles elle soumet ses utilisateurs.

Après un inventaire des services de haut-niveau attendus par les développeurs d'applications parallèles et les environnements de programmation, on a pu déterminer les problèmes à résoudre pour fournir ces services simplement à partir de la primitive de DMA inter-nœuds. On a alors construit au sein de modules noyau un ensemble de protocoles proposant des solutions aux divers problèmes exposés, la difficulté consistant à conserver dans tous ces protocoles la caractéristique zéro-copie de la communication.

En premier lieu, on a posé le problème de la connaissance de la localisation en mémoire physique des tampons distants pour effectuer une transmission de données. La solution proposée par le protocole SLR/P consiste à implémenter une communication à travers des canaux virtuels et sans copie : la localisation physique du tampon distant est remplacée par un numéro de canal. Un protocole de rendez-vous basé sur ce numéro gère l'échange des adresses physiques des tampons.

On s'est alors posé le problème de la sécurisation de SLR/P vis-à-vis des fautes matérielles du réseau. Pour le résoudre, on a ajouté un certain nombre de mécanismes de contrôle autour des opérations de base de SLR/P, le tout formant SCP/P, un ensemble de protocoles permettant de contourner les erreurs matérielles. La conservation du caractère zéro-copie a été une forte contrainte dans la spécification des mécanismes mis en jeu et de leur enchaînement.

Jusque là, les tampons locaux étaient désignés par leurs adresses physiques. Pour permettre à des processus d'utiliser MPC-OS, on a alors pu imaginer sans difficulté un module de conversion d'adresses, qui forme la couche SLR/V. Les difficultés sont apparues lorsqu'il a fallu sécuriser SLR/V vis-à-vis des comportements des applications. En effet, avec un mode de transmission zéro-copie, c'est le matériel distant qui va écrire les données directement dans l'espace mémoire du processus récepteur, sans que le système d'exploitation ne puisse intervenir. SCP/V, la version sécurisée de SLR/V, devait néanmoins garantir la pérennité des emplacements mémoire en cours de transfert, quelles que soient les opérations effectuées par le processus récepteur sur la structure de son espace d'adressage. Cela a nécessité une interaction délicate avec le module de gestion mémoire du système d'exploitation.

Dans tous ces protocoles, la signalisation de fin de transaction était implémentée sous forme de fonctions de *callback*, invoquées de manière asynchrone. On a alors simplifié l'API d'échange de données en construisant, au dessus de SCP/V, des canaux MDCP qui permettent des échanges de type *read()/write()* bloquants. MDCP effectue une copie dans un tampon de transit uniquement lorsque l'opération *write()* intervient avant l'opération *read()*. Cela permet, malgré le caractère bloquant des appels à MDCP, d'éviter les situations d'interblocage qui, comme on l'a montré, peuvent se produire avec un protocole à la fois zéro-copie et bloquant. Cette API dépourvue de signalisation asynchrone a donc pu être extraite du noyau et fournie à travers une bibliothèque de fonctions en mode utilisateur.

Tout au long de la construction de cet empilement de couches de protocoles et services, on a pris le parti d'implémenter une gestion statique des ressources (tâches, canaux, etc.), et de repousser vers le mode utilisateur leur allocation dynamique. On a pour cela créé un gestionnaire de ressource, le Manager, dont un représentant est présent sur chaque nœud. Il s'agit d'un gestionnaire distribué destiné à implémenter des algorithmes d'allocation de ressource distribués, services fournis simultanément à l'ensemble des tâches et applications distribuées présentes sur la machine. Pour cela, on a construit un cœur d'ORB *multi-thread*, afin d'une part de faciliter la conception les opérations réparties, et d'autre part de permettre leurs activations simultanées. On a ainsi pu développer une charpente d'objet et une charpente de verrou distribuées, à l'aide desquelles on a pu implémenter les algorithmes d'allocation de ressource distribués.

Des campagnes de mesures ont alors permis de quantifier les performances atteintes par les différentes couches de protocoles, mesures cohérentes avec les simulations extraites du modèle mathématique du fonctionnement de PCI-DDC. Les résultats obtenus ont fait apparaître un fossé important entre PUT et les autres couches de protocoles, en termes de latence : on passe de quelques micro-secondes par transmission à quelques dizaines de micro-secondes. La même constatation a été observée à la suite du portage de PVM sur la couche SLR/V, par un doctorant et un stagiaire de DEA du LIP6.

Cela a directement influencé le choix du protocole sur lequel le portage de MPI par un élève ingénieur de l'INT allait être engagé. À la lumière de l'expérience PVM, on a donc choisi PUT, ce qui a accru la difficulté du développement, mais a porté ses fruits : les performances obtenues sont similaires aux performances du portage de MPI sur BIP/Myrinet.

La perte de garantie d'acheminement à travers le réseau constitue l'inconvénient de ce retour aux API bas-niveau. On peut tirer de cette expérience une leçon simple : fournir des garanties de haut-niveau (sécurisation, garanties d'intégrité, canaux virtuels, etc.), au sein de protocoles génériques, apporte un coût important en latence, même si on préserve à tous niveaux le caractère zéro-copie des communications. Les débits ne s'en trouvent d'ailleurs pas pénalisés.

Suite à cette constatation, on a alors choisi une approche non traditionnelle, en entamant une analyse stochastique du problème de la sécurisation, qui s'inscrit dans le domaine de la fiabilité des systèmes redondants, et plus précisément dans celui de l'analyse du couplage des pannes. On a ainsi pu montrer que sous certaines conditions sur les délais de correction d'erreur, un protocole ne traitant que les cas les plus simples (i.e. pas de support de la double-faute), pouvait avoir une efficacité suffisante.

MPC-OS a été installé sur cinq plate-formes MPC, où il a constitué un environnement de programmation parallèle pour divers projets de recherche : les machines des thèmes ASIM et la machine libre-service du LIP6, la machine MPC du PRiSM de l'Université de Versailles, la machine MPC de l'Institut National des Télécommunications d'Evry, et l'émulateur MPC installé au sein de l'équipe SRC du LIP6.

Enfin, dans le cadre de la validation des développements matériels de machines construites à partir des composants PCI-DDC et/ou RCube, MPC-OS a été utilisé par des industriels : GEC Marconi Aerospace Systems (Grande-Bretagne), dans le cadre du projet EUROPRO, et Parsytec Computer (Allemagne), dans le cadre du projet Arches.

Les performances de l'interface PUT sont satisfaisantes, mais on paie au prix fort les services de plus haut niveau fournis par les différentes couches qui viennent s'empiler au dessus. L'expérience MPC-OS nous montre donc qu'une participation du matériel au support de ces services à valeur ajoutée est nécessaire si on veut conserver des performances analogues à celles de PUT. L'équipe de développement matériel de MPC a donc entrepris la réalisation d'un nouveau contrôleur réseau programmable [Desbarbieux, 2000]. On peut donc espérer introduire dans ce matériel programmable les opérations particulièrement coûteuses des couches logicielles actuelles, afin de profiter conjointement de bonnes performances et de services de haut-niveau.

Annexe A

Émulateur MPC

A.1 Daemons hslclient et hslserver

Un mode d'émulation est disponible sur la machine MPC. Il consiste à fournir tous les services d'une machine MPC complète, sans disposer du réseau HSL. Le réseau de contrôle, sous Ethernet, remplace alors le réseau rapide, de façon complètement transparente pour l'utilisateur et pour les couches de communication.

On offre ainsi la possibilité de développer des applications ou de nouveaux protocoles, et d'avoir une première approche de cette machine sans disposer du matériel qui lui est propre. Les performances ne seront pas au rendez-vous mais la machine se comportera exactement comme si elle était dotée des circuits PCI-DDC et RCube.

Pour ce faire, les modules d'émission et de réception de PCI-DDC sont émulés par deux daemons présents sur chaque nœud de calcul : hslclient (émulateur du module d'émission) et hslserver (émulateur du module de réception).

Au démarrage de la machine, le processus hslclient sur chaque nœud de calcul se connecte aux processus hslserver de l'ensemble des autres nœuds, *via* le protocole de communication RPC[1] à travers le réseau de contrôle. Le processus hslclient est un *client* des processus hslserver au sens RPC du terme. La figure A.1 page suivante représente les connexions RPC mises en jeu par un nœud.

Lorsqu'une requête de transfert traverse les couches de protocoles du module HSLDRIVER, la couche la plus basse qui constitue l'interface avec le matériel, et qui se contente en principe de remplir la LPE[2] avec un descripteur de page puis d'annoncer à PCI-DDC la présence de ce nouveau descripteur *via* un de ses registres, va alors se comporter différemment : elle va fournir le descripteur au processus hslclient.

Celui-ci peut alors émuler le module d'émission de PCI-DDC, en effectuant dans l'ordre

1. RPC : Remote Procedure Call, RFC-1050
2. Liste des pages à émettre

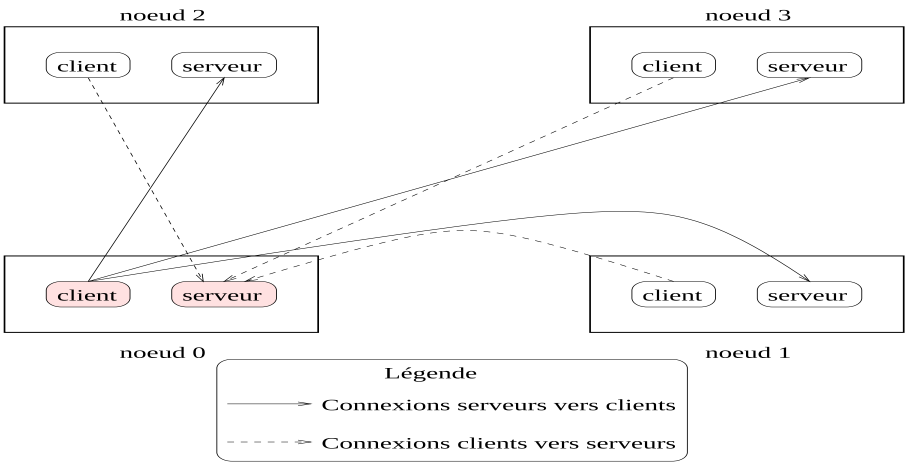


Fig. A.1 *Connexions RPC mises en jeu par le nœud 0 sur une machine à 4 nœuds*

les cinq opérations suivantes :

- ▷ hslclient consulte le descripteur ;
- ▷ Il va chercher les données locales dont il ne dispose que de l'adresse physique, en modifiant directement, et à plusieurs reprises, une entrée de table des pages de la MMU (ce qui est plus rapide que de passer par un remapping système). Cela permet de les rendre accessibles, page de 4 Ko par page de 4 Ko, dans l'espace virtuel du noyau, et par voie de conséquence de pouvoir y accéder en demandant au noyau de les recopier dans son espace virtuel. Cette étape n'est bien sûr pas effectuée s'il s'agit d'un message court ;
- ▷ Il simule, si le descripteur le demande, une interruption de fin d'émission d'une page ;
- ▷ Il consulte une table de routage afin de transcrire le numéro de nœud destinataire sur le réseau HSL en numéro IP sur le réseau de contrôle ;
- ▷ Il émet les données (sauf en cas de message court) et le descripteur vers le daemon récepteur hslserver, à travers une requête RPC.

Lorsque hslserver reçoit un descripteur *via* une requête RPC, il effectue alors les opérations suivantes, pour émuler le module de réception de PCI-DDC :

- ▷ Il consulte le descripteur ;
- ▷ S'il s'agit d'un message normal, il va déposer les données dont il ne connaît que l'adresse physique destinataire. Il utilise pour cela une entrée de table des pages, d'une manière similaire à hslclient ;
- ▷ Il simule, s'il y a lieu, une interruption de fin de réception d'un message.

Les daemons d'émulation sont construits pour émuler un réseau non adaptatif : les pages sont toujours reçues dans l'ordre avec lequel elles ont été émises.

Si on dispose d'un plus grand nombre de nœuds que de cartes HSL, on peut aussi construire une machine dont les nœuds dépourvus de carte utilisent le réseau de contrôle pour se raccorder aux autres. Une table dans le noyau de chaque nœud permet à la couche de communication la plus basse de HSLDRIVER de déterminer si un nœud est accessible *via* le réseau HSL ou s'il faut déléguer la transmission au processus hslclient.

Sur une machine MPC équipée de cartes FastHSL, les daemons hslclient et hslserver sont aussi présents, car ils ont aussi un autre rôle, celui d'initialiser les couches de communication en leur permettant de s'échanger des informations tandis que le réseau HSL n'est pas encore disponible. Ils permettent donc le *bootstrap* des couches de communication (voir section 4.3 page 82).

A.2 Architecture interne de PUT

La figure A.2 page suivante présente les interactions entre la couche PUT et les autres composants matériels et logiciels de la machine MPC. La couche PUT est composée de trois sous-modules :

- *Sous-module d'initialisation* : le sous-module d'initialisation est invoqué par un appel système provenant du daemon hslclient. Il permet de configurer les composants RCube et PCI-DDC, et de copier la table de routage de RCube dans ses registres internes (lien numéroté 1 de la figure A.2 page suivante) ;
- *Sous-module d'écriture distante* : lorsqu'il est invoqué par une application, il transmet des ordres d'écriture distante à PCI-DDC (lien numéroté 2 sur la figure). Mais lorsque l'ordre d'écriture distante est destiné à un nœud qui ne dispose pas de carte FastHSL, l'information est transmise au module d'émulation du daemon hslclient (lien numéroté 2 bis sur la figure) ;
- *Sous-module de traitement des interruptions* : le sous-module de traitement des interruptions reçoit et traite les interruptions provenant de PCI-DDC. Il peut directement être invoqué lors de l'activation d'une interruption matérielle par PCI-DDC (lien numéroté 3 pour une indication de fin d'émission, et 4 pour une indication de réception), ou par les modules d'émulation des daemons hslclient/hslserver (lien numéroté 3 bis pour hslclient, émulateur de la partie émettrice de PCI-DDC; lien numéro 4 bis pour hslserver, émulateur de la partie réceptrice de PCI-DDC).

  Une fois le traitement accompli, le module de traitement d'interruption de PUT signale l'évènement à l'application, à travers un point d'accès au service PUT.

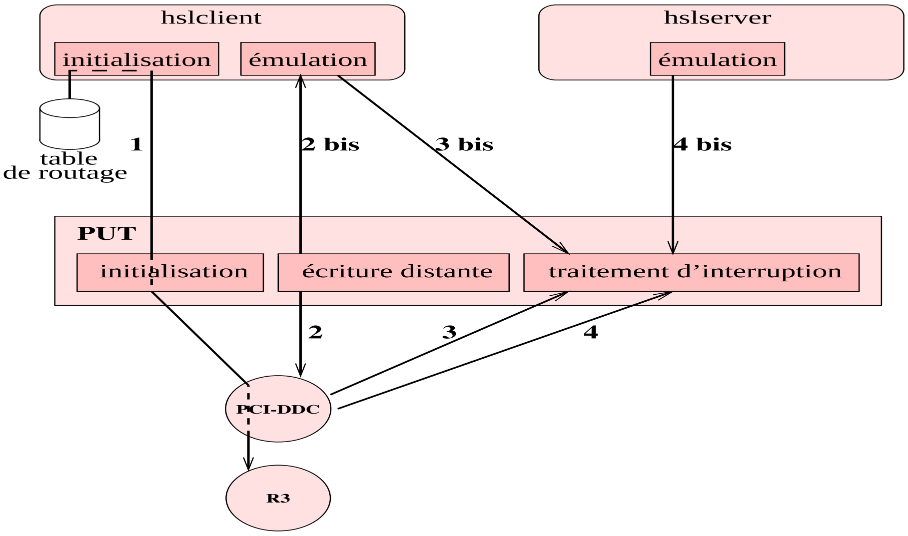


Fig. A.2 *La couche PUT*

Annexe B

Contournements logiciels

Des défauts de conception ont marqué certaines versions des composants matériels, et PUT, en fonction de paramètres de compilation, peut en corriger une partie de manière purement logicielle.

Deux version de PCI-DDC on successivement été fondues :

- ✓ Le document [Wajsbürt *et al.*, 1998] indique l'ensemble des défauts répertoriés suite à la première réalisation de PCI-DDC.
- ✓ Suite à des corrections matérielles au niveau de la couche métal, et à une refonte du circuit, une seconde version de PCI-DDC [1] a vu le jour, corrigeant les plus importants des défauts répertoriés.

Les défauts sont parfois cachés par l'environnement matériel. On a notamment remarqué que le type de contrôleur de bus PCI joue un rôle important dans l'apparition ou non de certains défauts de fonctionnement très pénalisants.

Ainsi, le choix d'apporter un correctif donné au sein de PUT dépend non seulement de la version de PCI-DDC utilisée, mais aussi des caractéristiques matérielles des nœuds de calcul.

Les correctifs ne sont pas directement mis en bijection avec les défauts constatés du PCI-DDC utilisé. On ne pouvait donc pas se contenter d'écrire deux versions de PUT, on a du écrire PUT afin de pouvoir appliquer au cas par cas chaque correctif. L'introduction au sein de PUT de chaque correctif a donc correspondu à déterminer les modifications à appliquer dans les cinq sous-modules de PUT, en prenant soin de ne pas interférer avec les autres correctifs, qu'ils soient activés ou non.

1. On nommera PCI-DDC 1st Run la première version de PCI-DDC, et PCI-DDC 2nd Run la seconde version.

Pour illustrer la problématique d'une telle phase de développement, examinons par exemple les contournements nommés `LPE_SLOW` et `NO_CONFIG_READ`, qui consistent respectivement à garantir que la LPE ne contient pas plus d'une entrée à un instant donné, et à interdire les lectures dans l'espace de configuration PCI lors d'une réception.

Le contournement `LPE_SLOW` est nécessaire dans des cas bien différents :

✓ Dans la première version de PCI-DDC, la condition de passage à l'entrée suivante de la LPE est erronée : il se peut que dans certaines conditions de dépassement de capacité de files internes à PCI-DDC, celui-ci décide de s'occuper de l'entrée suivante, alors que le traitement de l'entrée en cours n'est pas terminé. Le contournement proposé consiste donc à garantir qu'il n'y a pas plus d'une entrée à la fois dans la LPE, afin d'empêcher PUT de passer à l'entrée suivante tant que l'entrée courante n'est pas traitée complètement. Dans la deuxième version de PCI-DDC, la condition de passage a été corrigée par un patch au niveau métal, et le phénomène décrit ici ne se produit plus.

✓ En conditions d'utilisation intensive de PCI-DDC 2nd Run, avec un nœud basé sur un *chipset* de type Intel 440BX, on s'est aperçu que des paquets intermédiaires d'une page pouvaient se retrouver perdus. On ne constate pas ce phénomène avec un *chipset* PCI de type Intel 440LX. De manière empirique, il est apparu que le contournement précédent (`LPE_SLOW`) permettait de s'affranchir de ce phénomène. Les différentes analyses qui ont suivi, ainsi que des documents fournis par Intel, tendent à nous convaincre que ce problème découle ici d'une incartade du bridge 440BX avec la norme PCI, qui apparaît dans des conditions en principe exceptionnelles, mais mises en exergue par des caractéristiques très particulières du fonctionnement de PCI-DDC : celui-ci est commandé par des registres internes présentés dans l'espace de configuration PCI plutôt que dans l'espace I/O, ce qui est assez rare pour un circuit d'interface PCI.

Le contournement `LPE_SLOW` est donc nécessaire systématiquement avec PCI-DDC 1st Run et parfois nécessaire avec PCI-DDC 2nd Run. Pour l'implémenter, PUT remplit la LPE à chaque demande de l'application mais il ne signale à PCI-DDC l'ajout d'entrées que lorsque le traitement en cours est terminé. Il est donc nécessaire que PUT soit invoqué à la fin de chaque transmission, il doit donc rajouter une demande de signalisation de fin d'émission par interruption même si l'utilisateur ne l'avait pas requise. Lors de cette signalisation, PUT doit d'une part informer PCI-DDC de la présence d'une nouvelle entrée, si nécessaire, et d'autre part déterminer si l'utilisateur avait ou non requis une signalisation pour l'entrée dont le traitement vient de s'achever. PUT doit alors invoquer éventuellement la fonction de signalisation fournie par l'utilisateur.

Quant à lui, le contournement `NO_CONFIG_READ` consiste à interdire à PUT toute lecture d'un quelconque registre de PCI-DDC, car si une réception est en cours lors d'une telle lecture, les données déposées peuvent en être altérées. Un patch métal a été mis au point et cette restriction ne s'applique plus à PCI-DDC 2nd Run. Néanmoins, PUT ne peut donc pas, avec PCI-DDC 1st Run, s'informer par exemple de la cause d'une interruption : lorsque le gestionnaire d'interruption est invoqué, le sous-module de gestion d'interruption de PUT est censé déterminer, par la lecture d'un registre interne de PCI-DDC, la cause de cet évènement, et d'agir en conséquence. Il a donc fallu trouver un autre moyen pour déterminer la cause de l'interruption.

Or le contournement précédent, `LPE_SLOW`, force une interruption en émission pour être averti de la fin de traitement d'une entrée de LPE. On constate donc que la mise en œuvre de `NO_CONFIG_READ` a des implications sur la manière dont `LPE_SLOW` doit être codé. Malheureusement, ces deux contournements doivent pouvoir être activés en fonction de conditions matérielles particulières, et indépendantes. On doit pouvoir activer l'un, l'autre, ou les deux simultanément.

Des contournements distincts peuvent donc nécessiter une indépendance totale au niveau de leur activation, mais néanmoins être largement interdépendants au niveau de leur fonctionnement. La difficulté de la mise en place de corrections logicielles dans PUT a donc consisté à laisser la possibilité d'activer chaque contournement à la demande, tout en prenant soin de garantir le fonctionnement correct de chacun quel qu'en soit la combinaison choisie.

Plus précisément, on a procédé par étapes : on a déterminé les défauts nécessitant un même contournement, puis pour chaque contournement les conditions dans lesquelles les défauts associés apparaissent. On a ainsi pu regrouper les contournements qui apparaissent systématiquement dans les mêmes conditions matérielles, afin de les agréger. Puis pour les groupes de contournements restants, on a déterminé les chaînes de dépendances quand elles existaient. On a ainsi pu déterminer que lorsque tel ou tel contournement est nécessaire, tel ou tel autre doit alors être aussi activé. On a donc ainsi pu réduire le nombre de drapeaux de compilation que l'utilisateur doit positionner pour décrire les contournements qu'il désire introduire à la compilation suivant les conditions matérielles de la machine à exploiter.

Sachant qu'il y a plusieurs dizaines de défauts répertoriés au sein de [Wajsbürt *et al.*, 1998] pour PCI-DDC 1st Run, et encore plus d'une dizaine pour PCI-DDC 2nd Run (et parfois disjoints de ceux de PCI-DDC 1st Run), on ne présentera pas ces contournements dans ce manuscrit. Les fichiers sources de l'interface PUT sont très largement documentés en ligne, et incluent toutes les explications nécessaires à la compréhension en détail des méthodes appliquées.

On va se contenter ici d'esquisser quelques-uns des principaux contournements mis en œuvre, dans le tableau qui suit (on notera que l'utilisation d'un PC en mode multiprocesseur a de larges conséquences sur la configuration des circuits de gestion de l'acheminement des interruptions, et que la fréquence de certains défauts en est alors altérée) :

Symptôme	**Conditions**	**Contournement**	**Baisse de perf.**
Perte de données	1st Run et 440BX	LPE_SLOW	✓
Données invalides	1st Run	NO_CONFIG_READ	
Écriture à des emplacements incorrects et pertes de données	1st Run	limitation à 4 octets par paquet	✓
Initialisation de PCI-DDC	1st Run ou 2nd Run	Plusieurs phases d'initialisation successives	
Quelques IRQs non significatives	440BX et monopro	Ignorer IRQs	
Nombreuses IRQs non significatives	440BX et multipro	Ignorer IRQs	✓
Adresses de registres incorrectes	1st Run ou 2nd Run	Remapping par pilote bas niveau	
Lectures PCI incorrectes	440BX	Test de cohérence pour détecter, puis réitérer	
Fonctionnement incorrect de la LMR	1st Run ou 2nd Run	Tables de routage non adaptatives	
Messages courts non fonctionnels	1st Run	Transport dans les MI d'une salve de messages normaux vides	✓
Données désalignées non transmises	1st Run	Réalignement par Copie dans un tampon	✓

Annexe C

Implémentation de TCP/IP sur le réseau Gigabit

Pour démontrer le fonctionnement des composants matériels de MPC, une implémentation de TCP/IP a été mise en place sur une machine MPC exposée lors de la conférence DATE'98[1]. L'architecture matérielle du banc de démonstration est présentée sur la figure C.1 page suivante.

Cette implémentation de TCP/IP fonctionne grâce à une série de daemons PPP distribués sur les nœuds de calcul, et reliés deux à deux à travers un pseudo-pilote de périphérique série, construit au dessus d'un canal SLR/V. La figure C.2 page suivante présente le plan d'adressage utilisé et la figure C.3 page 243 montre les différents composants logiciels en jeu dans chaque nœud de calcul.

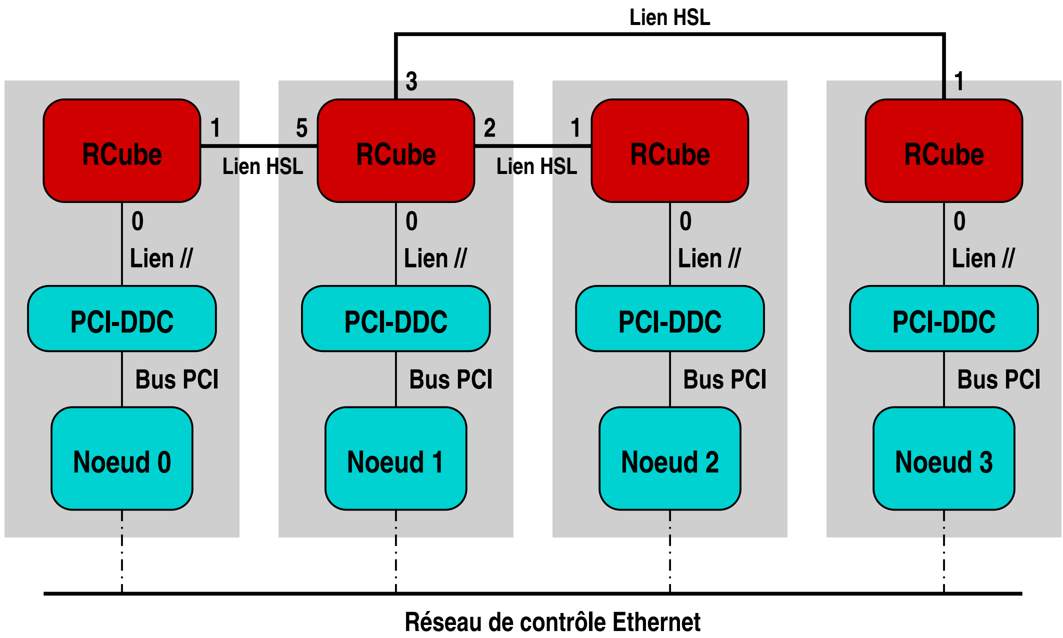


Fig. C.1 *Architecture matérielle du banc de test TCP/IP*

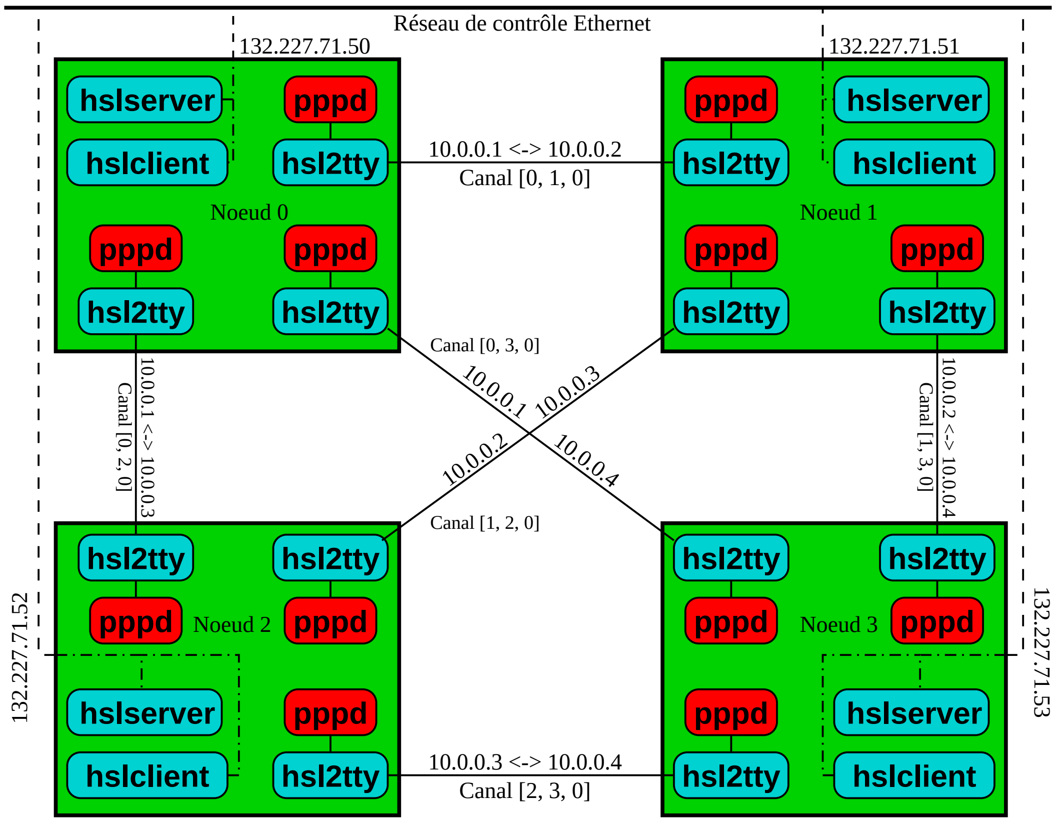


Fig. C.2 *Plan d'adressage IP*

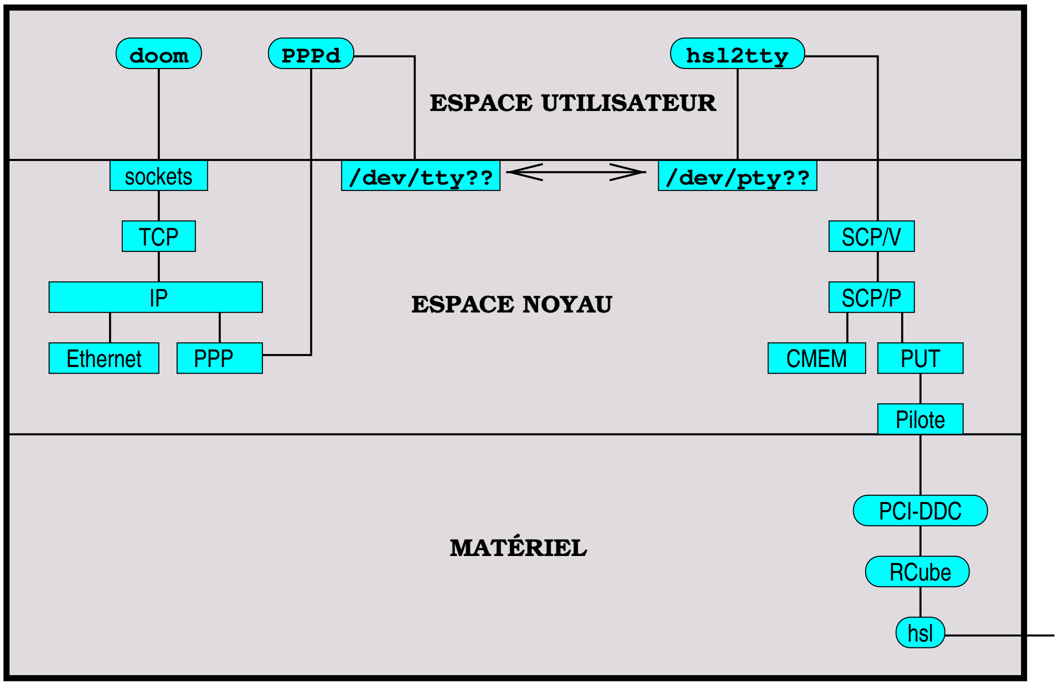


Fig. C.3 *Transmission des données*

Annexe D

Structure de données détaillée de SLR/P

On a décrit, section 5.3.3 page 109, les différentes tables qui forment les structures de données emmagasinant l'état du protocole SLR/P. Voici un aperçu schématique de ces tables ainsi que des champs qui les composent, et notamment ceux qui constituent leurs références croisées. Ce document est destiné à faciliter le travail d'analyse du code source de MPC-OS.

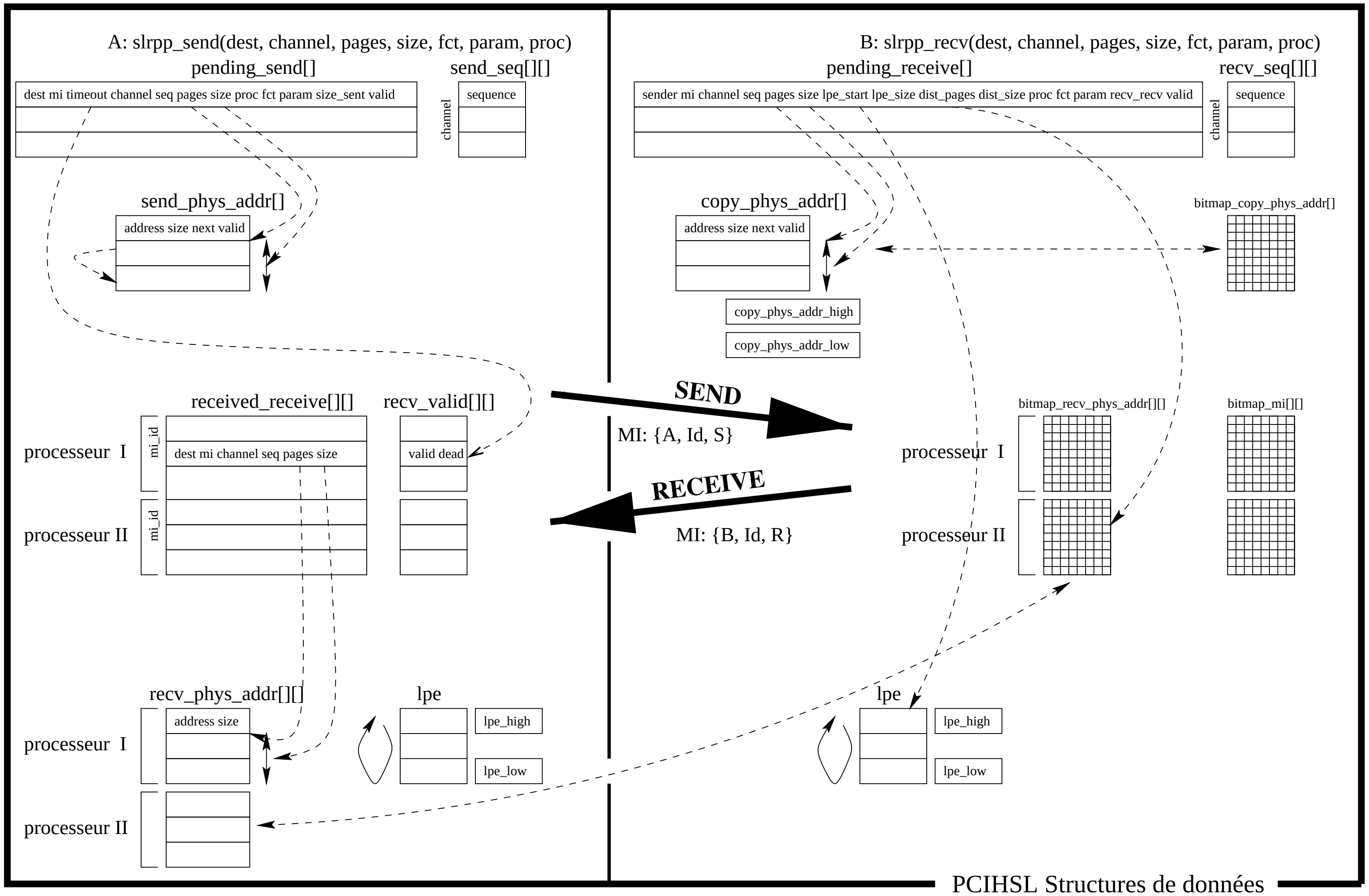


Fig. D.1 *Structures de données en jeu dans SLR/P*

Annexe E

Structure de données détaillée de MDCP

La protocole MDCP, étudié section 7.8.2 page 146, fournit un modèle de communication particulier : lorsqu'une demande d'émission de données se produit alors qu'aucune tâche destinataire n'a jusqu'alors indiqué d'adresse de réception, les données sont envoyées dans un tampon de transit dans le nœud récepteur. La figure E.1 page suivante présente les structures de données qui gèrent ce mode de communication, ainsi que les tampons de transit. Ce document est destiné à faciliter le travail d'analyse du code source de MPC-OS.

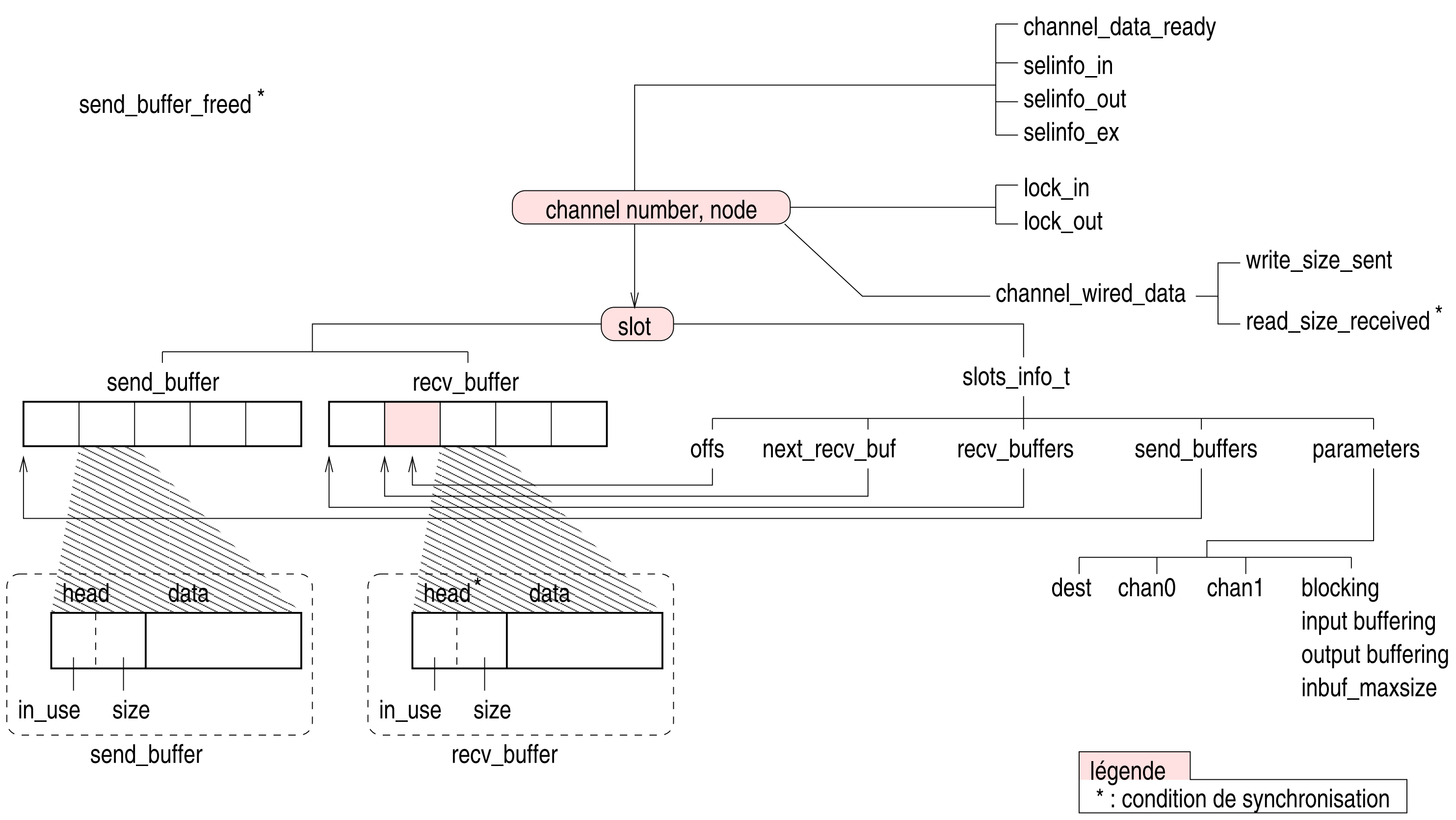


Fig. E.1 *Structures de données en jeu dans MDCP*

Annexe F

Modélisation du matériel

F.1 Objectif

Lorsque nous avons étudié les performances de la machine MPC, nous avons déterminé un certain nombre de constantes caractéristiques des performances ou du fonctionnement de la machine MPC lors d'un échange unidirectionnel. Cela nous a permis de modéliser les interactions entre le matériel et la couche logicielle bas niveau PUT.

Nous désirons maintenant fournir un modèle du comportement interne de PCI-DDC, le valider et enfin le paramétrer à l'aide des résultats précédents, **dans le cadre simplifié d'un échange unidirectionnel, en phase de découplage et au régime stationnaire**. Notre objectif est donc de décrire le fonctionnement interne de PCI-DDC grâce à l'information de son comportement apparent en réaction à des stimuli externes.

La figure F.1 page suivante présente la décomposition schématique interne de PCI-DDC en modules fonctionnels.

Le bus PCI transporte 32 bits de données à 33 MHz. Le lien entre PCI-DDC et RCube transporte 8 bits de données à 66 MHz. Il y a donc une nécessaire adaptation de débit, et, dans le cadre d'une émission, c'est le module FIFO TX qui va jouer ce rôle. C'est donc le comportement de ce module, ainsi que celui du module constructeur de paquet qui le précède et dont dépend son comportement, qu'on va modéliser.

F.2 Construction du paquet

La figure F.2 page suivante décrit l'organisation interne du module FIFO TX et son prédécesseur. Ce dernier comprend un registre à décalage juxtaposé à la FIFO de FIFO TX

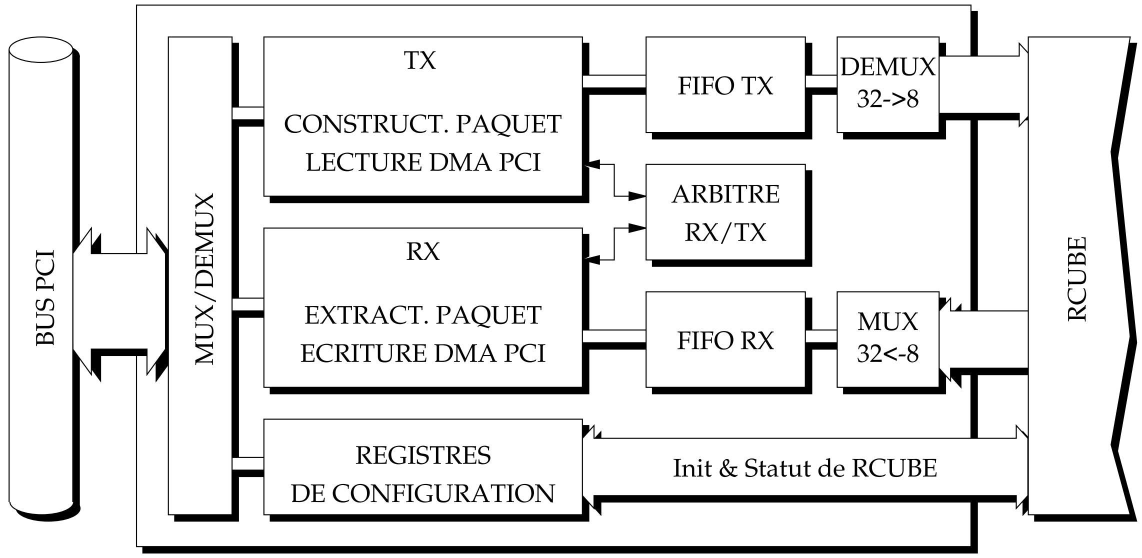


Fig. F.1 *Modules internes à PCI-DDC*

dont la capacité est de 80 mots. Le registre à décalage qui précède la FIFO est décomposé en 2 parties :

❶ Deux mots permettant de garantir la gestion des désalignements entre les plages de données émises et les adresses des tampons de réception ;

❷ Quatre mots qui permettent l'insertion soit d'une en-tête de paquet, soit des trois mots qui terminent un paquet (*End of Data, Data CRC, End of Paquet*).

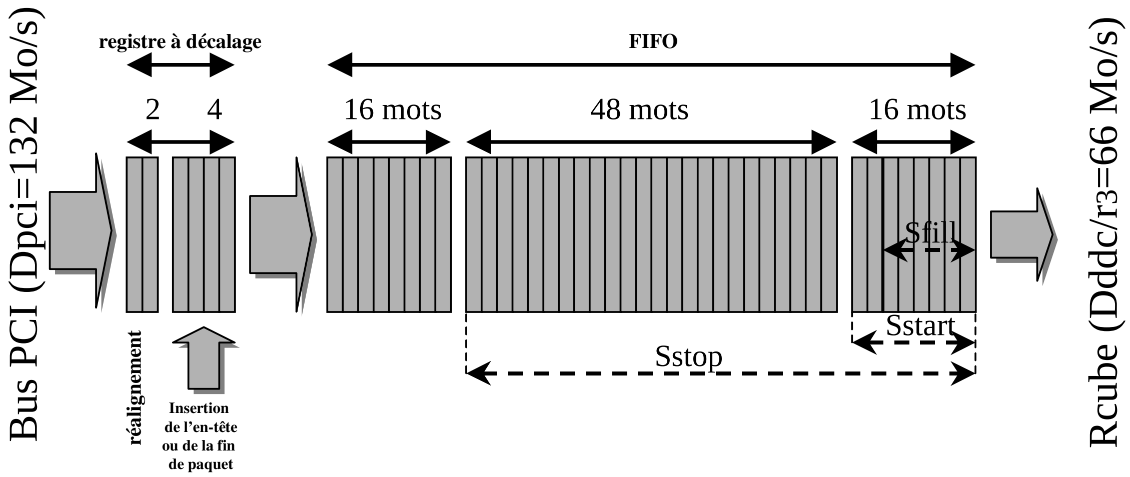


Fig. F.2 *Le module FIFO TX et le registre à décalage qui le précède*

Notons $D_{PCI} = 132\ Mo/s$, le débit avec lequel les données entrent dans FIFO TX. Elles en sortent avec le débit que nous avons déjà noté D_{DDC/R^3} (66 Mo/s).

Le fonctionnement de FIFO TX est le suivant :

▷ La FIFO se remplit jusqu'à ce que l'une des deux conditions suivantes soit satisfaite :
  – Lorsque la FIFO se remplit et arrive à un remplissage de $S_{stop} = 48 + 16$ mots,

la transmission du paquet en cours est interrompue (il faut 1 cycle pour l'interrompre, donc 1 mot de plus pour la taille limite) ;
  - La transmission est aussi interrompue quand toutes les données on été transmises.

▷ Le registre à décalage n'étant alors plus alimenté par le bus PCI, le nombre de mots qu'il contient diminue car il continue à se déverser dans la FIFO. Dès qu'il est vide, les 3 mots de clôture de paquet (notons $S_{footer} = 3$ mots) s'y retrouvent insérés, en un cycle.

▷ Lorsque la FIFO se vide et atteint la taille de $S_{start} = 16$ mots, PCI-DDC décide alors de déclencher la reprise de l'émission. Il demande donc au bridge l'accès au bus PCI.

▷ Une fois que l'accès au bus lui est fourni, PCI-DDC insère dans le registre à décalage en tête de FIFO les 4 mots constituant l'en-tête du paquet suivant (notons $S_{header} =$ 4 mots), en un cycle (s'il s'agit du premier paquet d'une page, PCI-DDC consulte avant cela l'entrée de LPE correspondant, afin de connaître les 4 mots d'en-tête). On désignera par S_{fill} la taille de la FIFO à cet instant.

Pour simplifier notre modèle, on va ignorer les problèmes de désalignement des données : on va supposer que les tampons d'émission et de réception sont toujours alignés sur une frontière de mot.

F.3 Modèle physique

Afin de pouvoir paramétrer notre modèle, définissons les quantités suivantes :

✓ T_{fill} est une inconnue qui représente la durée d'une phase de remplissage de la FIFO, c'est-à-dire la durée d'une phase pendant laquelle PCI-DDC lit sur le bus PCI des données à transférer à travers le réseau HSL.

✓ T_{wait} est une inconnue qui représente la durée d'une phase pendant laquelle PCI-DDC n'échange pas de données avec la mémoire centrale du nœud local. Ainsi, les phases de durées respectives T_{fill} et T_{wait} sont alternées.

✓ T_{grant} représente la durée d'allocation du bus PCI, commençant au moment où PCI-DDC désire débuter l'émission d'un paquet, et se terminant au moment où PCI-DDC reçoit le premier mot de données de ce paquet.

✓ La taille la plus grande qu'un paquet peut prendre, en phase de découplage et au régime stationnaire, a déjà été observée au paragraphe 9.3.2 page 193 : nous l'avions nommée S_{max} et le graphe des mesures expérimentales nous avait permis de déterminer sa valeur : 540 octets. **Nous n'allons pas ici reprendre cette valeur empirique, mais justement utiliser notre modèle pour la redécouvrir. $\mathbf{S_{max}}$ est donc ici considérée comme une inconnue.**

On va dans un premier temps étudier le comportement de la FIFO pour des pages contenant un et un seul paquet, le plus grand possible. Établissons donc les quatre équations qui régissent notre modèle paramétrique dans un tel cas.

T_{fill} et T_{wait} dépendent de la taille des données des paquets. On va donc considérer dorénavant leur valeur pour une taille de données fixée valant S_{max}.

Par définition du délai de remplissage T_{fill}, les données utiles d'un paquet transitent à travers le bus PCI pendant le temps d'une phase de remplissage. Cela se traduit par l'égalité suivante :

$$D_{pci}\, T_{fill} = S_{max} \tag{F.1}$$

PCI-DDC décide d'entrer en phase d'inactivité PCI quand la FIFO atteint 48+16 mots. Rappelons que PCI-DDC prend 1 cycle pour débuter cette phase. Le registre à décalage est plein à cet instant là, car on sort d'une phase d'activité PCI. La phase d'inactivité finit quand PCI-DDC se voit offert l'accès au bus et reçoit à nouveau des mots de données utiles. Par définition, cette phase a pour durée T_{wait}. Pendant ce délai, on vide, du côté RCube :

- ▷ Le contenu de la FIFO au début de la phase : $S_{stop} + 4$ octets (on ajoute 4 octets car il y a un glissement d'un cycle PCI pour l'arrêt de la réception) ;
- ▷ Le contenu des 6 mots du registre à décalage au début de la phase : notons $S_{reg} =$ 6 mots ;
- ▷ une fin de paquet qu'on insère dans le registre à décalage dès qu'il est vide : $S_{footer} =$ 3 mots ;
- ▷ Excepté le contenu de ce qui reste à la fin de la phase : S_{fill}.

Donc on peut établir la relation suivante :

$$D_{DDC/R^3}\, T_{wait} = S_{reg} + S_{footer} + S_{stop} + 4 - S_{fill} \tag{F.2}$$

Pendant la phase de remplissage (T_{fill}), on va extraire depuis le bus PCI toutes les données utiles d'un paquet, puisqu'on clôt le paquet à la fin de cette phase. On va donc vider, du côté RCube, ce qui était présent dans la FIFO au début de cette phase, l'en-tête de paquet qu'on va insérer dans le registre à décalage toujours au début de cette phase, et les données du paquet excepté celles qui vont rester dans la FIFO à la fin de la phase (c'est-à-dire au moment ou elle atteindra la taille $S_{stop} + 4$). On peut donc écrire la relation suivante :

$$D_{DDC/R^3}\, T_{fill} = S_{fill} + S_{max} + S_{header} - S_{stop} - S_{reg} - 4 \tag{F.3}$$

La latence d'attribution du bus et de début de réception de données est représentée par T_{grant}. Par définition, S_{fill} représente la taille de la FIFO au moment où on commence à recevoir des données en provenance du bus PCI. Sachant qu'on demande le bus quand la FIFO a pour taille S_{start}, on peut écrire la relation suivante entre ces trois paramètres :

$$D_{DDC/R^3}\, T_{grant} = S_{start} - S_{fill} \tag{F.4}$$

Avant de résoudre notre système constitué des quatre équations précédentes, déterminons précisément la valeur de T_{grant}. Il s'agit d'estimer le délai cumulé depuis la demande du bus jusqu'à la réception du premier octet de données utiles. Pour cela, nous allons comptabiliser le nombre de cycles PCI nécessaires à chacune des opérations matérielles effectuées pendant ce délai :

- ▷ acquisition du bus (1 cycle si le bus est libre) ;
- ▷ PCI-DDC pose sur le bus l'adresse physique de l'entrée de LPE à lire (1 cycle) ;
- ▷ PCI-DDC retourne le bus (1 cycle) ;
- ▷ 4 cycles pour obtenir depuis la cible PCI (mémoire SDRAM) les 4 mots qui forment une entrée de LPE, plus 1 cycle car on suppose que cette cible est moyennement rapide ;
- ▷ 1 cycle durant lequel le bus est inutilisable ;
- ▷ PCI-DDC pose sur le bus l'adresse physique du début du tampon d'émission, qu'il a lue dans le descripteur de LPE récupéré précédemment (1 cycle) ;
- ▷ PCI-DDC retourne le bus (1 cycle) ;

A partir de là, PCI-DDC reçoit tous les octets de données du tampon d'émission. Il aura donc fallut 11 cycles PCI, soit 330 ns, en supposant que le bus était libre initialement. Il faut noter qu'il n'est pas libre à tout moment, et que suivant les bridges, des cycles PCI supplémentaires peuvent être introduits lors de l'accès au bus. Pour quantifier l'écart entre le délai théorique et celui mis en jeu dans nos mesures, revenons à l'expérience de la section 9.2 page 188, et notamment à la formule de cumul des latences exprimée sur la figure 9.1 page 189. En prenant la valeur de 10 ns pour l'accès mémoire de signalisation, on en extrait une latence matérielle réelle supérieure à la latence théorique de 2100 - 1700 - 10 = 390 ns. On en déduit que $T_{grant} = 330 + 390 = 720\,ns$.

Classons maintenant les variables connues et inconnues qui peuplent nos quatre équations :

- ✓ 8 constantes : D_{pci}, D_{DDC/R^3}, S_{start}, S_{stop}, S_{footer}, S_{header}, S_{reg} et T_{grant}
- ✓ 4 inconnues : S_{max}, S_{fill}, T_{wait} et T_{fill}

La relation F.4 nous fournit $S_{fill} = 16$ octets. En introduisant ce résultat dans la relation F.2, on peut calculer $T_{wait} = 4{,}2\mu s$. La relation F.1 nous permet d'exprimer T_{fill} en fonction de S_{max}. On introduit donc cette expression de T_{fill} dans la relation F.3, et on dispose alors d'une équation contenant une seule inconnue, S_{max}, que l'on peut donc calculer : on trouve $S_{max} = 504$ octets. On injecte finalement cette valeur dans la relation F.1 et on obtient $T_{fill} = 3{,}81\,\mu s$. L'ensemble des inconnues vient d'être déterminé.

Rappelons que c'est l'expression de S_{max} qui nous intéressait particulièrement. En effet, il s'agit de la taille maximale d'un message contenant un seul paquet. Lorsqu'au début de ce chapitre on a établi l'expression théorique 9.3 page 195 du débit en fonction de la taille des données utiles à transporter, on avait paramétré cette expression justement à l'aide de S_{max}, dont on avait estimé la valeur en recherchant le point de discontinuité sur la figure 9.3 page 192 tirée de l'expérience.

On a ici pu déterminer la valeur théorique de S_{max} qui diffère de la valeur mesurée de seulement 6,6%, ce qui permet d'avoir une bonne confiance dans notre modèle décrivant le comportement interne de PCI-DDC.

Annexe G

Compléments mathématiques

Dans un souci de simplicité, le chapitre d'analyse du couplage des fautes ne présente pas les preuves de toutes les propositions qui y figurent, et simplifie parfois certaines définitions. On trouve ici l'ensemble de ces preuves, et les versions plus rigoureuses des définitions, d'un point de vue mathématique.

G.1 Modèle mathématique

Admettons tout d'abord l'existence d'un espace de probabilité $(\Omega,\mathcal{A},P)$ représentant les expériences aléatoires auxquelles on peut être soumis. Chaque expérience, qui commence au temps $t = 0$, consiste à observer le comportement d'une machine MPC constituée de deux nœuds reliés par un unique câble HSL. Ω constitue l'ensemble des épreuves, $\mathcal{A}$ est une σ-algèbre de Ω sur laquelle on a défini la probabilité P, fonction σ-additive telle que $P(\Omega) = 1$. $\mathcal{A}$ représente donc les évènements pour lesquels on peut définir la probabilité P d'occurrence.

Rappelons quelques notations habituelles dans le langage ensembliste et le langage des probabilités, qui vont nous servir par la suite. On notera $\mathcal{F}(X,Y)$ ou Y^X l'ensemble des applications de X dans Y. On notera $\mathcal{L}^p(\Omega,\mathcal{A},P)$ l'ensemble des variables aléatoires réelles définies sur $(\Omega,\mathcal{A},P)$ possédant un moment d'ordre p, et $\mathcal{L}^\infty(\Omega,\mathcal{A},P)$ l'espace des variables $\mathcal{A}$-mesurables P-p.s.[1] bornées. Tout élément de $\mathcal{L}^2(\Omega,\mathcal{A},P)$ sera dit de carré intégrable. Étant donné un espace topologique E, on notera $\mathcal{B}(E)$ la tribu borélienne de E, c'est-à-dire la tribu engendrée par la classe des ouverts de E.

Pour décrire les instants de faute matérielle, introduisons le processus stochastique A à

1. P-p.s. : P-presque sûrement

valeur dans $(\overline{\mathbb{R}},\mathcal{B}(\overline{\mathbb{R}}))$, constitué de la famille $A = (A_i,\, i \in \mathbb{N})$ des variables aléatoires définies sur l'espace probabilisé $(\Omega,\mathcal{A},P)$ et à valeur dans l'espace mesurable $(\overline{\mathbb{R}},\mathcal{B}(\overline{\mathbb{R}}))$. Il s'agit de l'espace des états de A.

La suite (A_i) est strictement croissante car elle décrit les instants successifs de panne. On peut en déduire le processus stochastique X, à valeur dans $(\overline{\mathbb{R}},\mathcal{B}(\overline{\mathbb{R}}))$, des délais d'inter-arrivées des pannes comme suit : $X = (X_i,\, i \in \mathbb{N}) = (A_i - A_{i-1},\, i \in \mathbb{N})$ avec la notation A_{-1} pour désigner une variable aléatoire nulle.

Nous disposons d'un protocole qui permet, en cas de faute simple, de recouvrer le bon fonctionnement de la machine. Définissons les délais de réparation à chacun des instants de panne par le processus $Y = (Y_i,\, i \in \mathbb{N}^*)$, suite de variables aléatoires à valeur dans $(\overline{\mathbb{R}},\mathcal{B}(\overline{\mathbb{R}}))$. Y_i désigne le délai de réparation de la faute qui s'est produite à l'instant A_{i-1}.

Avec les cartes FastHSL de troisième génération, l'expérience nous a montré que le temps moyen entre deux pannes est inférieur à l'heure et est caractérisé par une variance finie. Ces quantités dépendent néanmoins de l'application en cours, ou plus précisément de la charge réseau imposée par cette application : une erreur de lien qui se produit alors qu'aucun paquet ne transite sur le réseau est sans conséquence, et passe donc inaperçue.

Le processus X va représenter les délais d'inter-arrivées de pannes avec une des cartes de troisième génération, dont on vient de signaler qu'elles sont caractérisées par des inter-arrivées possédant une espérance et une variance finies. On va cependant travailler dans un cadre général, sans rien supposer sur les moments des variables en jeu.

Le protocole de correction d'erreur est simpliste car il n'a pas à gérer les doubles-fautes. Si le temps de réparation est majoré, alors Y est constitué d'une famille de variables aléatoires appartenant à $\mathcal{L}^\infty(\Omega,\mathcal{A},P)$. On va cependant, là encore, éviter toute supposition sur les moments de Y, pour ne pas restreindre le champ d'application de notre étude.

L'expérience montre que les erreurs sur les liens sont suffisamment séparées dans le temps pour pouvoir supposer que les variables aléatoires qui composent X sont indépendantes et identiquement distribuées.

Par essence, le fonctionnement du protocole de compensation des fautes matérielles nous permet d'affirmer que les variables aléatoires qui composent Y sont elles-aussi indépendantes et identiquement distribuées.

Enfin, l'apparition d'une faute n'a aucune raison d'être liée de quelque manière que ce soit au protocole de correction d'erreur. On va donc supposer que les variables aléatoires composant X et Y sont mutuellement indépendantes.

G.2 Construction du processus de panne

Nous allons rappeler la construction de la double-faute, puis vérifier qu'il s'agit bien d'une variable aléatoire.

Les processus A et Y étant fixés, il nous faut construire la variable aléatoire définissant l'instant de première double-faute.

Définition 3 Définissons l'application η de Ω dans $\overline{\mathbb{N}^*}$, qui a toute épreuve ω associe l'indice de la première panne double :

$$\forall\, \omega \in \Omega,\, \eta(\omega) = \begin{cases} +\infty & \text{si } \forall\, n \in \mathbb{N}^*,\, X_n(\omega) > Y_n(\omega), \\ min\{n \in \mathbb{N}^*,\, X_n(\omega) \leqslant Y_n(\omega)\} & \text{dans le cas contraire.} \end{cases}$$

Définissons à l'aide de η l'application Z de Ω dans $\overline{\mathbb{R}}$, qui à toute épreuve ω associe le premier délai de panne double :

$$\forall\, \omega \in \Omega,\, Z(\omega) = \begin{cases} A_{\eta(\omega)}(\omega) & \text{si } \eta(\omega) < +\infty, \\ +\infty & \text{dans le cas contraire.} \end{cases}$$

On a construit Z à partir de A, X et Y. Sachant que A peut s'exprimer à partir de X, on en déduit que Z dépend uniquement de X et Y.

Définition 4 Notons Ψ l'application à valeur dans $\mathcal{F}(\Omega; \overline{\mathbb{R}})$ qui à tout couple (X, Y) de processus stochastiques associe $\Psi_{X,Y} = Z$ selon la construction précédente.

On se propose de montrer que Z est une variable aléatoire réelle.

Par définition, Z est une variable aléatoire réelle si $Z^{-1}(\mathcal{B}(\overline{\mathbb{R}})) \subset \mathcal{A}$. C'est ce que nous allons montrer.

Définissons $(\Delta_n, n \in \mathbb{N}^*)$, $(\Lambda_n, n \in \overline{\mathbb{N}^*})$ et Γ_a les parties de Ω suivantes :

$$\begin{aligned} \forall\, n \in \mathbb{N}^*,\; \Delta_n &= \{\omega \in \Omega \,/\, X_n > Y_n\} \\ \forall\, n \in \mathbb{N}^*,\; \Lambda_n &= \{\omega \in \Omega \,/\, \forall i \in \{1, \cdots, n-1\}, X_i > Y_i \text{ et } X_n \leqslant Y_n\} \\ \Lambda_\infty &= \{\omega \in \Omega \,/\, \forall i \in \mathbb{N}^*, X_i > Y_i\} \\ \forall\, a \in \mathbb{R}, \quad \Gamma_a &= \bigcup_{n \in \mathbb{N}^*} A_n^{-1}([-\infty, a[) \cap \Lambda_n \end{aligned}$$

Proposition 1 $\forall\, a \in \mathbb{R}$, $\Gamma_a \in \mathcal{A}$, *la σ-algèbre de Ω sur laquelle on a défini P.*

démonstration :

Soit a un réel quelconque. A_n est une variable aléatoire et $[-\infty, a[$ est un borélien de $\overline{\mathbb{R}}$, donc $A_n^{-1}([-\infty, a[) \in \mathcal{A}$.

$\Delta_n = (X - Y)^{-1}(]0, +\infty])$, donc Δ_n est l'image réciproque d'un borélien de $\overline{\mathbb{R}}$ par une variable aléatoire, donc $\Delta_n \in \mathcal{A}$.

$\forall\, n \in \mathbb{N}^*$, $\Lambda_n = (\cup_{i \in \{1, \cdots, n-1\}} \Delta_i) \cap \Delta_n^c$, donc sachant qu'une tribu est stable par réunion finie, intersection et passage au complémentaire, on obtient $\Lambda_n \in \mathcal{A}$.

On a montré que $A_n^{-1}([-\infty, a[) \in \mathcal{A}$ et que $\Lambda_n \in \mathcal{A}$, donc $A_n^{-1}(-\infty, a[) \cap \Lambda_n \in \mathcal{A}$. D'après la stabilité d'une tribu par réunion dénombrable, on obtient $\Gamma_a \in \mathcal{A}$. □

Proposition 2 $\forall\, \omega \in \Omega, \eta(\omega) < +\infty \;\Rightarrow\; \omega \in \Lambda_{\eta(\omega)}$

démonstration:

Soit $\omega \in \Omega$, tel que $\eta(\omega) < +\infty$. Par définition, $\eta(\omega) = min\{n \in \mathbb{N}^*, X_n(\omega) \leqslant Y_n(\omega)\}$. Donc $\forall\, i \in \{1,\cdots,\eta(\omega)-1\}$, $X_i(\omega) > Y_i(\omega)$ et $X_{\eta(\omega)} \leqslant Y_{\eta(\omega)}$, et donc $\omega \in \Lambda_{\eta(\omega)}$. □

Proposition 3 $\forall\,(n,m) \in \mathbb{N}^{*2}$, $n \neq m \Rightarrow \Lambda_n \cap \Lambda_m = \varnothing$

démonstration :

On va raisonner par l'absurde. Soit m et n un couple d'entiers distincts de $\mathbb{N}^{*2}$, choisis tels que $m < n$. Supposons l'existence de ω appartenant à Λ_m et Λ_n.

Par définition de (Λ_i), on a :

$$\omega \in \Lambda_m \Rightarrow X_m(\omega) \leqslant Y_m(\omega)$$
$$\{\omega \in \Lambda_n \text{ et } m < n\} \Rightarrow X_m(\omega) > Y_m(\omega)$$

Ces deux résultats sont évidemment contradictoires. □

Proposition 4 $\forall\, a \in \mathbb{R}$, $\Gamma_a = (\Psi_{X,Y})^{-1}([-\infty,a[)$

démonstration :

Soit $\omega \in Z^{-1}([-\infty,a[)$. On a donc $Z(\omega) < a$, donc $Z(\omega) < +\infty$, donc $\eta(\omega) < +\infty$ et $A_{\eta(\omega)}(\omega) < a$.

Sachant que $\eta(\omega) < +\infty$, la proposition 2 prouve que $\omega \in \Lambda_{\eta(\omega)}$. On a aussi $A_{\eta(\omega)}(\omega) < a$, donc $\omega \in A^{-1}_{\eta(\omega)}([-\infty,a[)$.

En réunissant ces deux derniers résultats, on obtient $\omega \in A^{-1}_{\eta(\omega)}([-\infty,a[) \cap \Lambda_{\eta(\omega)}$, et par suite $\omega \in \Gamma_a$. Sachant que ceci est valable pour tout ω de $Z^{-1}([-\infty,a[)$, on obtient :

$$\forall\, a \in \mathbb{R},\ \Gamma_a \supset (\Psi_{X,Y})^{-1}([-\infty,a[)$$

Montrons maintenant l'inclusion réciproque.
Soit $\omega \in \Gamma_a$; par définition de Γ_a, $\exists\, n \in \mathbb{N}^* \,/\, \omega \in A_n^{-1}([-\infty,a[)$ et $\omega \in \Lambda_n$.

L'entier n vérifie $\omega \in \Lambda_n$, donc $X_n(\omega) \leqslant Y_n(\omega)$, donc $\eta(\omega) \leqslant n$, donc $\eta(\omega) < +\infty$. D'après la proposition 2, on obtient $\omega \in \Lambda_{\eta(\omega)}$. Or, on a aussi $\omega \in \Lambda_n$. Donc, d'après la proposition 3, $n = \eta(\omega)$.

Notre résultat $\omega \in A_n^{-1}([-\infty,a[)$ peut donc maintenant s'écrire $\omega \in A^{-1}_{\eta(\omega)}([-\infty,a[)$, c'est-à-dire $\omega \in (\Psi_{X,Y})^{-1}([-\infty,a[)$.

Sachant que ceci est valable pour tout ω de Γ_a, on obtient :

$$\forall\, a \in \mathbb{R},\ \Gamma_a \subset (\Psi_{X,Y})^{-1}([-\infty,a[)$$

On a donc démontré que $\Gamma_a = (\Psi_{X,Y})^{-1}([-\infty,a[)$ □

Théorème 3 Soient X et Y deux processus stochastiques à valeur dans $(\overline{\mathbb{R}},\mathcal{B}(\overline{\mathbb{R}}))$, l'application $\Psi_{X,Y}$ de $\mathcal{F}(\Omega;\overline{\mathbb{R}})$ est une variable aléatoire réelle définie sur l'espace probabilisé $(\Omega,\mathcal{A},P)$ et à valeur dans l'espace mesurable $(\overline{\mathbb{R}},\mathcal{B}(\overline{\mathbb{R}}))$.

démonstration :

Il nous faut montrer que l'image réciproque par $\Psi_{X,Y}$ de $\mathcal{B}(\overline{\mathbb{R}})$ est incluse dans $\mathcal{A}$.

Rappelons un résultat classique issu de la théorie de la mesure : pour montrer que Z est une variable aléatoire, il suffit de montrer que l'image réciproque par Z d'une classe d'ensembles boréliens engendrant $\mathcal{B}(\overline{\mathbb{R}})$ est incluse dans $\mathcal{A}$.

La proposition 4 permet d'affirmer que les images réciproques par $\Psi_{X,Y}$ des ensembles $(([-\infty,a[),a \in \mathbb{R})$ appartiennent à $\mathcal{A}$. Or la classe $(([-\infty,a[),a \in \mathbb{R})$ d'ouverts de $\overline{\mathbb{R}}$ engendre $\mathcal{B}(\overline{\mathbb{R}})$. Donc $\Psi_{X,Y}$ est une variable aléatoire réelle définie sur l'espace probabilisé $(\Omega,\mathcal{A},P)$ et à valeur dans l'espace mesurable $(\overline{\mathbb{R}},\mathcal{B}(\overline{\mathbb{R}}))$. $\square$

G.3 Temps moyen avant double-faute

Voici les preuves des propositions utilisées dans le calcul du temps moyen avant double-faute.

Proposition 5 $\forall\, n \in \mathbb{N},\, n \geqslant 2 \Rightarrow \Lambda_\infty \uplus (\uplus_{i \geqslant n} \Lambda_i) = \cap_{1 \leqslant i < n} \Delta_i$

démonstration :

Soit n un entier supérieur ou égal à 2, et soit ω élément de $\Lambda_\infty \uplus (\uplus_{i \geqslant n} \Lambda_i)$.

Supposons tout d'abord que $\omega \in \Lambda_\infty$. Dès lors, $\forall i \in \mathbb{N}^*, X_i(\omega) > Y_i(\omega)$, donc $\forall i \in \mathbb{N}^*, \omega \in \Delta_i$, donc $\forall i \in \{1,\cdots,n-1\}, \omega \in \Delta_i$, et donc $\Lambda_\infty \subset \cap_{1 \leqslant i < n} \Delta_i$.

Supposons maintenant que $\omega \in \uplus_{i \geqslant n} \Lambda_i$. Il existe donc un entier j supérieur ou égal à n, tel que $\omega \in \Lambda_j$, donc pour tout i inférieur strict à j, $\omega \in \Delta_i$, donc pour tout i inférieur strict à n, $\omega \in \Delta_i$, et donc $\uplus_{i \geqslant n} \Lambda_i \subset \cap_{1 \leqslant i < n} \Delta_i$.

Les deux résultats précédents se résument par $\forall n \in \mathbb{N}, n \geqslant 2 \Rightarrow \Lambda_\infty \uplus (\uplus_{i \geqslant n} \Lambda_i) \subset \cap_{1 \leqslant i < n} \Delta_i$.

Il nous faut maintenant montrer l'inclusion contraire. Pour cela, choisissons un entier n supérieur ou égal à 2, et un élément ω de $\cap_{1 \leqslant i < n} \Delta_i$.

Deux cas se présentent alors : soit $\eta(\omega) < +\infty$, soit $\eta(\omega) = +\infty$.

Premier cas : $\eta(\omega) < +\infty$. D'après la proposition 2, on a $\omega \in \Lambda_{\eta(\omega)}$. Sachant que $\omega \in \cap_{1 \leqslant i < n} \Delta_i$, on a évidemment $\eta(\omega) \geqslant n$, donc $\omega \in \uplus_{i \geqslant n} \Lambda_i$.

Deuxième cas : $\eta(\omega) = +\infty$. Par définition, on a, pour tout entier i positif strict, $X_i(\omega) > Y_i(\omega)$, donc $\omega \in \Delta_i$. Ainsi, $\omega \in \Lambda_\infty$.

En regroupant ces deux cas, on montre que $\forall\, n \in \mathbb{N},\, n \geqslant 2 \Rightarrow \Lambda_\infty \uplus (\uplus_{i \geqslant n} \Lambda_i) \supset \cap_{1 \leqslant i < n} \Delta_i$. $\square$

Proposition 6 $\Lambda_\infty \uplus (\uplus_{i \in \mathbb{N}^*} \Lambda_i) = \Omega$

démonstration :

Ω étant l'ensemble des épreuves, on a trivialement $\Lambda_\infty \uplus (\uplus_{i \in \mathbb{N}^*} \Lambda_i) \subset \Omega$.

Soit ω une épreuve quelconque. Deux cas se présentent :

Premier cas : $\eta(\omega) < +\infty$. D'après la proposition 2, on a $\omega \in \Lambda_{\eta(\omega)}$, donc $\omega \in \uplus_{i\in\mathbb{N}^*} \Lambda_i$.

Deuxième cas : $\eta(\omega) = +\infty$. Par définition, on a, pour tout entier i positif strict, $X_i(\omega) > Y_i(\omega)$, donc $\omega \in \Delta_i$. Ainsi, $\omega \in \Lambda_\infty$.

Ces deux cas nous prouvent que $\Lambda_\infty \uplus (\uplus_{i\in\mathbb{N}^*} \Lambda_i) \supset \Omega$. □

Proposition 7 *Soit $d \in \mathbb{N}^*$. Soient U et V deux variables aléatoires indépendantes. V est définie sur $(\Omega,\mathcal{A},P)$ et à valeur dans $(\overline{\mathbb{R}},\mathcal{B}(\overline{\mathbb{R}}))$, tandis que U est définie sur $(\Omega,\mathcal{A},P)$ et à valeur dans $(\mathbb{R}^d,\mathcal{B}(\mathbb{R}^d))$. Soit B un borélien de $\mathbb{R}^d$.*

Alors, la relation suivante est établie : $\int_{U^{-1}(B)} V\, dP = E(V)\, P(U \in B)$

démonstration :

Soit s l'application de $\mathcal{F}(\mathbb{R}^d,\{0,1\})$ qui associe 1 à tout élément de B, et 0 autrement.

Montrons tout d'abord que $s \circ U$ est une variable aléatoire : soit C un borélien de $\mathbb{R}^d$, il nous faut montrer que $(s \circ U)^{-1}(C) \in \mathcal{A}$.

L'image de s est $\{0,1\}$, donc $(s \circ U)^{-1}(C) = (s \circ U)^{-1}(C \cap \{0,1\})$.
On a donc $(s \circ U)^{-1}(C) \in \{(s \circ U)^{-1}(\varnothing),(s \circ U)^{-1}(\{0\}),(s \circ U)^{-1}(\{1\}),(s \circ U)^{-1}(\{0,1\})\}$.

Sachant que $(s \circ U)^{-1}(\varnothing) = \varnothing$, $(s \circ U)^{-1}(\{0\}) = U^{-1}(\mathbb{R}^-)$, $(s \circ U)^{-1}(\{1\}) = U^{-1}(\mathbb{R}^{+*})$ et $(s \circ U)^{-1}(\{0,1\}) = U^{-1}(\mathbb{R})$, on peut conclure que $(s \circ U)^{-1}(C) \in \mathcal{A}$.

Montrons maintenant que $s \circ U$ et V sont indépendantes. Pour cela, il faut montrer que quelque soient S et T deux boréliens respectivement de $\mathbb{R}^d$ et $\overline{\mathbb{R}}$, on a :

$$P((s \circ U)^{-1}(S) \cap V^{-1}(T)) = P((s \circ U)^{-1}(S))\, P(V^{-1}(T))$$

Cette équation est équivalente à la suivante, qu'il faut donc établir :

$$P((s \circ U)^{-1}(S \cap \{0,1\}) \cap V^{-1}(T)) = P((s \circ U)^{-1}(S \cap \{0,1\}))\, P(V^{-1}(T))$$

Il suffit pour conclure à l'indépendance, d'étudier séparément les quatre cas possibles, de la même façon que l'on a démontré que $s \circ V$ est une variable aléatoire.

Effectuons maintenant notre calcul, en utilisant l'indépendance de $s \circ U$ et V :

$$\int_{U^{-1}(B)} V\, dP = \int_\Omega (s \circ U)\, V\, dP = \int_\Omega s \circ U\, dP \int_\Omega V\, dP = \int_{U^{-1}(B)} dP\, E(V)$$

Et par suite, $\int_{U^{-1}(B)} V\, dP = P(U \in B)\, E(V)$. □

G.4 Critère d'existence des moments

Théorème 4 CRITÈRE D'EXISTENCE DES MOMENTS

Hypothèses :
On se place sous les hypothèses du théorème 1 page 212, auxquelles on ajoute la condition $P(X_1 > Y_1) < 1$.

Énoncé :
$\forall n \in \mathbb{N}^*,\ X_1 \in \mathcal{L}^n(\Omega,\mathcal{A},P) \Leftrightarrow \Psi_{X,Y} \in \mathcal{L}^n(\Omega,\mathcal{A},P)$

démonstration :

Soit n un entier naturel non nul.

Supposons tout d'abord que $\Psi_{X,Y} \in \mathcal{L}^n(\Omega,\mathcal{A},P)$. D'après la définition de $\Psi_{X,Y}$, on peut écrire :

$$E(X_1^n) \leqslant E(\Psi_{X,Y}^n) < +\infty$$

Donc $\Psi_{X,Y} \in \mathcal{L}^n(\Omega,\mathcal{A},P) \Rightarrow X_1 \in \mathcal{L}^n(\Omega,\mathcal{A},P)$.

Il nous faut maintenant montrer la relation symétrique. Pour cela, supposons que X_1 admette un moment d'ordre n et majorons $E(\Psi_{X,Y}^n)$. Par définition, on peut écrire :

$$E(\Psi_{X,Y}^n) = \int_\Omega \Psi_{X,Y}^n \, dP = \sum_{p=1}^{\infty} \int_{\Lambda_p} \left(\sum_{i=0}^{p} X_i \right)^n dP$$

Soient a et b deux réels positifs et n un réel supérieur ou égal à 1. La convexité de $t \mapsto t^n$ sur $\mathbb{R}^+$ permet d'établir que $(a+b)^n \leqslant 2^{n-1}(a^n+b^n)$. Cela nous permet de nous débarrasser de X_0 dans l'expression du n-ième moment de $\Psi_{X,Y}$:

$$\begin{aligned}
E(\Psi_{X,Y}^n) &\leqslant 2^{n-1} \sum_{p=1}^{\infty} \int_{\Lambda_p} \left[X_0^n + \left(\sum_{i=1}^{p} X_i \right)^n \right] dP \\
&\leqslant 2^{n-1} \sum_{p=1}^{\infty} \int_{\Lambda_p} X_0^n \, dP + 2^{n-1} \sum_{p=1}^{\infty} \int_{\Lambda_p} \left(\sum_{i=1}^{p} X_i \right)^n dP \\
&\leqslant 2^{n-1} E(X_0^n) + 2^{n-1} \sum_{p=1}^{\infty} \int_{\Lambda_p} \left(\sum_{i=1}^{p} X_i \right)^n dP
\end{aligned}$$

Rappelons la formule générale du développement de la puissance n-ième d'une somme de p éléments :

$$\left(\sum_{i=1}^{p} a_i \right)^n = \sum_{\substack{k_1=0 \\ k_1+\cdots+k_p=n}}^{n} \cdots \sum_{k_p=0}^{n} \frac{n!}{k_1! \cdots k_p!} \, a_1^{k_1} \cdots a_p^{k_p}$$

On injecte ce résultat dans l'expression majorant $E(\Psi^n_{X,Y})$:

$$E(\Psi^n_{X,Y}) \leqslant 2^{n-1}\, E(X_0^n) + 2^{n-1} \sum_{p=1}^{\infty} \left[\sum_{\substack{k_1=0 \\ k_1+\cdots+k_p=n}}^{n} \cdots \sum_{k_p=0}^{n} \frac{n!}{k_1!\cdots k_p!} \int_{\Lambda_p} X_1^{k_1}\cdots X_p^{k_p}\, dP \right]$$

Sachant que tous les termes sont positifs, on peut permuter les séries :

$$\begin{aligned}
E(\Psi^n_{X,Y}) &\leqslant 2^{n-1}\, E(X_0^n) + 2^{n-1} \sum_{\substack{k_1=0 \\ k_1+\cdots+k_\infty=n}}^{n} \cdots \sum_{k_\infty=0}^{n} \left[\sum_{p \geqslant max\{i\in\mathbb{N}^*,\, k_i\neq 0\}} \frac{n!}{k_1!\cdots k_p!} \int_{\Lambda_p} X_1^{k_1}\cdots X_p^{k_p}\, dP \right] \\
&\leqslant 2^{n-1}\, E(X_0^n) + 2^{n-1} \sum_{\substack{k_1=0 \\ k_1+\cdots+k_\infty=n}}^{n} \cdots \sum_{k_\infty=0}^{n} \left[\frac{n!}{k_1!\cdots k_p!} \int_{\substack{\uplus_{i\geqslant p}\Lambda_i \\ p=max\{i\in\mathbb{N}^*,\, k_i\neq 0\}}} X_1^{k_1}\cdots X_p^{k_p}\, dP \right] \\
&\leqslant 2^{n-1}\, E(X_0^n) + 2^{n-1} \sum_{\substack{k_1=0 \\ k_1+\cdots+k_\infty=n}}^{n} \cdots \sum_{k_\infty=0}^{n} \left[\frac{n!}{k_1!\cdots k_p!} \int_{\substack{\cap_{1\leqslant i<p}\Delta_i \\ p=max\{i\in\mathbb{N}^*,\, k_i\neq 0\}}} X_1^{k_1}\cdots X_p^{k_p}\, dP \right]
\end{aligned}$$

Il y a un nombre fini de termes pour lesquels $p \leqslant n$. Il existe donc un entier W_n tel que :

$$E(\Psi^n_{X,Y}) \leqslant W_n + 2^{n-1} \sum_{\substack{k_1=0 \\ k_1+\cdots+k_\infty=n}}^{n} \cdots \sum_{k_\infty=0}^{n} \left[\frac{n!}{k_1!\cdots k_p!} \int_{\substack{\cap_{1\leqslant i<p}\Delta_i \\ p=max\{i\in\mathbb{N}^*,\, k_i\neq 0\}>n}} X_1^{k_1}\cdots X_p^{k_p}\, dP \right] \qquad \text{(G.1)}$$

Si $p > n$, on dispose de l'égalité suivante :

$$\bigcap_{1\leqslant i<p} \Delta_i = \left[\bigcap_{i\in\mathbb{N}/k_i\neq 0} \Delta_{k_i} \right] \bigcap \left[\bigcap_{i\in\{1,\cdots,p\}\setminus\{k_i/k_i=0\}} \Delta_i \right]$$

Les X_i sont mutuellement indépendants, leurs puissances respectives le sont donc aussi (théorème de composition par des fonctions mesurables).

On peut alors exprimer l'intégrale comme suit :

$$\int_{\substack{\cap_{1\leqslant i<p}\Delta_i \\ p=max\{i\in\mathbb{N}^*,\, k_i\neq 0\}>n}} X_1^{k_1}\cdots X_p^{k_p}\, dP = P\left[\bigcap_{i\in\{1,\cdots,p\}\setminus\{k_i/k_i\neq 0\}} \Delta_i \right] \int_{\substack{\cap_{i\in\mathbb{N}/k_i\neq 0}\Delta_{k_i} \\ p=max\{i\in\mathbb{N}^*,\, k_i\neq 0\}>n}} X_1^{k_1}\cdots X_p^{k_p}\, dP$$

Or $\mathrm{Card}(\{i \in \mathbb{N}/k_i \neq 0\}) \leqslant n$ si $k_1 + \cdots + k_\infty = n$. Donc si $p > n$, on a :

$$P\left[\bigcap_{i\in\{1,\cdots,p\}\setminus\{k_i/k_i\neq 0\}} \Delta_i \right] \leqslant P(\Delta_1)^{p-n}$$

D'après les hypothèses, X_1 admet un moment d'ordre n, donc on peut définir un réel positif M_n comme suit :

$$M_n = \max_{p \in \{1, \cdots, n\}} \{E(X_1^p)\}$$

Si $k_1 + \cdots + k_\infty = n$, on dispose de l'encadrement suivant :

$$\int_{\substack{\cap_{i\in\mathbb{N}/k_i\neq 0}\Delta_{k_i} \\ p=max\{i\in\mathbb{N}^*,\, k_i\neq 0\}>n}} X_1^{k_1}\cdots X_p^{k_p}\, dP \leqslant \int_{\substack{\Omega \\ p=max\{i\in\mathbb{N}^*,\, k_i\neq 0\}>n}} X_1^{k_1}\cdots X_p^{k_p}\, dP = \int_\Omega X_1^{k_1}\, dP \cdots \int_{\substack{\Omega \\ p=max\{i\in\mathbb{N}^*,\, k_i\neq 0\}>n}} X_p^{k_p}\, dP \leqslant M_n^n$$

En regroupant les deux dernières majorations, on obtient :

$$\int_{\substack{\cap_{1\leqslant i<p}\Delta_i \\ k_1+\cdots+k_\infty=n,\, p=max\{i\in\mathbb{N}^*,\, k_i\neq 0\}>n}} X_1^{k_1}\cdots X_p^{k_p}\, dP \quad \leqslant M_n^n\, P(\Delta_1)^{p-n}$$

Ce qui nous permet de réécrire la relation G.1 page ci-contre de la manière suivante :

$$\begin{aligned} E(\Psi_{X,Y}^n) &\leqslant W_n + 2^{n-1} \sum_{\substack{k_1=0 \\ k_1+\cdots+k_\infty=n}}^{n} \cdots \sum_{k_\infty=0}^{n} \left[\frac{n!}{k_1!\cdots k_p!} \prod_{\substack{j=1 \\ p=max\{i\in\mathbb{N}^*,\, k_i\neq 0\}>n}}^{p} M_n^n\, P(\Delta_1)^{p-n} \right] \\ &\leqslant W_n + 2^{n-1} \sum_{\substack{k_1=0 \\ k_1+\cdots+k_\infty=n}}^{n} \cdots \sum_{k_\infty=0}^{n} \left[\frac{n!}{k_1!\cdots k_p!}\, \underset{p=max\{i\in\mathbb{N}^*,\, k_i\neq 0\}>n}{M_n^{np}\, P(\Delta_1)^{p(p-n)}} \right] \end{aligned}$$

Les termes étant tous positifs, on va les regrouper suivant les valeurs croissantes de p :

$$\begin{aligned} E(\Psi_{X,Y}^n) &\leqslant W_n + 2^{n-1} \sum_{p=n+1}^{\infty} \left[\sum_{\substack{k_1=0 \\ k_1+\cdots+k_p=n}}^{n} \cdots \sum_{k_{p-1}=0}^{n} \sum_{k_p=1}^{n} \left[\frac{n!}{k_1!\cdots k_p!} M_n^{np}\, P(\Delta_1)^{(p-n)p} \right] \right] \\ &\leqslant W_n + 2^{n-1} \sum_{p=n+1}^{\infty} \left[M_n^{np}\, P(\Delta_1)^{(p-n)p} \sum_{\substack{k_1=0 \\ k_1+\cdots+k_p=n}}^{n} \cdots \sum_{k_p=0}^{n} \frac{n!}{k_1!\cdots k_p!} \right] \end{aligned}$$

On reconnaît dans la série imbriquée le développement de l'expression de $(1 + \underset{\text{p fois}}{\cdots} + 1)^n$, et qui vaut p^n. On peut donc simplifier notre majorant :

$$\begin{aligned} E(\Psi_{X,Y}^n) &\leqslant W_n + 2^{n-1} \sum_{p=n+1}^{\infty} M_n^{np}\, P(\Delta_1)^{(p-n)p}\, p^n \\ &\leqslant W_n + 2^{n-1} \sum_{p=n+1}^{\infty} \left[M_n^n\, P(\Delta_1)^{p-n} \right]^p\, p^n \end{aligned}$$

Sachant que $P(\Delta_1) < 1$, on sait qu'il existe un entier q_n vérifiant $M_n^n\ P(\Delta_1)^{q_n - n} < 1$.

Donc il existe un triplet de réels $(\alpha_n, \beta_n, \gamma_n) \in \mathbb{R}^2 \times [0{,}1[$ tel que :

$$E(\Psi_{X,Y}^n) \leqslant \alpha_n + \beta_n \sum_{p=q_n}^{\infty} \gamma_n^p\, p^n$$

Cette dernière série est absolument convergente dans $\mathbb{C}$ sur le disque ouvert de rayon 1. Il suffit pour s'en persuader d'étudier la n-ième dérivée de $x \mapsto \frac{1}{1-x}$:

$$\frac{d^n}{dx^n}\left[\frac{1}{1-x}\right] = \frac{d^n}{dx^n}\left[\sum_{i=0}^{\infty} x^i\right]$$

Sachant que $P(\Delta_1) < 1$, $\Psi_{X,Y}$ possède un moment d'ordre n. $\square$

Annexe H

API de MPC-OS

H.1 CMEM

`cmem_virtaddr(slot)`: get the virtual start address of a slot.
slot : the slot number
return values : `virtual start address of the slot.`

`cmem_phys_addr(slot)`: get the physical start address of a slot.
slot : the slot number
return values : `physical start address of the slot.`

`cmem_getmem(size, name)`: get a slot of contig. memory.
size : the size of the slot – name : a string to identify the user of the slot
return values : `-1 on error, the slot number on success.`

`cmem_releasemem(id)`: release a slot of contig. memory.
id : the slot number
return values : `-1 on error, 0 on success.`

H.2 PUT

`set_mode_read(minor, mode)`: read HSL/Ethernet mode for a distant node.
minor : board number – mode : distant (p)node
return values : `SUCCESS, ERANGE`

`set_mode_write(minor, mode)`: set HSL/Ethernet mode for a distant node.
minor : board number – mode : distant (p)node and access mode
return values : `SUCCESS, ERANGE`

`put_get_node(minor)`: get the (p)node affected to a board.
minor : board number

return values : `-1 on error, node number on success`

`put_get_mi_start(minor, sap)`: get the first mi allocated for this board.
minor : the board – sap : the registration number affected to the user of this interface
return values : `the required MI`

`put_get_lpe_free(minor)`: the number of free entries in the LPE.
minor : the board number
return values : `the number required`

`put_add_entry(minor, entry)`: add an entry to the LPE.
This function corrects the bogus handling of Short Messages and Misaligned pages in PCIDDC 1st Run.
minor : board number – entry : entry to add.
return values : `SUCCESS, ENXIO, ENOENT, EAGAIN, ERANGE`

`put_register_SAP(minor, send, received)`: register a user of the PUT interface for one board.
minor : the board number – send : the interrupt procedure for data sent – received : the interrupt procedure for data received
return values : `-1 on error, registration number for this SAP on success`

`put_unregister_SAP(minor, id)`: unregister a user of the PUT interface.
minor : the board number – id : the registration number for the SAP
return values : `SUCCESS, ENXIO, ENOENT, EBADF`

`put_attach_mi_range(minor, sap, range)`: ask for the attribution of a range of MI.
minor : the board number – sap : the registration number for this SAP – range : the size of the range
return values : `SUCCESS, ENXIO, ENOENT, EBUSY, E2BIG, EINVAL, ENOMEM`

`put_flush_lpe()`: flush the LPE.
should be called at splhigh processor level, and not from an interrupt handler.
minor : the board number

`put_flush_lmi()`: flush the LMI.
should be called at splhigh processor level, and not from an interrupt handler.
minor : the board number

H.3 SCP/P

`slrpp_reset_channel(node, channel)`: reset the sequence numbers on a channel with a node.
node : remote node – channel : channel identifier

`slrpp_get_nodes()`: get a map of the active nodes on the network.
return values : `a bitmap of active nodes on the network.`

`slrpp_cansend(dest, channel)`: check that there is a pending receive on the other side of the channel.
dest : remote node – channel : channel identifier

return values : `TRUE, FALSE`

`slrpp_send(dest, channel, pages, size, fct, param, proc)`: try to send data to a remote node on a specific channel.
dest : remote node – channel : channel identifier – pages : description of pages to send – size : number of pages to send, each page MUST contain less than 64Kbytes – fct : callback function called when data have been correctly sent – param : parameter provided to the callback function when called – proc : proc structure of the calling process
return values : `SUCCESS, ERANGE, ENXIO, EINVAL, EWOULDBLOCK, EINTR, ERESTART, EFAULT, E2BIG, ENOENT, ENOTCONN, ESHUTDOWN`

`slrpp_recv(dest, channel, pages, size, fct, param, proc)`: ask to receive data from a remote node on a specific channel.
dest : remote node – channel : channel identifier – pages : description of pages ready to receive data – size : number of pages ready to receive data - each page MUST contain less than 64Kbytes – fct : callback function called when data have been correctly received – param : parameter provided to the callback function when called – proc : proc structure of the calling process
return values : `SUCCESS, ERANGE, ENXIO, EINVAL, EWOULDBLOCK, EINTR, ERESTART, ENXIO, E2BIG, ENOENT, ENOMEM, ENOTCONN, ESHUTDOWN`

H.4 SCP/V

`slrpv_cansend(dest, channel)`: check that there is a pending receive on the other side of the channel.
dest : remote node – channel : channel identifier
return values : `TRUE, FALSE`

`slrpv_send(dest, channel, pages, size, fct, param, proc) :`
`slrpv_send_prot(dest, channel, pages, size, fct, param, proc) :`
try to send data to a remote node on a specific channel. slrpv_send_prot() also protect the memory.
dest : remote node – channel : channel identifier – pages : virtual address of the beginning of the data – size : size of data – fct : callback function called when data have been correctly sent – param : parameter provided to the callback function when called – proc : proc structure of the calling process
return values : `SUCCESS, E2BIG, ENXIO, EINVAL, EWOULDBLOCK, EINTR, ERESTART, EFAULT, ENOENT, EIO, ENOTCONN, ESHUTDOWN`

`slrpv_send_piggy_back(dest, channel, pages, size, pages2, size2, fct, param, proc)`:
`slrpv_send_piggy_back_prot(dest, channel, pages, size, pages2, size2, fct, param,`proc) :
try to send data to a remote node on a specific channel. slrpv_send_piggy_back_prot() also protect the memory, except for the biggy backed data.
dest : remote node – channel : channel identifier – pages : virtual address of the beginning of the first set of data – size : size of data – pages2 : virtual address of the beginning of the second set of data – size2 : size of data – fct : callback function called when data have been correctly sent – param : parameter provided to the callback function when called – proc : proc structure of the

calling process
return values : `SUCCESS, E2BIG, ENXIO, EINVAL, EWOULDBLOCK, EINTR, ERESTART, EFAULT, ENOENT, EIO, ENOTCONN, ESHUTDOWN`

`slrpv_send_piggy_back_phys(dest, channel, ptab, size, pages2, size2, fct, param, proc)`:
`slrpv_send_piggy_back_prot_phys(dest, channel, ptab, size, pages2, size2, fct, param,`proc) :
try to send data to a remote node on a specific channel. slrpv_send_piggy_back_prot_phys() also protect the memory, except for the biggy backed data.
dest : remote node – channel : channel identifier – ptab : table of physical areas – size : size of the table – pages2 : virtual address of the beginning of the second set of data – size2 : size of data – fct : callback function called when data have been correctly sent – param : parameter provided to the callback function when called – proc : proc structure of the calling process
return values : `SUCCESS, E2BIG, ENXIO, EINVAL, EWOULDBLOCK, EINTR, ERESTART, EFAULT, ENOENT, EIO, ENOTCONN, ESHUTDOWN`

`slrpv_recv(dest, channel, pages, size, fct, param, proc) :`
`slrpv_recv_prot(dest, channel, pages, size, fct, param, proc) :`
ask to receive data from a remote node on a specific channel. slrpv_recv_prot() also protect the memory.
dest : remote node – channel : channel identifier – pages : virtual address of the beginning of the area ready to receive data – size : size of the area – fct : callback function called when data have been correctly received – param : parameter provided to the callback function when called – proc : proc structure of the calling process
return values : `SUCCESS, E2BIG, ENXIO, EINVAL, EWOULDBLOCK, EINTR, ERESTART, ENXIO, ENOENT, EIO, EACCES, ENOMEM, ENOTCONN, ESHUTDOWN`

`slrpv_recv_piggy_back(dest, channel, pages, size, pages2, size2, fct, param, proc)`:
`slrpv_recv_piggy_back_prot(dest, channel, pages, size, pages2, size2, fct, param,`proc) :
ask to receive data from a remote node on a specific channel. slrpv_recv_piggy_back_prot() also protect the memory, except for the biggy backed data.
dest : remote node – channel : channel identifier – pages : virtual address of the beginning of the first area ready to receive data – size : size of the area – pages : virtual address of the beginning of the second area ready to receive data – size : size of the area – fct : callback function called when data have been correctly received – param : parameter provided to the callback function when called – proc : proc structure of the calling process
return values : `SUCCESS, E2BIG, ENXIO, EINVAL, EWOULDBLOCK, EINTR, ERESTART, ENXIO, ENOENT, EACCES, ENOMEM, ENOTCONN, ESHUTDOWN`

`slrpv_recv_piggy_back_phys(dest, channel, ptab, size, pages2, size2, fct, param, proc)`:
`slrpv_recv_piggy_back_prot_phys(dest, channel, ptab, size, pages2, size2, fct, param,`proc) :
ask to receive data from a remote node on a specific channel. slrpv_recv_piggy_back_prot_phys() also protect the memory, except for the biggy backed data.
dest : remote node – channel : channel identifier – ptab : table of physical areas – size : size of the

table – pages : virtual address of the beginning of the second area ready to receive data – size : size of the area – fct : callback function called when data have been correctly received – param : parameter provided to the callback function when called – proc : proc structure of the calling process
return values : `SUCCESS, E2BIG, ENXIO, EINVAL, EWOULDBLOCK, EINTR, ERESTART, ENXIO, ENOENT, EACCES, ENOMEM, ENOTCONN, ESHUTDOWN`

`slrpv_mlock(lock_addr, lock_len, proc)`: locks into memory
the physical pages associated with the virtual address range starting at lock_addr for lock_len bytes.
lock_addr : beginning of the range – lock_len : length of the range – proc : proc structure of the calling process
return values : `SUCCESS, EINVAL, EAGAIN, ENOMEM`

`slrpv_munlock(lock_addr, lock_len, proc)`: unlocks into memory
the physical pages associated with the virtual address range starting at lock_addr for lock_len bytes.
lock_addr : beginning of the range – lock_len : length of the range – proc : proc structure of the calling process
return values : `SUCCESS, EINVAL, ENOMEM`

H.5 MDCP

`mdcp_init_com(dest, chan1, chan2, classname, proc)`: associate an indirect buffer with two channels.
dest : remote node – classname : classname assocoitated with this pair of channels – proc : calling process
return values : `SUCCESS, ERANGE, EEXIST, ENOMEM, EFAULT, ESHUTDOWN, EISCONN`

`mdcp_end_com(dest, chan, proc)`: free the slot of memory associated with a channel.
dest : remote node – chan : channel representing the slot – proc : proc structure of the calling process
return values : `SUCCESS, ERANGE, ENOENT, ENOMEM, ESHUTDOWN, ENOTCONN, EISCONN, EFAULT, EAGAIN`

`mdcp_getparams(dest, chan, param)`: get parameters.
dest : remote node – chan : channel affected – param : pointer to parameters buffer
return values : `SUCCESS, ERANGE, ENOENT`

`mdcp_setparam_blocking(dest, chan, param)`: set blocking parameter.
dest : remote node – chan : channel affected – param : blocking/non blocking behaviour
return values : `SUCCESS, ERANGE, ENOENT`

`mdcp_setparam_input_buffering(dest, chan, param)`: set input buffering.
dest : remote node – chan : channel affected – param : input buffering
return values : `SUCCESS, ERANGE, ENOENT`

`mdcp_setparam_input_buffering_maxsize(dest, chan, param)`: set input buffering max size.
dest : remote node – chan : channel affected – param : max size for input buffering
return values : `SUCCESS, ERANGE, ENOENT`

`mdcp_setparam_output_buffering(dest, chan, param)`: set output buffering.

dest : remote node – chan : channel affected – param : output buffering
return values : `SUCCESS, ERANGE, ENOENT`

`mdcp_write(dest, channel, buf, size, ret_size, proc)`: write data on a MDCP channel. dest : destination node – channel : main channel to write on – buf : local user data – size : size of local user data – ret_size : variable receiving the size sent – proc : process asking this job
return values : `SUCCESS, ERANGE, ENOENT, E2BIG, ENXIO, EINVAL, EWOULDBLOCK, EINTR, ERESTART, EFAULT, EIO, ENOTCONN, ESHUTDOWN, EMSGSIZE`

`mdcp_read(dest, channel, buf, size, ret_size, proc)`: read data from a MDCP channel. dest : destination node – channel : main channel to write on – buf : local user data – size : size of local user data – ret_size : variable receiving the size sent – proc : process asking this job
return values : `SUCCESS, ERANGE, ENOENT, E2BIG, ENXIO, EINVAL, EWOULDBLOCK, EINTR, ERESTART, ENXIO, ENOENT, EIO, EACCES, ENOMEM, ENOTCONN, ESHUTDOWN, EMSGSIZE`

H.6 SELECT

`hsl_select(...)`: select a source of events from file descriptors and channels.

H.7 LIBMPC

Initialization of the library :

```
void mpc_init(void);
void mpc_close(void);
```

Management of application classes :

```
appclassname_t make_appclass(void);
int delete_appclass(appclassname_t cn);
make_subclass_prefnode(...);
make subclass_raw(...);
```

Channel management :

```
int mpc_get_channel(...);
int mpc_close_channel(...);
```

Access to MDCP read()/write() kernel layers :

```
int mpc_write(pnode_t dest, channel_t, channel, const void *buf,
                                                          size_t nbytes);
int mpc_read(pnode_t dest, channel_t, channel, void *buf, size_t nbytes);
```

Access to hsl_select() kernel layers :

```
void MPC_CHAN_SET(pnode_t dest, channel_t chan, mpc_chan_set *mpcchanset);
```

```
void MPC_CHAN_CLR(pnode_t dest, channel_t chan, mpc_chan_set *mpcchanset);
void MPC_CHAN_ISSET(pnode_t dest, channel_t chan, mpc_chan_set *mpcchanset);
void MPC_CHAN_ZERO(mpc_chan_set *mpcchanset);
int mpc_select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
               mpc_chan_set *mpcreadfds, mpc_chan_set *mpcwritefds,
               mpc_chan_set *mpcexceptfds, struct timeval *timeout);
```

Topology description :

```
int mpc_get_local_infos(u_short *ret_cluster, pnode_t *ret_pnode,
                                                  int *ret_nclusters);
int mpc_get_node_count(int cluster);
```

Task creation :

```
int mpc_spawn_task(char *cmdline, u_short cluster, pnode_t pnode,
                                            appclassname_t cn);
```

H.8 SOCKETWRAP

Fully-supported entry points :

```
socket(); close(); dup(); dup2(); listen(); bind(); accept(); connect();
read(); recv(); recvfrom(); recvmsg(); write(); send(); sendto();
sendmsg(); select();
```

Partially-supported entry points :

```
ioctl(); fcntl(); getsockopt(); setsockopt(); fork(); exec();
```

Bibliographie

[Acher *et al.*, 1999] G. Acher, W. Karl, and M. Leberecht. The TUM PCI/SCI adapter. *SCI: scalable coherent interface. Architecture and software for high-performance computer clusters*, 1999.

[Alasdair *et al.*, 1994] R. Alasdair, A. Bruce, J. Mills, and A. Smith. Technical Report EPCC-KTP-CHIMP-V2-USER 1.2 – CHIMP/MPI User Guide – Parallel Computing Centre, Edinburgh, UK, June, 1994.

[Anderson *et al.*, 1995] Thomas E. Anderson, David E. Culler, and David A. Patterson. A Case for Networks of Workstations : NOW. *IEEE Micro*, February 1995.

[ANL, 2001] ANL. Mpich – a portable implementation of mpi – argonne national laboratory, mpich – a portable implementation of mpi, http://www.mcs.anl.gov/mpi/mpich/, 2001.

[Aumage *et al.*, 2000] O. Aumage, L. Bouge, A. Denis, J.-F. Mehaut, G. Mercier, R. Namyst, and L. Prylli. Madeleine II: a portable and efficient communication library for high-performance cluster computing. In *Proceedings IEEE International Conference on Cluster Computing. CLUSTER 2000*, dec 2000.

[Avresky *et al.*, 1999] D.R. Avresky, V. Shurbanov, R. Horst, and P. Mehra. Performance evaluation of the ServerNet SAN under self-similar traffic. In *Proceedings 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing. IPPS/SPDP*, 1999.

[Bertozzi *et al.*, 2001] M. Bertozzi, M. Panella, and M. Reggiani. Design of a VIA based communication protocol for LAM/MPI suite. In *Proceedings Ninth Euromicro Workshop on Parallel and Distributed Processing*, feb 2001.

[Boden *et al.*, 1995] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W-K. Su. Myrinet, A Gigabit-per-Second Local Area Network. *IEEE Micro*, feb 1995.

[Bouaraoua, 1998] Abdelhafid Bouaraoua. *Mise en œuvre, l'evaluation des performances et la vérification de la validité de topologies et de schémas de routage pour l'aide à la conception de réseaux d'interconnexion pour architectures parallèles.* PhD thesis, Université Paris VI, May 1998.

[Bryant *et al.*, 2000] R. Bryant, B. Hartner, Qi He, and G. Venkitachalam. SMP scalability comparisons of Linux kernels 2.2.14 and 2.3.99. In *Proceedings of 4th Annual Linux Showcase and Conference*, 2000.

[BSD Report, 2000] BSD Report. BSD, the other open source Unix. *Information Systems Control Journal*, 2000.

[Butler and Lusk, 1994] R. Butler and E. Lusk. parallel programming system – Monitors, messages, and clusters : The P4 parallel programming system. Parallel Comput. 20 (April 1994), 547–564., 1994.

[Buyya, 1999] Rajkumar Buyya. *High Performance Cluster Computing, vol.1 : Architecture and Systems*. Éd. Prentice Hall, 1999.

[Cadinot *et al.*, 1997] P. Cadinot, N. Dorta, and B. Folliot. MAÎS : un système de pagination en mémoire distante dans un environnement réseau à haut débit. *RenPar'97*, May 1997.

[Cappello *et al.*, 2001] F. Cappello, O. Richard, and D. Etiemble. Understanding performance of SMP clusters running MPI programs. *Future Generation Computer Systems*, 2001.

[Carriero *et al.*, 1994] Nicholas J. Carriero, David Gelernter, Timothy G. Mattson, and Andrew H. Sherman. The Linda alternative to message-passing systems. *Parallel Computing*, 20(4):633–655, 1994.

[Chelius, 2001] G. Chelius. Implementing Virtual Interface Architecture on top of the GM message passing interface. In *Proceedings of First IEEE/ACM International Symposium on Cluster Computing and the Grid*, may 2001.

[Chung *et al.*, 2000] Sang-Hwa Chung, Soo-Cheol Oh, Sejin Park, Hankook Jang, and Chi-Jung Ha. A CC-NUMA prototype card for SCI-based PC clustering. In *Proceedings IEEE International Conference on Cluster Computing. CLUSTER 2000*, 2000.

[Cochran, 1999] R. Cochran. ATM: sales finally match the hype. In *Business Communications Review*, 1999.

[Cocozza-Thivent, 1997] Christiane Cocozza-Thivent. *Processus stochastiques et fiabilité des systèmes*. Éd. Springer, 1997.

[Cook, 1993] Jeremy Cook. A First Course in Programming the Intel Paragon - http://www.cica.indiana.edu/iu_hpc/paragon/pgon-tutorial/pgon.html, 1993.

[Cray Systems, 1993] Cray Systems. Cray Research expands successful CRAY Y-MP C90 system into series. *Cray Channels*, 1993.

[Cray Systems, 2000] Cray Systems. Cray t3d http://www.cray.com/products/systems/gallery/t3d.html, 2000.

[Cyliax, 2000] I. Cyliax. Catching the PCI bus - A spin through the background. *Circuit Cellar Ink*, 2000.

[Desbarbieux, 2000] Jean-Lou Desbarbieux. *Conception et réalisation d'un contrôleur réseau programmable pour machine parallèle de type grappe de PC.* PhD thesis, Université Paris VI, June 2000.

[Dunning *et al.*, 1998] Dave Dunning, Greg Regnier, Gary McAlpine, Don Cameron, Bill Shubert, Frank Berry, Anne Marie Merritt, Ed Gronke, and Chris Dodd. The Virtual Interface Architecture. *IEEE Micro*, 18(2), 1998.

[Eicken *et al.*, 1995] T. Eicken, A. Basu, V. Buch, and W. Vogels. U-Net : A user-level network interface for parallel and distributed computing – Proceedings of the 15th ACM Symposium on Operating Systems Principles, December, 1995.

[Eicken, 1994] T. Eicken. Building parallel programming models using Active Messages. In *Proceedings of the Second Workshop on Environments and Tools for Parallel Scientific Computing*, 1994.

[Evans *et al.*, 2000] G. Evans, J. Blackledge, and P. Yardley. *Analytic methods for Partial Differential Equations.* Éd. Springer, 2000.

[Fagg and Dongarra, 2000] G.E. Fagg and J.J. Dongarra. FT-MPI: fault tolerant MPI, supporting dynamic applications in a dynamic world. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 7th European PVM/MPI Users' Group Meeting*, 2000.

[Felderman *et al.*, 1994] R. Felderman, A. Deschon, D. Cohen, and G. Finn. ATOMIC : a high-speed local communication architecture. *Journal of High Speed Networks*, 1994.

[Felten *et al.*, 1996] E. W. Felten, R. D. Alpert, A. Bilas, M. A. Blumrich, D. W. Clark, S. M. Damianakis, C. Dubnicki, L. Iftode, and K. Li. Early Experience with Message-passing on the SHRIMP Multicomputer. In *Proc. of the 23rd Annual Int'l Symp. on Computer Architecture (ISCA'96)*, pages 296–307, 1996.

[Foata and Fuchs, 1998] Dominique Foata and Aimé Fuchs. *Calcul des probabilités.* Éd. Dunod, 1998.

[Foldvik and Meyer, 1995] R.G. Foldvik and D. Meyer. Moving towards ATM: LAN/WAN evolution and experimentation at the University of Oregon. In *Proceedings of 20th*

Conference on Local Computer Networks, 1995.

[fos, 1996] *Double Standards : Bringing Task Parallelism to HPF Via the Message Passing Interface*, 1996.

[Gao, 2000] Guohong Gao. Designing efficient fault-tolerant systems on wireless networks. In *Proceedings of ISW 2000. 34th Information Survivability Workshop*, 2000.

[Geist *et al.*, 1994] A. Geist, A. Beguelin, Jack Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing.* MIT Press, Cambridge, Mass., 1994.

[Getchell and Rupert, 1992] D. Getchell and P. Rupert. Fiber Channel in the Local Area Network – IEEE LTS, pages 38–42, may, 1992.

[Gillett and Kaufmann, 1997] R. Gillett and R. Kaufmann. Using the Memory Channel Network – IEEE Micro, 17(1), 1997.

[Gillett, 1996] R. Gillett. Memory Channel : An Optimized Cluster Interconnect – IEEE Micro, 16(2):12-18, feb, 1996.

[Girardi *et al.*, 1996] G. Girardi, D. Ercole, M. Devault, and Y. Rouaud. IP applications through public ATM networks using switched VCs. In *ATM, Networks and LANs, NOC '96. Proceedings of the European Conference on Networks and Optical Communications*, 1996.

[Gnédenko *et al.*, 1972] B. Gnédenko, Y. Bélliaev, and A. Soloviev. *Méthodes mathématiques en théorie de la fiabilité.* Éd. MIR, 1972.

[Greiner *et al.*, 1998] Alain Greiner, Pierre David, Jean-Lou Desbarbieux, Alexandre Fenyö, Jean-Jacques Lecler, Frédéric Potter, Vincent Reibaldi, Franck Wajsbürt, and Belkacem Zerrouk. La machine MPC. In Hermes, editor, *Calculateurs Parallèles, Réseaux et Systèmes répartis*, volume 10, pages 71–84, February 1998.

[Gustavason, 1992] D. Gustavason. Scalable Coherent Interface and Related Standards Projects – IEEE MICRO, Vol. 12, No. 1, 10–22, feb, 1992.

[Heath *et al.*, 2001] T. Heath, S. Kaur, R.P. Martin, and T.D. Nguyen. Quantifying the impact of architectural scaling on communication. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, 2001.

[Hey, 1992] A.J.G. Hey. Parallel universal message-passing and applications. In *Transputers '92. Proceedings of of the International Conference : Advanced Research and Industrial Applications*, 1992.

[Huang *et al.*, 1999] Yiqing Huang, Z. Kalbarczyk, and R.K. Iyer. Error recovery schemes and fast simulation for ServerNet network software interface. In *Proceedings of PDCS-99: 12th International Conference on Parallel and Distributed Computing Systems*, 1999.

[Hwang *et al.*, 1997] Kai Hwang, Choming Wang, and Cho-Li Wang. Evaluating MPI collective communication on the SP2, T3D, and Paragon multicomputers. In *Proceedings. Third International Symposium on High-Performance Computer Architecture*, 1997.

[Iannello *et al.*, 1998] G. Iannello, M. Lauria, and S. Mercolino. LogP performance characterization of fast messages atop Myrinet. In *Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Processing - PDP'98*, 1998.

[Ionescu *et al.*, 1999] D.C. Ionescu, N. Limnios, and editors. *Stastistical and Probabilistic Models in Reliability.* Birkhäuser, 1999.

[Ito, 2000] Y. Ito. Fujitsu cluster technology : Synfinity series. *Fujitsu*, 2000.

[Itoh *et al.*, 2000] M. Itoh, T. Ishizaki, and M. Kishimoto. Accelerated socket communications in system area networks. In *Proceedings IEEE International Conference on Cluster Computing. CLUSTER 2000*, dec 2000.

[Kaieda *et al.*, 2001] A. Kaieda, Y. Nakayama, A. Tanaka, T. Horikawa, T. Kurasugi, and I. Kino. Analysis and measurement of the effect of kernel locks in SMP systems. *Concurrency and Computation Practice and Experience*, 2001.

[Kallenberg, 1997] Olav Kallenberg. *Foundations of Modern Probability.* Springer, 1997.

[Kishimoto *et al.*, 2000] M. Kishimoto, N. Ogawa, T. Kurosawa, K. Fukui, N. Tachino, A. Savva, and N. Shiratori. High performance communication system for UNIX cluster system. *Transactions of the Information Processing Society of Japan*, 2000.

[Kleinrock, 1975] Leonard Kleinrock. *Queueing Systems, vol.1 : theory.* Éd. Wiley-Interscience, 1975.

[Korolyuk, 1997] Vladimir S. Korolyuk. Stochastic Models of Systems in Reliability Problems. In *Proceedings of 1st International Conference on Mathematical Methods in Reliability*, Bucarest, Roumanie, September 1997.

[Kurmann *et al.*, 2001] C. Kurmann, F. Rauch, and T.M. Stricker. Speculative defragmentation - leading Gigabit Ethernet to true zero-copy communication. In *Proceedings the Ninth International Symposium on High-Performance Distributed Computing*, 2001.

[Lee *et al.*, 2000] Jae Min Lee, Wook Hyun Kwon, Young Shin Kim, and Hong-Ju Moon. Physical layer redundancy method for fault-tolerant networks. In *Proceedings of WFCS*

2000. 3rd IEEE International Workshop on Factory Communication, 2000.

[Liaaen and Kohmann, 1999] M.C. Liaaen and H. Kohmann. Dolphin SCI adapter cards. *SCI: scalable coherent interface. Architecture and software for high-performance compute clusters*, 1999.

[McKusick *et al.*, 1996] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quaterman. *The Design and Implementation of the 4.4BSD Operating System.* Addison Wesley, 1996.

[Mei-E, 2000] Dai Mei-E. Analysis for model LogP of high performance network parallel computation. *Mini-Micro Systems*, 2000.

[Metcalfe and Boggs, 1976] D. E. Metcalfe and R.M. Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the Association of Computing Machinery*, 19(7):395–404, 1976.

[Michalickova *et al.*, 2000] K. Michalickova, M. Dharsee, and C.W.V. Hogue. Sequence analysis on a 216-processor Beowulf cluster. In *Proceedings of 4th Annual Linux Showcase and Conference*, 2000.

[Moore, 1965] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics, vol 38*, 1965.

[Namyst *et al.*, 1995] R. Namyst, J. Mehaut, and P. Parallel. PM2: Parallel multithreaded machine. A computing environment for distributed architectures – In Parallel Computing (ParCo'95), pages 279–285. Elsevier Science Publishers, Sept. 1995., 1995.

[Nishimura *et al.*, 2000] S. Nishimura, T. Kudoh, H. Nishi, J. Yamamoto, K. Harasawa, N. Matsudaira, S. Akutsu, and H. Amano. 64-Gbit/s highly reliable network switch (RHiNET-2/SW) using parallel optical interconnection. *Journal of Lightwave Technology*, dec 2000.

[Ong and Farrell, 2000] Hong Ong and P.A. Farrell. Performance comparison of LAM/MPI, MPICH, and MVICH on a Linux cluster connected by a gigabit ethernet network. In *Proceedings of 4th Annual Linux Showcase and Conference*, oct 2000.

[Pakin *et al.*, 1997] S. Pakin, V. Karamcheti, A. Chien, and M. Efficient. Fast Messages: Efficient, portable communication for workstation clusters and MPPs – IEEE Concurrency, 5(2):60–73, apr, 1997.

[Potter, 1996] Frédéric Potter. *Conception et réalisation d'un réseau d'interconnexion à faible latence et haut débit pour machines multiprocesseurs.* PhD thesis, Université Paris VI, April 1996.

[Prakash, 2000] K. Prakash. An attempt to completely utilize the bandwidth capability of PCI-X 133 MHz devices in a 66 MHz PCI-X. In *Proceedings Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region*, 2000.

[Prylli and Tourancheau, 1998] Loïc Prylli and Bernard Tourancheau. BIP : A New Protocol Designed for High Performance Networking on Myrinet. In *IPPS/SPDP Workshops*, pages 472–485, 1998.

[Prylli, 1998] Loïc Prylli. BIP Messages User Manual for BIP 0.94. Technical report, Université de Lyon, 1998. `http://www-bip.univ-lyon1.fr/bip.html`.

[Pétrot *et al.*, 1997a] Frédéric Pétrot, Denis Hommais, and Alain Greiner. A Simulation Environment for Core Based Embedded Systems. In *Proceeding of the* 30^{th} *IEEE International Simulation Symposium*, pages 86–91, Atlanta, Georgia, April 1997. IEEE.

[Pétrot *et al.*, 1997b] Frédéric Pétrot, Denis Hommais, and Alain Greiner. Cycle Precise Core Based Hardware/Software System Simulation with Predictable Event Propagation. In *Proceeding of the* 23^{rd} *Euromicro Conference*, pages 182–187, Budapest, Hungary, September 1997.

[Ramachandran *et al.*, 1996] U. Ramachandran, G. Shah, S. Ravikumar, and J. Muthukumarasamy. Scalability Study of the KSR-1. *Parallel Computing, vol 22*, 1996.

[Rangarajan and Iftode, 2000] M. Rangarajan and L. Iftode. Software distributed shared memory over Virtual Interface Architecture: implementation and performance. In *Proceedings of 4th Annual Linux Showcase and Conference*, oct 2000.

[Raynal, 1992] Michel Raynal. *Synchronisation et état global dans les systèmes répartis.* Éd. Eyrolles, 1992.

[Reibaldi, 1997] Vincent Reibaldi. *Conception et réalisation d'un routeur de paquets à hautes performances.* PhD thesis, Université Paris VI, July 1997.

[Renault *et al.*, 2000] E. Renault, P. David, and P. Feautrier. PCI-DDC application programming interface: performance in user-level messaging. In *Proceedings of Euro-Par 2000 Parallel Processing. 6th International Euro-Par Conference.*, sept 2000.

[Revuz, 1997] Daniel Revuz. *Probabilités.* Éd. Hermann, 1997.

[Éric Renault, 2000] Éric Renault. *Étude de l'impact de la sécurité sur les performances dans les grappes de PC.* PhD thesis, Université de Versailles Saint-Quentin-en-Yvelines, 2000.

[Rindos *et al.*, 1996] A. Rindos, S. Woolet, L. Nicholson, and M. Vouk. A performance evaluation of emerging Ethernet technologies: switched/high-speed/full-duplex Ethernet and Ethernet LAN emulation over ATM. In *Proceedings of SOUTHEASTCON '96. Bringing Together Education, Science and Technology*, 1996.

[Rowe, 1997] M. Rowe. PCI bus cards and PCs must be compatible. *Test and Measurement World*, 1997.

[Shiryaev, 1989] A.N. Shiryaev. *Probability*. Springer, 1989.

[Shurbanov *et al.*, 2000] V. Shurbanov, D. Avresky, P. Mehra, and W. Watson. Flow control in ServerNet clusters. In *Proceedings of Euro-Par 2000. European Conference on Parallel Computing*, 2000.

[Silva and Mana, 1998] Fabricio Silva and Karim Mana. PVM port to MPC. Technical report, Laboratoire d'informatique de Paris VI, 1998. `http://mpc.lip6.fr/pvm.html`.

[Stallings, 2000] W. Stallings. Gigabit Ethernet. *Dr. Dobb's Journal*, 2000.

[Sterling *et al.*, 1995] 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 Processing*, pages I:11–14, Oconomowoc, WI, 1995.

[Stunkel *et al.*, 1995] C. B. Stunkel, D. G. Shea, B. Abali, M. G. Atkins, C. A. Bender, D. G. Grice, P. Hochschild, D. J. Joseph, and B. J. The SP2 High-Performance Switch. *IBM Systems Journal 34, No. 2*, 1995.

[Sun Microsystems, 2001] Sun Microsystems. Sun SBus
http://www.sun.com/io_technologies/sbus, 2001.

[Tanenbaum, 1995] Andrew S. Tanenbaum. *Distributed Operating Systems.* Prentice Hall, 1995.

[Thinking Machines, 1998] Thinking Machines. CM-2, CM-2a, CM-200 and CM-5 - http://nhse.npac.syr.edu/hpccsurvey/orgs/tmc/tmc.html, 1998.

[Tolmie *et al.*, 1999] D. Tolmie, T.M. Boorman, A. DuBois, D. DuBois, W. Feng, and I. Philp. From HiPPI-800 to HiPPI-6400: A changing of the guard and gateway to the future. In *Proceedings. 6th International Conference on Parallel Interconnects (PI'99) (Formerly Known as MPPOI)*, 1999.

[Tourancheau and Westrelin, 2000] B. Tourancheau and R. Westrelin. Study of the medium message performance of BIP/Myrinet. In *Proceedings IEEE International Conference on Cluster Computing. CLUSTER 2000*, dec 2000.

[Touyama and Horiguchi, 2001] T. Touyama and S. Horiguchi. Performance evaluation of practical parallel computer model LogPQ. *International Journal of Foundations of*

Computer Science, 2001.

[Uthayopas *et al.*, 2000] P. Uthayopas, S. Paisitbenchapol, T. Angskun, and J. Maneesilp. System management framework and tools for Beowulf cluster. In *Proceedings Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region*, 2000.

[VMEbus, 1996] VMEbus. Proceedings of VITA. Europe Congress. VMEbus: The Systems Architecture for the 21st century, 1996.

[Wagschal, 1998] Claude Wagschal. *Topologie et analyse fonctionnelle*. Éd. Hermann, 1998.

[Wah, 2000] K. S. How Tai Wah. A theoretical study of fault coupling. *Software testing, verification and reliability*, 10:3–46, March 2000.

[Wajsbürt *et al.*, 1997] Franck Wajsbürt, Jean-Lou Desbarbieux, Cyril Spasevski, Stéphane Penain, and Alain Greiner. An Integrated PCI component for IEEE 1355 Networks. In *Proceeding of the European Multimedia Microprocessor Systems and Electronic Commerce Conference and Exhibition*, Florence, Italy, May 1997.

[Wajsbürt *et al.*, 1998] Franck Wajsbürt, Jean-Lou Desbarbieux, and Cyril Spasevski. PCIDDC mask 1 Chip Errata. Technical report, Laboratoire d'informatique de Paris VI, 1998. `http://mpc.lip6.fr/pciddc/PCIDDC_CE.ps.gz`.

[Wallach *et al.*, 1995] Deborah A. Wallach, Wilson C. Hsieh, Kirk L. Johnson, M. Frans Kaashoek, and William E. Weihl. Optimistic active messages: A mechanism for scheduling communication with computation. In *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'95*, pages 217–226, Santa Barbara, California, 1995.

[Westrelin, 2001] R. Westrelin. A new software architecture for the BIP/Myrinet firmware. In *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2001.

[Woodward *et al.*, 2000] T.K. Woodward, A.L. Lentine, J.D. Fields, G. Giaretta, and R. Limacher. First demonstration of native Ethernet optical transport system prototype at 10 Gbit/s based on multiplexing of gigabit Ethernet signals. *IEEE Photonics Technology Letters*, 2000.

Publications personnelles

Ouvrages

en langue française :
Alexandre Fenyo, Frédéric Le Guern, Samuel Tardieu. *Raccorder son réseau d'entreprise à l'Internet*, Ed. Eyrolles, 520 pages & CD-Rom – février 1997.
`www.editions-eyrolles.com/sommaire/2-212-08951-1.asp`

en langue espagnole :
Alexandre Fenyo, Frédéric Le Guern, Samuel Tardieu. *Connecte su red local a Internet*, Ed. Gestión 2000, 440 pages – février 1998.
`www.gestion2000.com`

Revues

Pierre David, Jean-Lou Desbarbieux, Alexandre Fenyo, Alain Greiner, Jean-Jacques Lecler, Frédéric Potter, Vincent Reibaldi, Franck Wajsbürt et Belkacem Zerrouk. *La machine MPC*, Calculateurs Parallèles, Réseaux et systèmes répartis – réseaux à haut débit de stations pour le support d'applications parallèles et réparties – vol.10, février 1998.
`lib.univ-fcomte.fr/REVUE/CPRSR`

Alexandre Fenyo, *Le fonctionnement de l'Internet, architecture, routage*, Tribunix no. 57 – septembre/octobre 1994.
`www.afuu.fr, www.enst.fr/~fenyo/article-tribunix57.ps`

Conférences

Amal Zerrouki, Olivier Glück, Jean-Lou Desbarbieux, Alain Greiner, Franck Wajsbürt, Alexandre Fenyo, Cyril Spasevski, *The MPC Parallel Computer : Hardware, Low Level Protocols and Performances*, PDCS 2000, 12^{th} International Conference on Parallel and Distributed Computing and Systems, Las Vegas, USA – Novembre 2000

Alexandre Fenyo, Pierre David, Alain Greiner, *Noyau de communication sécurisé pour la machine parallèle MPC*, RenPar'10, Strasbourg, France – juin 1998.
`icps.u-strasbg.fr/renpar, mpc.lip6.fr/renpar/renpar10`

Jean-Jacques Lecler, Alexandre Fenyo, Alain Greiner, Frédéric Potter, *SmartHSL : An Evaluation Board for the IEEE 1355 Technology*, European Multimedia, Microprocessor Systems and Electronic Commerce Conference and Exhibition (EMMSEC'97), Florence, Italy – novembre 1997.
`mpc.lip6.fr/emmsec97-3/emmsec`

Abréviations

ATM ..... Asynchronous Transfer Mode
ASIM ..... Architecture des Systèmes Intégrés et Micro-électronique
BIP ..... Basic Interface for Parallelism
BSCP ..... Basic Secure Channelized Protocol
CISC ..... Complex Instruction Set Computer
CSCT ..... Core System Class Tree
CMEM ..... Contiguous Memory
DDC ..... Direct Deposit Component
DMA ..... Direct Memory Access
DTCT ..... Distributed Template Class Tree
ENST ..... École Nationale Supérieure des Télécommunications
HSL ..... High Speed Link
INRIA ..... Institut national de recherche en informatique et en automatique
L2TP ..... Loader To Task Protocol
LIP6 ..... Laboratoire d'Informatique de Paris VI
LMI ..... Liste des MI
LMR ..... Liste des Messages Reçus
LPE ..... Liste des Pages à Émettre
M2LP ..... Manager To Loader Protocol
MDCP ..... Multi-Deposit Channelized Protocol
MI ..... Message Identifier
MICP ..... Message Identifier Cleaner Protocol
MOODT ..... Multi-threaded Object Oriented Distributed Toolkit
MPI ..... Message Passing Interface
ORB ..... Object Request Broker
PVM ..... Parallel Virtual Machine
RFC ..... Request For Comments
RFCP ..... Receiver Flow Control Protocol
RISC ..... Reduced Instruction Set Computer
RPC ..... Remote Procedure Call
SAP ..... Service Access Point
SCP/P ..... Secure Channelized Protocol / Physical addresses
SCP/V ..... Secure Channelized Protocol / Virtual addresses
SFCP ..... Sender Flow Control Protocol
SLR/P ..... State Less Receiver Protocol / Physical addresses
SLR/V ..... State Less Receiver Protocol / Virtual addresses

SMP .... Symetric Multi-Processors
T2MP .... Task To Manager Protocol
TCP/IP .... Transport Control Protocol / Internet Protocol
TLI .... Transport Layer Interface
VHDL .... Very High Description Langage
VLSI .... Very Large Scale Integration
VMR .... Virtual Memory Referencer
WAN .... Wide Area Network

Notations

$p.s.$	presque surement
$(\Omega,\mathcal{A},P)$	espace de probabilité
$\mathcal{L}^p(\Omega,\mathcal{A},P)$	espace des variables $\mathcal{A}$-mesurables et de puissance $p^{\text{ième}}$ P-intégrable
$L^p(\Omega,\mathcal{A},P)$	espace quotient du précédent
$\mathcal{L}^\infty(\Omega,\mathcal{A},P)$	espace des variables $\mathcal{A}$-mesurables P-p.s. bornées
$L^\infty(\Omega,\mathcal{A},P)$	espace quotient du précédent
$\emptyset$	évènement impossible
$A \subset B$	A implique B
$A \cap B$	conjonction de A et B
$A \cup B$	union de A et B
$A \uplus B$	union de A et B si $A \cap B = \emptyset$
$A \setminus B$	différence de A et B
$A^c = \Omega \setminus A$	évènement contraire à A
$\cup_n A_n$	«au moins l'un des A_n se réalise»
$\cap_n A_n$	«tous les A_n se réalisent»
$\mathcal{B}(E)$	tribu borélienne d'un espace topologique E
$\mathbb{R}$	droite réelle
$\overline{\mathbb{R}} = [-\infty, +\infty]$	droite réelle achevée
$\mathbb{N}$	ensemble des entiers naturels
$\mathbb{Z}$	ensemble des entiers relatifs
$\mathbb{N}^*$	ensemble des entiers naturels privé de 0
$\mathrm{Card}(A)$	cardinal de A
P_X	loi de la variable aléatoire X
Φ_X	fonction caractéristique de la variable aléatoire X
$E(X)$	espérance mathématique de la variable aléatoire X
$P(X)$	probabilité de l'évènement X
$P(X \geqslant a)$	probabilité de l'évènement $X \geqslant a$
$\mathrm{Var}(X)$	variance de la variable aléatoire X
$\sigma(X)$	écart-type de la variable aléatoire X
f_X	densité de la variable aléatoire X
$f^{(n)}$	$\text{n}^{\text{ième}}$ dérivée de f

Index