N° d'ordre : 328
N° attribué par la bibliothèque : 05ENSL0 328

École normale supérieure de Lyon
Laboratoire de l'Informatique et du Parallélisme

THÈSE

en vue d'obtenir le grade de

Docteur de l'École normale supérieure de Lyon

Spécialité : Informatique

au titre de l'école doctorale de Mathématiques et Informatique Fondamentale

présentée et soutenue publiquement le 11/10/2005

par Monsieur Brice Goglin

Réseaux rapides et stockage distribué dans les grappes de calculateurs : propositions pour une interaction efficace

Directeur de thèse : Pascale VICAT-BLANC PRIMET

Après avis de Brigitte PLATEAU et Raymond NAMYST

Devant la commission d'examen formée de :

Yves	DENNEULIN	Maître de conférences	Membre
Olivier	GLÜCK	Maître de conférences	Co-encadrant de thèse
Bernard	LECUSSAN	Professeur	Membre
Raymond	NAMYST	Professeur	Membre/Rapporteur
Thierry	PRIOL	Directeur de recherche	Président du jury
Pascale	VICAT-BLANC PRIMET	Directrice de recherche	Directrice de thèse

N° d'ordre : 328
N° attribué par la bibliothèque : 05ENSL0 328

École normale supérieure de Lyon
Laboratoire de l'Informatique et du Parallélisme

THÈSE

en vue d'obtenir le grade de

Docteur de l'École normale supérieure de Lyon

Spécialité : Informatique

au titre de l'école doctorale de Mathématiques et Informatique Fondamentale

présentée et soutenue publiquement le 11/10/2005

par Monsieur Brice Goglin

Réseaux rapides et stockage distribué dans les grappes de calculateurs : propositions pour une interaction efficace

Directeur de thèse : Pascale VICAT-BLANC PRIMET

Après avis de Brigitte PLATEAU et Raymond NAMYST

Devant la commission d'examen formée de :

Yves	DENNEULIN	Maître de conférences	Membre
Olivier	GLÜCK	Maître de conférences	Co-encadrant de thèse
Bernard	LECUSSAN	Professeur	Membre
Raymond	NAMYST	Professeur	Membre/Rapporteur
Thierry	PRIOL	Directeur de recherche	Président du jury
Pascale	VICAT-BLANC PRIMET	Directrice de recherche	Directrice de thèse

Remerciements

Je tiens tout d'abord à remercier mes deux encadrants, Pascale Vicat-Blanc Primet et Olivier Glück, pour m'avoir guidé au cours de cette thèse. J'associe également Loïc Prylli qui a été à l'origine de ces travaux et a assuré une partie importante de l'encadrement technique, même après son départ du laboratoire.

Je remercie ensuite chaleureusement les membres de mon jury, Yves Denneulin, Bernard Lecussan, Raymond Namyst et Thierry Priol, ainsi que Brigitte Plateau.

Merci à ma famille et en particulier à mes parents qui m'ont permis de mener ces longues études plutôt que de rester cantonné aux activités agricoles dans mes Ardennes natales.

Merci ensuite à Émilie pour son soutien moral pendant toute la thèse, pour avoir relu avec attention l'intégralité du manuscrit sans partir en courant, pour m'avoir expliqué comment marche Openoffice.org, et surtout pour le pot qu'elle a entièrement concocté de ses petites mains de cuisinière au détriment de ses activités de recherche contre le cancer.

Je tiens ensuite à remercier tous les membres du laboratoire, notamment les directeurs Jean-Michel Müller et Frédéric Desprez, les secrétaires Sylvie, Isabelle et Corinne dont l'efficacité n'a d'égal que la gentillesse, Serge et Dominique pour le traitement rapide de tous mes problèmes informatiques, et enfin Jean-Christophe pour avoir partagé le désastre de l'administration des machines parallèles après le départ de Loïc.

Merci également à tous mes différents cobureaux, dans l'ordre chronologique David, Pascal, Benjamin, Jean-Patrick et Dino, ainsi qu'à tous les membres de l'équipe notamment Laurent et Cong Duc.

Je remercie ensuite tous les élèves que j'ai eu en cours durant le monitorat qui a accompagné cette thèse, en particulier smimou pour m'avoir fait prendre conscience que les MIM1 etaient nuls en programmation en C au point de déréférencer NULL.

Merci à tous ceux qui ont distrait mes journées sur le channel `#sos` d'IRC, que ce soit mes ex-camarades de l'ENS Lyon, mes (anciens) élèves ou encore des espions du CRI. Un merci particulier à Heimdall et tonfa qui ont contribué à assurer mon minimum de sport hebdomadaire en venant à la randonnée roller le vendredi soir au lieu de recompiler le dernier noyau `mm` d'Andrew Morton.

Merci également à Loïc (encore lui), Patrick, Drew, Marty, Susan et Abby pour leur

accueil lors de mes séjours chez Myricom.

Je remercie également tous les autres qui se reconnaîtront, ceux avec qui j'aurai partagé des discussions scientifiques (ou pas) et ceux qui auront contribué à rendre ces 3 années agréables.

Et enfin, merci à Maître Yoda et Jabba the Hutt de s'être proposés pour combler les deux seules initiales qui manquaient dans l'index de ce manuscrit.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

1.1 Les besoins du stockage

Le stockage est une composante majeure des systèmes informatiques. La généralisation des périphériques de stockage persistant permet de conserver de manière sûre les informations lorsque l'ordinateur est éteint ou tombe en panne. Les données manipulées sont lues en début de session, maintenues dans la mémoire centrale de la machine pendant les traitements puis sauvées en fin de session, voire également en cours de session pour prévenir une éventuelle panne. Si la persistance des données reste la fonction la plus importante du stockage, la capacité des périphériques de stockage permet également de répondre efficacement à la demande croissante en mémoire des applications, en particulier lorsque les données manipulées ne peuvent pas tenir intégralement en mémoire centrale.

Le développement rapide des réseaux informatiques a permis aux utilisateurs d'accéder à leurs données de manière uniforme depuis différentes machines, le stockage étant effectué sur un serveur distant. Le coût de la traversée du réseau pour contacter le serveur et accéder aux données a donné lieu à de nombreuses recherches scientifiques visant d'une part à améliorer les performances du système, et d'autre part à en optimiser le modèle.

1.2 Les grappes de calcul

Les besoins croissants en puissance de calcul informatique dans des domaines aussi variés que la simulation numérique, la physique des particules, la génomique ou la réalité virtuelle ont nécessité une évolution régulière du matériel. Cela s'est traduit par le développement dans les années 1970 de super-calculateurs parallèles utilisant des technologies réseau dédiées. Le coût prohibitif de ces machines a conduit à leur disparition progressive dans les années 1990 au profit des grappes de stations. Cette solution, même si elle peut être moins puissante, se révèle plus extensible et générique, et par conséquent moins onéreuse.

La généralisation des grappes depuis une dizaine d'années a engendré des contraintes

bien plus grandes sur le stockage distribué. En effet, le nombre croissant de nœuds connectés, la quantité de données manipulées et la nécessité de les partager entre les différentes instances d'une application parallèle qui s'y exécute imposent des transferts de données très performants. Il a donc fallu développer des mécanismes permettant au stockage distribué d'être plus efficace.

1.3 Les réseaux haute performance

L'obtention de performances élevées lors de l'exécution d'applications parallèles sur des grappes de stations a nécessité de réduire au maximum le coût des communications entre les différents processus. On sait en effet depuis longtemps que le gain maximal obtenu en parallélisant est limité par le sous-système le plus lent, c'est-à-dire les entrées-sorties [Amd67].

De nombreuses recherches ont donc été menées sur les systèmes de communication dans les grappes, permettant d'une part des transferts de données très rapides, et d'autre part une gestion autonome des communications par le matériel. Ces travaux ont abouti à des réseaux dédiés aux grappes, présentant une latence très faible et une bande passante très élevée. Les implantations logicielles ayant été conçues pour les supporter efficacement, les communications entre les différentes instances d'une application parallèle peuvent bénéficier aisément de ces performances très élevées.

L'exécution performante des applications parallèles passe également par un stockage distribué efficace. Des travaux ont été menés dans ce contexte pour améliorer la qualité du serveur, notamment en utilisant des techniques de parallélisation. Cependant, l'utilisation du réseau dans ce cadre reste très simple et profite peu des performances du matériel sous-jacent. Pourtant les besoins du stockage distribué, notamment en débit, sont similaires voire supérieurs à ceux des communications inter-processus au sein des applications parallèles.

1.4 Objectifs de la thèse

Le but de ce travail est d'étudier l'utilisation efficace du réseau dans le cadre du stockage distribué. Cette idée doit permettre d'améliorer les performances de l'accès au stockage distant en profitant au maximum du matériel réseau disponible dans les grappes. Il s'agit tout d'abord d'étudier les apports potentiels des réseaux haute performance des grappes pour le stockage distribué. En particulier, il faut voir dans quelle mesure les caractéristiques principales de ces réseaux, notamment la très faible latence et la grande bande passante, permettent d'envisager du stockage distribué à haute performance.

Cependant, la spécialisation des technologies réseau pour obtenir des communications performantes au sein des applications parallèles les rend difficiles à utiliser dans d'autres contextes où les contraintes peuvent être très différentes. L'adaptation de ces

technologies au stockage distribué, en particulier l'intégration du modèle de programmation de ces réseaux dans les différents contextes du stockage, doit être étudiée. Cela implique d'une part de s'intéresser aux transferts de données entre les différents nœuds, notamment entre un client et un serveur, et d'autre part au contrôle des communications sur ces nœuds, c'est-à-dire à la gestion des communications et des événements qu'elles engendrent.

1.5 Contributions de la thèse

La contribution principale de cette thèse consiste en la proposition et l'étude d'une interface de programmation des réseaux suffisamment flexible pour répondre efficacement à la fois aux besoins des communications dans les applications parallèles et des différents contextes de stockage distribué.

Tout d'abord, différentes lacunes dans les systèmes d'exploitation et dans les interfaces de programmation des réseaux de grappes existantes sont mises en évidence et analysées. Ensuite, des solutions sont proposées pour modifier l'interface GM des réseaux MYRINET afin de pouvoir exploiter efficacement les performances du matériel sous-jacent dans le cadre du stockage distribué.

Ces travaux validés en laboratoire sont également validés industriellement puisque nos propositions ont été intégrées dans la nouvelle interface MX (*Myrinet Express*) des réseaux MYRINET, permettant ainsi une exploitation aisée du matériel.

Ces réalisations correspondent à plus de 50 000 lignes de code dont 12 000 dans le noyau LINUX, environ 1500 dans les pilotes des réseaux MYRINET et une centaine dans le microprogramme qui s'exécute dans leurs cartes d'interface. Notre contribution dans l'interface MX est d'ores et déjà utilisée par le logiciel SOCKETS-MX distribué par MYRICOM et devrait l'être rapidement par d'autres projets. Ces travaux ont par ailleurs fait l'objet de plusieurs publications citées en page 171 et de plusieurs séminaires.

1.6 Plan du manuscrit

Le contexte de notre étude et l'état de l'art sont présentés au chapitre 2. Nous y rappelons les mécanismes mis en jeu dans les systèmes de stockage distribué ainsi que les principaux travaux qui ont été menés dans le domaine spécifique des grappes de stations pour obtenir des performances à la hauteur des besoins des applications parallèles. Nous y présentons ensuite les réseaux haute performance des grappes, leurs spécificités et mettons en évidence les différentes interactions mises en jeu lorsqu'un système de stockage accède au réseau. Les différentes questions qui ont guidé notre étude sont exposées au chapitre 3.

Le chapitre 4 présente une étude préliminaire de l'accès aux fichiers distants sur un réseau haute performance. La conception d'un protocole expérimental simple nommé

ORFA (*Optimized Remote File-system Access*) nous permet d'extraire les goulots d'étranglement de l'utilisation des réseaux existants. Nous étudions ensuite précisément au chapitre 5 les problèmes liés aux transferts de données et en particulier au mécanisme d'enregistrement mémoire imposé par l'interface GM des réseaux MYRINET. Le chapitre 6 est consacré au contrôle des communications. Nous y étudions tout d'abord l'impact de la gestion des messages inattendus dans un modèle client-serveur de stockage distribué, puis la gestion des événements liés aux entrées-sorties vers le réseau et les périphériques de stockage.

Enfin, nous proposons au chapitre 7 l'intégration de nos travaux dans l'interface MX des réseaux MYRINET. Nous montrons tout d'abord que le contrôle des communications dans MX évite les problèmes rencontrés préalablement. Nous présentons l'application de nos idées concernant les transferts des données sous la forme d'une interface de programmation flexible. Les résultats expérimentaux montrent que MX se comporte particulièrement bien dans le contexte du stockage distribué, et apporte également un gain significatif à d'autres applications. Ces observations nous permettent de conclure sur l'état actuel de l'interaction entre les systèmes de stockage et les réseaux haute performance des grappes et de dégager plusieurs perspectives de poursuite de notre travail.

Chapitre 2

État de l'art

Le stockage des données dans les grappes de calcul haute performance comprend deux aspects : l'accès aux données stockées à distance et les réseaux haute performance qui relient les différents nœuds.

L'accès aux données distantes peut être mis en œuvre à différents niveaux logiciels. C'est, depuis une vingtaine d'années, le sujet de nombreux travaux de recherche. Il nécessite des protocoles complexes pour maintenir la cohérence entre les données locales et leurs originaux distants, la réplication des données pour assurer leur disponibilité, mais aussi, dans le contexte des grappes, des mécanismes de parallélisation du serveur afin de servir efficacement de nombreux clients. De leur coté, les réseaux haute performance se caractérisent surtout par une très grande spécificité. Cette opposition rend complexe leur interaction avec un système de fichiers distribués pour grappe.

Pour examiner cette interaction, nous étudions les systèmes de fichiers distribués dans la partie 2.1. Les spécificités des réseaux de grappe sont ensuite présentées en 2.2. L'interaction des accès aux fichiers avec ces réseaux est étudiée en partie 2.3.

2.1 Les systèmes de fichiers distribués

Avec la généralisation des réseaux informatiques au début des années 1980, les machines de travail sont devenues de simples clients accédant à travers le réseau à des services hébergés à distance.

Le stockage a été une des premières applications de ce modèle client-serveur puisque l'accès distant aux données, et en particulier aux espaces utilisateur (*home*), permettait d'accéder aux mêmes fichiers depuis n'importe quelle machine du réseau. Il s'agissait tout d'abord de rapatrier les fichiers à chaque nouvelle session par un protocole de type FTP (*File Transfer Protocol* [FTP85]) puis de les enregistrer sur le serveur en fin de session. Ce mode déconnecté a été révolutionné en 1983 avec l'apparition de DOMAIN, le système de fichiers des machines APOLLO qui pour la première fois permettait d'accéder et modifier de manière transparente des fichiers stockés sur un serveur distant [Lev86]. Cette approche s'est banalisée à partir de 1985 grâce au *Network File System* du système d'exploitation SOLARIS de SUN [WLS+85] et s'est depuis développée sous le nom géné-

rique de *système de fichiers distribués*.

Après des rappels de terminologie, nous présentons les principaux travaux qui ont été menés sur le stockage distribué, c'est-à-dire les problèmes de cache et de maintien de la cohérence, puis la parallélisation des serveurs pour passer à l'echelle dans les grappes.

2.1.1 Concepts de base et terminologie

2.1.1.a Concepts

Organisation logique et métadonnées Un des rôles principaux des systèmes d'exploitation est d'exposer à l'application une vue logique du système en masquant les spécificités du matériel. Dans le cadre du stockage, l'application manipule les données dans des objets logiques appelés *fichiers* dans lesquels elle peut lire ou écrire des octets. Chaque fichier est caractérisé par un certain nombre d'attributs tels qu'une taille, un propriétaire et des droits d'accès. Ces caractéristiques sont nommées *métadonnées*, par opposition aux *données* réelles stockées dans les fichiers. Certains fichiers particuliers (les *répertoires*) forment une arborescence où sont placés tous les autres fichiers, ce qui permet de leur associer un *chemin d'accès* (*Nommage*).

Cette organisation logique masque le fait que le stockage réel dans les dispositifs physiques (disques durs, bandes, ...) est basé sur un ensemble de blocs de taille fixe. Le système d'exploitation est en charge de faire correspondre les fichiers et répertoires logiques manipulés par l'application avec les blocs physiques. Dans les systèmes UNIX, ces objets logiques sont nommés *Nœuds d'index* (ou *I-nœuds*).

Stockage distribué Le *stockage distribué* consiste à utiliser des données stockées sur une machine distante appelée *serveur*. L'application qui accède à ces données (et par abus de langage la machine sur laquelle l'application s'exécute) est appelée *client*. Le système se charge de transférer sur le réseau reliant le client et le serveur les données et les métadonnées (voir la figure 2.1).

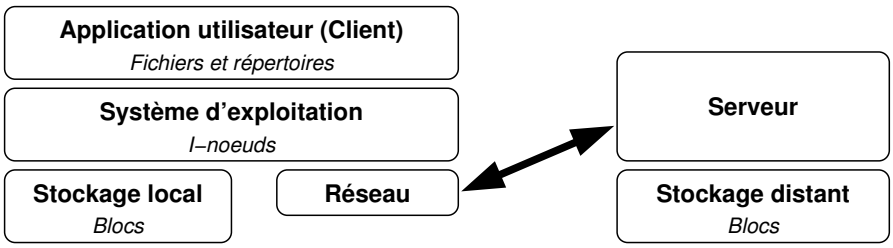


FIG. 2.1: *Stockage local et distribué.*

Accès distant Un accès à un fichier distant est composé de l'ensemble des communications réseau mises en jeu lorsqu'un client effectue une requête au serveur distant. Il s'agit d'un aller-retour sur le réseau accompagné d'un transfert de données, du client vers le serveur dans le cas d'une écriture et du serveur vers le client dans le cas d'une lecture.

Transparence Le stockage physique dépend en fait non seulement du matériel, mais aussi du type de *système de fichiers* utilisé. L'organisation réelle des données et des métadonnées dans les blocs physiques varie par exemple beaucoup selon l'utilisation d'un stockage EXT3 (*Extended File System 3*, le système généralement utilisé sous LINUX) ou VFAT (*Virtual File Allocation Table*, un des systèmes utilisés sous WINDOWS). Le système d'exploitation est également chargé de rendre ces différentes organisations logiques accessibles à l'application de manière uniforme et transparente.

Dans le cas des systèmes de stockage distribué, le stockage physique n'est pas situé sur la machine manipulant les données (*transparence de localisation*). La *transparence de nommage* nécessite quant à elle de rendre les données accessibles à l'application par le même *espace de nommage* quelque soit leur localisation réelle.

Cohérence Le stockage distribué permet à plusieurs machines clientes d'utiliser les mêmes données voire de modifier les mêmes fichiers. Cela engendre des problèmes d'accès *concurrents* aux :

métadonnées : par exemple quand un client supprime un fichier qu'un autre est en train de lire.

données : lorsque deux clients modifient le même fichier.

Le système de stockage distribué doit ici être capable de maintenir une certaine *cohérence* entre les différents clients [NWO88]. Si les modifications d'un client sont immédiatement repercutées sur le serveur et sur les autres clients (par exemple via une opération atomique), la cohérente est dite *forte*. Si la propagation des modifications n'est pas visible immédiatement, la cohérence est *faible* ou *lâche*. Dans ce cas, certaines modifications concurrentes peuvent être perdues.

2.1.1.b Anatomie des fichiers dans UNIX

Pour accéder aux fichiers, l'application manipule des descripteurs de fichiers ouverts et passe des requêtes à la bibliothèque standard (`read`, `write`, ... voir en annexe A.1.1). Cette bibliothèque se charge de les traduire en appels-système pour invoquer le système d'exploitation.

L'implantation des accès aux fichiers dans le noyau s'articule autour d'une couche de virtualisation qui uniformise les interfaces d'accès aux différents systèmes de fichiers qu'ils soient locaux ou distants. Cette couche est appelée VFS dans LINUX (*Virtual File System*) ou VNODES dans BSD et SOLARIS [Kle86]. Ainsi l'application manipule de la

même façon les fichiers d'une partition EXT3 ou VFAT. La couche VFS se charge de convertir les accès logiques à des fichiers en accès à des blocs disque.

Les accès aux disques sont eux aussi virtualisés par une couche dédiée afin, notamment, de manipuler de manière uniforme les disques IDE (*Integrated Drive Electronics*) ou SCSI (*Small Computer System Interface*). Les pilotes des différents types de disques conduisent enfin aux blocs physiquement stockés sur les disques ou leurs partitions.

Les dispositifs de stockage peuvent cependant être très lents. Si le débit d'un disque moderne peut atteindre 50 Mo/s, le temps initial d'accès reste de l'ordre de plusieurs millisecondes car la tête de lecture doit se déplacer mécaniquement. Ce temps d'accès initial au matériel ne devient négligeable que lorsqu'on transfère de grandes quantités de données. Les systèmes d'exploitation modernes cachent les données et métadonnées pour éviter des accès répétitifs aux mêmes blocs physiques. Cette stratégie est connue sous le nom de *Buffer-Cache* dans les système UNIX [Bac86], ou *Page-Cache* plus récemment dans LINUX. Les applications communiquent en fait avec ce cache tandis que le système sous-jacent maintient la cohérence entre le cache et le dispositif de stockage physique. L'annexe B détaille plus précisément cette mise en œuvre dans le VFS des noyaux LINUX.

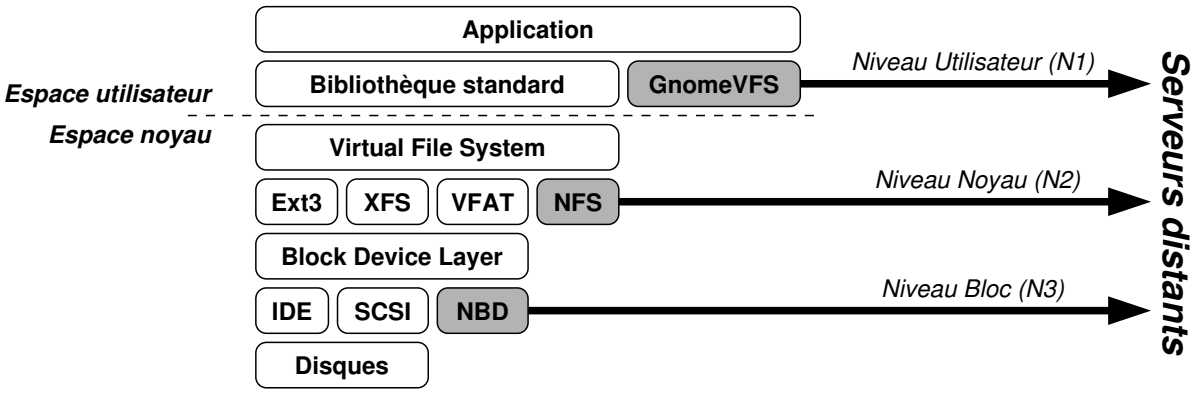


FIG. 2.2: *Mise en œuvre des accès distants aux fichiers dans les systèmes* UNIX.

La figure 2.2 présente la mise en œuvre habituelle des accès aux fichiers distants dans UNIX. Le système de fichiers distribués est un cas particulier (N2). Les deux autres possibilités sont l'accès en espace utilisateur (N1) et au niveau des périphériques bloc (N3). Nous détaillons maintenant ces trois stratégies.

2.1.1.c Accès en espace utilisateur (N1)

Le développeur d'une application n'est pas obligé d'utiliser l'interface standard d'accès aux fichiers fournie par la bibliothèque standard (voir l'annexe A.1.1). L'utilisation d'interfaces spécifiques peut répondre plus précisément aux besoins de certaines applications. Il est possible de construire des routines spécifiques pour contacter un serveur

distant et rapatrier des fichiers. Cette mise en œuvre peut se faire soit de manière directe dans l'application, soit dans une bibliothèque spéciale telle que GNOMEVFS [Gno] (qui permet par exemple d'accéder directement à des fichiers stockés dans une archive `tar`) ou PVFS (voir en 2.1.3.a).

Cette bibliothèque exporte une interface de programmation spécifique qui nécessite la réécriture de l'application, ou bien surcharge les routines de la bibliothèque standard pour les remplacer par des routines contactant le serveur distant (voir l'annexe C). Les données échangées sur le réseau utilisent alors soit un protocole calqué sur l'interface offerte par la bibliothèque spécifique manipulée par l'application (descripteurs de fichiers et des segments quelconques) soit un protocole plus évolué si le client propose des fonctionnalités du type cache.

2.1.1.d Système de fichiers distribués (N2)

Le *Virtual File System* des systèmes UNIX présente l'avantage de fournir une méthode très simple pour ajouter le support d'un nouveau type de système de fichiers. Il suffit de préciser les routines de traitement spécifique du nouveau système et elles seront automatiquement appelées lorsqu'une application accédera à un fichier de ce type. Pour utiliser une instance de ce système de fichiers, il faut le *monter* (l'intégrer à l'arborescence virtuelle) et son contenu devient accessible aux applications.

La mise en œuvre d'un système de fichiers distribués consiste à insérer un client dans le VFS pour transmettre au serveur distant toutes les requêtes d'accès aux fichiers concernés. Cette stratégie est la plus couramment utilisée pour accéder aux fichiers distants. C'est notamment le cas de NFS et SMBFS (*Samba File System* [Sam], pour accéder aux dossiers partagés par les systèmes WINDOWS) sur les machines classiques, et de LUSTRE (voir en 2.1.3.a) et de certains systèmes propriétaires tels que GPFS (voir en 2.1.3.b) dans le contexte des grappes. Les objets échangés sur le réseau sont alors des métadonnées et pages logiques de fichiers.

2.1.1.e Network Block Device (N3)

Dans les cas (N1) et (N2) précédents, le VFS se charge de traduire les requêtes logiques d'accès aux fichiers en accès à certains blocs des disques. Mais il est également possible de construire des disques virtuels dont les accès seront redirigés vers un serveur distant. Les communications réseau sont alors réalisées depuis le bas de la pile d'accès aux fichiers du client. Une telle implantation est réalisée en ajoutant un nouveau type de disque dans la couche de virtualisation des périphériques blocs du système d'exploitation, la *Block Device Layer*.

Cette stratégie communément appelée *Network Block Device* permet de manipuler une partition ou un disque physique distant comme s'il était local. Les messages échangés sur le réseau ont la granularité du bloc disque. C'est le modèle utilisé dans ISCSI (*Internal Small Computer System Interface* [Sys01]). C'est également dans ce modèle qu'ont

été réalisés les travaux OPIOM (*Off-Processor I/O with Myrinet* [Geo01]) et READ^2 (*Remote Efficient Access to Distant Device* [CRU03]) visant à optimiser le transfert des blocs entre des disques et une carte réseau du côté serveur.

2.1.1.f Synthèse des méthodes d'accès aux fichiers distants

Type	Bibliothèque partagée (N1)	Système de fichiers distribué (N2)	*Network Block Device* (N3)
Données manipulées	Descripteurs et octets	I-nœuds et pages	Blocs
Interface	personnalisable	standard	standard
Transparence	peut être mise en place	automatique	automatique
Cache	peut être mis en place	automatique (désactivable)	automatique (désactivable)
Implantation	Espace utilisateur	Noyau Niveau logique (VFS)	Noyau Niveau périphérique (*Block Device Layer*)

TAB. 2.1: *Synthèse des principales caractéristiques des différentes implantations possibles de l'accès aux fichiers distants.*

La table 2.1 synthétise les trois types d'accès aux fichiers distants. La bibliothèque utilisateur permet une grande liberté d'interface et de mise en œuvre tandis que les implantations dans le noyau ont une interface fixée mais assurent un accès transparent pour n'importe quelle application et fournissent un cache.

Nous étudions maintenant l'organisation du système de stockage distribué global et les différents domaines de recherche qui ont permis d'améliorer ses performances.

2.1.2 Cache et réplication

Accéder aux données distantes requiert *a priori* de traverser le réseau pour chaque requête. Les premières implantations de NFS sur les protocoles IP (*Internet Protocol*) et ETHERNET dans les années 1980 subissaient une latence très importante, de l'ordre de la milliseconde. De la même façon que les données sont cachées dans le système d'exploitation lors des accès locaux, les systèmes de fichiers distribués maintiennent un cache du côté client. Ainsi, la traversée du réseau est évitée de la même manière que les accès physiques aux disques sont évités.

2.1.2.a Cohérence entre clients et serveur

La grande différence entre un cache local et un cache sur un client distant réside dans le fait que les disques locaux ne sont accédés que par le système d'exploitation local, tandis qu'un serveur distant peut potentiellement être accédé par différentes machines clientes. Dans le premier cas, seul le cache local a accès aux données physiquement stockées sur les disques locaux. La synchronisation est alors facile à assurer. Dans le second cas, si plusieurs clients modifient simultanément les mêmes données, il faut mettre en place un mécanisme pour propager les modifications parmi les autres clients (voir la figure 2.3).

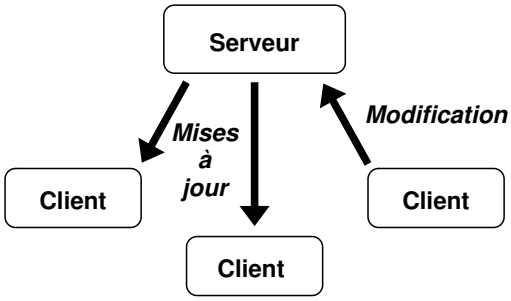


FIG. 2.3: *Propagation des modifications par un client vers les autres clients accédant au même fichier.*

De nombreux travaux ont été menés sur ce problème de maintien de cohérence. NFS se contente d'assurer une cohérence faible du type *close-open*, c'est-à-dire que les modifications d'un fichier sur un client ne sont vraiment propagées qu'à sa fermeture [SGK+85]. AFS se contente également de cette cohérence lâche [HKM+88]. Ces modèles suffisent pour une utilisation du type accès au répertoire *home* d'un utilisateur puisque les accès concurrents y sont rares. SPRITE [NWO88] fixe par ailleurs un délai maximum de 30 secondes entre une modification et sa propagation. Ce délai permet dans le cas de fichiers ouverts très longtemps de propager leur modification régulièrement, ce qui permet aux autres client d'en être informé assez rapidement. Avec un délai plus petit, le coût de la propagation deviendrait prohibitif. Il s'agit ici d'un compromis entre coût et cohérence qui se révèle intéressant dans le cas d'accès concurrents limités.

Par contre, dans le cas d'accès réellement concurrents tels qu'on peut en observer dans des applications parallèles traitant un même fichier sur de multiples nœuds, une cohérence beaucoup plus forte est requise. En effet, il peut être nécessaire que les clients soient informés immédiatement des modifications concurrentes, bien avant qu'un délai fixé se soit écoulé. Dans ce cas, le maintien d'un état dans le serveur avec des verrous partagés est utilisé, mais de telles méthodes de synchronisation peuvent avoir un impact important sur les performances. Une gestion hiérarchique de ces verrous et de jetons permet par exemple d'y remédier [SH02].

2.1.2.b Réplication du serveur

Les serveurs des modèles initiaux de type NFS avaient du mal à supporter la charge imposée par un grand nombre de machines clientes. AFS (*Andrew File System* [HKM+88]) a introduit la réplication des données sur de multiples serveurs pour améliorer le passage à l'échelle des systèmes de fichiers distribués. Le *Coda File System* [BN99] puis sa version allégée, INTERMEZZO [Bra99], ont encore amélioré ce modèle en autorisant le client à travailler de manière complètement déconnectée (voir la figure 2.4). Le client peut donc répliquer les données distantes sur son disque local et tolérer des pannes du serveur. De plus, ce modèle permet de rapatrier des données très lointaines pour les manipuler aussi rapidement que des données locales, tandis que le maintien de la cohérence avec le serveur distant est effectué de manière transparente en arrière plan.

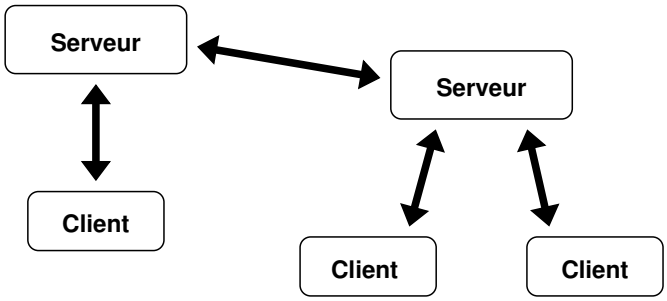


FIG. 2.4: *Réplication des données sur différents serveurs pour répartir la charge des clients.*

La réplication des données sur différents serveurs répartit la charge sur différentes machines. Le maintien de la cohérence impose alors non seulement de synchroniser les caches des clients et leur serveur, mais aussi d'effectuer des mises à jour entre les différents serveurs. Mais ces mises à jour entre les différents réplicats restent souvent asynchrones et les serveurs ne travaillent pas en collaboration étroite. La cohérence maintenue est de fait intrinsèquement faible, ce qui rend difficile une utilisation concurrente des mêmes fichiers par de multiples clients.

2.1.3 Parallélisation et répartition

L'accès aux fichiers dans les grappes présente des besoins bien plus grands que dans le cadre d'une utilisation classique telle que l'accès d'un utilisateur à son répertoire *home*. Tout d'abord, les applications parallèles peuvent être beaucoup plus gourmandes en entrées-sorties. La quantité de machines clientes connectées dans une grappe et leurs grands besoins en stockage rendent difficile l'utilisation d'un serveur unique. En effet, les grappes de calcul regroupent désormais couramment plusieurs centaines de machines qui vont traiter des teraoctets de données.

Ensuite, les accès concurrents au même fichier par de multiples clients sont très courants puisque ces derniers travaillent sur les mêmes ensembles de données. Par

exemple, les calculs matriciels en parallèle imposent de répartir le traitement des blocs d'une même matrice entre les différents clients. Il est donc important dans ce contexte de maintenir une cohérence forte à travers les caches des différents clients et les éventuels multiples serveurs.

Il faut distinguer la gestion des *métadonnées* (qui incluent ici la distribution des données sur les différents périphériques de stockage physique) et les entrées-sorties réelles (les accès aux blocs physiques sur les périphériques de stockage). Ces dernières imposent une charge de travail beaucoup plus grande car les applications parallèles manipulent beaucoup plus de données que de métadonnées.

2.1.3.a Serveurs parallèles

La solution la plus classique consiste à paralléliser le serveur se chargeant des entrées-sorties réelles et à utiliser un serveur unique dédié aux métadonnées. Le serveur de métadonnées traite les requêtes logiques puis indique aux clients quels serveurs I/O contacter pour accéder aux données réelles (voir la figure 2.5). C'est notamment le modèle utilisé par PVFS (*Parallel Virtual File System*, [LR99, CLRT00]) qui est un des systèmes dédiés aux grappes les plus aboutis. Il a été conçu pour s'interfacer efficacement avec les applications de type MPI-IO [1] en supportant la distribution d'un même fichier sur l'ensemble des serveurs I/O, de la même façon qu'un contrôleur RAID répartit les blocs sur différents disques. Ce *stripping* (répartition) permet de paralléliser efficacement les accès aux fichiers en répartissant la charge sur tous les serveurs.

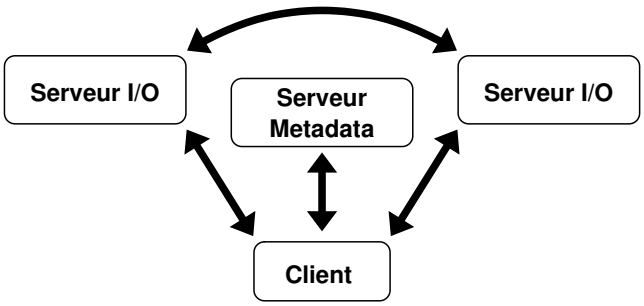


FIG. 2.5: *Parallélisation du stockage en répartissant les entrées/sorties et les accès aux métadonnées sur différents serveurs accédés simultanément par le même client.*

NFSP [LD02] propose une version parallèle de NFS utilisant les disques locaux des machines de la grappe pour y stocker les données plutôt que d'ajouter un système de stockage dédié. Tous les nœuds deviennent alors des serveurs I/O potentiels. Un serveur central de métadonnées, *nfspd*, reçoit les requêtes des clients NFS et les transmet

[1] La norme MPI propose une interface de communication (voir en 2.2.1.b). Elle a ensuite été étendue, notamment avec l'interface d'accès aux fichiers MPI-IO (voir en 2.3.1.a).

aux serveurs I/O concernés. Ceux-ci répondent ensuite directement au client pour éviter de concentrer le trafic sur le serveur de métadonnées.

La répartition des données utilisées dans ces différents systèmes peut ensuite être accompagnée de redondance afin d'une part de tolérer d'éventuelles pannes des systèmes de stockage et/ou des serveurs, mais aussi de répartir encore plus la charge. Mais plus qu'une simple réplication des données, il est nécessaire ici de mettre en place une collaboration étroite entre les différents serveurs pour maintenir la cohérence nécessaire dans le contexte des grappes. Le maintien de la cohérence sur les données est rendu difficile puisque les serveurs I/O sont démultipliés. Les systèmes précédents assurent donc plutôt l'atomicité des écritures (un lecteur verra l'intégralité d'une modification ou rien du tout). La cohérence des métadonnées est par contre plus simple à maintenir si un seul serveur de métadonnées est utilisé.

PVFS2 [Lig01] et les versions récentes de NFSP [LDVL03] proposent également de distribuer le serveur de métadonnées pour éviter le goulot d'étranglement qu'il représente. Il est alors nécessaire de mettre en place un protocole complexe de mise à jour des différents serveurs entre eux. Mais l'impact sur les performances peut être très important si les accès sont très concurrents. La réplication du métaserveur de NFSP permet par exemple de multiplier la performance des écritures par 5 [ANL+04]. Le projet le plus prometteur dans ce domaine est LUSTRE [Clu02, Sch03] qui a été directement conçu pour supporter de telles contraintes et semble d'ores et déjà passer efficacement à l'échelle des grandes grappes (plus de 1 000 nœuds).

2.1.3.b Stockage partagé sans serveur

Plutôt que d'améliorer le passage à l'échelle en utilisant de multiples serveurs, GFS (*Global File System* [SRO96]) choisit de supprimer totalement le serveur pour éviter le goulot d'étranglement qu'il peut représenter. Ainsi, les accès au système de stockage ne sont plus centralisés, mais des verrous globaux sont utilisés pour assurer la synchronisation des caches des différents clients (voir la figure 2.6). Le système de stockage est attaché directement aux nœuds de calcul par un réseau dédié, par exemple de type FIBRE CHANNEL [Jur95]. L'accès au stockage se fait au niveau bloc tandis que la synchronisation des clients est assurée au niveau logique, là où la notion de fichier a un sens. Ce modèle est également utilisé dans XFS [ADN+96] qui permet de stocker des blocs sur un système distant ou sur les nœuds eux-mêmes.

GFS et XFS n'ont pas été conçus spécifiquement pour les grappes de calcul haute performance, mais aussi par exemple pour les applications de type base de données où les requêtes sont petites et aléatoires. Dans les applications parallèles, les requêtes sont assez larges et régulières. La régularité des tailles de requêtes et de leur répartition a été montrée dans [CACR95, NKP+96]. La taille de requêtes augmente avec la taille des problèmes et la puissance des machines. Il est courant aujourd'hui de manipuler des blocs de données de 64 kilooctets.

Le coût du verrouillage de GFS est assez élevé car il est effectué à très bas niveau, au niveau des blocs physiques dans les disques, ce qui restreint les performances. Le *Ge-*

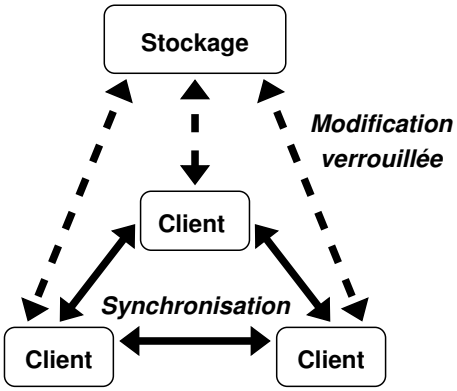


FIG. 2.6: *Accès non-centralisé à des disques partagés.*

neral Parallel File System (GPFS [SH02]) est un système propriétaire utilisé par IBM sur certaines des plus grandes grappes, par exemple `ASCI White` qui est composée de 512 nœuds de 16 processeurs. Il utilise une architecture similaire à GFS mais le verrouillage y est moins coûteux car implanté plus haut dans les couches logicielles. Les verrous sont gérées par un processus centralisé, le *Global Lock Manager* qui dirige des *Lock Manager* locaux.

2.1.4 Synthèse des caractéristiques du stockage distribué

Nous venons de présenter les travaux qui ont été menés sur l'accès aux données à distance en particulier dans le domaine des grappes. Les solutions proposées varient beaucoup selon le **nombre de clients**, l'**étendue géographique du système**, la **concurrence entre leurs accès** et les **besoins en performances** (voir la table 2.2).

	Clients	Étendue (m)	Concurrence	Performance
NFS	10^2	$< 10^2$	faible	faible
AFS, *Coda*, INTERMEZZO	**10-10^4**	**10-10^7**	faible	faible
ISCSI	1	< 10	aucune	**élevée**
GFS, GPFS, PVFS, LUSTRE	**10^2-10^3**	$< 10^2$	**élevée**	**élevée**

TAB. 2.2: *Réponses des différents systèmes de stockage distribué aux différents besoins des utilisateurs et des applications.*

Ces différents types de stockage distribué impliquent des utilisations du réseau très différentes. Nous nous concentrons ici sur le stockage dans les grappes, c'est-à-dire un contexte où les clients sont nombreux, gourmands et effectuent des accès concurrents. Les modèles qui nous intéressent sont ceux du dernier type, et en particulier les modèles client-serveur les plus répandus, PVFS et LUSTRE, qui ont l'avantage d'être ouverts,

tandis que GPFS est un système propriétaire d'IBM du type stockage partagé sans serveur. Nous cherchons à améliorer l'utilisation du réseau haute performance dans ce cadre.

Nous présentons maintenant les spécificités de ces réseaux avant de détailler leur interaction avec l'accès aux fichiers distants en partie 2.3.

2.2 Les réseaux haute performance

L'évolution des technologies réseaux depuis leur apparition a été assez simple. La nécessité de faire inter-opérer chaque machine avec le reste de la planète a conduit à utiliser des technologies et protocoles de communication difficiles à faire évoluer, tels que ETHERNET et TCP/IP. Les performances ont heureusement été grandement améliorées grâce à l'augmentation des bandes passantes (de quelques Mbit/s à 1 Gbit/s en local et de quelques kbit/s à quelques Gbit/s en longue distance). Mais ces technologies ont été conçues il y a une trentaine d'années et ne sont plus forcément adaptées aux besoins actuels. L'utilisation de ces protocoles sur un réseau très performant ne permet pas d'en tirer les meilleures performances [BGM99].

Les besoins des applications parallèles ont provoqué le développement de recherches sur des réseaux et protocoles associés spécifiques beaucoup plus performants. Leur bande passante dépasse le Gigabit/s depuis une dizaine d'années tandis que la latence atteint facilement moins de quelques microsecondes. Cependant, ces technologies ont été explicitement conçues pour les applications parallèles, rendant possiblement difficile leur utilisation dans d'autres contextes. Le développement de ces réseaux a néanmoins eu un grand impact sur le stockage distribué puisqu'il a permis des échanges haute performance entre les clients et les serveurs.

Avant d'étudier précisément l'interaction complexe entre le stockage et les réseaux en partie 2.3, nous rappelons le fonctionnement des réseaux traditionnels et leurs limites avant de présenter les principes clés des réseaux haute performance des grappes et leur mise en œuvre concrète.

2.2.1 Les limites des réseaux traditionnels

Les réseaux IP (*Internet Protocol* [IP81]) ont été conçus à la fin des années 1970. Même si quelques optimisations ont été apportées depuis, l'implantation des couches TCP/IP dans les systèmes d'exploitation modernes repose sur les mêmes principes.

2.2.1.a Les réseaux ETHERNET et l'interface SOCKET

La figure 2.7 situe l'implantation des différentes couches logicielles d'accès au réseau dans le système UNIX. Les applications communiquent à travers l'interface SOCKET qui permet l'accès à un service de transport fiable et connecté, TCP (Transmission Control

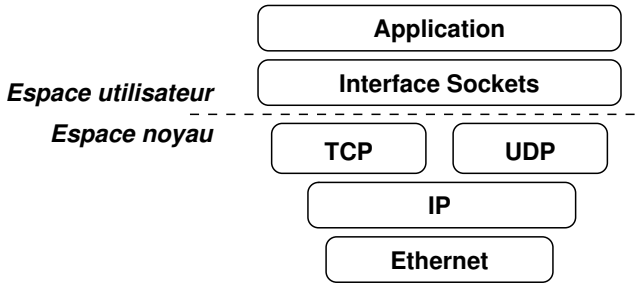


FIG. 2.7: *Accès aux réseaux* ETHERNET *par les protocoles* TCP *et* UDP *et l'interface* SOCKET.

Protocol [TCP81]), et un service de transport non-fiable et non-connecté, UDP (User Datagram Protocol [UDP80]). Ces deux services implantés par les protocoles de même nom assurent la transmission des données de haut niveau entre les applications distantes. La couche IP assure l'acheminement des paquets à travers différents équipements intermédiaires (routeurs et passerelles) jusqu'au destinataire. Enfin, la couche de niveau liaison, typiquement ETHERNET en réseau local, se charge de la transmission de paquets sur le lien physique sous la forme de trames.

L'interface SOCKET d'accès au réseau offre une syntaxe du même type que l'interface d'accès aux fichiers UNIX (voir l'annexe A.1.1). Les primitives sont généralement bloquantes : le système ne rend pas la main à l'application tant que les données n'ont pas été envoyées ou reçues. En fait, à l'émission, le système copie les données dans une zone spéciale (le *Socket Buffer*) puis rend la main à l'application (voir la figure 2.8). Le système d'exploitation se charge ensuite d'envoyer les données en arrière plan de cette zone vers la carte d'interface réseau par un accès direct à la mémoire (DMA, *Direct Memory Access*). Ainsi, l'application n'est bloquée que pendant la copie, au lieu d'attendre la réception du dernier message d'acquittement du dernier paquet (ce qui peut être long si le correspondant se trouve à l'autre bout de la planète). En réception, la stratégie est symétrique mais le récepteur est vraiment bloqué jusqu'à l'arrivée des données en mémoire utilisateur.

Ce modèle a l'inconvénient d'être gourmand en temps processeur et en occupation du bus mémoire (3 accès coté émetteur et 3 côté récepteur) puisqu'il faut effectuer une copie mémoire à l'envoi et à la réception. Par ailleurs, l'empilement des différents protocoles est assez lourd puisqu'ils ajoutent chacun leur en-tête aux paquets et imposent le calcul de sommes de contrôle (*Checksum*) pour vérifier l'intégrité des données. Des communications à haut débit consommeront donc une très grande quantité de temps processeur (les traitements protocolaires d'une connexion à 10 Gigabit/s saturent un ordinateur moderne). De plus, ces traitements protocolaires augmentent la latence des communications.

Par ailleurs, la capacité de ce modèle à fournir des notifications rapides des événements réseau est très limitée. En effet, en présence de communications simultanées vers de nombreux nœuds, l'interface SOCKET impose de vérifier l'état de chaque connexion

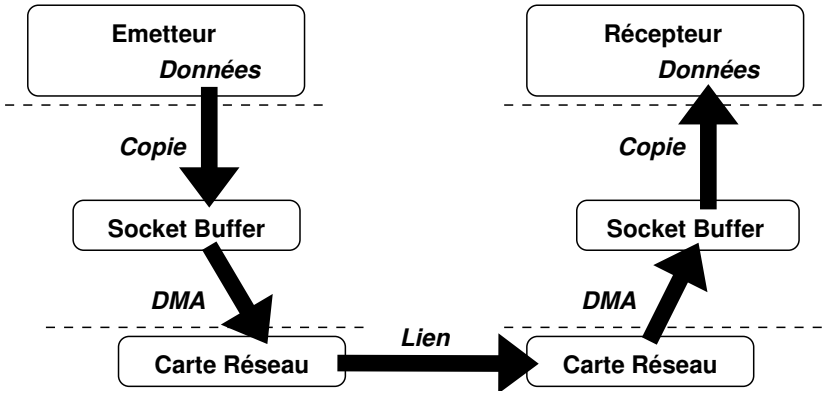


FIG. 2.8: *Transfert de données via l'interface* SOCKET.

pour savoir si des données sont arrivées. La stratégie classique (`poll/select`) permet de tester l'état de plusieurs connexions ou bloquer en attendant qu'un état soit modifié. Les *événements* attendus ici peuvent être la réception de données (l'application peut les recevoir sans bloquer) ou la terminaison de l'émission précédente de données (l'application peut émettre à nouveau sans bloquer). Cette stratégie de *notification d'événements* s'est révélée incapable de supporter efficacement un grand nombre de descripteurs de connexion [BM98]. De nombreux travaux ont été menés dans ce domaine, en particulier en ce qui concerne les serveurs WWW [Keg]. Cela s'est traduit par des variantes de `poll` permettant l'enregistrement de sources d'intérêt [BMD99] puis par des supports implantés directement dans les systèmes d'exploitation, tout d'abord par l'intermédiaire de `kevent/kqueue` dans FREEBSD [Lem00] et enfin l'apparition de `epoll` et LINUX AIO très récemment dans LINUX.

D'autres travaux ont étudié l'utilisation des threads pour traiter ces événements. En effet, les bibliothèques de threads modernes, en particulier NPTL (*Native POSIX Thread Library* [DM03]) ou encore les threads ordonnancés en espace utilisateur [DN03] permettent de réagir rapidement aux entrées-sorties. Cependant, l'utilisation de threads n'a pas été conçue pour des applications fortement *événementielles* mais plutôt pour des applications nécessitant une réelle concurrence entre files d'exécution [Ous96]. Par contre, le compromis consistant à utiliser un certain nombre de threads se répartissant les sources d'événements a prouvé son efficacité [Goo].

Ces nouvelles stratégies ont permis aux applications de supporter efficacement des communications réseau vers un très grand nombre de machines. Mais la notification des événements conserve une latence importante, notamment en raison de l'appel-système nécessaire pour accéder aux événements réseau.

2.2.1.b Les applications parallèles et MPI

Les applications de calcul parallèle ne se satisfont pas toujours de ces contraintes puisque, d'une part le temps processeur est nécessaire pour le calcul, et d'autre part la latence peut avoir un impact notable sur les performances de l'application. Le déport du traitement des couches protocolaires dans des cartes réseau programmables a donc été proposé.

Dans le cadre des grappes, l'interface SOCKET et la pile TCP/IP ne sont que rarement utilisées. Les communications sont plutôt effectuées par une interface dédiée au calcul parallèle, MPI (*Message Passing Interface* [For94, MPIa]) qui se base sur le paradigme du *Rendez-vous* (par opposition aux connexions du modèle SOCKET). Les applications postent ici des requêtes *asynchrones* d'envoi et réception de données puis testent plus tard leurs terminaisons.

L'aspect asynchrone est primordial puisqu'il permet théoriquement de recouvrir les communications par du calcul, au lieu d'attendre sans rien faire jusqu'à la terminaison de la requête. Il faut noter qu'il n'est pas utile ici de copier les données dans une zone temporaire de type *Socket Buffer* puisque le caractère asynchrone des communications impose à l'application de ne pas toucher aux zones mémoire concernées jusqu'à la notification de terminaison de la communication.

L'interface MPI a semblé initialement très complexe (voir en annexe A.2.1) mais s'est en fait rapidement imposée comme le standard des communications pour applications parallèles. Des industriels ont tenté d'imposer d'autres interfaces de programmation à la place de MPI. C'est notamment le cas de VIA (*Virtual Interface Architecture* [SASB99, BAP00]) qui propose des primitives asynchrones en mode connecté (voir l'interface en annexe A.2.2). Cependant, ce modèle ne s'est jamais imposé, notamment car le modèle du *Rendez-vous* proposé par MPI se révèle plus adapté et efficace pour concevoir des applications de calcul parallèle [BM00].

Le remplacement de l'interface SOCKET par MPI dans les grappes s'est accompagné du développement de réseaux dédiés pour profiter au maximum de ses avantages.

2.2.2 Principes clés des réseaux haute performance de grappe

Les besoins en communication des applications parallèles étant très particuliers, des réseaux et des protocoles très spécifiques ont été conçus pour ce contexte. L'idée générale est de raccourcir au maximum le chemin critique suivi par les données échangées entre deux processus s'exécutant sur des nœuds différents. Les principaux mécanismes proposés sont :

Communications zéro-copie : Les applications parallèles ne doivent pas voir leur temps processeur consommé par les communications. Supprimer les copies intermédiaires, consommatrices en temps processeur, est possible en utilisant une carte d'interface capable d'initier des DMA (*Direct Memory Access*) pour transférer directement des données entre la mémoire utilisateur des applications et le réseau sans intervention du processeur central.

Suppression du système d'exploitation du chemin critique : Le coût d'un appel-système est de l'ordre d'une demi-microseconde sur les machines modernes (environ 0,4 μs sur un processeur cadencé à 2 GHz). Il est possible d'éviter les appels-système (*OS-bypass*) en autorisant les applications à accéder directement à la carte d'interface réseau. Le gain en latence est alors appréciable puisque les latences sont du même ordre de grandeur (quelques microsecondes).

Réactivité aux événements : Les périphériques peuvent notifier le processeur central d'un événement réseau en l'interrompant. Le coût d'une *interruption* étant assez élevé (plusieurs microsecondes), faire une attente active (*Scrutation*) permet de récupérer plus rapidement les événements. La contre-partie est que l'attente active consomme du temps processeur. Mixer intelligemment la scrutation (pour les événements arrivant tôt) et les interruptions (pour les événements arrivant tard) permet de récupérer efficacement tous les événements réseau [MGH+96].

Réseau physique fiable : Les topologies des grappes sont statiques et fermées, et le trafic y est déterministe. Il est donc tout à fait possible de prévoir la quantité de données qui va circuler dans les différents liens. Un dimensionnement suffisant du cœur du réseau permet d'éviter les problèmes de congestion, donc des pertes de messages. La fiabilité des liens physiques, en particulier les fibres optiques, évite les corruptions de données. Les protocoles de transmission (et retransmission en cas d'erreur) sont donc beaucoup moins complexes que ceux que l'on trouve dans l'Internet (TCP/IP) qui est par essence un réseau non-fiable et fortement congestionné.

Traitement du protocole dans la carte d'interface : La simplicité du protocole permet de le traiter entièrement sur les cartes d'interface dotées d'un processeur et de mémoire embarqués. L'application soumet à la carte ses requêtes de communication et peut recouvrir leur traitement par du calcul.

Routage par la source : La régularité et la staticité des réseaux des grappes permet de fixer le routage. Le processeur embarqué sur la carte d'interface peut placer les informations de routage en début de paquet. Le travail des commutateurs se réduit alors au simple fait d'extraire ce petit en-tête puis de diriger le paquet, ce qui en améliore les performances. Les congestions peuvent être gérées par un contrôle de flux très simple au niveau des liens qui permet à un récepteur d'indiquer à l'émetteur qu'il doit attendre avant d'émettre les paquets suivants.

Nous détaillons maintenant la mise en œuvre de ces idées dans le support logiciel du réseau. Nous nous concentrons sur les aspects qui intéressent notre étude de l'interaction entre les interfaces de programmation du réseau et les couches système.

2.2.3 Support logiciel pour les réseaux haute performance

2.2.3.a Modèles de programmation

Le traitement des requêtes de communication par la carte d'interface permet l'implantation de modèles asynchrones de programmation. L'application soumet des requêtes à la carte, recouvre leur traitement par des calculs (*Overlap*) puis teste leur terminaison plus tard.

Ces requêtes sont de deux types, qui correspondent aux deux paradigmes de programmation des applications parallèles, le *Rendez-vous* (passage de messages, utilisé dans MPI) et les accès mémoire à distance. Dans le premier cas, les requêtes sont du type émission et réception de message. Les paramètres de ces requêtes sont comparés pour savoir quelle émission va être reçue par quelle requête de réception. Cette correspondance (*Matching*) peut être effectuée dans l'hôte (ce qui réduit le recouvrement) ou dans la carte d'interface (ce qui peut nécessiter beaucoup de ressources mémoire et de puissance).

Dans le second modèle, les primitives de communication sont du type lecture ou écriture en mémoire distante, des RDMA (*Remote Direct Memory Access*). L'application définit des fenêtres de RDMA (*Window*) dont elle donne les identifiants aux autres nœuds. Ces derniers peuvent ensuite lire et écrire à distance dans les zones mémoire décrites par ces fenêtres. Là aussi, plus la carte d'interface offre de ressources matérielles et des fonctionnalités évoluées, plus le protocole pourra y être déporté, et plus le recouvrement dans l'hôte sera amélioré.

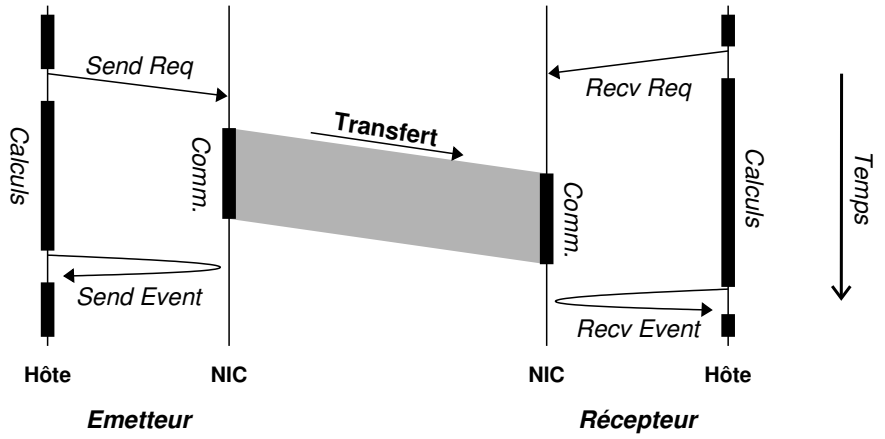


FIG. 2.9: *Diagramme temporel d'une communication selon le paradigme du* Rendez-vous. *L'application du nœud émetteur poste une requête d'émission (*Send Req*) tandis que le récepteur poste une réception (*Recv Req*). Les cartes d'interfaces (*NIC*) traitent cette communication en tâche de fond (*Comm.*) pendant que les applications continuent leurs calculs. Les applications viennent tester plus tard la terminaison des requêtes (*Send Event *et* Recv Event*).*

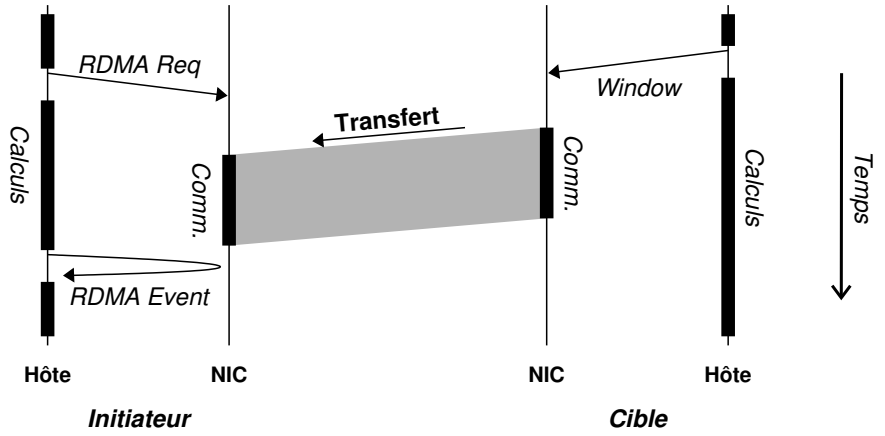


FIG. 2.10: *Diagramme temporel d'une lecture mémoire à distance par* RDMA*. L'application du nœud cible crée une fenêtre de* RDMA *(*Window*) tandis que l'initiateur poste une requête (*RDMA Req*). Les cartes d'interfaces (*NIC*) traitent cette communication en tâche de fond (*Comm.*) pendant que les applications continuent leurs calculs. L'application initiatrice vient tester plus tard la terminaison de la requête (*RDMA Event*).*

Dans le modèle du *Rendez-vous*, les deux nœuds sont notifiés quand une communication est effectuée (voir la figure 2.9). Par contre, lors d'un RDMA, l'application cible n'est pas notifiée (voir la figure 2.10). S'il est assez facile de mettre en œuvre des accès mémoire à distance sur une interface de programmation de type passage de messages, le contraire peut-être difficile. En effet, l'absence de notification distante oblige à effectuer des attentes actives, ce qui consomme du temps processeur, en particulier si on doit attendre différents messages simultanément.

2.2.3.b Transferts zéro-copie

Les transferts de données zéro-copie par DMA (voir la figure 2.11) ne sont en fait importants que pour les larges messages pour lesquels une copie mémoire n'est pas envisageable. Ils permettent de saturer les liens physiques, c'est-à-dire jusqu'à 1 Go/s de nos jours (c'est notamment la bande passante des bus PCI courants et des liens des technologies réseau les plus avancées).

Mais ces DMA ont l'inconvénient de nécessiter une interruption pour notifier leur terminaison sans faire d'attente active. Le coût est de plusieurs microsecondes, ce qui est négligeable pour un large transfert de données. Mais pour les petits messages, c'est la latence qui est importante. En effet, avec une latence L et une bande passante B, le temps de transfert d'un message de taille s est $T = L + s/B$. Comme L est de l'ordre de quelques microsecondes et B plusieurs centaines de megaoctets par seconde, $T \simeq L$ pour s inférieur à quelques kilooctets. Une modélisation générale de ces délais dans le cadre des machines parallèles a été proposée dans le modèle LogP [CKP+93].

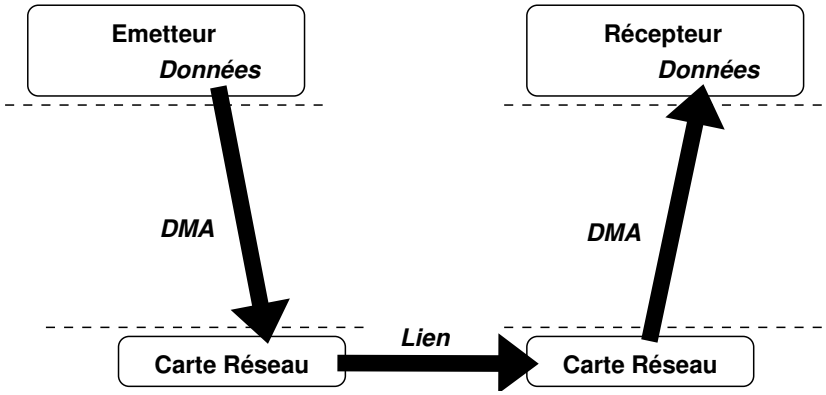


FIG. 2.11: *Transfert de données zéro-copie* OS-bypass *sur un réseau de grappe.*

Il est préférable de gaspiller quelques cycles processeur pour être notifié rapidement plutôt que de subir le surcoût d'une interruption. Ceci permet de conserver une très petite latence des petits messages au prix de quelques cycles processeur. Au lieu des DMA, on peut utiliser ici les PIO (*Programmable Input/Output*) c'est-à-dire des communications entièrement contrôlées par le processeur central. Le processeur n'a pas besoin dans ce cas d'attendre une notification de la carte d'interface réseau puisque c'est lui qui dirige la communication.

2.2.3.c Notification des événements

La notification de terminaison des requêtes est un point important dans la conception des réseaux haute performance. En effet, comme on vient de le voir, il est nécessaire de notifier rapidement les événements pour abaisser la latence. On utilise alors des méthodes mixant scrutation pour les notifications rapides (l'application fait une attente active) et interruption pour les notifications tardives (l'application s'endort et est réveillée par une interruption de la carte réseau). Ces mécanismes permettent de notifier rapidement une application qui a demandé à être bloquée jusqu'à la terminaison d'une requête. Cependant, il faut également être capable de réagir efficacement au grand nombre d'événements qui peuvent arriver sur un réseau haute performance.

Les stratégies modernes du type `epoll` ou LINUX AIO ont permis aux applications d'attendre efficacement des événements réseau (en particulier l'arrivée de messages dans des connexions TCP) sans souffrir du nombre de sources à écouter simultanément (voir en 2.2.1.a). Cependant, ces mécanismes imposent à l'application de recourir à un appel-système pour aller tester l'état des connexions dans le système d'exploitation. Le coût de la traversée des couches système étant de l'ordre d'une microseconde, il est préférable de l'éviter pour atteindre des latences très faibles.

Ce modèle événementiel nécessite de faire remonter en espace utilisateur les événementiels notifiés par le matériel (généralement par interruption). Plutôt que de traverser

les couches système, les interfaces de programmation des réseaux rapides préfèrent tester directement l'arrivée d'événements depuis l'espace utilisateur en accédant directement au *mapping* (projection) mémoire de la carte d'interface[2]. Le système d'exploitation n'est utilisé que lorsque l'application doit dormir en attendant une interruption.

Par ailleurs, il peut être important de prévenir l'application sans qu'elle ne soit obligée d'interrompre les calculs pour demander explicitement l'état des requêtes devant être traitées. La plupart des interfaces de programmation des réseaux rapides ne font progresser les communications que lorsque l'application appelle explicitement une fonction de la bibliothèque. Il est parfois souhaitable d'effectuer des traitements en tâche de fond sans attendre l'intervention de l'application, par exemple pour libérer les ressources ou répondre à une requête distante. Pour ce faire, AM (*Active Messages* [vECGS92]) utilise un thread spécial chargé d'appeler les routines de traitement (*receive handler*) des messages reçus. Ainsi, au lieu de ne faire progresser les communications que lorsque l'application appelle des fonctions de la bibliothèque, les échanges se poursuivent de manière permanente en arrière plan. Cela permet d'ailleurs d'améliorer le recouvrement des communications par le calcul puisque l'application peut se concentrer sur ses calculs.

Ces méthodes de notification sont efficaces mais diffèrent considérablement des stratégies classiques de type `poll`, `epoll` ou LINUX AIO. Cela rend difficile leur utilisation dans un modèle classique de programmation. Il est par exemple impossible de recevoir de façon uniforme les événements du réseau haute performance et des disques, ce qui réduit l'efficacité d'un serveur de stockage. Nous détaillerons ce problème en partie 6.2.

2.2.3.d Le problème des traductions d'adresses

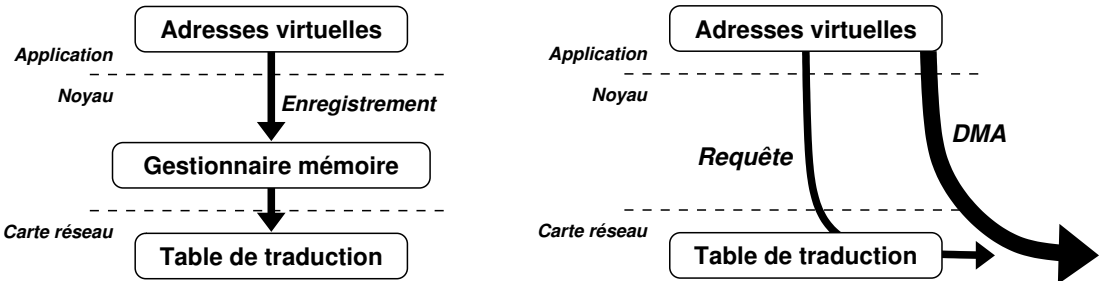
La soumission directe par l'application de requêtes impliquant des DMA entre la mémoire de l'application et le réseau a soulevé un problème technique assez nouveau. L'application ne manipule que des adresses virtuelles, tandis que le matériel ne manipule que des adresses physiques. La traduction est habituellement effectuée dans le système d'exploitation. L'application passe des adresses virtuelles au noyau par un appel-système, puis le noyau les traduit en adresses physiques avant de les donner au matériel.

Dans les réseaux haute performance, les communications *OS-bypass* évitent l'intervention du système d'exploitation pour réduire la latence. Mais cela interdit donc l'assistance du système d'exploitation pour traduire les adresses.

La stratégie la plus répandue pour résoudre ce problème est appelée *Enregistrement mémoire*. Il s'agit de déclarer à l'initialisation les zones mémoire qui vont être utilisées dans des communications. On effectue alors un appel-système pour traduire leurs adresses et enregistrer définitivement les correspondances dans la carte d'interface (voir

[2] Les périphériques peuvent exposer une partie de leur mémoire embarquée au processeur central. Le système peut alors projeter cette mémoire en espace virtuel, c'est-à-dire la faire correspondre à des adresses virtuelles (*mapping*), ce qui permet ensuite d'y accéder de manière transparente.

la figure 2.12(a)). Toutes les communications peuvent ensuite être *OS-bypass*, l'application passant des adresses virtuelles à la carte et celle-ci les traduisant grâce aux correspondances préalablement enregistrées (figure 2.12(b)). Cette idée a été introduite dans les travaux sur le système U-NET [vEBBV95] sous la forme d'une zone statiquement pré-enregistrée puis généralisée dans U-NET/MM [WBvE97].



(a) Enregistrement mémoire.

(b) Communication zéro-copie *OS-bypass* après enregistrement.

FIG. 2.12: *Modèle zéro-copie* OS-bypass *avec enregistrement mémoire.*

L'enregistrement mémoire est une méthode efficace mais il présente deux gros inconvénients. Tout d'abord, les applications ne sont généralement pas écrites pour préparer ainsi les zones mémoire utilisées pour les entrées-sorties. L'utilisation d'un tel modèle dans une application normale peut donc être difficile. Une solution couramment utilisée consiste à charger une bibliothèque partagée d'enregistrer à la volée et de manière transparente les zones mémoire utilisées par l'application dans ses entrées-sorties.

Le second problème est le coût souvent très élevé de l'enregistrement et du désenregistrement mémoire, qui peut être de l'ordre de la centaine de microsecondes comme nous le verrons en 5.1.1. Ce coût est notamment beaucoup plus élevé que la latence des petits messages (quelques microsecondes). Il n'est donc rentable que si les zones mémoire sont grandes et réutilisées plusieurs fois. Pour les petits messages, il reste préférable d'utiliser une copie intermédiaire dans une zone mémoire statiquement pré-enregistrée. Le gaspillage de quelques cycles processeur est au final plus économique que l'enregistrement. Dans les autres cas, on utilise une stratégie appelée *cache d'enregistrement mémoire* (*Pin-down Cache*, introduite dans [TOHI98]). Plutôt que de désenregistrer une zone mémoire dès la terminaison de la communication qui la mettait en jeu, l'idée est ici de retarder au maximum le désenregistrement des zones mémoire. En effet, tant qu'il est possible d'enregistrer de nouvelles pages, il est inutile de perdre du temps à désenregistrer les anciennes. Le coût du désenregistrement réel est ainsi évité. De plus, si une zone est laissée enregistrée dans le cache, ses réutilisations suivantes ne nécessiteront pas un nouvel enregistrement. Le coût global est donc largement réduit, en particulier si les mêmes zones mémoire sont ré-utilisées plusieurs fois. Par ailleurs, on peut également combiner les désenregistrements de différent zones (*Batch Deregis-*

tration [ZBJ+02]) afin d'en diminuer le coût apparent.

2.2.4 Description des principaux réseaux haute performance

Nous présentons ici les principales technologies réseaux qui ont dominé le domaine des grappes de calculateurs depuis une dizaine d'années. L'ordre chronologique nous conduit a présenter en premier lieu SCI puis MYRINET et QUADRICS avant de terminer par la récente norme INFINIBAND.

2.2.4.a Scalable Coherent Interface

Le système SCI (*Scalable Coherent Interface* [Gus92]) a été introduit au début des années 1990 par la société DOLPHIN [Dol]. Son but est de fournir un mécanisme de mémoire partagée à très faible latence. Pour cela, les cartes d'interface mettent en place un pont (*Bridge*) entre le bus mémoire d'une machine et ceux des nœuds distants à travers le réseau.

Le réseau SCI peut-être utilisé par l'interface de programmation SISCI (*Software Interface for SCI*). Elle propose deux principales méthodes de communication par accès mémoire à distance. La première consiste à projeter (*mapper*) des segments de mémoire distante dans la mémoire virtuelle locale (*Mémoire partagée*). Les accès à cette mémoire mappée provoquent un déroutement par la carte d'interface. La seconde méthode consiste à demander explicitement un accès mémoire à distance par des primitives de type RDMA (*Remote Direct Memory Access*). L'interface de programmation SISCI est détaillée en annexe A.3.3.

SCI a été utilisé dans de nombreux travaux dans le domaine des grappes, par exemple dans [ISSW97]. Les modèles récents offrent une latence d'accès mémoire à distance de 1,4 μs tandis que la bande passante atteint théoriquement 340 Mo/s. Cependant, cette technologie souffre de certaines limites. Tout d'abord, la topologie du réseau est un anneau (ou tores 2D ou 3D plus récemment). Le partage de bande passante que cela implique limite la bande passante réellement observée ainsi que la possibilité de passer à l'échelle des grandes grappes. Ensuite, le traitement des communications est très largement effectué par le processeur de l'hôte, ce qui réduit considérablement la capacité de recouvrement des communications.

Enfin, le modèle de programmation orienté *Mémoire partagée* rend difficile son utilisation dans des applications de type MPI, qui sont actuellement les plus courantes sur les grappes. En effet, comme nous l'avons développé en partie 2.2.3.a, le modèle du *Rendez-vous* nécessite une notification sur les deux machines mises en jeu dans la communication. Dans le cas de l'utilisation du modèle de *Rendez-vous* au dessus du modèle à *Mémoire partagée* de SISCI, la latence observée au niveau de l'application passe de 1,4 à 3,8 μs.

2.2.4.b Myrinet

MYRINET [BCF+95] est le leader du marché des réseaux d'interconnexion pour grappes de calcul. Il est produit par la société MYRICOM [Myr] depuis 1995. Les caractéristiques principales sont un excellent passage à l'échelle (la plus grande grappe MYRINET actuelle est le *MareNostrum* au *Barcelone Supercomputing Center*, composée de 2282 nœuds) et des cartes performantes et faciles à programmer.

Le réseau se base sur une topologie en *clos* basée sur des commutateurs pouvant relier jusqu'à 256 nœuds. Le routage par la source permet un transfert rapide des paquets sur le réseau physique tandis que la fiabilité est assurée par une vérification des *Checksum* dans le matériel. Les liens physiques soutiennent 250 Mo/s mais certaines cartes d'interface permettent d'utiliser deux liens par nœud et donc d'atteindre à 500 Mo/s.

Les cartes d'interface embarquent un processeur RISC, le LANAI, dont les dernières versions tournent à 333 MHz. Il dispose de sa propre mémoire SRAM (2 Mo en général) très rapide (accessible en 2 cycles), et d'un moteur DMA puissant. Les spécifications libres de ce matériel, la puissance du LANAI et la facilité de reprogrammation de ces cartes ont conduit à de nombreux travaux académiques sur les protocoles réseaux et leurs mises en œuvre dans MYRINET, par exemple dans VMMC (*Virtual Memory-Mapped Communication* [DBLP97]) ou FM (*Fast Messages* [LPC98]).

Parmi les travaux les plus intéressants, il faut citer BIP (*Basic Interface for Parallelism* [PT97]) qui a longtemps été la référence en matière d'utilisation efficace des réseaux MYRINET. Ses optimisations indépendantes des petits messages (par PIO et *OS-bypass*) et des gros (par DMA et zéro-copie) permet d'atteindre une latence de 3 μs et d'utiliser 96 % de la capacité des liens. L'utilisation du processeur central reste très faible, ce qui permet un très bon recouvrement. L'interface de programmation logicielle de BIP est orientée passage de messages.

Le pilote officiel actuel des réseaux MYRINET s'appelle GM. Son interface est également orientée passage de messages (voir en annexe A.3.1 ou dans [GM03]). Ce logiciel industriel est plus robuste que le logiciel académique BIP (notamment vis-à-vis des erreurs de transmissions, par exemple lorsqu'un message est perdu ou corrompu) mais a des performances inférieures. La latence atteint 6 μs sur les machines les plus performantes, au prix d'une consommation processeur assez importante. Il a par exemple été montré dans [LCY+03] que le traitement de la terminaison nécessitait 4 μs de temps processeur dans l'hôte.

La bande passante observée avec GM est bonne mais cela impose à l'application d'enregistrer explicitement les zones mémoire impliquées dans les communications. Par ailleurs, les méthodes de notification sont très limitées. Attendre la terminaison d'une requête particulière est impossible. Cela impose d'utiliser un thread recevant les événements réseau dans une file unique et réveillant le thread concerné par l'événement reçu. Nous présenterons plus précisément l'utilisation de GM dans nos travaux aux chapitres 5 et 6.

Les performances assez moyennes de GM, en particulier par rapport à BIP, ont conduit MYRICOM à développer un nouveau pilote, nommé MX (*Myrinet Express*). Les

buts de MX sont notamment de fournir des performances similaires à BIP avec la robustesse de GM et une interface de programmation plus souple. La plupart des applications tournant sur les grappes étant des applications MPI, MX propose en fait une interface très proche de MPI (voir en annexe A.3.2 et A.2.1 et dans [MX05]). Elle fournit principalement des routines asynchrones de soumission de requête d'émission ou réception supportant le *Matching* (voir en 2.2.3.a) et les communications vectorielles, mais aussi les accès mémoire à distance et les communications collectives. MX est en cours de développement mais propose déjà des performances inédites, 2,6 μs de latence MPI et une très bonne utilisation de la capacité du lien. Nous présenterons plus précisément MX en partie 7.1.1.

2.2.4.c Quadrics

Les réseaux QsNet de Quadrics [Qua] sont probablement les plus performants (et les plus chers) du marché, avec 1,8 μs de latence et 900 Mo/s de bande passante. Pour ce faire, la technologie QsNet repose sur du matériel très performant et quelques innovations logicielles.

Le réseau est là aussi composé de commutateurs formant une topologie de type réseau de clos et reliant des cartes d'interface (nommées Elan) par des liens de capacité 1,06 Go/s. Ces cartes d'interface sont elles aussi très puissantes et programmables. Elles proposent des fonctionnalités avancées, par exemple du multicast. Mais contrairement à Myrinet, peu de travaux ont été effectués sur ce matériel, notamment parce que ses spécifications ne sont pas ouvertes.

Les cartes d'interface Elan ont la particularité d'embarquer une MMU (*Memory Management Unit*) comme dans les processeurs des machines. Ce circuit électronique embarqué est capable de traduire directement les adresses virtuelles passées par l'application en adresse physique. C'est un avantage important puisque les concurrents doivent émuler une MMU logiciellement pour effectuer cette traduction. Cette innovation technologique est assistée d'une modification dans les systèmes d'exploitation pour maintenir la MMU de la carte constamment à jour vis-à-vis de la MMU de l'hôte. Ainsi, les techniques du type enregistrement mémoire sont inutiles, et les communications zéro-copie deviennent très faciles. C'est par de telles innovations et également par la puissance du matériel qu'une latence très faible et une bande passante très élevée sont atteintes [PcFH+02, PFHC03].

Les réseaux QsNet sont pilotés par l'interface de programmation Elanlib du type accès mémoire à distance. Pour répondre aux besoins des applications de type MPI, Quadrics fournit également la bibliothèque Tports (*Tagged Message Ports*) implantée au dessus de Elanlib (voir l'annexe A.3.4). Contrairement aux réseaux SCI ou Infiniband, la mise en œuvre de ce modèle *Rendez-vous* au-dessus des accès mémoire à distance est efficace car QsNet fournit des techniques de notification à distance.

La nécessité de modifier le système d'exploitation pour bénéficier de communication zéro-copie de manière transparente rebute cependant certains utilisateurs. Quadrics fournit donc depuis peu des pilotes logiciels basés sur l'enregistrement mémoire et ne

nécessitant pas un système d'exploitation modifié. Les performances sont alors probablement moins bonnes puisque le coût logiciel nécessaire pour compenser l'absence de modification du noyau est forcément supérieur au coût des modifications elle-mêmes.

2.2.4.d Infiniband

INFINIBAND est une norme créée à la fin des années 1990 par un consortium de nombreux constructeurs de matériel informatique afin de définir l'architecture des entrées-sorties de l'avenir. L'idée initiale était de définir un standard de bus d'entrées-sorties pour remplacer le bus PCI mais aussi les systèmes d'accès au stockage et au réseau [Inf01]. Finalement, après le retrait de certains constructeurs, et notamment INTEL qui a annoncé PCI EXPRESS pour remplacer le bus PCI, les travaux ont été recentrés autour des réseaux haute performance [Pfi01], essentiellement pour les communications dans les grappes de calcul, mais aussi pour accéder aux dispositifs de stockage attaché au réseau (NAS, *Network Attached Storage*), notamment FIBRE CHANNEL.

Les processeurs des cartes d'interface INFINIBAND sont désormais essentiellement produits par MELLANOX. Les équipements réseau et logiciels sont ensuite distribués par un certain nombre de constructeurs dont TOPSPIN, VOLTAIRE et INFINICON. Les spécifications de la norme précisent à la fois la mise en œuvre matérielle et logicielle. Cependant, chaque revendeur distribue ses propres couches logicielles d'accès au réseau respectant plus ou moins la norme. Le projet OPENIB [Ope] propose une alternative libre à ces différentes distributions logicielles propriétaires.

La conception du réseau INFINIBAND ressemble en fait plus à un réseau IP haute performance qu'à un réseau fondamentalement conçu pour les grappes de calcul. Le protocole de bas niveau est très proche de IP. La topologie n'est pas fixée et le routage est effectué dans les commutateurs. Le réseau peut donc être organisé en *clos* où en topologies plus simples si les besoins en bande passante au cœur sont moindres.

La principale interface est les VERBS qui ressemble fortement à VIA (voir en annexe A.3.5 et A.2.2). Les communications bas niveau profitent des capacités du matériel à faire du RDMA pour obtenir de très bonnes performances. Les interfaces de type DAPL sont également souvent utilisées sur INFINIBAND [VRC+03].

La bande passante théorique est actuellement de 2 Go/s en utilisant des cartes d'interface munies de deux liens INFINIBAND 4x. Cependant, les limites des bus d'entrées-sorties des machines rendent encore actuellement difficile l'obtention de telles performances. La généralisation de PCI EXPRESS devrait y remédier. La latence obtenue se situe entre 5 et 10 μs. En pratique, l'implantation entièrement basée sur les RDMA rend difficile l'obtention de performances aussi élevées dans les applications MPI sur les grandes grappes. En effet, comme dans SISCI, l'absence de notification sur la machine cible du RDMA l'oblige à effectuer une attente active. En plus de consommer du temps processeur, cette stratégie ne passe pas du tout à l'échelle lorsque le nombre de messages ou de nœuds augmente.

2.2.5 Synthèse des caractéristiques des réseaux haute performance

Technologie (Protocole)	ETHERNET (IP)	SCI (SISCI)	MYRINET (MX)	QSNET (ELANLIB)	INFINIBAND (VERBS)
Interface	SOCKET	RDMA	*Rendez-vous*	RDMA + notification	RDMA
Zéro-copie	non	non	sauf petits messages	sauf petits messages	sauf petits messages
Enregistrement	non	oui	oui	non si système modifié	oui
Utilisation CPU	grande (copie)	grande (copie PIO)	faible (DMA)	faible (DMA)	faible (DMA)
Topologie	Variable	Anneau ou Tore	Clos	Clos	Clos ou variable
Bande passante théorique (Mo/s)	$\simeq$ 100+100	340 partagés	500+500	900+900	800+800
Latence (μs)	$\simeq$ 50	2	3	2	5

TAB. 2.3: *Comparatif des réseaux haute performance en terme d'interface de programmation, communications zéro-copie, nécessité de préparer les zones mémoire impliquées (enregistrement), utilisation du processeur central pour traiter les communications, topologie et performances brutes.*

La table 2.3 résume les principales caractéristiques des différents réseaux haute performance utilisés dans les grappes de calcul. Les réseaux des grappes obtiennent des performances beaucoup plus élevées qu'ETHERNET en utilisant des interfaces de programmation dédiées et des fonctionnalités avancées dans la carte. Bénéficier de ces performances dans le cadre du stockage distribué permet d'envisager des systèmes très performants. Cependant, ces réseaux ont été essentiellement conçus pour les communications entre nœuds d'une application parallèle, notamment par MPI. Cette specialisation ne répond pas forcément aux besoins du stockage distribué puisque les contraintes y sont différentes. Nous étudions maintenant ce problème en détaillant les interactions entre les interfaces de programmation des réseaux des grappes et les couches système d'accès aux fichiers.

2.3 Interaction entre les interfaces d'accès aux fichiers et au réseau

Nous résumons tout d'abord l'évolution conjointe des réseaux et de l'accès aux fichiers distants dans les grappes ces 20 dernières années avant de détailler les problèmes d'interaction entre leurs interfaces de programmation respectives.

2.3.1 Évolution

Les grappes de calculs sont apparues au milieu des années 1990, avec notamment le projet BEOWULF [SSB+95]. Leurs besoins spécifiques ont eu une influence importante sur la conception des systèmes de fichiers. La figure 2.13 présente les principales étapes de l'évolution des réseaux et de l'accès aux fichiers dans les grappes.

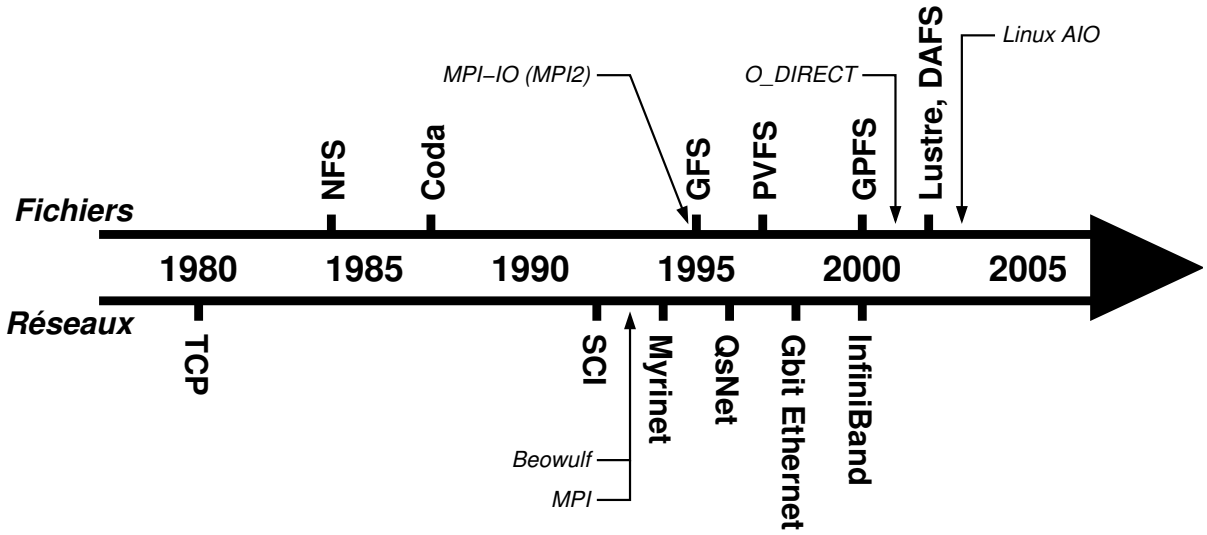


FIG. 2.13: *Évolution des technologies réseaux et de l'accès aux fichiers distants.*

Avant l'apparition des grappes, les applications avaient peu de besoins en latence ou en bande passante. Les protocoles simples d'accès aux fichiers distants du type NFS suffisaient à répondre à leurs besoins. Avec la généralisation de l'Internet, les travaux se sont concentrés autour du premier domaine de recherche que nous avons présenté en partie 2.1.2 (les problèmes de cache et réplication).

2.3.1.a De NFS aux bibliothèques spécifiques

La popularité de NFS a conduit les administrateurs à l'utiliser dans les premières grappes de calcul. Seuls les quelques super-calculateurs disponibles au début des années 1990 ont donné lieu à quelques travaux très spécifiques mais peu diffusés. Ce n'est que quelques années plus tard, surtout à partir de 1995 avec la généralisation des grappes de calcul haute performance, que les lacunes de NFS et des autres systèmes ont été réellement mises en évidence. L'état des principaux systèmes de fichiers distribués disponibles dans les grappes à la fin des années 1990 est résumé dans [Sch99].

La sensibilité de NFS aux réseaux haute performance est assez faible. Il se révèle incapable de profiter des grandes bandes passantes et des faibles latences. Son temps

de réponse sera par exemple le même si la latence du réseau passe de 100 à 10 μs ou si la bande passante passe de 2,5 à 25 Mo/s [MC99]. C'est notamment lié à son coût protocolaire trop important. En effet, il a été montré dans [HSCL97] que la traversée de la couche RPC (*Remote Procedure Call*) pouvait représenter jusqu'à 50 % du coût protocolaire total.

Des travaux ont été menés pour améliorer les performances de NFS sur les réseaux haute performance, par exemple dans le cadre du projet ATOMIC [Fab98], mais ils n'ont pas été introduits dans la version standardisée. NFS3 [NFS95] a simplement légèrement amélioré les performances en supportant le *Write-back Caching* qui permet de différer les écritures réelles sur le serveur distant pour ne pas bloquer le client trop longtemps. Le client reprend donc la main rapidement pendant que le serveur écrit réellement les données dans le dispositif de stockage physique. La dernière version, NFS4 [NFS00] n'apporte quant à elle que des améliorations destinées à la sécurité et aux réseaux longue distance. En dehors de sa simplicité et de sa disponibilité, NFS ne présente donc que peu d'intérêt pour les applications parallèles dans le domaine des grappes. Il s'est rapidement révélé incapable de soutenir efficacement la demande de plusieurs centaines de clients.

La répartition de la charge de travail sur différents serveurs a été très étudiée et a donné lieu à tous les travaux décrits en partie 2.1.3. Si le problème des performances est souvent cité, c'est surtout un défaut d'interface de programmation et de support système adapté qui s'est révélé. Nous avons expliqué en partie 2.2.1 que l'interface standard d'accès au réseau (SOCKET) s'était révélée incapable de répondre aux besoins des applications parallèles, notamment le recouvrement calcul/communication et les communications zéro-copie. Cela s'est traduit par la généralisation de l'utilisation de l'interface MPI. De la même façon, les limites de l'interface standard d'accès aux fichiers (voir par exemple [BdRC93]) ont conduit à la conception de l'extension MPI-IO [MPIb].

En effet, les applications parallèles utilisent les accès disque de manière très spéciale [SR97]. Les accès peuvent notamment être :

Vectoriels : plusieurs segments non-contigus sont accédés simultanément ;

Parallèles : les sous-parties d'un même accès sont traitées par différents serveurs ;

Collectifs : les requêtes de plusieurs clients sont regroupées avant d'être traitées ;

Asynchrones : les requêtes sont traitées en arrière plan tandis que l'application continue de calculer.

Les systèmes d'exploitation existants et leurs interfaces de programmation ne proposaient aucune de ces fonctionnalités avancées dans leur interface standard.

Les concepteurs de systèmes de fichiers pour les grappes ont donc commencé à contourner le modèle traditionnel, c'est-à-dire à ne plus implanter leur client dans le système d'exploitation (voir la partie 2.1.1). Les bibliothèques de type MPI-IO définissent leur propre interface (voir en annexe A.2.1) en espace utilisateur et y implantent directement les accès aux fichiers distants. Ainsi, au lieu d'être échangées entre le système d'exploitation du client et le serveur, les données sont échangées entre une bibliothèque utilisateur et le serveur (voir la figure 2.14).

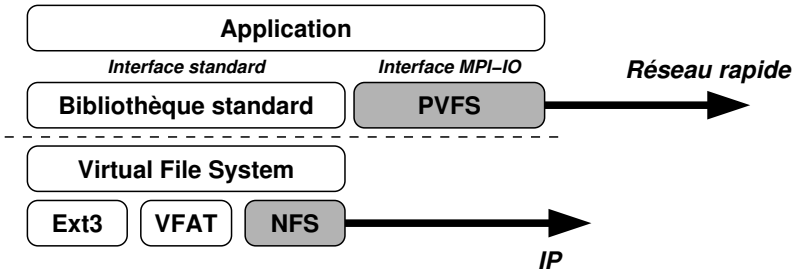


FIG. 2.14: *Mise en œuvre des accès distants par une interface spécifique (ici* MPI-IO*) en espace utilisateur.*

Une telle mise en œuvre permet par ailleurs de profiter des communications zéro-copie proposées par les réseaux haute performance des grappes. En effet, les interfaces de programmation de ces réseaux sont essentiellement disponibles en espace utilisateur. Il est donc tout à fait possible de les utiliser ici pour transférer efficacement les données entre les clients et le serveur de fichiers distant. C'est notamment ce qu'a fait PVFS en utilisant le réseau MYRINET [CLRT00].

Cette idée a été poussée à l'extrême dans DAFS (*Direct Access File System* [MAFS03]) où l'interface d'accès aux fichiers a été entièrement calquée sur l'interface des réseaux rapides, en l'occurence VIA (voir en annexe A.2.3 et A.2.2). Les applications peuvent alors utiliser très efficacement le réseau sous-jacent pour accéder aux fichiers distants. Mais elles doivent être entièrement réécrites pour tenir compte des spécificités de l'interface utilisée. Ceci est envisageable pour les bibliothèques spéciales comme MPI-IO mais pose des problèmes pour les applications standard. Par ailleurs, la disparition progressive de VIA a conduit à la création de DAPL, une couche d'accès au réseau dédié à DAFS [DCK+03]. Son interface est en fait très proche de VIA et se compose d'une partie en espace utilisateur (UDAPL) et d'une partie dans le noyau (KDAPL).

Un inconvénient de ces implantations dans une bibliothèque utilisateur est qu'un éventuel cache du côté client sera beaucoup plus difficile à partager entre les différents processus. En effet, avec la généralisation des clusters de SMP (*Symmetric Multi-Processing*), il est courant de placer sur un même nœud plusieurs processus d'une application parallèle, typiquement un processus par processeur. Il est alors intéressant de partager les caches d'accès aux fichiers distants. Dans le cas d'une implantation dans le noyau, ce partage est immédiat. Par contre, dans une implantation en espace utilisateur, il nécessitera des techniques de partage mémoire et communication inter-processus (IPC) qui peuvent se révéler coûteuses.

2.3.1.b Vers un retour dans le noyau

Cette situation a depuis évolué puisque les systèmes d'exploitation proposent désormais des interfaces d'accès aux fichiers beaucoup plus adaptées aux applications parallèles. Tout d'abord, les accès vectoriels ont été ajoutés à l'interface standard par les

appels `readv/writev` (voir l'annexe A.1.1).

Ensuite, les accès zéro-copie ont été introduits pour permettre aux applications de préciser au système d'exploitation qu'il ne doit pas cacher les données d'un fichier (en passant le paramètre `O_DIRECT` à l'ouverture, voir l'annexe A.1.1). La suppression de copie réduit l'utilisation du processeur. Cette fonctionnalité a été introduite pour que les applications gourmandes en mémoire, notamment les bases de données et le calcul *out-of-core* (calcul dont les données ne tiennent pas dans la mémoire physique de la machine) puissent maîtriser leurs accès disque. En effet, lorsque ces applications veulent écrire, elles ne veulent pas être *swappées*[3] du fait de l'allocation d'une page par le système pour cacher les données.

Dans le cadre des applications parallèles sur les grappes, les accès zéro-copie aux fichiers réduisent la consommation processeur en évitant une copie. Par ailleurs, l'absence de cache du côté client évite d'avoir à implanter un protocole complexe pour les synchroniser vis-à-vis du serveur et maintenir la cohérence. Cependant, l'absence de cache impose de contacter le serveur pour chaque lecture ou écriture, même s'il s'agit de ne lire qu'un seul octet qui a déjà été lu peu avant. C'est donc à l'application de choisir intelligemment quels fichiers doivent être cachés ou non.

Enfin, les entrées-sorties asynchrones ont été ajoutées, notamment dans les noyaux LINUX 2.6 avec LINUX AIO [Lah02, BPPM03, BTSM04] (voir en annexe A.1.2). Le cœur des systèmes d'exploitation était déjà naturellement asynchrone depuis longtemps, notamment par le fait que les entrées-sorties notifiées par interruption (*Interrupt-Driven I/O*) sont explicitement asynchrones. Cependant, cette fonctionnalité n'était auparavant pas exposée en espace utilisateur. Seules des primitives bloquantes étaient offertes aux applications. Désormais, avec les accès aux fichiers asynchrones, il devient possible de recouvrir des accès disque (qui sont généralement très lents) par du calcul.

Maintenant que ces fonctionnalités avancées sont disponibles dans les systèmes d'exploitation, les raisons ayant motivé la mise en œuvre d'interfaces spécifiques telles que PVFS ou MPI-IO en espace utilisateur ne sont plus vraiment valables. Par ailleurs, l'utilisation de l'interface standard est généralement plus facile et portable. On observe donc un retour des systèmes de fichiers distribués dans les grappes vers les implantations dans le noyau. C'est notamment le cas de PVFS [CLRT00, Lig01] (qui propose en fait les deux stratégies) et surtout LUSTRE [Clu02, Sch03] (voir en partie 2.1.3.a).

2.3.2 Modélisation des délais d'accès aux fichiers distants

Pour analyser les délais des différentes étapes lors de l'accès aux fichiers distants, nous introduisons les notations suivantes :

[3] Le *swap* consiste à utiliser une partition disque comme extension de la mémoire physique de la machine. Le système déplace des pages vers le disque (*Swap-out*) quand la mémoire physique se remplit, et les ramène (*Swap-in*) quand il doit y accéder.

s	Quantité de données à transférer
B_{Mem}	Bande passante mémoire de l'hôte
T_{Copie}	Temps de copie mémoire dans l'hôte
B_{Net}	Capacité du lien d'accès
L_{Net}	Latence du réseau
B_{Disk}	Bande passante des disques
L_{Disk}	Temps d'accès aux disques
T_{Sto}	Temps logiciel d'accès au gestionnaire de stockage
T_{Reg}	Temps logiciel d'enregistrement mémoire
T_{Async}	Coût logiciel d'utilisation de l'interface asynchrone de programmation réseau
T_{Event}	Coût logiciel de coordination des événements disque et réseau

Les coûts logiciels pouvant être différents côté client et serveur, nous utilisons le suffixe C ou S pour les distinguer.

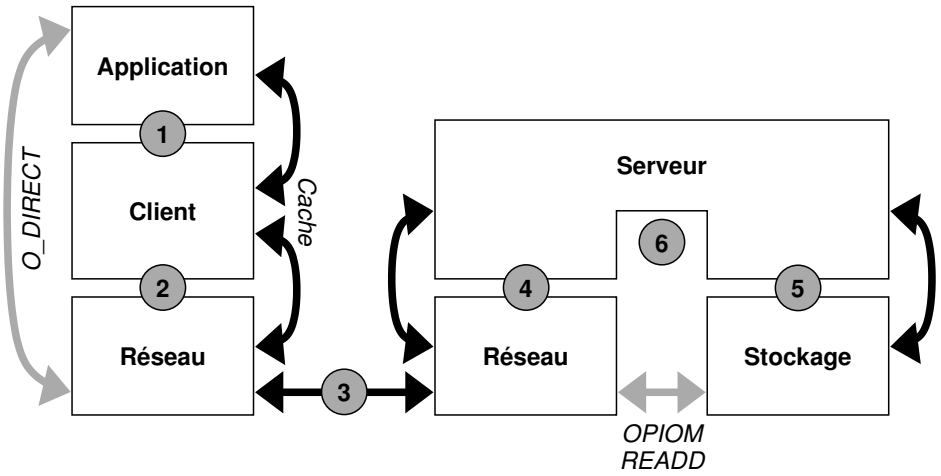


FIG. 2.15: *Différentes étapes d'accès aux fichiers distants. Les différentes interactions logicielles sont numérotées de 1 à 6. Les flèches noires désignent le chemin habituel des données. Les flèches grises présentent des optimisations des transferts de données.*

La figure 2.15 illustre les différentes étapes d'accès aux fichiers distants. Le délai total T_{Total} d'accès aux fichiers distants se décompose en trois termes correspondant au client, réseau et serveur (équation 2.1). Le délai réseau T_{Reseau} décrit par l'équation 2.3 correspond ici à l'étape (3), c'est-à-dire au délai matériel d'un aller-retour réseau accompagné du transfert de données dans un des deux sens.

Coût de l'accès aux fichiers distants :

$$T_{Total} = T_{Client} + T_{Reseau} + T_{Serveur} \quad (2.1)$$

$$T_{Client} = T_{StoC} + T_{CopieC} + T_{RegC} + T_{AsyncC} \quad (2.2)$$

$$T_{Reseau} = 2 \times L_{Net} + s/B_{Net} \quad (2.3)$$

$$T_{Serveur} = T_{RegS} + T_{AsyncS} + T_{EventS} + T_{StoS} + T_{CopieS} + L_{Disk} + s/B_{Disk} \quad (2.4)$$

Nous avons mesuré à titre indicatif les ordres de grandeur de différents termes sur nos machines[4]. La bande passante mémoire B_{Mem} de l'hôte atteint aujourd'hui 2 Go/s, ce qui permet de copier $s = 64$ ko en $T_{Copie} = 32$ μs. La bande passante réseau B_{Net} atteint plusieurs centaines de megaoctets par seconde tandis que la latence L_{Net} est de quelques microsecondes. Les accès disque restent très lents puisque le déplacement mécanique de la tête de lecture impose un délai L_{Disk} de plusieurs millisecondes. Le débit B_{Disk} est ensuite de plusieurs dizaines de megaoctets par secondes, qui peuvent être agrégés par des technologies du type RAID. Cependant, ces accès physiques sont masqués par les mécanismes de cache dans le système d'exploitation du serveur, ils ne sont pas donc importants ici.

Les coûts logiciels varient selon la mise en œuvre du système, notamment son implantation dans le noyau ou en espace utilisateur, et l'interface de programmation qu'elle utilise. Ils peuvent par exemple impliquer des appels-système (400 ns) ou des copies mémoire. Contrairement aux coûts précédents qui sont limités par les performances du matériel, ces coûts logiciels permettent d'envisager un certain nombre d'optimisations. Nous les détaillons donc maintenant.

Les délais du côté client correspondent aux étapes (1) et (2). Il s'agit donc d'une part du coût de l'accès au client de stockage T_{StoC} et d'autre part des coûts d'utilisation du réseau par ce client, T_{RegC} et T_{AsyncC}.

L'interaction (1) entre l'application et le client d'accès aux fichiers distants a été abondamment étudiée. Nous ne nous attacherons donc pas à l'optimiser. Des travaux ont mené à l'amélioration de l'interface de programmation pour satisfaire les besoins des applications parallèles, avec notamment des accès parallèles, collectifs, vectoriels et/ou asynchrones dans MPI-IO, ou des fonctionnalités dédiées à une meilleure utilisation du réseau sous-jacent dans DAFS. Son coût T_{StoC} dépend de l'implantation du client en espace noyau ou utilisateur. Dans le premier cas, le coût reste assez faible. Dans le second, un appel système est nécessaire ainsi que la traversée des couches supérieures du système d'exploitation. Par ailleurs, une copie mémoire dans le cache du client est souvent nécessaire (T_{CopieC}). Nous avons mesuré un surcoût d'environ 15 μs dans ce dernier cas.

L'interaction (2) entre le client de stockage distribué et le réseau peut poser problème du fait de la spécialisation de ce dernier pour les communications dans les applications parallèles. Le coût est fonction d'une part de l'enregistrement mémoire T_{RegC}

[4] Les mesures de délai ont été réalisées sur les machines équipées de 2 processeurs PENTIUM 4 XEON cadencés à 2,6 GHz.

(préparation des zones mémoire ou copie dans une zone intermédiaire) et d'autre part de l'utilisation efficace du modèle de programmation asynchrone T_{AsyncC} (soumission de requête, recouvrement calcul-communication puis notification). Ces deux coûts sont étroitement lié à la capacité de l'application d'utiliser finement le modèle de programmation sous-jacent. Mais ces coûts sont également fonction de l'implantation en espace utilisateur ou noyau. Nous nous y intéressons tout particulièrement dans notre étude.

Les accès à la carte d'interface réseau ou les copies intermédiaires qui peuvent être mis en jeu dans T_{Reg} imposent un coût pouvant atteindre une centaine de microsecondes. Par contre, ce coût pourra être moindre si l'application a été conçue pour enregistrer les zones mémoire utilisées dans les communications et optimiser son organisation mémoire.

Les accès du type `O_DIRECT` qui court-circuitent le cache du client (flèche grise de gauche) obligent l'interface de programmation du réseau à pouvoir manipuler directement des données de l'application à travers le client. Ils permettent donc de supprimer T_{CopieC} même avec un client dans le noyau mais risquent d'augmenter T_{RegC}.

Le coût d'utilisation de l'interface asynchrone de programmation (T_{Async}) correspond par exemple à des attentes actives lors de l'attente d'évenements ou la gestion de files d'événements, ce qui peut représenter quelques centaines de nanosecondes ou quelques microsecondes.

Les délais du côté serveur correspondent aux étapes (4), (5) et (6), c'est-à-dire le coût d'utilisation du réseau, d'accès au système de stockage et de leur interaction.

L'accès au stockage (5) nécessite de traverser les couches basses du système d'exploitation (T_{StoS}). L'accès physique aux périphériques de stockage ($L_{Disk} + s/B_{Disk}$) a un coût très supérieur à tous les autres mais il est masqué car le serveur met en place un système de cache avec des accès aux disques en tâche de fond. Si le processus serveur est implanté dans le noyau, le coût T_{StoS} d'accès au stockage sera alors très faible. Par contre, s'il est mis en œuvre en espace utilisateur, il lui faudra traverser les couches supérieures du système d'exploitation pour accéder aux données. Le coût est alors d'environ 20 μs sur nos machines auquel s'ajoute une copie mémoire (T_{CopieS}). Nous ne nous attacherons pas à optimiser ces délais qui ont déjà été très largement étudiés.

L'interaction (4) entre le réseau et le processus serveur de stockage présente des caractéristiques proches de celles de (1). Là aussi, le coût d'accès à la mémoire T_{RegS} et l'utilisation du modèle asynchrone T_{AsyncS} vont varier suivant l'implantation en espace utilisateur ou noyau.

Enfin, l'utilisation commune (6) du réseau et du stockage est un problème bien connu puisque de nombreux travaux ont été consacrés aux différents types de serveurs. Par contre, l'utilisation d'un réseau haute performance dans ce cadre peut poser problème puisque l'interface de programmation est très différente, notamment en ce qui concerne la notification d'événement, ce qui peut nuire à l'efficacité de l'application. En effet, les stratégies de notification dans ces réseaux sont très particulières et peuvent ne pas s'intégrer finement aux stratégies standard utilisées pour l'accès aux périphériques de stockage (voir en partie 6.2.2). Le coût T_{EventS} de cette interaction est une des grandes

inconnues ici. Il peut se traduire par des attentes actives ou la gestion de file d'attente, ce qui peut représenter quelques centaines de nanosecondes ou quelques microsecondes.

Les transferts directs entre réseau et dispositifs de stockage (flèche grise de droite) ont été beaucoup étudiés, notamment dans OPIOM et READ[2]. Ils permettent de supprimer T_{CopieS} mais augmentent beaucoup T_{RegS} puisqu'il faudra ici être capable de transférer les données depuis la carte réseau vers les disques physiques.

Nous détaillons maintenant ces interactions concrètes entre stockage et réseau en revenant sur les différentes méthodes d'accès vues en partie 2.1.1 en insistant tout particulièrement sur les coûts d'enregistrement mémoire T_{RegC} et T_{RegS} car ils varient beaucoup selon l'implantation du client et du serveur.

2.3.2.a Accès en espace utilisateur

La mise en œuvre de l'accès aux fichiers distants dans une bibliothèque client en espace utilisateur repose sur, d'une part, des transfert de grandes quantités de données entre les zones mémoire des processus clients et le serveur (dans le cas d'appels de type `read/write`) , et d'autre part, des échanges de métadonnées entre la bibliothèque et le serveur (pour les appels de type `stat` ou `readdir` par exemple). Le coût d'accès au stockage T_{StoC} est dans ce cas très faible puisqu'il s'agit d'un simple appel de fonction (équation 2.5).

Coût dans un client en espace utilisateur :

$$T_{Client} = T_{RegC} + T_{AsyncC} \tag{2.5}$$

Les réseaux rapides peuvent apporter un gain indéniable pour transférer les données puisque les protocoles zéro-copie permettent d'obtenir une très grande bande passante avec un très faible coût processeur dans l'hôte. Cependant, comme nous l'avons vu en partie 2.2.3.b, ces mécanismes nécessitent souvent d'enregistrer la mémoire préalablement, ce qui n'est jamais fait dans une application normale.

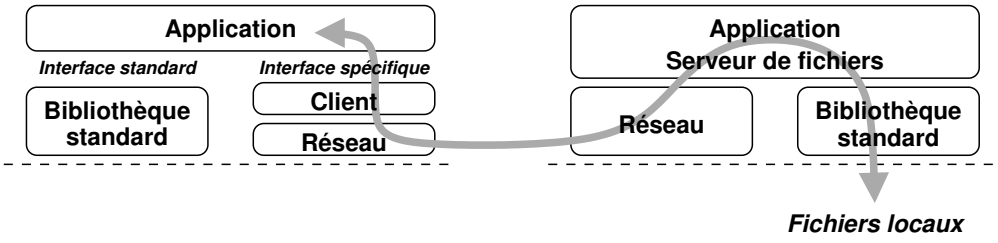


FIG. 2.16: *Accès aux fichiers distants en espace utilisateur par une interface de programmation spécifique.*

Interface spécifique L'utilisation d'une interface de programmation spécifique telle que PVFS ou MPI-IO pour accéder aux fichiers distants impose d'écrire l'application de façon adaptée (voir la figure 2.16). Il est donc envisageable ici de rendre l'interface encore plus spécifique en ajoutant des primitives pour réaliser l'enregistrement mémoire. L'application devra donc enregistrer préalablement toutes les zones mémoire impliquées dans les accès aux fichiers distants. C'est la stratégie proposée par DAFS (voir en annexe A.2.3 et [MAF+02]). Le coût T_{RegC} (voire également T_{AsyncC}) est alors entièrement reporté dans l'application.

La plupart des bibliothèques d'accès aux fichiers distants existantes ne poussent pas la spécificité de l'interface de programmation aussi loin. L'enregistrement mémoire est alors assuré à la volée (de manière transparente) par la bibliothèque, par exemple dans l'implantation de PVFS en espace utilisateur. C'est dans ce cas que le coût T_{RegC} pourra devenir important.

Le transfert des métadonnées entre le serveur et la bibliothèque cliente va se caractériser par des messages de taille assez faible, qui peuvent donc bénéficier de la latence très faible proposée par les réseaux des grappes. L'enregistrement mémoire ne sera pas un problème dans le cas des messages de faible taille. Il est préférable de faire une copie intermédiaire dans une zone pré-enregistrée.

Surcharge de l'interface standard L'utilisation d'une interface spécifique de programmation a l'avantage de mieux répondre aux besoins des applications parallèles, mais a l'inconvénient d'imposer de ré-écrire les applications pour se conformer à cette interface. Certains projets préfèrent donc *surcharger* l'interface des bibliothèques standard. (voir la figure 2.17). C'est par exemple ce que proposaient les premières versions de PVFS [CLRT00].

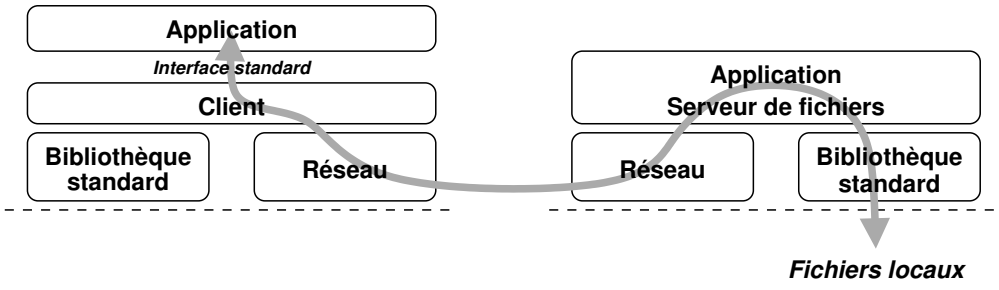


FIG. 2.17: *Accès aux fichiers distants en espace utilisateur en surchargeant l'interface standard.*

L'idée est alors de placer la bibliothèque client entre les applications et la bibliothèque standard. Son rôle est de rediriger les accès aux fichiers locaux vers la bibliothèque standard tandis que les accès aux fichiers distants sont envoyés sur le réseau. Les échanges sur le réseau sont alors identiques au cas précédent. Cependant, l'enregistrement mémoire est ici forcément transparent puisque seule l'interface standard est utilisée par les applications.

2.3.2.b Maintien d'un cache du côté client

Les deux implantations que nous venons de décrire utilisent des transferts directs entre l'application et le serveur de données distantes. Certains systèmes préfèrent maintenir un cache dans le client de façon à éviter d'avoir à contacter le serveur distant pour chaque requête de l'application.

Le maintien d'un cache de métadonnées, notamment les attributs des fichiers ou le contenu des répertoires ne change pas vraiment l'utilisation du réseau. La quantité de messages échangés entre le client et le serveur va certes diminuer, mais leur taille restera assez faible. L'utilisation de copies intermédiaires au lieu de l'enregistrement mémoire reste donc préférable.

Par contre, les messages transportant les données réelles des fichiers diffèrent en présence d'un cache puisqu'au lieu d'être déposées directement dans les zones mémoire de l'application, les données seront déposées dans la bibliothèque client, là où le cache est maintenu. L'enregistrement mémoire n'est donc alors nécessaire que pour des zones mémoire internes à la bibliothèque client, qui peut par conséquent faire le nécessaire pour utiliser des zones pré-enregistrées. Le coût T_{RegC} peut alors être moindre. Les transferts réels vers l'application se font ensuite par simple copie entre ses zones mémoire et le cache de la bibliothèque (comme dans n'importe quel système de fichiers distribués maintenant un cache du côté client).

Coût dans un client en espace utilisateur avec cache :

$$T_{Client} = T_{CopieC} + T_{RegC} + T_{AsyncC} \quad (2.6)$$

2.3.2.c Mise en œuvre du serveur dans le noyau

Les modèles d'accès en espace utilisateur reposent généralement sur un processus serveur implanté en espace utilisateur sur la (ou les) machine distante. Ce processus se charge de transférer les données entre les clients et les disques locaux. La stratégie standard consiste à stocker temporairement ces données dans l'espace mémoire du processus, dans des zones préalablement enregistrées.

Cette stratégie a l'inconvénient de nécessiter une copie entre l'espace utilisateur et le système d'exploitation (T_{CopieS}), ce qui réduit la puissance du serveur. Ce problème a également été rencontré dans le cadre des serveurs WWW haute performance, qui doivent lire des pages HTML sur le disque et les envoyer sur le réseau. Il a notamment été proposé de *mapper* (projeter) les fichiers dans le processus serveur pour les envoyer sans copie. Une solution plus attrayante consiste à utiliser l'appel-système `sendfile` qui envoie directement des données stockées dans le cache de fichiers du système d'exploitation [NBK99]. Ainsi, la traditionnelle copie depuis le cache vers l'application avant l'envoi sur le réseau est évitée. Cependant, il n'est pas clair que ces solutions sont applicables aux serveurs de fichiers, notamment car les accès y sont en lecture et écriture, alors qu'elles ont été conçues pour les serveurs WWW où les accès sont généralement en lecture seule (voir en partie 4.2.6).

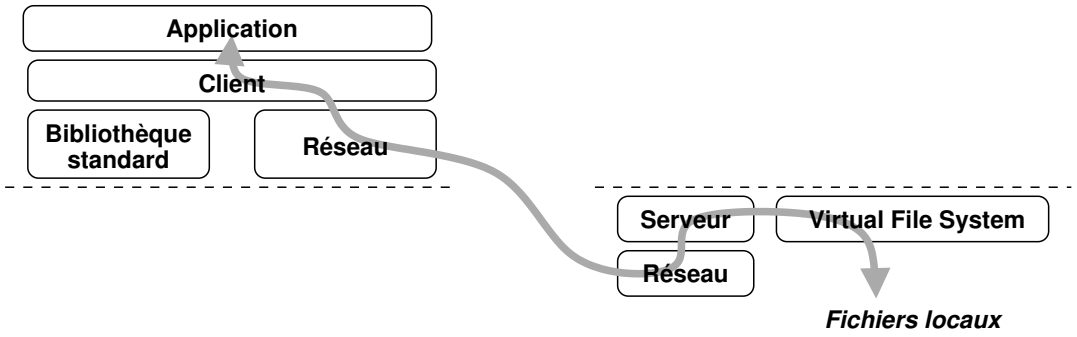


FIG. 2.18: *Accès en espace utilisateur vers un serveur noyau pour réduire les copies intermédiaires.*

Une autre solution a été proposée pour supprimer cette copie de données entre le processus serveur et le système d'exploitation. Il s'agit de déplacer ce serveur dans le système. Cette méthode était utilisée pour obtenir des serveurs WWW performants avant l'avènement de l'appel-système `sendfile` [JKN+01]. Dans le contexte des serveurs de fichiers, c'est notamment ce que DAFS propose [Mag02a].

Coût dans un serveur en espace noyau :

$$T_{Serveur} = T_{RegS} + T_{AsyncS} + T_{EventS} + L_{Disk} + s/B_{Disk} \quad (2.7)$$

Ce modèle, décrit sur la figure 2.18 permet par ailleurs d'éviter le coût de la traversée des couches système T_{StoS} et de la copie mémoire T_{CopieS} (équation 2.7). Il impose cependant deux contraintes importantes.

Tout d'abord, l'utilisation du réseau haute performance dans une application interne au noyau impose la disponibilité d'une interface de programmation de ce réseau dans le noyau. De plus, cette interface doit être compatible avec l'interface utilisateur puisqu'elle devra dialoguer avec un client en espace utilisateur. C'est par exemple impossible avec QUADRICS QSNET puisque l'interface noyau KCOMM (*Kernel Communications*) est basée sur des services et des RPC tandis que les interfaces utilisateur ELANLIB et TPORTS sont basées respectivement sur du RDMA et du passage de messages (voir en annexe A.3.4).

Ensuite, les zones mémoire manipulées du côté serveur sont en fait constituées des pages du système où les données sont cachées. Les caractéristiques très spéciales de ces pages (notamment le fait qu'elles n'ont généralement pas d'adresse virtuelle) peuvent les rendre difficile à utiliser avec l'interface de programmation du réseau (voir en partie 5.2.3.a), le coût T_{RegS} étant alors très important.

2.3.2.d Accès par un système de fichiers dans le noyau

Comme nous l'avons vu en partie 2.1.1, la stratégie la plus courante pour accéder aux fichiers distants consiste à implanter le client dans le système d'exploitation. Ainsi,

les applications accèdent aux fichiers distants comme à n'importe quel fichier. C'est une couche de *Virtualisation* du système d'exploitation qui se charge de transmettre les requêtes au client, qui va ensuite les envoyer au serveur.

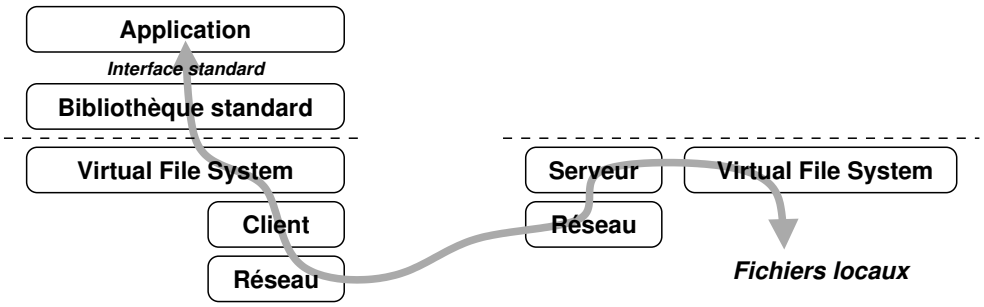


FIG. 2.19: *Accès par un système de fichiers implanté dans le noyau du client.*

Ce modèle, représenté sur la figure 2.19, est de nouveau utilisé dans le cadre des applications parallèles car les systèmes d'exploitation supportent désormais les accès vectoriels, asynchrones et zéro-copie. La mise en œuvre en espace utilisateur n'a donc plus l'avantage de fournir une interface de programmation plus adaptée que celle du système d'exploitation. C'est la raison pour laquelle les systèmes de stockage distribué tels que PVFS et LUSTRE sont maintenant souvent utilisés dans le noyau.

Les communications mises en jeu dans ce modèle vont être initiées depuis le système d'exploitation du client. Il faut là aussi que l'interface de programmation du réseau supporte ce type de communication. Dans le cas des métadonnées, il s'agira ici de transférer des messages de petites tailles vers des zones mémoire du système d'exploitation distant. Les copies étant envisageables pour ce type de message, aucun problème technique lié aux traductions d'adresse n'apparaît.

Par contre, les échanges de données réelles présentent plus de difficultés. Dans le cas où l'application a demandé que les fichiers qu'elle manipule ne soient pas cachés du côté client (par le paramètre `O_DIRECT`), les données doivent être échangées directement entre l'espace mémoire de l'application et le serveur distant. Ce type de transfert est conceptuellement très proche des communications zéro-copie traditionnelles sur un réseau haute performance, à ceci près qu'ici ils sont initiés dans le contexte du système d'exploitation, et non dans celui de l'application elle-même. C'est une contrainte forte sur l'interface de programmation du réseau que de supporter ce type de communication puisque elle n'a généralement pas été prévue pour genre de zone mémoire (voir en partie 5.2.3.a). Le coût de ces accès est détaillé par l'équation 2.8. Les transferts zéro-copie entre l'application et le réseau permettent d'éviter le coût de la copie T_{CopieC}.

Coût dans un client zéro-copie en espace noyau :

$$T_{Client} = T_{StoC} + T_{RegC} + T_{AsyncC} \quad (2.8)$$

Si l'application ne demande pas d'accès zéro-copie, les transferts de données sont effectués entre le cache de fichiers du système d'exploitation et le serveur. Comme nous venons de le voir dans le cas d'un serveur implanté dans le noyau, les pages constituant ce cache de fichiers sont très spéciales. Le support de communication impliquant ce genre de pages est une autre contrainte forte sur l'interface de programmation du réseau. Le coût des accès avec copie est détaillé par l'équation 2.9.

Coût dans un client en espace noyau avec cache :

$$T_{Client} = T_{StoC} + T_{CopieC} + T_{RegC} + T_{AsyncC} \quad (2.9)$$

Pour ces deux types d'accès, le coût T_{RegC} peut être important.

2.3.2.e Accès par bloc

La dernière méthode d'accès aux fichiers distants (*Network Block Device*) consiste à placer le client au niveau des périphériques bloc et à manipuler une partition ou un disque distant (voir la figure 2.20). Les données transférées sont alors exclusivement des blocs. Des travaux ont été réalisés du côté serveur de ce modèle, en particulier dans OPIOM et READ[2] (voir en 2.1.1.e), mais très peu du côté client.

Ce modèle est utilisé dans ISCSI où les couches protocolaires de TCP sont insérées entre les différentes couches logicielles d'accès aux périphériques de stockage SCSI. Il s'agit dans ce cas de réseaux traditionnels (ETHERNET) éventuellement longue distance, pour lesquels nos objectifs de performances ont peu d'intérêt. Il est également mis en œuvre dans les systèmes de stockage partagé sans serveur tels que GFS ou GPFS (voir en partie 2.1.3.b). Mais dans ce cas, l'accès au stockage distant utilise un réseau dédié au stockage, notamment FIBRE CHANNEL. Le problème de l'utilisation du réseau haute performance pour l'accès au stockage n'intervient pas.

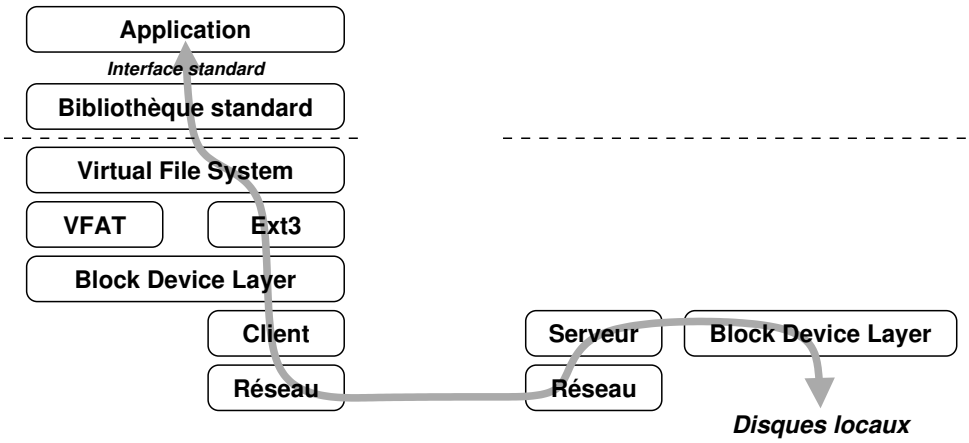


FIG. 2.20: *Accès en mode bloc par montage d'un périphérique distant.*

Dans le modèle général du *Network Block Device*, les communications impliquent des zones mémoire similaires au cas précédent (équation 2.9). Les blocs sont cachés dans le système d'exploitation du client de la même façon que les pages des fichiers le sont (les deux caches sont d'ailleurs fusionnés dans le *Page-Cache* du noyau LINUX). Les contraintes sur l'interface de programmation du réseau sont les mêmes, elle doit pouvoir manipuler ces zones mémoire spéciales. Du côté serveur, les contraintes sont similaires.

Le serveur peut en fait facilement être implanté en espace utilisateur. Il est en effet possible d'accéder directement aux blocs physiques d'un périphérique dans une application si on ouvre ce périphérique (par exemple `/dev/hdb1` pour la première partition du second disque IDE) au lieu d'ouvrir les fichiers qu'il contient. Dans ce cas, les communications se font entre un client dans le système d'exploitation et un serveur en espace utilisateur, ce qui impose à l'interface de programmation du réseau d'être similaire dans ces deux mondes (comme dans le cas d'un client utilisateur et d'un serveur noyau).

2.3.3 Synthèse des interactions entre stockage et réseau

Nous avons présenté en partie 2.1 les différentes techniques utilisées pour mettre en œuvre un stockage distribué performant. L'utilisation de techniques du type cache et réplication permet d'améliorer les performances mais ce sont surtout les stratégies de parallélisation du serveur qui ont permis d'assurer un support efficace des besoins des applications parallèles dans les grappes de calcul. Le développement conjoint des réseaux haute performance dans les grappes a permis des communications très performantes entre les nœuds des applications parallèles mais cette spécialisation a rendu difficile leur utilisation pour le stockage distribué.

Comme nous venons de le voir, les besoins de l'accès distant varient beaucoup selon l'implantation du client et du serveur. L'interaction entre les couches logicielles du stockage et du réseau haute performance présente un certain nombre de difficultés.

L'utilisation de l'interface standard d'accès aux fichiers sans cache du côté client contraint à enregistrer à la volée les zones mémoire utilisées par l'application. Par contre, l'utilisation d'une interface spécifique permet de demander à l'application de préparer ces zones. La présence d'un cache du côté client permet de limiter les contraintes de l'enregistrement mémoire aux zones du cache dans le client. Dans le cas d'une implantation dans le noyau, que ce soit du côté client ou serveur, les zones mémoire manipulées sont très différentes des habituels segments de mémoire en espace utilisateur. Les pages du cache du système ont des caractéristiques très différentes et ne correspondent pas forcément aux hypothèses de l'enregistrement mémoire (voir en partie 5.2.3.a).

Les coûts d'utilisation de l'interface asynchrone de programmation, que ce soit seule (T_{Async}) ou en interaction avec les entrées-sorties de stockage (T_{Event}), varient en revanche peu avec le type d'implantation. Nous nous attendons à ce que ces coûts atteignent quelques centaines de nanosecondes ou quelques microsecondes au maximum.

Coût	Valeur actuelle	Valeur envisageable
T_{Sto}	Jusqu'à 20 μs, fixé par l'implantation	
T_{Copie}	$\simeq$ 32 μs pour 64 ko de données ou 0 selon l'implantation	
T_{Reg}	1-100 μs	faible si utilisation efficace
T_{Async}	100 ns-1 μs	très faible si utilisation efficace
T_{Event}	100 ns-1 μs	très faible si utilisation efficace

TAB. 2.4: *Synthèse des différents coûts logiciels lors de l'accès aux fichiers distants et des gains envisageables.*

La table 2.4 synthétise les différents coûts logiciels de l'accès au fichiers distants. Nous avons montré que les coûts d'accès au stockage ou certaines copies mémoire sur le chemin d'accès aux fichiers distants étaient liés à la conception du client ou du serveur, en particulier la présence d'un cache et l'implantation en espace utilisateur ou dans le noyau. Par contre, les coûts d'utilisation du réseau, que ce soit l'enregistrement mémoire T_{Reg}, l'utilisation de l'interface asynchrone de programmation T_{Async} et l'interaction des stratégies de notification du réseau et des disques T_{Event} restent présents dans tous les modèles, parfois avec des impacts différents. C'est à ces problèmes que nous nous intéressons.

Chapitre 3

Problématique et démarche

3.1 Cadre de l'étude

Dans l'état de l'art, nous avons identifié les principaux problèmes du stockage distribué. Les travaux sur le cache du côté client ont permis de réduire la nécessité de contacter le serveur distant pour chaque accès. Les protocoles sans état dans le serveur ont ensuite permis au client de travailler de manière complètement déconnectée, même en cas de panne du serveur (partie 2.1.2.a).

Dans le contexte des grappes de calcul où les besoins des applications parallèles sont très grands, les travaux se sont concentrés autour de la suppression du goulot d'étranglement que représentait le serveur lors de l'accès aux fichiers par des centaines de clients gourmands. Les stratégies basées sur la parallélisation du serveur (voire sa suppression dans les systèmes à stockage partagé) ont permis de supprimer ce goulot d'étranglement (partie 2.1.3).

Par ailleurs, l'utilisation du réseau haute performance permet de démultiplier le débit utile des serveurs. Ainsi, remplacer un réseau FAST ETHERNET par MYRINET permet de mutiplier les performances de PVFS par 3 [CLRT00]. L'apport des réseaux haute performance est donc indéniable et permet d'envisager une amélioration significative des performances de l'accès aux fichiers distants. Pourtant, cet apport est bien inférieur au gain en bande passante proposé par ce réseau (plus d'1 Gbit/s pour MYRINET contre 100 Mbit/s pour FAST ETHERNET). C'est à cette sous-utilisation du réseau haute performance dans le cadre du stockage distribué que nous nous intéressons.

Il faut d'ailleurs noter que l'implantation de PVFS2 dans le noyau a la particularité de retourner en espace utilisateur avant d'accéder au réseau. Cette stratégie est justifiée par l'**absence potentielle de support suffisant pour les accès au réseau depuis le noyau** : "*it is not clear that we will have ready access to all networking APIs from within the kernel*" [PVF03]. De la même façon, LUSTRE [Sch03] n'est plus disponible sur les réseaux MYRINET. Ceci révèle les difficultés que l'on peut rencontrer lorsqu'on cherche à utiliser le réseau haute performance pour l'accès aux fichiers distants.

Nous allons présenter un travail visant à utiliser efficacement le réseau sous-jacent pour améliorer les performances du stockage afin de répondre aux besoins des applications parallèles. Il s'agit d'une stratégie complémentaire aux traditionnels mécanismes

de parallélisation du serveur. L'idée est de fournir un support efficace dans l'interface de programmation du réseau haute performance pour les besoins du stockage distribué.

Ce travail est basé sur des développements logiciels que nous avons réalisés et des études expérimentales détaillées afin de mettre en évidence les difficultés d'utilisation du réseau dans ce contexte puis de proposer des solutions. Il ne s'agit pas ici d'écrire un nouveau système de stockage distribué. Nous avons concentré nos travaux autour de l'**optimisation des transferts point-à-point** entre un client et un serveur utilisant les réseaux rapides des grappes car l'analyse des performances des systèmes actuels tels que PVFS montre que le réseau est sous-utilisé. Nos optimisations se placent **à l'interface entre ces systèmes et la couche logicielle d'accès au réseau** et concernent d'une part les transferts de données et d'autre part le contrôle des communications. Il s'agit donc de faire en sorte que le réseau puisse être très efficacement utilisé dans ce cadre pour ensuite permettre aux systèmes existants de bénéficier des optimisations que nous proposons.

Nous nous intéressons spécifiquement aux **modèles organisés autour d'un serveur** (ou plusieurs), par exemple PVFS ou LUSTRE (voir en partie 2.1.3.a) qui sont les plus répandus et les plus ouverts car développés dans le monde académique.

L'idée est d'étudier les **différents types d'accès distant**, au niveau utilisateur ou noyau, en respectant l'**interface standard** d'accès aux fichiers. En effet, les modifications d'interface de programmation ont été étudiées dans MPI-IO et surtout dans DAFS où il a été montré que cela permettait une utilisation efficace du réseau [WP02, MAFS03]. Nous nous concentrons sur l'interface standard, ce qui va permettre à n'importe quelle application de tirer profit de nos travaux et ainsi d'assurer la portabilité des performances. Par ailleurs, la disponibilité récente des primitives d'accès vectoriels, zéro-copie et asynchrones dans l'interface standard doit permettre de répondre efficacement aux besoins des applications parallèles (voir en 2.3.1.b).

Le protocole de haut niveau doit être adapté à notre contexte, c'est-à-dire des grappes constituées de machines homogènes. Il est donc inutile ici de traverser les couches du type XDR (*External Data Representation*) pour assurer la portabilité du protocole comme dans NFS. De plus, ces machines étant reliées par un réseau fiable, les couches logicielles du type RPC (*Remote Procedure Call*) utilisées dans NFS pour mettre en place un protocole fiable semblent également inutiles. Même si leurs coûts n'est plus très élevé sur un processeur cadencé à plusieurs gigahertz, ces étapes sont inutiles dans notre contexte.

Les réseaux MYRINET sont utilisés, via leur interface de programmation GM, d'une part puisqu'ils sont les plus répandus du marché et d'autre part parce que leur facilité de reprogrammation a permis de nombreux développements logiciels. Les technologies réseau plus modernes, en particulier INFINIBAND, sont cependant également très utilisées de nos jours puisque leur modèle de programmation n'a pas encore été définitivement fixé et permet donc des modifications aisées. Ce serait donc une plateforme de développement intéressante. Mais l'utilisation d'une technologie matûre comme MYRINET et de son modèle de programmation orienté *Rendez-vous* permettra ici de mettre plus clairement en évidence les problèmes d'interaction entre une couche réseau conçue pour les

communications entre nœuds d'une application parallèle et les couches système d'accès au stockage.

L'objectif majeur de notre étude est de voir :

Dans quelle mesure les réseaux haute performance peuvent-ils être utilisés efficacement dans un système de stockage distribué de type client-serveur où les applications manipulent les données par l'intermédiaire de l'interface standard de programmation des accès aux fichiers ?

3.2 Problématique

Pour étudier ce problème général, plusieurs sous-questions doivent être examinées :

Q1 – En quoi les performances des réseaux des grappes peuvent-ils améliorer l'accès aux fichiers distants ?

Tout d'abord, il convient de se demander quel peut être l'apport réel des réseaux des grappes pour les performances de l'accès aux fichiers distants. Leurs principales caractéristiques sont leurs **très faible latence** (quelques microsecondes) et **très grande bande passante** (plusieurs centaines de megaoctets par seconde). Dans quelle mesure ces performances sont-elles intéressantes pour l'accès aux fichiers ? Cela peut dépendre du type de message :

Métadonnées : La faible latence peut être très intéressante pour ces requêtes. Elle peut permettre d'envisager une remise en cause de la nécessité du cache du côté client si le temps d'accès au serveur distant est proche du temps d'accès local.

Données : La grande bande passante devrait avoir un intérêt pour les transferts de grandes quantités de données. La faible latence peut être intéressante pour les petites (comme dans le cas des métadonnées).

Selon l'architecture globale du système, on peut avoir uniquement des échanges entre clients et serveurs, mais aussi entre différents serveurs (dans les systèmes parallèles, en particulier si les données sont répliquées, comme dans PVFS ou LUSTRE) ou entre les clients (dans un modèle à stockage partagé sans serveur, dans GFS ou GPFS).

Cette question est étudiée au chapitre 4.

Q2 – Comment les mécanismes des réseaux haute performance peuvent-ils améliorer le transfert de données lors de l'accès aux fichiers distants ?

Les interfaces de programmation des réseaux des grappes ont été conçues pour répondre aux besoins en communication dans les applications parallèles. Les grandes performances de ces réseaux sont notamment dues à la suppression de copie sur le chemin critique entre deux applications qui communiquent. Il peut être intéressant de regarder si le même genre d'optimisation est possible pour l'accès aux fichiers distants.

Cependant, nous avons vu que les besoins en transferts de données dans les systèmes de stockage sont différents de ceux des communications dans une application parallèle. Par ailleurs, ils diffèrent beaucoup selon que la mise en œuvre du client ou du serveur est faite en espace utilisateur ou dans le noyau, avec cache ou sans.

Du côté client, il s'agira donc de permettre des transferts zéro-copie entre l'application (ou le cache) et le réseau. Du côté serveur, il peut s'agir de supprimer une copie intermédiaire en utilisant des stratégies similaires à `sendfile` sur TCP qui envoie directement les données depuis l'espace mémoire du système d'exploitation (voir en 2.3.2.c), ou encore des mécanismes liés à la mémoire virtuelle, notamment la projection des fichiers dans l'espace d'adressage du processus serveur.

Il convient donc de se demander dans quelle mesure les interfaces de programmation très spécifiques de ces réseaux, et notamment l'enregistrement mémoire peuvent s'interfacer efficacement avec ces besoins. Ce problème se révèle notamment important pour les implantations dans le noyau car, d'une part les interfaces de programmation des réseaux n'ont pas été conçues pour ce contexte, et d'autre part les problèmes d'adressage mémoire y sont plus nombreux (voir en 2.3.2).

Cette question est étudiée au chapitre 5.

Q3 – Le trafic généré par l'accès distant aux fichiers est-il bien géré par le contrôle des communications des réseaux des grappes ?

Les applications parallèles répartissent les calculs sur l'ensemble de la grappe. Cette répartition homogène de la charge dans la grappe conduit naturellement à une répartition assez homogène du trafic sur l'ensemble du réseau. On évite par exemple de centraliser certains éléments car cela pourrait engendrer un goulot d'étranglement en terme de charge de la machine et de son lien d'accès.

La gestion des communications dans les réseaux des grappes a été conçue pour des utilisations de ce type. Les systèmes de stockage partagé sans serveur (comme GFS [SRO96] ou GPFS [SH02]) permettent de conserver un trafic du même type puisque la gestion globale de l'accès au stockage n'est pas du tout centralisée. Par contre, les systèmes utilisant un ou plusieurs serveurs, notamment PVFS [CLRT00] et LUSTRE [Sch03], conduisent indéniablement à une concentration du trafic autour de ces nœuds spéciaux. On peut donc se demander dans quelle mesure ce trafic moins bien réparti peut être supporté par les réseaux des grappes et leurs éventuels protocoles de contrôle.

Cette question est étudiée en partie 6.1.

Q4 – Peut-on mixer les notifications d'événements des réseaux rapides avec les événements traditionnels du stockage ?

La mise en œuvre d'un serveur de stockage impose de convertir les requêtes provenant des clients sur le réseau en requêtes pour les disques locaux. De nombreux travaux ont été effectués dans le cadre des réseaux traditionnels (voir en partie 2.2.3.c). Il est désormais possible de manipuler par la même interface les événements générés par les connexions réseau et les notifications de terminaison d'accès asynchrones aux fichiers.

Sans une telle uniformisation des interfaces, les stratégies distinctes d'attente d'événements issus des disques et des réseaux traditionnels rendraient les serveurs beaucoup moins performants. Il est donc important de voir dans quelle mesure les modèles asynchrones proposés dans les réseaux des grappes peuvent eux aussi s'interfacer avec les méthodes traditionnelles d'attente d'événements, en particulier `poll/select` (voir en 2.2.1.a).

Il peut également être intéressant d'étudier l'utilisation de ces stratégies asynchrones du côté client où les notifications nécessaires varient selon le type d'accès aux fichiers demandés par l'application.

Cette question est étudiée en partie 6.2.

3.3 Démarche

Notre étude commence par des mesures simples de l'impact des réseaux haute performance sur les différents types d'accès distant. Il s'agit de mettre en évidence quelques points clés pour, d'une part, quantifier l'apport réel des performances du réseau, et d'autre part, distinguer les différents problèmes, que ce soit au niveau des performances ou de la mise en œuvre elle-même (Q1).

Nous utilisons un protocole expérimental très simple nommé ORFA (*Optimized Remote File-system Access*) pour étudier la connexion point-à-point entre un client mis en œuvre en espace utilisateur et respectant l'interface standard d'accès aux fichiers et un serveur distant (chapitre 4). La simplicité du modèle permet d'en faire ressortir les goulots d'étranglement et de déterminer les domaines où des optimisations sont envisageables et utiles. L'idée est de proposer un certain nombre d'idées visant à améliorer l'utilisation du réseau dans ce cadre et de voir par la suite comment elles peuvent s'appliquer aux systèmes de fichiers existants, notamment les systèmes parallèles.

L'implantation en espace utilisateur permet d'étudier l'interaction entre l'accès aux fichiers et le réseau. Il faut cependant également s'intéresser au cas où l'accès est mis en œuvre dans le noyau. Cette étude concerne l'accès aux fichiers distants à travers les couches système et l'interface standard de programmation, qui permettent un accès transparent pour n'importe quelle application et autorisent l'utilisation du cache système. Il s'agit donc d'une étude de l'interaction entre réseaux rapides et stockage dans le contexte d'un système du type LUSTRE. Nous utilisons pour ce faire un protocole similaire à ORFA, ORFS (*Optimized Remote File System*), mis en œuvre à travers un client implanté dans le noyau LINUX (chapitre 5).

La distinction entre ces deux cas est importante ici car ils présentent une utilisation très différente du réseau (Q2). Comme nous l'avons vu en partie 2.3.2.d, les types de mémoire mis en jeu dans les communications dans le second cas sont très différents des habituelles zones de mémoire utilisateur manipulées dans les communications de type MPI pour lesquelles ont été conçues les interfaces de programmation du réseau. Nous nous attendons donc à ce que l'interaction entre ces interfaces et les couches d'accès aux fichiers soient difficiles.

Cette étude permet de mettre en évidence certains autres problèmes liés au contrôle des communications dans le contexte général du stockage client-serveur. Notre étude se focalise donc ensuite (chapitre 6) sur le trafic réseau engendré par le stockage distribué (Q3) et la gestion des événements réseau dans le cadre général des entrées-sorties (Q4).

À la lumière de ces expérimentations et de leurs résultats, nous étudions systématiquement les fonctionnalités existantes dans les réseaux rapides pour voir leur adéquation à l'accès distant aux fichiers. Ceci nous conduit à proposer de nouveaux mécanismes améliorant l'efficacité et la simplicité de l'utilisation du réseau dans le cadre général du stockage distribué (chapitre 7). Ainsi, la combinaison des stratégies traditionnelles telles que la parallélisation et de nos travaux sur l'interaction avec le réseau devrait permettre de répondre efficacement aux grands besoins en stockage des applications parallèles s'exécutant sur les grappes.

Chapitre 4

Étude préliminaire

Nous nous concentrons sur l'accès aux fichiers par l'intermédiaire de l'interface standard d'accès aux fichiers (voir en annexe A.1.1). Nous avons expliqué en partie 2.3.1.a que les interfaces spécifiques (telles que MPI-IO ou DAFS) développées pour améliorer l'accès aux fichiers dans les grappes imposent un travail de réécriture de l'application. Nous souhaitons tout d'abord étudier dans quelle mesure on peut se passer de telles interfaces pour obtenir un accès performant aux fichiers distants, via une utilisation efficace du réseau haute performance.

Il s'agit ici de répondre à la question Q1 : **en quoi les performances des réseaux des grappes peuvent-ils améliorer l'accès aux fichiers distants ?**

La complexité des systèmes d'accès aux fichiers distants existants rend difficile leur instrumentation. Par exemple, la présence d'un cache du côté client conduit à segmenter chaque accès en pages et à différer les écritures, ce qui empêche la mesure directe de l'utilisation du réseau. Nos expérimentations ont donc été menées sur un système dédié, que nous avons nommé ORFA (*Optimized Remote File-system Access*).

Nous présentons cette architecture et réalisons des études de performance d'accès aux fichiers distants afin d'identifier quels peuvent être les apports des réseaux des grappes.

4.1 Architecture de ORFA

ORFA a été conçu pour permettre une étude aisée des différentes techniques d'accès au réseau dans le cadre de l'accès distant aux fichiers. L'implantation a tout d'abord été réalisée en espace utilisateur puisque beaucoup de systèmes de fichiers distribués pour grappe y sont implantés, mais aussi pour ne pas subir toutes les contraintes du développement dans le noyau au cours de notre étude préliminaire.

4.1.1 Généralités

L'implantation de ORFA en espace utilisateur est décrite dans [L]. Elle repose sur un client transmettant les requêtes des applications à un serveur distant. Notre volonté

de respecter l'interface standard d'accès aux fichiers permet d'utiliser n'importe quelle application traditionnelle. Ces applications dialoguent habituellement avec les bibliothèques standard du système d'exploitation. Le client ORFA s'insère entre l'application et le système pour intercepter les appels de l'application qui nécessitent des communications via le réseau rapide vers le serveur de fichiers distant (voir la figure 4.1).

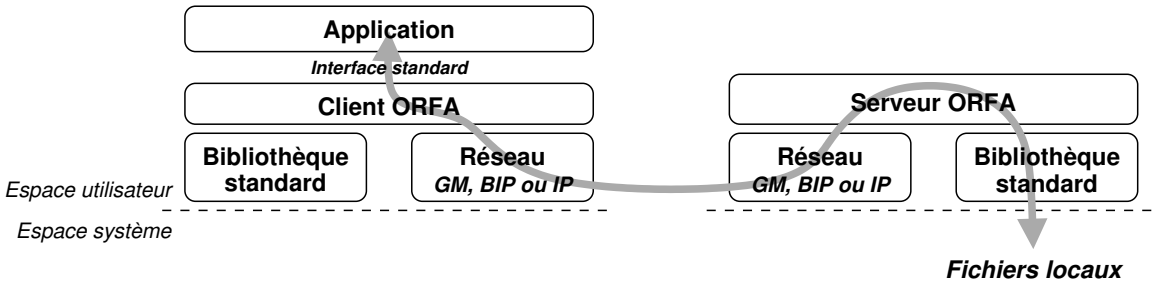


FIG. 4.1: *Modèle* ORFA.

Les requêtes de l'application sont transmises au serveur distant en utilisant différents protocoles et réseaux. Le protocole utilisé entre le client et le serveur correspond exactement aux appels de l'application (`open`, `read`, ...). Le client ne fait preuve d'aucune forme d'intelligence (aucun cache n'est maintenu, les requêtes ne sont pas regroupées, ...). Ainsi, on peut étudier précisément l'impact du réseau sur les différents types de requêtes de l'application.

Le protocole de haut niveau mis en place entre le client et le serveur ORFA se décompose en l'émission par le client d'un descripteur de requête (description d'une lecture ou écriture par exemple) éventuellement suivi de données à traiter, puis de la réponse du serveur composée d'un autre descripteur et éventuellement de données résultantes. Ces descripteurs sont des structures de contrôle décrivant le type et les paramètres de la requête ou le resultat de son traitement dans la réponse.

ORFA a été conçu pour utiliser efficacement les réseaux haute performance. Nous nous sommes concentrés dans cette étude sur les réseaux MYRINET, qui sont les plus couramment utilisés dans les grappes (voir en partie 2.2.4.b). ORFA les manipule par l'intermédiaire des interfaces GM, BIP ou IP. Le choix entre ces couches réseau est fait à la compilation.

Pour extraire de nos expérimentations l'apport des réseaux haute performance, une implantation sur TCP a été développée. Cependant, il est plus important ici de montrer l'apport des couches logicielles plutôt que celui du matériel. C'est donc la couche d'émulation IP sur MYRINET qui sera utilisée lors de nos expérimentations sur TCP.

Nous détaillons maintenant l'implantation du client puis celle du serveur.

4.1.2 Architecture du client

Lorsqu'un client du système de fichiers distribués est implanté dans le système d'exploitation, les couches du système (le VFS) se chargent de le rendre accessible de manière transparente à toutes les applications (voir en partie 2.1.1.d). L'implantation du client ORFA en espace utilisateur nécessite ici d'intercepter les appels de l'application. En effet, il faut détourner le chemin habituel des requêtes de l'application vers le système d'exploitation et passer ces requêtes au client ORFA.

Pour ce faire, on utilise la modularité des applications. Les systèmes d'exploitation modernes utilisent des bibliothèques partagées pour éviter la duplication de code. Ces bibliothèques sont chargées dynamiquement selon les besoins de l'application. Le chargeur permet de préciser différents paramètres, parmi lesquels le fait de forcer le préchargement d'une bibliothèque spéciale. Comme l'édition de liens entre le code de l'application et ceux des bibliothèques se fait en cherchant les symboles dans l'ordre de chargement des bibliothèques, la bibliothèque préchargée va être utilisée en priorité au lieu des bibliothèques habituelles. L'implantation du client ORFA se base donc sur une bibliothèque préchargée qui redéfinit les fonctions d'accès aux fichiers de la bibliothèque standard. Les détails d'implantation de l'interception par le client ORFA ont été publiés dans [B] et sont présentés en annexe C.2.

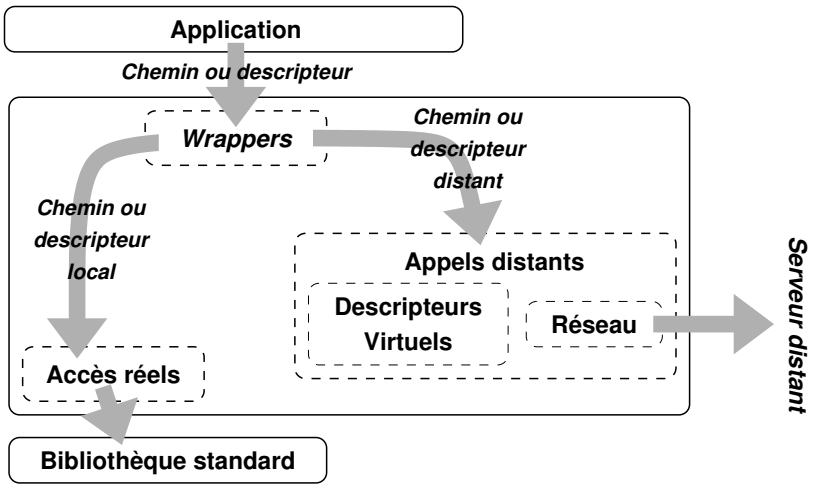


FIG. 4.2: *Implantation du client* ORFA.

Le client ORFA exporte une interface d'accès aux fichiers identique à l'interface standard d'accès aux fichiers dans UNIX. Les fichiers sont représentés par des chemins d'accès ou par des descripteurs de fichiers. Dans les deux cas, le client ORFA regarde si le chemin ou descripteur est local ou distant (voir la figure 4.2) puis redirige le traitement vers la bibliothèque standard (pour les fichiers locaux) ou le serveur distant (fichiers distants).

Lorsqu'un fichier est manipulé par son chemin, son traitement ne nécessite que

de faire suivre la requête de l'application au serveur distant. Par contre, si l'application manipule un fichier par un descripteur (identifiant entier renvoyé par `open`), le client ORFA doit faire en sorte que le descripteur virtuel du fichier distant se comporte comme un descripteur traditionnel, ce qui est bien plus complexe. Il faut en effet émuler la sémantique de partage de fichier entre processus (*File Sharing Semantics*). Par exemple, les descripteurs peuvent être partagés après un `fork`, conservés après un `exec` ou dupliqués lors d'un `dup`. Le client ORFA supporte toutes ces fonctionnalités, ce qui permet à n'importe quelle application de manipuler les fichiers distants de manière complètement transparente. Les détails de mise en œuvre des descripteurs virtuels de fichiers distants sont présentés en annexe C.1.

Le principal rôle du client ORFA consiste à transférer les données entre l'application et le serveur distant. Pour ce faire, il utilise efficacement les protocoles zéro-copie en passant les zones mémoire de l'application à la couche d'accès au réseau (délai T_{RegC} dans notre modélisation en partie 2.3.2). Cela impose de mettre en œuvre dans le client le même genre de techniques que dans une couche MPI, notamment pour gérer l'enregistrement mémoire. En effet, l'interface standard d'accès aux fichiers que les applications utilisent ne propose aucune primitive destinée à préparer les zones mémoire mises en jeu dans les accès aux fichiers.

Le client ORFA doit donc réaliser cette préparation de manière transparente. Les techniques habituelles d'optimisation de l'enregistrement mémoire (voir en partie 2.2.3.d) sont utilisées, notamment le fait d'utiliser des copies pour les petits messages et un cache d'enregistrement pour les gros. Ainsi, les données peuvent être transférées efficacement depuis l'application vers le réseau. Nous étudierons plus précisément cette implantation dans la partie 5.1.

4.1.3 Architecture du serveur

Le serveur ORFA se charge de traiter les requêtes des clients distants. Il doit d'une part être en attente des événements du réseau et d'autre part traiter les requêtes correspondantes. Il s'agit essentiellement de gérer efficacement et simultanément des entrées-sorties vers le réseau et les fichiers.

Nous étudions tout d'abord un serveur en espace utilisateur accédant aux fichiers locaux (voir la figure 4.1). Les requêtes des clients distants sont traduites en requêtes similaires vers les fichiers locaux. Ce modèle très simple permet d'étudier différentes optimisations au niveau de l'interaction entre le réseau et l'accès aux fichiers, notamment l'influence de la réduction des copies ou des méthodes de notification d'événements (délais T_{RegS}, T_{AsyncS} et T_{Event}).

Cependant, ce modèle a l'inconvénient de pouvoir présenter certains goulots d'étranglement du côté serveur. Par exemple, la traversée des couches système (notamment le VFS) impose un certain nombre de verrous pour assurer l'intégrité des données accédées à la fois par le serveur ORFA et par d'autres processus locaux. Lorsque le volume de données n'est destiné à être accédé que par des clients via le système distribué (par

opposition à des accès locaux par les processus de la machine serveur), ces verrous sont inutiles. Éviter la traversée de ces couches système permet alors de réduire la charge du serveur. Ce modèle peut par exemple être mis en œuvre en manipulant le périphérique de stockage en mode bloc depuis le processus serveur (qui se charge d'y implanter une structure logique de système de fichiers).

Ensuite, les accès aux périphériques de stockage physique imposent des contraintes temporelles assez importantes et peu prévisibles. Les performances observées du côté client peuvent donc non seulement dépendre de la traversée des couches système, mais aussi des accès disque du côté serveur. Afin que notre étude des optimisations du côté client ne soit pas gênée, nous avons également développé un modèle où un serveur virtuel évite tous ces problèmes potentiels. Le but est alors simplement de fournir aux clients un accès performant et prévisible.

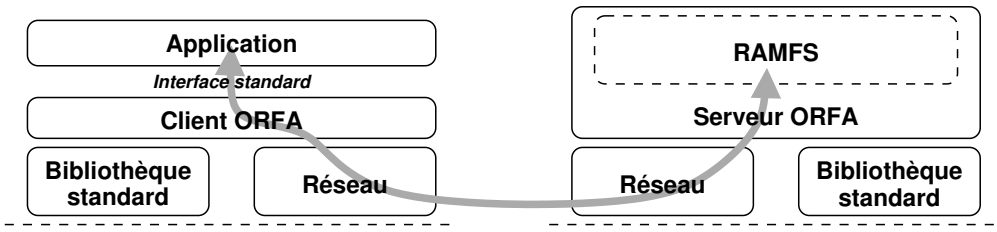


FIG. 4.3: *Implantation du serveur* ORFA *en espace utilisateur accédant à un système de fichiers en mémoire (*RAMFS*).*

ORFA propose donc un serveur entièrement implanté en espace utilisateur, c'est-à-dire que le système de fichiers exporté aux clients distants (RAMFS) est lui aussi implanté dans le processus serveur (voir la figure 4.3). La bibliothèque standard du côté serveur n'est alors plus utilisée. Les accès aux fichiers sont donc uniquement des accès à des blocs stockés dans la mémoire utilisateur du processus serveur. Cela réduit considérablement les risques de surcharge du serveur et permet des mesures cohérentes des performances du côté client qui ne sont pas polluées par des phénomènes pouvant survenir du côté serveur. Ainsi, les optimisations du côté client sont plus simples à étudier.

4.2 Estimation des performances

L'architecture d'ORFA permet, du fait de son absence de cache du côté client, de mesurer facilement les performances brutes de l'accès aux fichiers distants entre l'espace mémoire de l'application et le serveur.

Nous présentons tout d'abord une évaluation de la latence des requêtes simples pour en extraire les coûts des différentes étapes du chemin critique et voir dans quelle mesure la latence très faible des réseaux est utile pour l'accès aux fichiers distants. Ces premiers résultats sont ensuite complétés par une étude des performances des accès aux métadonnées. Nous nous concentrons alors sur les accès aux données réelles pour

montrer l'influence de la grande bande passante des réseaux haute performance. Enfin, les évaluations des performances lors d'accès concurrents nous conduisent à étudier la suppression de copies de données pour réduire la charge du côté serveur.

4.2.1 Méthodologie

Ces études de performances sont menées en comparant, d'une part, les accès distants aux fichiers via ORFA et NFS et, d'autre part, l'utilisation de l'émulation IP sur MYRINET/GM et l'implantation native de ORFA sur GM. La plate-forme de test est une grappe constituée de 24 machines équipées de 2 processeurs PENTIUM 4 XEON cadencés à 2,6 GHz et 2 Go de mémoire vive. Le réseau MYRINET 2000 utilise des cartes PCI64C munies d'un processeur LANAI 9.3 cadencé à 200 MHz.

Une machine est consacrée au processus serveur (ORFA ou NFS) tandis que les 23 autres sont chargées d'exécuter entre 1 et 8 clients chacune. Nous faisons ainsi varier le nombre de clients entre 1 et 184, ce qui permet d'étudier les performances des accès simples (point-à-point, avec 1 seul client) ainsi que le coût des travaux sur le serveur (avec beaucoup de clients).

Les données manipulées sont d'une part de grands fichiers accédés séquentiellement pour les tests d'accès aux données réelles (tests en lecture et écriture), et d'autre part une arborescence de 10 000 fichiers de quelques kilo-octets pour les tests d'accès aux métadonnées (commandes `du` et `tar`).

4.2.2 Latence

Nos premières expérimentations consistent à mesurer la latence des requêtes envoyées au serveur par le client. La table 4.1 présente les temps d'accès aux fichiers mesurés au niveau de l'application par des requêtes courtes et simples (récupération des attributs d'un fichier).

Notre étude débute par une comparaison des temps d'accès aux fichiers distants dans ORFA et NFS (dont le cache côté client est initialement vide). Les mesures sont effectuées d'une part sur un réseau IP traditionnel (FAST ETHERNET relié par un seul commutateur) et d'autre part sur l'émulation IP des réseaux MYRINET. Les temps d'accès observés au niveau applicatif avec ORFA et NFS montrent que le coût protocolaire des deux systèmes est similaire et négligeable devant la latence observée au niveau de la couche réseau. Notre système ORFA a donc un comportement cohérent au regard de NFS. Ces résultats montrent d'ailleurs que la traversée dans NFS des couches XDR (*External Data Representation* pour la portabilité [XDR87]) et RPC (*Remote Procedure Call* pour transmettre les requêtes [RPC88]) est moins coûteuse sur les machines modernes que par le passé [HSCL97, Fab98].

Nous poursuivons ensuite notre étude en observant l'apport de l'utilisation native d'un réseau haute performance. Les comparaisons des temps d'accès au niveau applicatif avec ORFA sur l'émulation IP de MYRINET et sur GM montrent un apport indéniable

Méthode d'accès aux fichiers	Protocole réseau	Temps d'accès aux fichiers	Latence réseau (aller-retour)	Coût hors-réseau
NFS	TCP/ETHERNET	$\simeq 310$	$\simeq 300$	$\simeq 10$
	TCP/GM	$\simeq 120$	$\simeq 110$	$\simeq 10$
ORFA	TCP/ETHERNET	$\simeq 310$	$\simeq 300$	$\simeq 10$
	TCP/GM	$\simeq 120$	$\simeq 110$	$\simeq 10$
	GM	28,7	16,6	11,1
Local	—	0,66	—	0,66
Local + ORFA	—	0,71	—	0,71

TAB. 4.1: *Temps d'accès aux fichiers distants sur* NFS *et* ORFA *(en microsecondes). Ces temps ont été mesurés en moyennant les temps totaux de 1000 accès à un fichier distant au niveau applicatif. Les latences au niveau de la couche réseau sont obtenues avec* NETPIPE *[Net] pour* TCP *et un programme dédié à* MYRINET/GM.

de la faible latence du réseau. Le temps d'accès aux fichiers distants passe de 120 à 28 μs lorsque la latence au niveau réseau passe de 110 à 16 μs.

Par contre la comparaison avec les temps d'accès aux fichiers locaux montre que l'accès distant reste très lent même sur un réseau rapide. Le coût de traitement du protocole au niveau applicatif, par exemple la gestion des files de requêtes ou la traduction des descripteurs de requête en appel de fonction, reste en effet plus important que le simple appel d'une fonction dans le cas d'un accès local. La ligne **Local + ORFA** permet par ailleurs de signaler que le surcoût de l'interception dans ORFA est très faible (environ 50 ns).

Ces résultats nous amènent à remettre en cause la nécessité du cache du côté client. En effet, si le temps d'accès aux fichiers distants sur les réseaux rapides reste supérieur au temps d'accès local, il est très inférieur à celui observé sur les réseaux traditionnels. Or, les modèles actuels tels que NFS ont été conçus en supposant que la latence réseau est beaucoup trop élevée pour que le client puisse contacter le serveur à chaque accès, d'où l'introduction d'un cache du côté client (voir en partie 2.1.2) La latence très faible apportée par les réseaux rapides permet d'envisager de se passer de ce cache et de contacter plus souvent le serveur distant. C'est ce que nous étudions dans la partie suivante.

4.2.3 Accès aux métadonnées

Le test précédent visait à extraire la latence des accès, nous observons maintenant le comportement de ORFA et NFS lorsqu'une application parcourt une arborescence de répertoires et récupère les attributs (métadonnées) de tous les fichiers (commande UNIX `du`), puis lorsqu'elle lit le contenu de ces fichiers (commande UNIX `tar`).

La table 4.2 présente la quantité de fichiers traités par le serveur. Nous comparons

Accès distant	Couche réseau	Fichiers traités avec `du` par seconde	Fichiers traités avec `tar` par seconde
NFS	TCP/GM	8697	869
ORFA	TCP/GM	4014	1139
ORFA	GM	7455	2202

TAB. 4.2: *Comparaison des performances d'accès aux métadonnées avec* NFS *et* ORFA. *Un client parcourt une arborescence de 10 000 fichiers répartis dans 4 niveaux de répertoires et occupant un espace total de 130 Mo. Le test avec* `du` *consiste à récupérer les attributs de chaque fichier.* `tar` *lit non seulement les attributs mais aussi le contenu des fichiers. Les mesures sont réalisées en moyennant 100 tests successifs.*

tout d'abord les performances d'accès aux métadonnées (tests avec `du`) avec NFS et ORFA sur l'émulation IP sur MYRINET/GM. À réseau équivalent, NFS est capable de traiter plus de deux fois plus de requêtes concernant les métadonnées. Cet avantage de NFS sur ORFA est dû à l'intelligence du client NFS et à son cache. En effet, il optimise ce type d'accès en lisant directement plusieurs entrées d'un répertoire à l'avance (*Read-Ahead*), en récupérant les attributs des fichiers dans la même requête (agrégation de requêtes) et en les gardant dans son cache du côté client. La simplicité du client ORFA l'oblige à effectuer deux requêtes au serveur distant pour chaque entrée de répertoire (lecture de l'entrée puis récupération des attributs du fichier correspondant). NFS compense donc la latence réseau en réduisant considérablement le nombre de requêtes envoyées au serveur. L'utilisation native du réseau MYRINET dans ORFA/GM améliore beaucoup les performances mais ne suffit pas à rattraper l'avantage de NFS. L'apport du cache dans le client NFS est donc indéniable ici, quelles que soient les performances du réseau sous-jacent.

Lorsque l'expérience implique également la lecture du contenu des fichiers (tests avec `tar`), le client ORFA se comporte beaucoup mieux, notamment parce que les optimisations de NFS pour accéder aux métadonnées perdent de leur importance devant la quantité de données transférées.

Accès distant	Couche réseau	Optimisations	Fichiers traités par `du`
NFS	TCP/GM	*Read-Ahead* + cache	8697
ORFA	GM	–	7455
ORFA	GM	RAMFS	8697
ORFA	GM	RAMFS + *Read-Ahead*	11597

TAB. 4.3: *Impact des optimisations sur les accès aux métadonnées avec* ORFA. *On mesure l'apport du serveur* ORFA *implanté en mémoire (*RAMFS*) puis celui de la lecture à l'avance des métadonnées (*Read-Ahead*). Les mesures sont réalisées comme pour la table 4.2.*

Pour clarifier l'apport du cache pour les accès aux métadonnées, nous présentons ensuite dans la table 4.3 les résultats de la même expérience (test avec `du`) menée avec des systèmes ORFA optimisés. L'utilisation du serveur ORFA en mémoire (RAMFS, voir sur la figure 4.3) apporte un gain d'environ 15 %. Compte tenu de la forte réduction du travail sur le serveur (car les couches système ne sont plus traversées et il n'y a plus d'accès disque sur le serveur), ce gain se révèle assez faible, ce qui montre que la quantité de travail sur le serveur n'est pas un facteur limitant dans cette expérience.

La seconde optimisation consiste à ajouter du *Read-Ahead* pour lire les entrées de répertoire à l'avance. Le client ORFA peut alors récupérer plusieurs entrées par requête (mais la récupération de leurs attributs nécessite toujours une requête par entrée). Le nombre de requêtes au serveur est donc presque divisé par deux par rapport aux expériences précédentes. Le gain d'environ 20 % observé ici montre l'apport important de ce genre d'optimisations. Pourtant, le nombre de requêtes reste très supérieur à celui de NFS puisque le client NFS est capable de récupérer plusieurs entrées et leurs attributs en une seule requête.

Nous avons montré que le cache du côté client avait un impact très important sur les performances de l'accès aux métadonnées et que, en comparaison, la faible latence offerte par les réseaux des grappes apportait finalement un gain de performances assez faible.

4.2.4 Accès aux données réelles

Après l'étude de l'influence des accès aux métadonnées, nous nous intéressons maintenant aux performances des accès aux données réelles contenues dans les fichiers. Un fichier de 130 megaoctets est lu séquentiellement par un client en utilisant différentes tailles de requêtes. Les résultats sont présentés sur la figure 4.4.

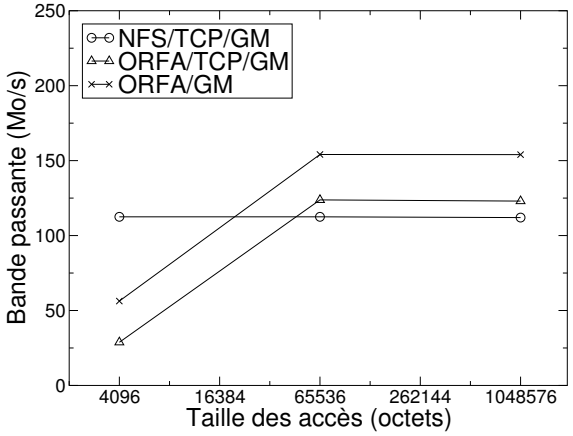


FIG. 4.4: *Performances des lectures séquentielles dans un fichier pour des requêtes de 4 ko, 64 ko et 1 Mo.*

L'accès par requête de 4 ko montre là encore la supériorité de NFS. Le *Virtual File System* de la machine cliente utilise la technique du *Read-Ahead* qui consiste à charger à l'avance les pages suivantes du fichier pour anticiper une lecture séquentielle. Le client NFS lit ici quelques pages en avance, ses requêtes sur le réseau sont donc en fait de l'ordre de 2 ou 4 pages. Cela explique pourquoi ses performances sont proches de celles d'ORFA pour des requêtes 8 ou 16 ko.

Les performances de NFS et ORFA pour des requêtes de 64 ko et 1 Mo montrent la saturation des interfaces réseaux IP vers 120 Mo/s. Cela peut s'expliquer par le coût de ces couches protocolaires, notamment en terme de copies mémoire, qui deviennent prédominantes quand la taille des requêtes augmente [Fab98]. Par contre, l'utilisation de l'interface native GM permet d'améliorer grandement les performances des transferts grâce aux communications zéro-copie et à la bande passante élevée offerte par MYRINET/GM.

4.2.5 Accès concurrents

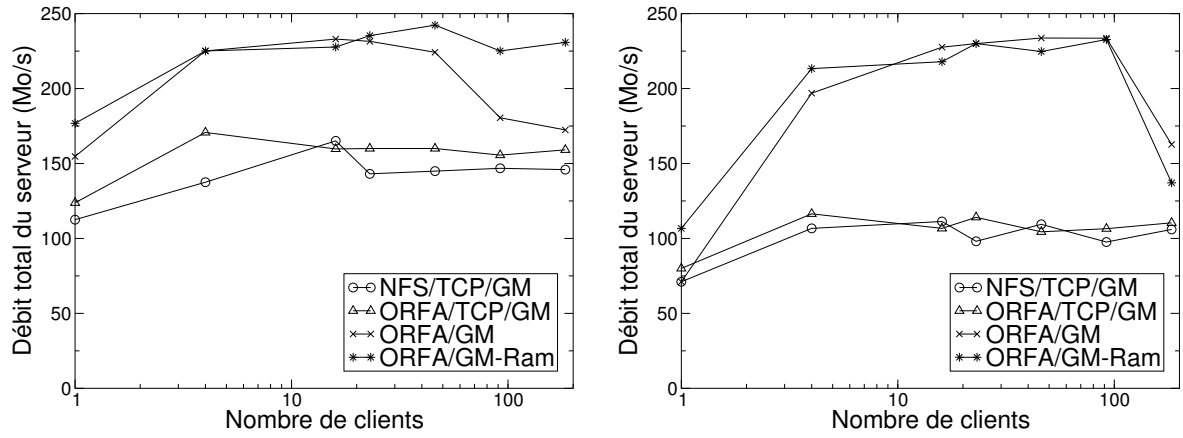
Nous nous intéressons maintenant aux accès concurrents en observant les performances lorsque de nombreux clients émettent simultanément des requêtes au même serveur. Les clients manipulent ici des fichiers distincts. Nous étudions donc les accès concurrents au même serveur et non pas les modifications concurrentes dans un même fichier.

Le serveur reçoit des données lors des requêtes en écriture alors qu'il en émet dans le cas des lectures. La comparaison des performances de ces deux types de requêtes permet donc de voir si la direction du trafic a un impact sur les performances et d'observer d'éventuels goulots d'étranglement dans la gestion des accès concurrents au niveau du serveur.

La figure 4.5 présente le débit que peut soutenir le serveur lorsque les clients accèdent à des fichiers distincts par des requêtes de 64 ko. La courbe **Ram** présente les performances lorsque le serveur accède à un système de fichiers en mémoire (RAMFS), c'est-à-dire un serveur où le traitement des requêtes est moindre (pas de traversée des couches système et moins de copies mémoire).

Comme nous l'avons vu précédemment, les débits supportés par NFS et ORFA sur les interfaces IP saturent rapidement, à 150 Mo/s en lecture et 100 en écriture. Le débit en écriture est inférieur car le traitement des modifications de fichiers sur le serveur est plus complexe que celui des lectures. Cette différence n'apparaît pas dans le cas ORFS/GM car le coût protocolaire y est bien moindre et permet donc d'éviter une saturation du serveur dans les cas des écritures.

Les performances d'ORFA sur l'interface GM sont bien supérieures, le débit maximal du lien physique (250 Mo/s) est quasiment atteint. Cependant, en présence de très nombreux clients, le débit utile du serveur finit par diminuer. Il s'agit en fait d'une saturation du serveur puisque cette diminution n'apparaît pas lors de lectures depuis le serveur implanté en mémoire.



(a) Lectures de 64 ko.

(b) Écritures de 64 ko.

FIG. 4.5: *Performances des lectures et écritures séquentielles dans un fichier pour des requêtes de 64 ko (de 1 à 184 clients).*

Par contre, en écriture, la chûte du débit utile est brutale et ce, avec ou sans le système de fichiers en mémoire. La raison est que le serveur ORFA ne parvient plus à recevoir les données suffisamment vite. En effet, le serveur doit préparer un tampon de réception pour chaque message qu'un client envoie. Quand le nombre de clients et la charge du serveur augmente, ce dernier ne parvient plus à préparer ces tampons assez vite. Le contrôle des communications dans GM ne supporte pas bien ce cas et provoque le gaspillage d'une grande partie de la bande passante, ce qui explique la chute des performances d'ORFA. Nous détaillerons ce problème en partie 6.1.

4.2.6 Réduction des copies du côté serveur

Nous venons de voir dans les expérimentations précédentes que la charge du serveur est un facteur limitant important. Une des principales différences de charge entre le serveur ORFA implanté en mémoire et celui accédant aux fichiers locaux réels est le nombre de copies mémoire mises en jeu. Les accès aux fichiers locaux du serveur imposent de copier les données dans le cache de fichiers du système d'exploitation tandis que l'accès au système de fichiers en mémoire est direct. Cette copie mémoire supplémentaire et la traversée des couches du système d'exploitation ont un impact conséquent sur les performances. La figure 4.5(a) montre par exemple une différence de 20 % sur le débit utile du serveur par les grandes requêtes.

Nous avons vu en partie 2.3.2.c que des travaux avaient été menés pour éviter cette copie supplémentaire, en particulier dans les serveurs WWW. Nous présentons ici quelques mesures visant à confirmer ces résultats.

Tout d'abord, la suppression de cette copie peut être réalisée en *mappant* (projetant)

le fichier cible dans la mémoire du processus serveur[1]. Il suffit alors d'émettre ou de recevoir depuis cette zone de mémoire virtuelle pour accéder directement au cache de fichiers du système d'exploitation. En fait, le coût du mapping d'un fichier dépend beaucoup de l'architecture matérielle, et en particulier du nombre de processeurs car le mapping impose de mettre à jour la table de pages et donc de vider le cache de traduction d'adresses (TLB, *Translation Lookaside Buffer*) dans chaque processeur. Par ailleurs, la mise en œuvre dans le système d'exploitation peut beaucoup varier. Par exemple, LINUX 2.4 présente un coût de mapping de quelques centaines de microsecondes. Dans ce cas, la copie mémoire due à une lecture ou écriture dans le cache restera plus rapide pour moins de 100 ko de données à traiter. LINUX 2.6 introduit des améliorations à ce système en différant par exemple le mapping réel jusqu'à l'accès réel aux pages mappées. Cependant, le coût moyen reste supérieur à celui d'une lecture/écriture. L'utilisation du mapping mémoire ne semble donc pas utile en terme de performances, sauf peut-être pour de très larges portions de fichiers.

L'autre méthode permettant de supprimer cette copie mémoire est d'utiliser l'appel-système `sendfile` qui permet d'envoyer directement les données du cache de fichiers vers une connexion TCP (voir en partie 2.3.2.c). L'utilisation de cette stratégie dans le serveur ORFA sur TCP lorsque les clients lisent des données apporte un gain en débit utile du serveur de 10 % pour des requêtes de 64 ko et 30 % pour 1 Mo. L'apport de cette stratégie est donc indéniable pour les accès larges aux fichiers distants.

Aucun appel `recvfile` n'est cité dans la littérature, probablement parce que les travaux menés sur `sendfile` ciblaient les serveurs WWW où les accès se font principalement en lecture seule. Dans le cas d'un serveur de fichiers, les accès se font également en écriture. Pour prouver le concept, nous avons mis en place `sendfile` et `recvfile` sur BIP (voir en partie 2.2.4.b) et avons observé un gain en performance identique pour les accès en lecture et écriture aux fichiers distants. Cependant, le stabilité limitée de BIP et l'arrêt de son développement ne nous a pas permis de continuer ces travaux. Aucune mise en œuvre d'un appel `sendfile` (ou `recvfile`) sur GM n'est citée dans la littérature. Mais nos travaux sur BIP nous conduisent à penser que cela optimiserait de façon conséquente les accès au serveur.

4.3 Synthèse

Le but de notre étude était de répondre à la question Q1 de la problématique : **en quoi les performances des réseaux des grappes peuvent-ils améliorer l'accès aux fichiers distants ?** Les réseaux haute performance fournissent une latence très faible (jusqu'à quelques microsecondes) et une bande passante très élevée.

Les mesures présentées en partie 4.2.2 laissent supposer que la latence très faible du réseau permet d'abaisser singulièrement les temps d'accès aux fichiers distants. Cependant, les mesures dans une véritable application en 4.2.3 montrent que l'impact de la

[1] Le mapping d'un fichier consiste en fait à utiliser les pages du cache de fichiers dans l'espace d'adressage d'un processus.

latence est en fait assez faible. Le bénéfice attendu pour des petites requêtes (quelques centaines d'octets), c'est-à-dire pour les accès aux métadonnées, est inférieur à l'apport d'un cache du côté client et de l'utilisation de techniques du type *Read-Ahead* ou agrégation de requêtes comme nous l'avons vu en observant les performances de NFS. C'est notamment la forte latence des réseaux traditionnels qui avait conduit à l'implantation d'un tel cache (voir en 2.1.2). La faible latence des réseaux haute performance ne permet pas de remettre cette idée en cause : **un transfert réseau reste beaucoup plus coûteux qu'un accès dans le cache du client.**

En ce qui concerne les données réelles des fichiers, la taille des messages peut énormément varier. Des études sur des applications réelles, par exemple dans [SR97], ont montré qu'il était nécessaire d'utiliser des optimisations différentes pour traiter les différents types de requêtes, par exemple en maintenant un cache du côté client pour les petits accès mais en évitant le cache pour les gros. La bande passante du réseau se révèle très importante pour ces derniers, comme nous l'avons vu sur la figure 4.5. La bande passante élevée du réseau rapide apporte un gain incontestable pour l'accès aux fichiers distants.

Ces expérimentations et résultats ont été publiés dans [C]. Ils nous ont permis de mettre en évidence de nouveaux problèmes :

Type d'implantation Le protocole expérimental ORFA que nous avons utilisé dans les expérimentations précédentes est entièrement implanté en espace utilisateur. Ce type de mise en œuvre ne concerne qu'une partie des systèmes de stockage distribué existants. Beaucoup de systèmes modernes tels que PVFS2 et LUSTRE sont implantés dans le noyau. Il nous faut donc également étudier ce type de systèmes (Q2).

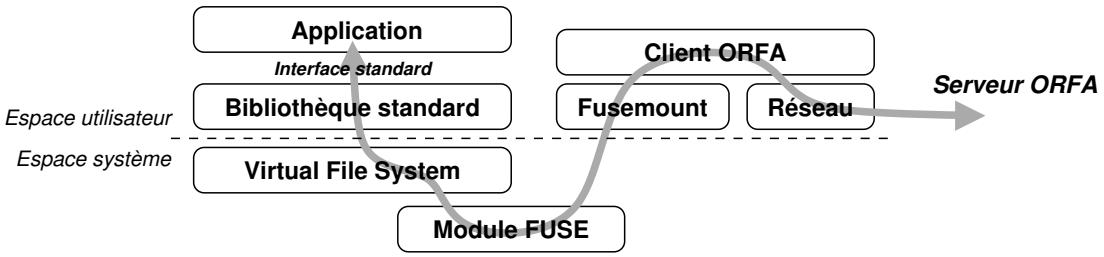


FIG. 4.6: *Mise en œuvre de* ORFA *à travers le noyau* LINUX *avec* FUSE.

Lors de nos études préliminaires, nous nous sommes intéressés à une implantation mixte réalisée par le système FUSE (*File-system in USEr-space* [FUS]) qui permet de monter dans le noyau LINUX un système de fichiers implanté en espace utilisateur. Nous avons conçu un client ORFA pour FUSE, ce qui permet aux applications d'accéder aux fichiers distants par le protocole ORFA sans nécessiter aucune interception des primitives d'accès aux fichiers. Le module noyau de FUSE transmet les requêtes à un processus dédié (*Fusemount*) qui les passe au client ORFA (voir la figure 4.6). Ces travaux n'ont

pas abouti à des résultats intéressants car les performances étaient trop limitées par le fait que l'aller-retour dans le noyau impose deux copies mémoire supplémentaires. Nous avons notamment mesuré que la bande passante observée par l'application était divisée par deux par rapport à l'implantation classique d'ORFA en espace utilisateur.

En revanche, cette implantation a l'avantage de bénéficier automatiquement des caches natifs du système d'exploitation. En effet, comme toute implantation dans le noyau, le système FUSE est placé sous le VFS (*Virtual File System*) qui cache les structures logiques des systèmes de fichiers. En particulier, les métadonnées sont toujours cachées tandis que les données le sont à condition que l'application ne demande pas explicitement des accès directs (par le paramètre `O_DIRECT` passé à l'ouverture du fichier). Le cache permet d'éviter de contacter plusieurs fois le serveur distant si les mêmes données ou métadonnées sont accédées consécutivement. De plus, le principe de localité conduit à des optimisations du type *Read-Ahead* qui sont très utiles quand on parcourt un répertoire ou quand on lit un fichier de manière séquentielle.

Nous avons vu en 4.2.3 que le cache des métadonnées était très important pour accéder aux fichiers distants de manière performante, beaucoup plus que la faible latence du réseau. La présence du cache natif dans le système d'exploitation nous permet d'étudier immédiatement la performance d'un système avec cache du côté client, sans avoir à l'implanter nous même (cette quantité de travail aurait présenté peu d'intérêt dans notre étude), à condition de placer le client ORFA dans le système d'exploitation.

Nous présentons au chapitre 5 une étude détaillée de l'implantation de ORFA dans le noyau LINUX en nous intéressant à la fois aux accès à travers le cache et aux accès directs.

Uniformité du trafic Nous avons vu en partie 4.2.5 que les accès en écriture avaient un impact réseau très particulier lorsqu'on utilise GM. En effet, lorsqu'un nombre conséquent de clients accèdent au serveur en lecture, le flux sortant du serveur est explicitement contrôlé par ce dernier qui envoie lui-même les données aux clients. Par contre, en écriture, les clients émettent les messages sans se soucier a priori de l'état des autres clients. Cela conduit à une congestion côté serveur.

Il est important de mettre en place un mécanisme de contrôle de communication soit dans la couche réseau soit dans l'application client-serveur elle-même. Dans nos expérimentations, les communications entre les clients et le serveur ORFA utilisent le paradigme de passage de messages. Or, GM se révèle incapable de traiter de manière intelligente les messages inattendus, c'est-à-dire les messages dont la zone de réception est inconnue car la réception correspondante n'a pas encore été preparée. L'émetteur réémet alors continuellement son message jusqu'à ce que la réception correspondante ait été soumise, ce qui conduit à un important gaspillage de bande passante lorsque le message est volumineux. Le nombre de requêtes de réception simultanées étant limitées, l'augmentation du nombre de clients amplifie ce phénomène et accroît le gaspillage de ressources. Le débit utile du serveur diminue donc fortement et ce linéairement. L'utilisation du serveur ORFA en mémoire réduit ce problème car la charge du serveur est

notablement diminuée puisque les couches système n'ont plus à être traversées pour accéder aux fichiers réels.

Nous étudierons le problème du contrôle des communications dans cet environnement à trafic non-régulier (Q3) dans la partie 6.1.

Efficacité du serveur Nos expérimentations ont montré que le serveur était généralement le goulot d'étranglement. L'utilisation d'un réseau haute performance améliore notablement les accès aux fichiers distants mais augmente la charge et révèle donc plus rapidement les limites du serveur. Le serveur ORFA implanté entièrement en mémoire (RAMFS) a beaucoup moins de travail à effectuer. Nous avons vu en partie 4.2.6 que des optimisations, notamment en terme de copies mémoire étaient tout à fait envisageables. Cependant, ce serveur finit lui aussi par saturer quand le nombre de clients augmente.

Il est important d'utiliser un serveur peu chargé pour mettre en évidence les problèmes d'utilisation du réseau. Mais il faut ensuite utiliser un serveur chargé pour y mettre en évidence le goulot d'étranglement. Il s'agit notamment de voir dans quelle mesure les événements réseau peuvent être traités efficacement dans le cadre d'un processus serveur accédant simultanément aux fichiers locaux et au réseau rapide (Q4).

Les accès aux fichiers se font de manière synchrone dans notre serveur ORFA. Il faudrait voir dans quelle mesure l'utilisation d'un modèle événementiel couplé aux entrées-sorties asynchrones permettrait d'améliorer les performances. Ce problème ainsi que la gestion des événements du côté client seront étudiés en partie 6.2.

Chapitre 5

Optimisations des transferts de données

Nous présentons maintenant une étude détaillée de l'utilisation d'un réseau haute performance pour transférer les données lors de l'accès aux fichiers. Nous nous intéressons au réseau MYRINET de MYRICOM et à son interface de programmation officielle, GM (voir la partie 2.2.4.b). Cette interface est spécifique mais est assez représentative des interfaces actuelles des réseaux rapides. Il s'agit de s'intéresser à la question Q2 : **comment les mécanismes des réseaux haute performance peuvent-ils améliorer le transfert de données lors de l'accès aux fichiers distants ?**

GM a été conçue pour les applications de passage de message en espace utilisateur (voir l'annexe A.3.1 et [GM03]), par exemple MPI. Cependant, elle propose également la même interface dans le noyau LINUX permettant ainsi des communications entre des applications en espace utilisateur et en espace noyau.

Nous détaillons tout d'abord l'utilisation de GM dans ORFA pour voir dans quelle mesure GM répond aux besoins en transferts de données des fichiers distants en espace utilisateur. Nous étudions ensuite l'implantation de ORFS, le portage de ORFA dans le noyau LINUX, pour montrer les problèmes d'interaction entre l'interface de programmation de GM et les couches système d'accès aux fichiers.

Nous nous concentrons sur le côté client des implantations car c'est celui qui impose le plus de contraintes. En effet, notre volonté de nous conformer à l'interface standard d'accès aux fichiers limite notre marge de manœuvre puisque nous ne pouvons pas remonter jusqu'à l'application les contraintes liées à l'interface de programmation du réseau rapide sous-jacent.

Nous avons mené nos expériences entre deux machines équipées de deux processeurs PENTIUM 4 XEON cadencés à 2,6 GHz et de 2 Go de mémoire vive. Elles sont reliées par un réseau MYRINET 2000 qui utilise des cartes PCI64C munies d'un processeur LANAI 9.3 cadencé à 200 MHz. L'interface GM 2.0.13 et un noyau LINUX 2.4.26 ont été utilisés.

5.1 Utilisation traditionnelle en espace utilisateur

Nous détaillons tout d'abord l'implantation d'ORFA au dessus de MYRINET/GM. Comme nous l'avons vu en 4.1.2, le client ORFA se présente sous la forme d'une biblio-

thèque partagée automatiquement préchargée. Cette bibliothèque intercepte les appels d'entrées-sorties des applications et redirige vers un serveur ceux qui concernent les fichiers distants accessibles via le réseau rapide MYRINET.

Cette interception transparente doit être capable de transférer efficacement les données entre les zones mémoire de l'application (celles qu'elle fournit pour accéder aux fichiers) et le réseau. Il nous faut donc utiliser l'enregistrement mémoire sans aucune assistance possible de l'application puisque nous avons choisi de ne pas la modifier.

Comme nous l'avons vu en 2.2.3.d, le coût important de cette stratégie nécessite de la mettre en œuvre intelligement. Nous détaillons maintenant cette mise en œuvre dans le client ORFA.

5.1.1 Impact de l'enregistrement mémoire

L'enregistrement mémoire est une stratégie consistant à stocker dans la carte d'interface des traductions d'adresses virtuelles en adresses physiques. Ainsi, le passage des adresses virtuelles manipulées par l'application aux adresses physiques manipulées par la carte peut être effectué sans intervention du système d'exploitation (*OS-bypass*). La carte d'interface réseau peut alors transférer les données par DMA, c'est-à-dire sans copie ni intervention du processeur central.

L'enregistrement mémoire est donc une étape d'initialisation des zones mémoire. Il s'agit de demander l'assistance du système d'exploitation pour parcourir ces zones, verrouiller les pages qu'elles contiennent, traduire leurs adresses virtuelles et stocker les traductions dans une table dans la mémoire de la carte. Toutes les communications peuvent ensuite être entièrement traitées par la carte sans assistance du système.

Le désenregistrement consiste à enlever une traduction d'adresse de la mémoire de la carte. Il est notamment nécessaire lorsqu'une entrée n'est plus valide. Pour éviter qu'une entrée ne soit utilisée par une communication en cours alors que son désenregistrement vient d'être demandé, le désenregistrement doit être synchrone. Il faut interrompre le fonctionnement normal de la carte (c'est-à-dire le traitement des communications) et ne pas le reprendre avant que le désenregistrement soit terminé. Ces deux opérations sont coûteuses, en particulier dans MYRINET/GM qui n'a pas été conçu pour désenregistrer couramment des zones mémoire.

Nous avons mesuré le coût de l'enregistrement mémoire dans GM et l'avons comparé au coût d'une copie mémoire. Les résultats sont présentés sur la figure 5.1. L'enregistrement mémoire coûte environ 2,6 μs par page (4 ko sur nos machines), ce qui correspond au temps de verrouillage, traduction d'adresse et stockage dans la table de la carte. Le désenregistrement a un coût initial de 215 μs (interruption de la carte) auquel s'ajoute 2,1 μs par page (déverrouillage et suppression dans la table de la carte). En comparaison, la copie d'une page mémoire coûte 2 μs sur nos machines.

Il est donc important ici de bien choisir le compromis entre copie et enregistrement puisque le gaspillage de quelques cycles processeurs pour copier les données dans une zone statiquement pré-enregistrée peut être facilement plus rentable que tout le méca-

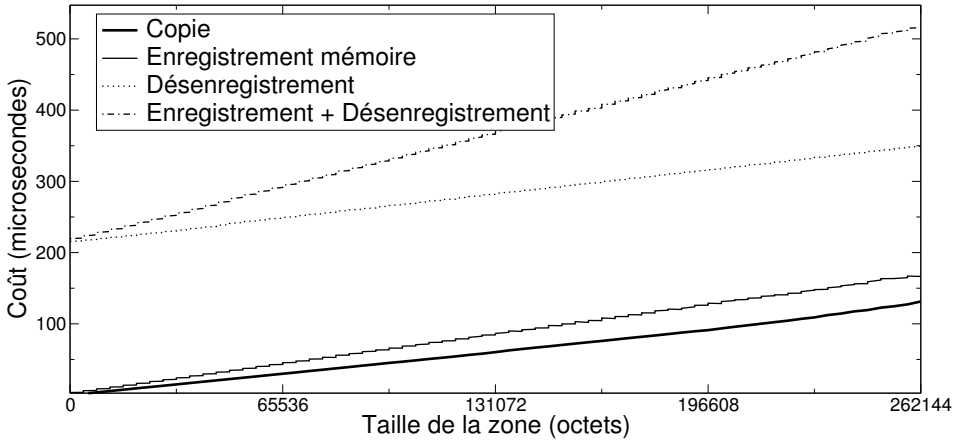


FIG. 5.1: *Comparaison des coûts de l'enregistrement mémoire dans* GM *et d'une copie mémoire. Les mesures ont été réalisées sur une machine équipée de deux processeurs* PENTIUM 4 XEON *cadencés à 2,6 GHz.*

nisme d'enregistrement.

La stratégie du cache d'enregistrement mémoire (voir en 2.2.3.d) semble particulièrement adaptée à GM puisque c'est le coût du désenregistrement qui est prohibitif. Tandis que la carte stocke les traductions d'adresses, ce cache conserve dans la mémoire de l'hôte la liste des zones enregistrées ainsi que leur nombre d'utilisateurs actuels. Ainsi, au lieu de désenregistrer les zones à la fin de leur utilisation, elles sont conservées dans le cache pour éviter le coût d'un désenregistrement et d'un ré-enregistrement plus tard.

Cependant, cette stratégie impose de maintenir à jour le cache d'enregistrement vis-à-vis des éventuelles modifications de l'espace d'adressage de l'application. En effet, l'application n'étant pas consciente que certaines de ses pages ont été enregistrées à la volée par les bibliothèques sous-jacentes, elle peut modifier son adressage sans prévenir le gestionnaire du cache. C'est notamment le cas lorsqu'un *mapping* (projection) de fichier est modifié, en particulier lorsque la bibliothèque `malloc` utilise des mappings anonymes[1].

Si le cache n'est pas invalidé dans ce cas, la carte d'interface réseau continue à utiliser des traductions d'adresse virtuelle invalides. Elle peut donc lire et écrire des données dans des zones physiques qui ont pu être réutilisées par le système pour stocker des données critiques. Cette stratégie transparente d'enregistrement mémoire impose donc de suivre les modifications d'adressage de l'application en interceptant tous les appels susceptibles d'invalider des entrées du cache d'enregistrement.

[1] L'allocation de mémoire par `malloc` se fait soit en agrandissant le *tas* par l'appel-système `sbrk` pour les petites allocations, soit en créant un espace de mapping anonyme (mapping du fichier `!/dev/zero!` pour obtenir de la mémoire initialisée à 0) pour les grandes allocations.

5.1.2 Mise en œuvre de l'enregistrement mémoire dans ORFA

Nous détaillons l'implantation concrète de mécanismes de gestion de l'enregistrement mémoire dans le client ORFA. Les problèmes mis en jeu sont identiques à ceux rencontrés lorsqu'on développe un *middleware* du type MPI. En effet, là aussi, l'interface proposée à l'application ignore l'enregistrement mémoire. Il faut donc enregistrer à la volée en interceptant les appels de l'application qui nécessitent des transferts sur le réseau.

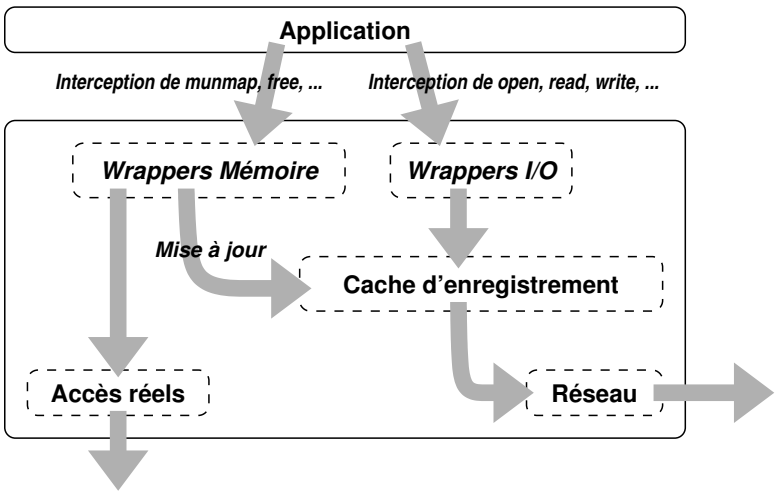


FIG. 5.2: *Mise en œuvre du cache d'enregistrement transparent dans le client* ORFA.

La figure 5.2 représente le schéma de fonctionnement de l'accès au réseau dans ORFA. Les entrées-sorties de l'application (`open`, `read`, `write`, ...) sont interceptées par la bibliothèque client préchargée. Les zones mémoire mises en jeu sont passées au cache d'enregistrement qui les enregistre si cela n'a pas encore été fait. Les routines susceptibles de modifier l'espace d'adressage de l'application (en particulier `munmap` et `free`) sont également interceptées pour invalider le cache avant de les traiter réellement.

Cette méthode fonctionne très bien mais son efficacité est fortement tributaire de la réutilisation des mêmes zones mémoire par l'application. La figure 5.3 présente les performances comparées du client ORFA utilisant des copies mémoire dans une zone statiquement préenregistrée, un enregistrement puis un désenregistrement à chaque requête et enfin un cache d'enregistrement (enregistrement lors du premier transfert uniquement). L'enregistrement systématique est beaucoup trop coûteux, même pour les gros messages, puisque moins de 75 % de la bande passante réseau est utilisée. L'utilisation des copies mémoire est intéressante pour les petites requêtes mais finit par écrouler les performances lorsque leur taille atteint 64 ko car la consommation processeur devient trop importante. L'impact du cache d'enregistrement mémoire est indéniable puisqu'il

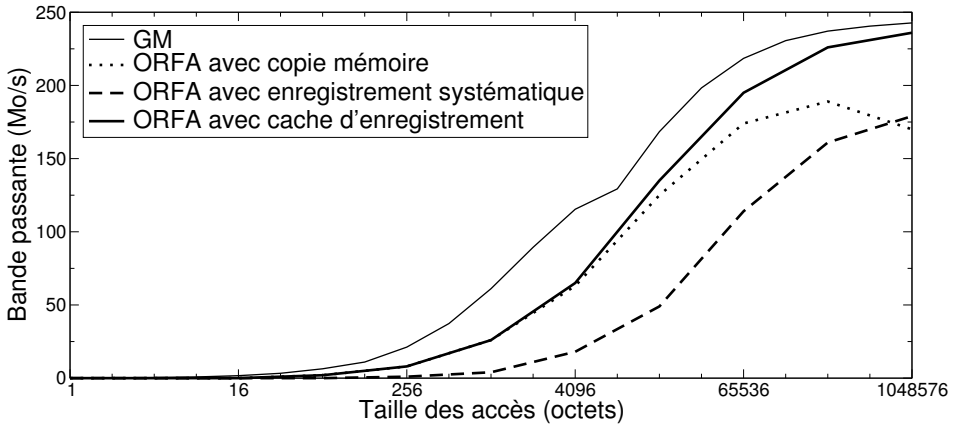


FIG. 5.3: *Performance comparée des accès aux données distantes avec* ORFA *sur* GM. *Un seul client accède séquentiellement à un fichier stocké dans le serveur* ORFA *en mémoire.*

permet d'utiliser quasiment toute la bande passante supportée par GM. C'est essentiellement le cas pour les requêtes larges puisque le coût du protocole ORFA devient négligeable devant le temps de transfert des données. Des expérimentations menées sur PVFS dans [WWP03a] confirment le gain important apporté par le cache d'enregistrement.

Par contre, une application provoquant beaucoup de défauts du cache d'enregistrement verra ses performances se rapprocher du cas sans cache, c'est-à-dire une chûte de 20 % de la bande passante.

Le suivi des modifications de l'espace d'adressage n'a pas d'impact sur les performances. On est donc en droit de se demander si la modification de l'interface standard d'accès aux fichiers pour s'affranchir de l'enregistrement mémoire est utile. En effet, elle rejette la quantité de travail développée dans le client ORFA vers le concepteur de l'application. Les performances obtenues seraient également tributaires de la réutilisation des mêmes zones mémoire dans l'application.

Les études menées sur DAFS dans [RI04] ont cependant montré que l'application parvient mieux à optimiser l'utilisation de la mémoire si la couche réseau lui fournit des indications, notamment la taille du cache d'enregistrement. Le développeur de l'application maîtrise son utilisation de la mémoire et peut alors l'organiser en tenant compte de ces indications. Un gain de 25 % en bande passante a ainsi été observé pour une application (base de données) sur DAFS. Des résultats similaires ont été obtenus avec MPI-IO sur DAFS [WP02]. L'implantation du cache d'enregistrement dans ADIO [2] avec assistance de la couche DAFS sous-jacente fournit les meilleures performances.

L'utilisation du réseau MYRINET dans notre implantation ORFA en espace utilisa-

[2] ADIO (*Abstract Device Interface for I/O*) est la couche d'accès aux fichiers dans l'implantation ROMIO [TGL99] de MPI-IO.

teur montre une bonne efficacité en utilisant le cache d'enregistrement mémoire. Sa mise en œuvre n'a pas présenté de problème majeur puisque les techniques d'interception et de cache d'enregistrement sont désormais bien connues. Ces techniques restent efficaces dans le contexte des accès aux fichiers distants.

Nous nous intéressons maintenant à la mise en œuvre de ces techniques dans le noyau LINUX.

5.2 Utilisation dans le cadre d'un système de fichiers distribués en espace noyau

Les principaux systèmes de fichiers distribués existants souffrent de la non-adaptation (ou non-disponibilité) des interfaces de programmation des réseaux dans le noyau. Par exemple, LUSTRE, qui utilise la couche de virtualisation PORTALS [BBS02, BRLM02], n'est disponible que sur très peu de réseaux haute performance (QSNET de QUADRICS et certaines versions d'INFINIBAND). Les premières implantations étaient disponibles sur GM mais utilisaient toujours des copies car l'interface d'enregistrement mémoire ne convenait pas à leurs besoins. Le support de GM a apparemment été abandonné depuis.

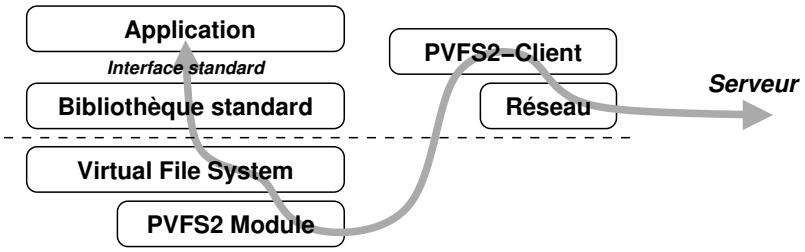


FIG. 5.4: *Mise en œuvre des accès aux fichiers distants depuis le noyau dans* PVFS2.

Les développeurs de PVFS2 ont décidé de ne pas accéder au réseau depuis l'espace noyau, mais au contraire, de remonter en espace utilisateur où un processus dédié se charge des communications (voir la figure 5.4). La documentation de PVFS2 justifie ce choix par l'**absence potentielle de support suffisant pour les accès au réseau depuis le noyau** : "*it is not clear that we will have ready access to all networking APIs from within the kernel.*" [PVF03].

Nous avons étudié précisément une implantation d'un client de système de fichiers distribués dans le noyau afin de voir dans quelle mesure il est possible de faire interagir efficacement et simplement l'interface de programmation réseau et les couches système d'accès au stockage. Cette étude est réalisée à travers le portage de notre client ORFA dans le noyau LINUX. Nous continuons à utiliser MYRINET/GM comme couche d'accès au réseau puisque nous souhaitons d'une part voir les besoins du stockage dans le noyau (pour répondre au problème de PVFS2) et d'autre part étudier précisément

les contraintes liées à l'interface GM dans le noyau (pour voir pourquoi LUSTRE ne la supporte plus).

5.2.1 Architecture de ORFS

ORFS (*Optimized Remote File-System*) est l'implantation équivalente à ORFA dans le noyau LINUX. L'objectif est de proposer les mêmes fonctionnalités qu'en espace utilisateur, c'est-à-dire un accès transparent et performant aux fichiers distants. Comme nous l'avons vu à la partie 2.1.1.d, le client se place sous la couche système du VFS (*Virtual File System*).

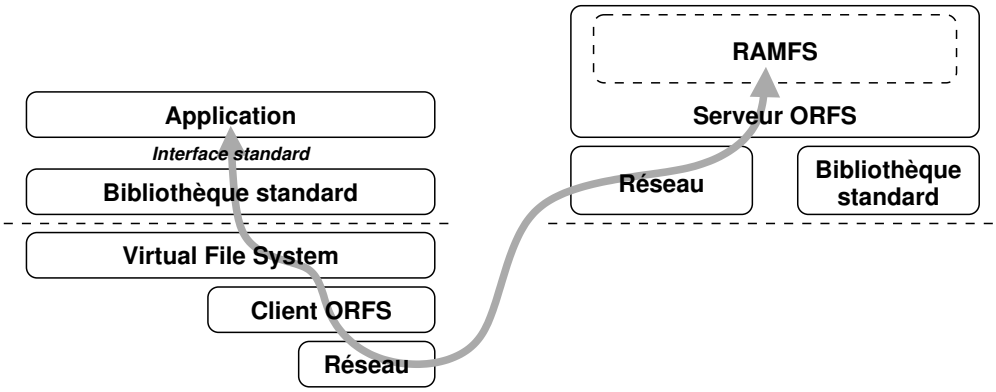


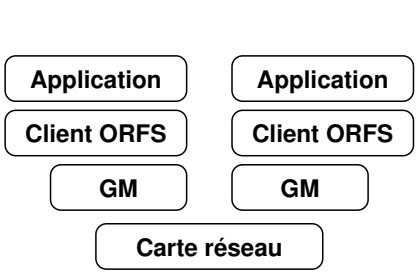
FIG. 5.5: *Modèle de* ORFS *accédant à un serveur de fichiers en mémoire (*RAMFS*).*

Nous détaillons ci-après la mise en œuvre des accès aux fichiers distants dans le client ORFS. Comme nous souhaitons mettre en évidence les problèmes d'interaction entre les couches système et l'interface de programmation réseau du côté client, il nous faut supprimer les goulots d'étranglement du côté serveur pour le rendre très performant. Nous utilisons là encore un système de fichiers en mémoire RAMFS similaire à celui utilisé dans ORFA (voir la figure 5.5).

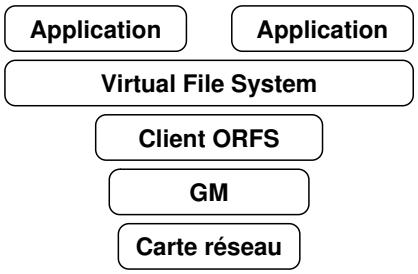
L'implantation du client ORFS comme système de fichiers intégré au VFS de LINUX permet à l'utilisateur de monter un système de fichiers distants (les fichiers stockés dans RAMFS) comme n'importe quel autre type de système de fichiers (EXT3, VFAT ou NFS par exemple). Les couches système, le VFS et la bibliothèque standard, se chargent de rendre l'accès à ce point de montage complètement transparent. L'interface standard d'accès aux fichiers est complètement respectée et aucune technique d'interception des appels à cette bibliothèque n'est nécessaire, contrairement à l'implantation de ORFA.

Cependant, contrairement au client ORFA en espace utilisateur qui pouvait intercepter de nombreux appels de l'application, le client ORFS ne peut pas savoir tout ce que les applications font. Le VFS n'informe ORFS que des accès aux fichiers, pas des autres activités. En effet, la mise en œuvre d'un nouveau système de fichiers dans le

noyau LINUX consiste à fournir un ensemble de routines que le VFS va appeler pour effectuer les traitements spécifiques. Aucune routine ne concernant les événements autres que les accès aux fichiers ne peut être spécifiée.



(a) Un port GM est utilisé pour chaque client ORFA et donc pour chaque application.



(b) Un port GM est utilisé pour chaque client ORFS, c'est-à-dire pour chaque montage ORFS, et donc pour toutes les applications y accédant.

FIG. 5.6: *Partage du réseau entre différentes applications clients accédant aux fichiers distants avec* ORFA *ou* ORFS.

Le premier problème qui se présente est le **multiplexage des canaux de communication** reliant les différentes applications clients à un même serveur. Le client ORFA utilise un canal pour chaque application (voir la figure 5.6(a)). Ce modèle n'est pas applicable à ORFS car le client ne saura pas quand l'application va terminer et ne pourra donc pas libérer le canal et les ressources associées. Les deux seuls objets dont ORFS est conscient sont les points de montage et les fichiers. Créer un canal de communication par fichier ouvert risque de consommer trop de ressources système. ORFS utilise donc un canal de communication vers le serveur distant pour chaque système de fichiers monté (voir la figure 5.6(b)). Sur le réseau IP le client ouvre une SOCKET depuis le noyau. Sur MYRINET, le client ouvre un *port* GM (un point d'entrée vers le réseau MYRINET, voir en annexe A.3.1) dans le noyau.

Le second problème qui se pose est celui des **transferts de données entre le réseau et le cache de pages** du système d'exploitation. Le noyau UNIX propose en effet essentiellement des accès aux fichiers à travers son cache de pages (*Page-Cache*). Les données sont échangées entre le cache de pages et l'application tandis que le système d'exploitation maintient en arrière plan son cache à jour vis-à-vis du serveur. Cette stratégie impose de transférer des données entre le cache et le serveur distant, ce qui impose des contraintes très différentes de l'utilisation traditionnelle du réseau dans ORFA en espace utilisateur. Nous détaillons cette mise en œuvre dans ORFS en 5.2.3.

Pour répondre aux besoins spécifiques des applications telles que les bases de données réparties ou le calcul *out-of-core*, une stratégie court-circuitant le *Page-Cache* a été

ajoutée dans les systèmes d'exploitation (voir en partie 2.3.1.b). Ces accès `O_DIRECT` se révèlent en fait très intéressants pour les applications parallèles. Nous commençons par présenter la mise en œuvre de cette stratégie dans ORFS.

5.2.2 Accès directs

Les accès `O_DIRECT` consistent à transférer directement les données depuis les zones mémoire de l'application vers le dispositif de stockage, c'est-à-dire le serveur distant dans le cas de ORFS. Ce modèle est très proche du modèle ORFA en espace utilisateur, mais il impose de manipuler les zones mémoire de l'application depuis l'espace noyau, et en particulier avec un port GM ouvert dans le noyau. Les techniques d'optimisation utiles dans ce cas devraient *a priori* être les mêmes que dans le client ORFA, c'est-à-dire le cache d'enregistrement mémoire. Cependant, comme nous l'avons vu en partie 5.1.1, il faut suivre les modifications de l'espace d'adressage de l'application pour maintenir le cache à jour, et ce depuis le noyau. Par ailleurs, il faut pouvoir gérer le multiplexage des canaux de communication. Cette mise en œuvre nécessite beaucoup plus de travail qu'en espace utilisateur.

5.2.2.a Partage de port GM dans le noyau

Les communications directes entre la mémoire de l'application et le serveur distant sont effectuées dans GM avec le port qui a été ouvert pour le point de montage correspondant aux fichiers cibles. Ce port est utilisé par toutes les applications accédant à des fichiers de ce point de montage. Comme toute structure du système d'exploitation, il est partagé entre les processus qui l'utilisent. Il faut donc être capable de manipuler simultanément des zones de mémoire de différents processus. Par exemple, une adresse virtuelle enregistrée par une application devra pouvoir être distinguée de la même adresse virtuelle d'une autre application. Un cache d'enregistrement effectuant ces distinctions a dû être implanté.

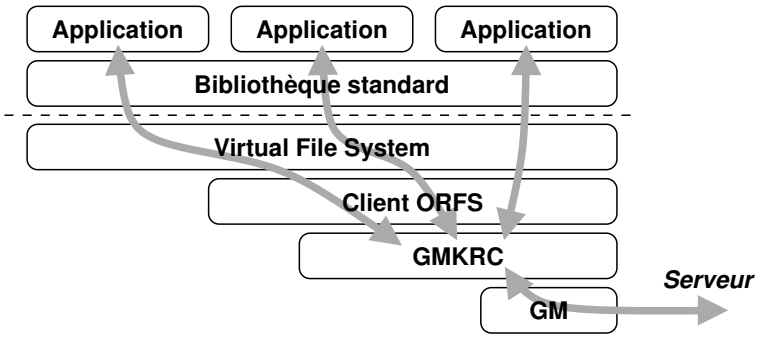


FIG. 5.7: *Partage de port* GM *entre les applications utilisant le client* ORFS.

À la différence du client ORFA en espace utilisateur où un port et un cache étaient maintenus pour chaque application, un port et un cache sont partagés ici entre toutes les applications accédant à même un point de montage. Nous avons choisi de ne pas inclure ces mécanismes dans le client ORFS mais dans un module noyau séparé nommé GMKRC (GM *Kernel Registration Cache*). Il propose une interface similaire à celle de GM à qui il transmet les requêtes réseau après avoir géré les entrées du cache d'enregistrement (voir la figure 5.7) correspondant aux données transférées.

La distinction des différents espaces d'adressage est en fait bien plus difficile qu'il n'y paraît car la carte d'interface MYRINET considère que GM n'autorise qu'une seule application par port. Les entrées stockées dans la table d'enregistrement de la carte MYRINET sont composées du numéro de port GM, d'une adresse virtuelle et de l'adresse physique associée. La carte est donc incapable de distinguer deux mêmes adresses virtuelles de deux espaces d'adressage différents si le port utilisé est le même. Pour résoudre ce problème, nous avons modifié le *firmware* de la carte, c'est-à-dire le microprogramme qu'elle exécute pour permettre cette distinction. Pour ce faire, le microprogramme a été compilé avec des pointeurs de taille supérieure et nous utilisons les bits supplémentaires pour identifier l'espace d'adressage cible. Dans notre implantation, les pointeurs passent de 32 à 64 bits et les premiers 32 bits contiennent un pointeur vers la structure de noyau qui décrit l'espace d'adressage (`struct mm_struct`).

Cette modification a un impact important sur les ressources stockées dans la carte d'interface car l'espace occupé par la table de traductions d'adresse est presque doublée. La taille limitée de la mémoire embarquée sur la carte impose de réduire soit le nombre d'entrées de traduction, soit les autres ressources, par exemple le nombre de descripteurs de communications en cours ou la taille de la table de routage.

Par contre, notre modification de la taille des pointeurs dans la carte n'est pas visible par l'application. C'est en fait GMKRC qui se charge de traduire les adresses normales en pointeurs agrandis. L'application, c'est-à-dire le client ORFS ici, utilise donc GMKRC comme il utiliserait normalement GM. C'est GMKRC qui met en place de manière transparente le cache d'enregistrement et la distinction des espaces d'adressage.

Il faut noter ici que le problème aurait pu être plus complexe sur un système d'exploitation où une même adresse virtuelle peut correspondre à des zones différentes en espace utilisateur et noyau. Ce n'est pas le cas des noyaux LINUX standard (par opposition aux noyaux *4G/4G*, voir en annexe D.1), mais ça l'est dans MACOS X et WINDOWS. Dans ce cas, en plus de l'espace d'adressage du processus, ce sont également les adresses virtuelles utilisateur et noyau qu'il nous aurait fallu distinguer. Nous reviendrons sur ce problème en partie 7.2.1.

5.2.2.b Suivi des modifications d'espace d'adressage

GMKRC met en œuvre un cache d'enregistrement mémoire au dessus de GM. Il est nécessaire de maintenir ce cache à jour vis-à-vis des modifications de l'espace d'adressage des applications. GMKRC doit donc être informé de ces changements.

Cependant, le client ORFS ne peut pas être informé de ce genre d'événement. En effet, le noyau LINUX permet de définir des routines devant être appelées lors de certains événements : des *hooks*. Mais ce mécanisme a été conçu pour que seul le créateur d'un objet puisse en définir les routines. Ainsi, si le client ORFS peut préciser les routines de traitement des objets logiques décrivant un montage ORFS, notamment les fichiers (voir en annexe B.3), il ne peut pas en préciser pour les objets du gestionnaire mémoire. Les changements d'espace d'adressage correspondent à des modifications de VMA (*Virtual Memory Area*, les objets composant cet espace). Seul le créateur d'un VMA peut être informé de sa modification.

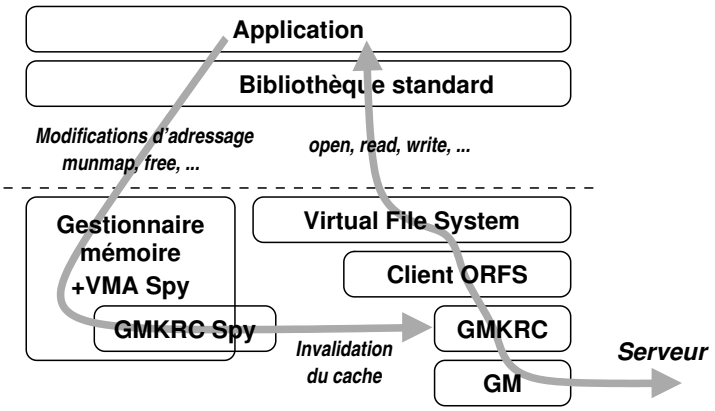


FIG. 5.8: *Gestion des accès directs dans* ORFS *avec* GMKRC *et* VMA SPY.

Aucune stratégie de type interception n'étant possible dans le noyau LINUX, il nous a fallu mettre en place un mécanisme dédié pour permettre à des systèmes externes d'être notifiés d'événements par le gestionnaire mémoire. Dans notre cas, c'est GMKRC qui demande à être informé des modifications des espaces d'adressage. Le mécanisme VMA SPY que nous proposons consiste en des *hooks* dans le code de la gestion mémoire de LINUX permettant à des *espions* (SPY) d'être informés de certains événements particuliers comme la modification d'un espace d'adressage. GMKRC définit un espion dédié qui, pour chaque VMA contenant des pages enregistrées, demande à ce que la modification du *mapping* de cette VMA provoque l'invalidation du cache d'enregistrement (voir la figure 5.8).

Cette mise en œuvre est détaillée en annexe E. Elle a l'avantage d'être extrêmement peu coûteuse puisque d'une part les modifications d'espace d'adressage sont rares, et d'autre part seul un test supplémentaire est ajouté dans le code normal du noyau. Par contre, l'ajout de ce mécanisme dans le code du noyau LINUX impose ici de le modifier.

L'idée du *patch* VMA SPY est en fait assez proche du patch utilisé par QUADRICS pour mettre à jour la MMU de la carte d'interface (voir en 2.2.4.c). L'implantation est par contre assez différente puisque nos espions surveillent des VMA tandis que les leurs surveillent la totalité de l'espace d'adressage d'un processus. Nous n'avons pas pu étudier le comportement réel des deux modèles avec des applications réelles. Mais

une étude qualitative laisse supposer que le nombre de VMA contenant des pages enregistrées est souvent très limité. L'utilisation de l'espion pour surveiller l'intégralité de l'espace d'adressage conduirait à de nombreux appels inutiles. Cependant, le coût de ces appels est probablement assez faible puisqu'il s'agit uniquement de savoir si les pages cibles sont enregistrées ou non. Nous avons discuté de ces idées avec les développeurs de QUADRICS qui souhaitent voir un patch de ce type intégré au noyau LINUX. Cela leur permettrait de ne plus imposer à leurs clients d'utiliser un noyau modifié pour manipuler la MMU des cartes QSNET, et cela nous permettrait de maintenir GMKRC à jour sans modification du noyau.

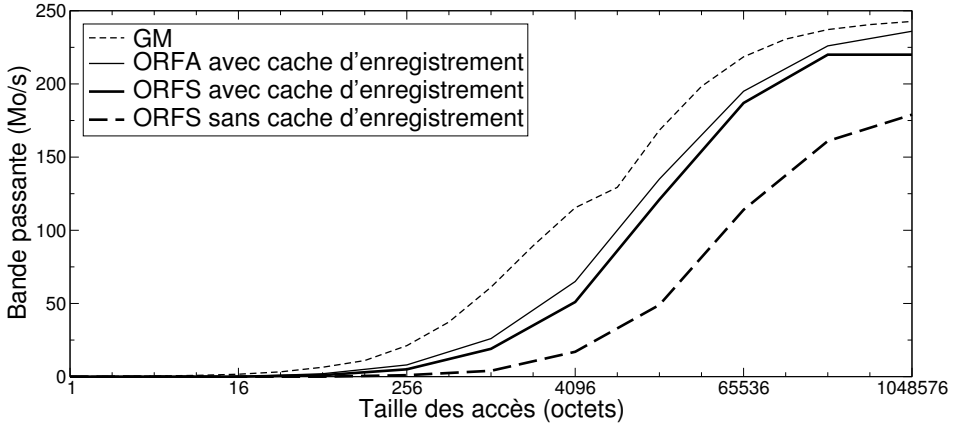


FIG. 5.9: *Performance comparée des accès directs aux données distantes avec* ORFA *et* ORFS *sur* GM.

Les performances des accès `O_DIRECT` aux fichiers distants par ORFS sont présentées sur la figure 5.9. Elles montrent, tout comme pour l'implantation d'ORFA en espace utilisateur, l'apport très net du cache d'enregistrement (avec GMKRC et VMA SPY) dans le cas où le nombre de défauts de cache est faible. ORFS se montre légèrement moins efficace que ORFA car sa mise en œuvre impose la traversée des couches système (appel-système et VFS). Ces transferts zéro-copie entre les zones mémoire de l'application et le réseau permettent dans le cas de grosses requêtes, d'accéder aux fichiers distants très efficacement (les accès en écriture présentent en fait des performances très similaires).

5.2.3 Accès par le cache

Les accès `O_DIRECT` permettent à l'application de maîtriser les accès réels au dispositif de stockage en évitant les transferts via le cache du système d'exploitation. Cependant, dans le cas de petites requêtes ou d'accès répétitifs aux mêmes données, le coût de la traversée du réseau devient prohibitif. Nous l'avions constaté en partie 4.2.3 lors de l'étude des accès aux métadonnées avec ORFA, cela reste valable avec ORFS. Par

exemple, la bande passante utilisée lors d'accès `O_DIRECT` de 256 octets sur ORFS est de 5 Mo/s, c'est-à-dire 2 % de la bande passante du réseau. Dans ce cas, l'application a tout intérêt à utiliser un cache du côté client, même si cette stratégie peut nécessiter de maintenir la cohérence entre les différents caches si plusieurs clients modifient simultanément les mêmes fichiers. Nous présentons ci-après le support des accès distants aux fichiers depuis le cache de LINUX dans ORFS.

5.2.3.a Enregistrement mémoire des pages du noyau

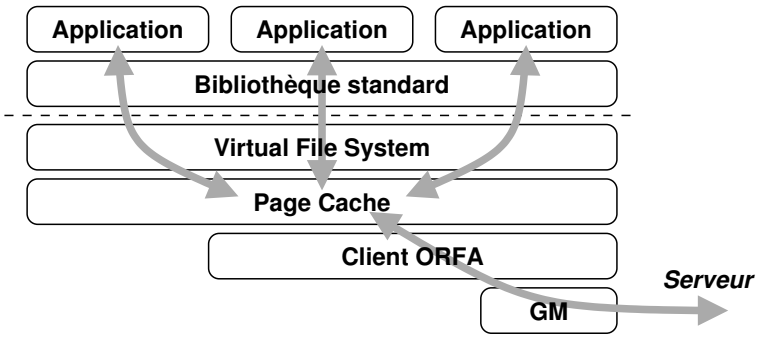


FIG. 5.10: *Accès aux fichiers distants par* ORFS *à travers le* Page-Cache *du système d'exploitation.*

Accéder aux fichiers distants à travers le cache du système d'exploitation consiste à transférer des données entre ce cache et la mémoire de l'application tandis que le système maintient le cache à jour vis-à-vis du serveur distant (voir la figure 5.10). Le modèle d'implantation d'ORFS dans ce contexte est donc identique au modèle `O_DIRECT` mais les communications doivent mettre en jeu des zones mémoire du noyau au lieu de celles des applications.

Le cache du système d'exploitation (c'est-à-dire le *Page-Cache* dans LINUX) est constitué de zones mémoire dont les caractéristiques sont très différentes de celles de l'espace utilisateur que nous enregistrions avec GMKRC. Il s'agit d'un ensemble de pages physiques qui ne sont pas forcément mappées en mémoire virtuelle. En effet, le système n'a pas besoin de les mapper tant que leurs adresses ne doivent pas être déréférencées. Il les mappe temporairement pendant les transferts de données puis les relâche pour libérer de la mémoire virtuelle (voir en annexe D.2).

L'enregistrement mémoire d'une page du cache imposerait de la mapper puis de donner l'adresse virtuelle générée et l'adresse physique correspondante à la carte d'interface réseau pour qu'elle retrouve l'adresse physique de la page au moment du transfert. C'est inutile ici puisque c'est l'adresse physique qui est initialement disponible. Par ailleurs, le noyau verrouille d'ores et déjà ces pages pendant les transferts de données. La stratégie de l'enregistrement mémoire ne semble donc pas du tout adaptée ici. Il est d'ailleurs important de se souvenir que l'enregistrement mémoire a été inventé

pour permettre aux applications des communications zéro-copie *OS-bypass* (voir en partie 2.2.3.d). L'*OS-bypass* n'a aucun sens ici puisque le client ORFS s'exécute justement dans le contexte du système d'exploitation !

Une solution simple consisterait à utiliser des copies dans des zones statiquement enregistrées. Cependant, les copies mémoire d'une page de 4 ko sont coûteuses et les utilisateurs d'application parallèles préfèrent souvent éviter ce genre de gaspillage de temps processeur puisque leurs applications ont besoin de ces cycles pour le calcul. Des travaux théoriques ont été menés dans [Bru99] puis [PDZ00] pour concevoir une sémantique claire de partage et de copie des pages mémoire dans les caches du type cache de fichiers ou les couches protocolaires d'accès au réseau. Ils ont abouti à des propositions de fusion du cache de fichiers du système d'exploitation et des zones mémoire intermédiaires utilisées par le réseau (par exemple les *Socket Buffer* pour TCP ou les zones préenregistrées pour GM). Cela peut permettre de transmettre directement des données entre le réseau et le cache de fichiers [WWPR04].

Cependant, ces stratégies basées sur l'enregistrement mémoire nous semblent beaucoup trop complexes compte tenu du contexte d'exécution dans le noyau et de la spécificité des pages mises en jeu. Nous présentons maintenant une alternative basée sur l'utilisation directe des adresses physiques des zones mémoire dans les communications.

5.2.3.b Utilisation des adresses physiques

Le *Page-Cache* de LINUX manipule l'ensemble des pages du système, que ce soit le cache des fichiers ou des pages de mémoire anonyme d'une application. Ces pages sont manipulées par l'intermédiaire de structures `struct page` qui décrivent les cadres de mémoire physique. Le VFS maintient des références vers ces structures lorsqu'elles sont mises en jeu dans un transfert, et les relâche ensuite pour que le système puisse éventuellement utiliser ces pages pour d'autres besoins. L'intégralité des routines du VFS manipule donc les éléments du cache de fichiers par la structure `page` qui les contient (voir en annexe B.1). Comme le VFS verrouille ces pages pendant les transferts de données (soit vers l'application, soit vers le dispositif de stockage), la carte d'interface réseau peut y accéder sans risque. Plutôt que d'adapter l'enregistrement mémoire à ces pages spéciales, nous avons modifié GM pour autoriser l'utilisation de ces structures via l'adresse physique associée dans les primitives de communication.

Nous avons ajouté aux traditionnelles primitives basées sur des adresses virtuelles enregistrées des primitives utilisant des adresses physiques (voir en annexe A.3.1). GM passe directement ces adresses à la carte d'interface qui pourra les utiliser immédiatement, sans avoir à parcourir sa table de traductions enregistrées. Il en résulte un léger gain en latence puisque le travail de la carte d'interface est moindre. Nous avons mesuré un gain d'environ 0,8 μs de chaque côté (émetteur et récepteur) sur nos machines. L'utilisation de cartes MYRINET plus récentes (PCIXD) diminue ce gain à 0,4 μs (voir la figure 5.11). Cela a également nécessité la modification du *firmware* pour prendre en compte ces nouvelles primitives.

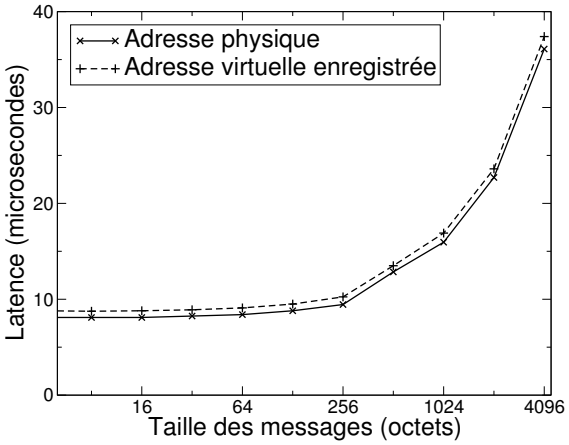


FIG. 5.11: *Latence des communications par adresses physiques au niveau* GM *et des communications traditionnelles par adresse virtuelle enregistrée sur des cartes* MYRINET PCIXD.

Ce gain en latence est en fait peu significatif sur les performances d'ORFS puisque, comme nous l'avons vu en 4.3, la latence a peu d'impact sur les performances de l'accès aux fichiers distants. Par contre, il faut bien voir que les performances fournies par notre stratégie ne peuvent être inférieures à celles d'une méthode basée sur l'enregistrement mémoire puisque nous avons réduit le chemin critique à son minimum.

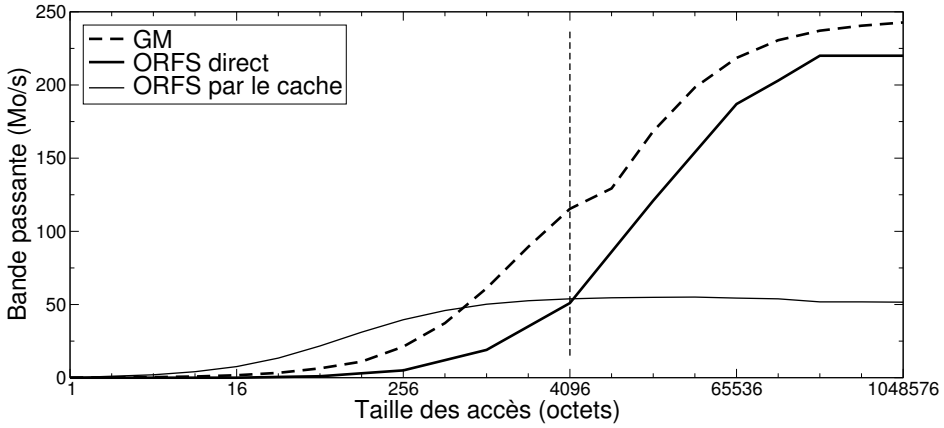


FIG. 5.12: *Performance comparée des accès aux données distantes dans* ORFS *sur* GM *avec et sans l'intervention du cache.*

La figure 5.12 présente les performances d'accès aux fichiers distants par le cache de LINUX sur ORFS. Il s'agit ici des performances vues de l'application, c'est-à-dire au dessus du cache de fichiers. Vu de l'application, il ne s'agit que de copies mémoires depuis ce cache. Le VFS de LINUX maintient ce cache à jour en tâche de fond en utilisant

une requête au serveur distant pour chaque page du cache (4 ko). Quelle que soit la taille des accès demandés par l'application, chaque accès par le cache se traduit par un ensemble de requêtes de la taille d'une page. La bande passante réseau générée par des messages de cette taille est assez mauvaise (115 Mo/s dans GM).

Par contre, chaque accès direct consiste en une seule requête échangeant directement les données entre le réseau et l'application. L'utilisation réseau est alors bien meilleure. Il est donc difficile de comparer directement les performances observées à travers le cache avec celles des accès directs ou les performances brutes de GM.

On peut constater sur la figure que lorsque l'application fait des accès de la taille d'une page, c'est-à-dire quand ORFS utilise un accès réseau pour chaque requête de l'application (que ce soit un accès direct ou par le cache), les deux types d'accès ont des performances très proches vues de l'application. En fait, le temps d'un accès par le cache est de 76 μs contre 80,3 pour les accès directs. Cependant, dans le premier cas, les accès incluent un appel système et la traversée des couches système (environ 1 μs en tout) puis la copie des données entre le cache de fichiers et l'application (environ 2 μs sur nos machines). Le temps réel de traitement de la requête réseau utilisée pour mettre à jour une page du cache n'est donc en fait que de 73 μs. Cela représente donc un gain d'environ 7 μs par rapport à une requête directe avec enregistrement mémoire.

L'obtention de performances réellement intéressantes à travers le cache nécessite que le système d'exploitation soit capable de regrouper les requêtes destinées au serveur distant, en lisant par exemple plusieurs pages à l'avance (*Read-Ahead*) ou en combinant les écritures différées (*Write-Back Caching*). Ces stratégies ne sont pas disponibles nativement dans les noyaux LINUX 2.4. Certains systèmes regroupent eux-mêmes les accès du VFS, par exemple NFS qui implante le mécanisme de *Read-Ahead* comme nous l'avons vu en 4.2.4. Mais ce type d'implantation est assez complexe et peu intéressante pour notre étude. Nous n'avons donc pas mis en œuvre ces techniques dans ORFS.

LINUX 2.6 fournit désormais aux systèmes de fichiers de tels accès agrégés (voir en annexe B.2). Il est alors possible d'utiliser beaucoup plus efficacement le réseau puisque les messages transférés sont bien plus gros. Par contre, cela impose de disposer de primitives de communications vectorielles, ce que GM ne propose pas. Nous n'avons donc pas étudié ce type d'accès ici mais nous y reviendrons en partie 7.1.3.d. Les besoins en communications vectorielles sont en effet accrus par l'utilisation directe de l'adresse physique, mais la stratégie habituelle basée sur l'enregistrement mémoire ne permet pas non plus de les éviter.

Nous n'avons pas présenté ici d'accès en écriture puisque les performances vues de l'application dépendent uniquement des copies mémoire vers le cache. Les transferts ORFS vers le réseau sont en effet effectués entièrement en tâche de fond par le système d'exploitation (*Write-Back Caching*). Les requêtes ORFS en écriture présentent les mêmes performances qu'en lecture, et le besoin en accès vectoriels est identique puisque le système va chercher à combiner les écritures.

5.3 Synthèse

Nous avons étudié dans cette partie l'utilisation concrète des réseaux MYRINET et de leur interface GM pour transférer des données dans un système de stockage distribué. Il s'agissait donc de répondre à la question Q2 : **comment les mécanismes des réseaux haute performance peuvent-ils améliorer le transfert de données lors de l'accès aux fichiers distants ?**

Nous nous sommes concentrés sur le côté client et avons insisté sur les problèmes d'enregistrement mémoire puisque c'est la stratégie généralement utilisée pour mettre en place des communications zéro-copie. Ces résultats ont été publiés dans [A].

L'étude en espace utilisateur dans le client ORFA en partie 5.1 a montré que les stratégies habituellement utilisées dans des *middlewares* du type MPI fonctionnaient également bien pour mettre en œuvre des accès aux fichiers distants performants. **L'utilisation du cache d'enregistrement montre tout son intérêt** en particulier pour les grosses requêtes pour lesquelles la bande passante est très bien utilisée.

Nous avons détaillé la mise en œuvre de ces techniques dans le système d'exploitation LINUX pour répondre aux besoins des systèmes de fichiers distribués qui se trouvent généralement dans le noyau. Les problèmes rencontrés lors de la mise en œuvre des communications dans le noyau ont été beaucoup plus nombreux. Pour les accès directs (évitant le cache), il a tout d'abord fallu **partager l'utilisation d'un même port de communication** GM par tous les processus accédant aux fichiers ORFS. Cela a notamment nécessité de **modifier le programme de contrôle de la carte MYRINET pour agrandir les pointeurs d'adresses** puis de **modifier le noyau LINUX pour autoriser le cache d'enregistrement à être notifié des modifications d'espace d'adressage**.

Les accès indirects à travers le cache de fichiers ont ensuite nécessité de **modifier l'interface de programmation GM pour pouvoir gérer la spécificité des zones mémoire** mises en jeu. Plutôt que d'adapter les transferts depuis des zones du cache de pages au mécanime de l'enregistrement mémoire de GM, nous avons choisi d'utiliser une **stratégie basée sur l'utilisation directe de l'adresse physique**. En effet, les pages du cache de fichiers sont déjà verrouillées en mémoire physique et l'adresse physique est connue. Il nous a fallu modifier l'interface de programmation et le firmware de GM pour rendre ceci possible.

Ce modèle se révèle très facile à utiliser et apporte même un gain en latence. Ses performances sont limitées par l'implantation du cache lui-même dans le système puisqu'une requête de l'application est divisée en autant de requêtes au serveur distant qu'il y a des pages à mettre à jour dans le cache. Mais il ne faut pas oublier que c'est l'application elle-même qui choisit si ses accès doivent être cachés ou non. Le cache est souvent utile pour les petites requêtes mais moins pour les grandes. C'est à l'utilisateur de tenir compte de ces contraintes dans son application.

Toutes les modifications que nous avons dû apporter à GM et au noyau LINUX montrent comme **il peut être difficile de faire interagir les couches système d'accès aux fichiers et les interfaces de programmation des réseaux des grappes** (Q2). Par contre, les performances ainsi obtenues sont très intéressantes. Nous utilisons ces conclusions

dans le chapitre 7 pour proposer un certain nombre d'améliorations aux interfaces de programmation existantes. Nous y étudions notamment les problèmes d'adressage mémoire et les besoins en communications vectorielles.

Nous n'avons pas étudié le troisième type d'implantation de client de stockage distribué, le *Network Block Device* (voir en partie 2.1.1.e). Ses besoins sont en fait très proches de ceux d'un système de fichiers distribués utilisant le cache du système d'exploitation.

Les transferts de données du côté serveur n'ont pas non plus été examinés car c'est du côté client que les contraintes sont les plus fortes, notamment en raison du respect de l'interface standard de programmation. Cependant, nous nous attendons à ce que les besoins du côté serveur, que ce soit dans une implantation en espace utilisateur ou dans le noyau, soient couverts par l'étude que nous avons présentée ici.

Les autres interfaces réseaux supportant les communications depuis le noyau ne souffrent pas toutes de problèmes similaires à GM. Lorsque l'enregistrement mémoire renvoie un identifiant unique de zone mémoire (notamment dans KCOMM de QSNET ou les VERBS INFINIBAND, voir en annexes A.3.4 et A.3.5), les collisions entre espaces d'adressage de différents processus (voir en 5.2.2.a) peuvent *a priori* être facilement évitées. Par contre, le problème du suivi des modifications d'espace d'adressage subsiste tant que le système d'exploitation n'a pas été modifié pour notifier la couche réseau (soit avec notre patch VMA SPY, soit avec le patch de QSNET). Le support des zones mémoire spéciales telles que le *Page-Cache* peut utiliser un enregistrement mémoire spécifique (dans les VERBS) ou nécessiter un mapping en mémoire virtuelle (dans KCOMM). Mais nous pensons que ces techniques sont beaucoup trop complexes pour ce problème finalement assez simple.

Chapitre 6

Optimisations du contrôle des communications

Nous venons d'étudier au chapitre 5 les problèmes liés au transfert de données lorsqu'on utilise les interfaces de programmation des réseaux haute performance dans le contexte du stockage distribué. Nous traitons ici les problèmes de contrôle mis en évidence dans notre étude préliminaire au chapitre 4. Le modèle client-serveur utilisé dans les systèmes de stockage auxquels nous nous intéressons présentent en effet des besoins différents des applications parallèles en matière de contrôle.

Nous présentons tout d'abord une étude des messages inattendus dans les réseaux haute performance afin de voir l'impact du trafic hiérarchisé du stockage distribué qui est très différent du trafic homogène des communications entre les nœuds d'une application parallèle. Nous traitons donc la question Q3 : **le trafic généré par l'accès distant aux fichiers est-il bien géré par le contrôle des communications des réseaux des grappes ?**

Nous étudions ensuite les problèmes de notifications d'événements pour mettre en évidence les limites de l'interaction entre l'interface de programmation classique des entrées-sorties et l'interface asynchrone des réseaux haute performance. Il s'agit alors de la question Q4 : **peut-on mixer les notifications d'événements des réseaux rapides avec les événements traditionnels du stockage ?**

6.1 Trafic hiérarchisé et messages inattendus

Nous avons présenté en partie 2.2.2 les différentes innovations qui ont conduit au succès des réseaux haute performance dans les grappes. En dehors des traditionnels mécanismes permettant les communications zéro-copie traitées dans la carte d'interface, d'autres idées moins célèbres ont également permis une amélioration notable des performances.

Les réseaux et protocoles traditionnels ont été conçus pour permettre à n'importe quelle machine de la planète de communiquer, quelque soit la topologie du réseau. Cela permet une très grande flexibilité mais implique un très grand coût puisqu'il faut être capable d'une part, de faire interopérer un très grand nombre de machines, et d'autre

part, de pouvoir réagir aux phénomènes de congestion dans les goulots d'étranglement. La grande dynamicité de la topologie de l'internet impose un algorithme de routage adaptatif. La grande probabilité de perte de paquets impose quant à elle des mécanismes efficaces de reprise sur erreur.

Ces deux idées ne sont pas valables dans les réseaux interconnectant les grappes de calcul. Tout d'abord, la topologie de ces grappes étant fixée et regulière, la table de routage peut être fixée au début de la session tandis que le routage par la source assure la propagation des paquets à moindre coût. Mais surtout, le cœur de réseau y est suffisamment dimensionné pour qu'un trafic régulier entre toutes les extrémités du réseau ne subisse aucune congestion. Il arrive bien entendu qu'un lien, commutateur ou nœud tombe en panne, mais le taux de panne est très faible. Le protocole de transport dans les couches bas niveau de gestion du réseau est donc mis en place autour du cas sans erreur, tandis que la reprise sur erreur est un cas particulier peu optimisé.

6.1.1 Répartition du trafic dans les grappes

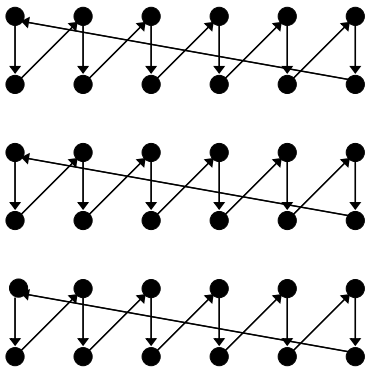


FIG. 6.1: *Exemple de trafic uniforme entre les nœuds d'une application parallèle.*

Les applications parallèles ont généralement un trafic assez uniforme à travers la grappe. En effet, une bonne parallélisation d'un problème nécessite d'une part de répartir des charges de calcul similaires sur les machines, mais aussi de faire en sorte que le trafic réseau soit bien réparti. Ainsi, les goulots d'étranglement sont évités, que ce soit en terme de puissance de calcul ou en terme de capacité des liens [GGKK94] (voir la figure 6.1).

L'utilisation de ces réseaux dans un modèle de stockage distribué ne génère pas le même trafic. En effet, les modèles client-serveur ont intrinsèquement un trafic centralisé autour du serveur, ce qui crée un goulot d'étranglement. L'utilisation de système de fichiers parallèles a été initialement proposée pour répartir la charge de traitement sur différents serveurs. Cette idée a également l'avantage de diffuser la charge réseau sur les liens des différents serveurs. Le problème consiste à dimensionner correctement le système puisqu'un trop grand nombre de clients forme là encore un goulot au niveau

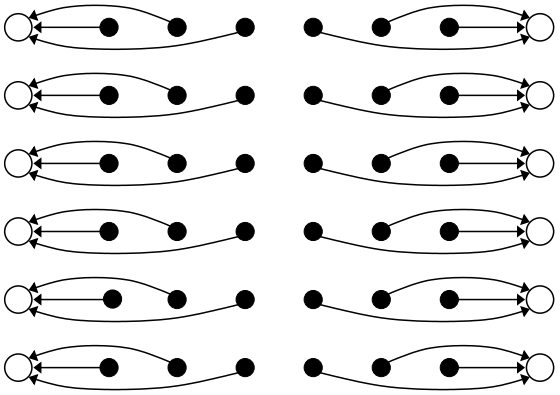


FIG. 6.2: *Exemple de trafic engendré par une grappe de 36 machines accédant aux fichiers répartis sur 12 serveurs.*

des serveurs (voir la figure 6.2). Il faut noter que la parallélisation de chaque requête des clients sur l'ensemble des serveurs ne modifie pas la charge de chacun des liens des serveurs.

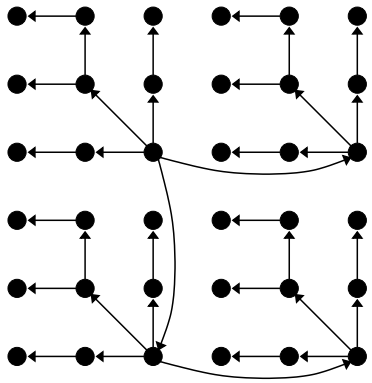


FIG. 6.3: *Exemple de trafic engendré par la prise d'un verrou dans un système de stockage partagé sans serveur. Un algorithme binaire est utilisé pour propager la prise du verrou aux n nœuds en $log(n)$ étapes en diffusant progressivement le trafic réseau.*

Le seul modèle où ce problème est évité est celui du stockage partagé sans serveur (voir en partie 2.1.3.b). Dans ce cas, le réseau d'interconnexion de la grappe est utilisé par les clients pour se synchroniser. La qualité de ce modèle distribué sans serveur est alors directement liée à la capacité des algorithmes de synchronisation à ne pas être centralisés (voir la figure 6.3). Ce problème a beaucoup été étudié et des solutions performantes ont été mises en œuvre, par exemple dans GPFS où les performances ne semblent pas souffrir du nombre de nœuds [SH02].

Les problèmes de contrôle de flot sont donc *a priori* moindres dans ces systèmes sans

serveur que dans les modèles client-serveur tels que PVFS ou LUSTRE sur lesquels se concentre notre étude.

6.1.2 Impact des messages non-reçus

Nous revenons maintenant sur les problèmes rencontrés en partie 4.2.5. Nous avions observé que les performances du serveur diminuaient quand le nombre de clients dépassait environ 100. Ce phénomène est étrange puisque, en règle générale, quand le nombre de clients augmente alors que le serveur est déjà chargé au maximum, seule la latence de traitement des requêtes augmente. Le serveur continue à traiter autant de requêtes par unité de temps. Cependant, les requêtes attendent plus longtemps avant d'être traitées.

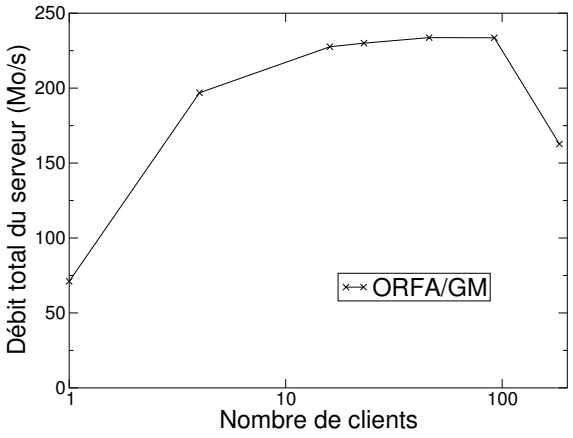


FIG. 6.4: *Impact de l'augmentation du nombre de clients sur la bande passante utile du serveur* ORFA *sur* GM. *Les clients* ORFA *soumettent des requêtes d'écriture de 64 ko tandis que le serveur ne peut pas préparer plus de 125 zones de réception simultanées.*

La figure 6.4 reprend les valeurs incriminées. On mesure la bande passante traitée par le serveur ORFA lorsque les clients lui envoient simultanément des requêtes d'écriture de 64 ko. Avec 92 clients, 233 des 250 Mo/s disponibles sur le réseau MYRINET sont utilisés. Par contre, avec 184 clients, seuls 163 sont apparemment traités.

Le phénomène qui se produit ici est un gaspillage de la bande passante utile par les messages non-reçus. En effet, le serveur doit préparer des zones de mémoire pour recevoir les données que lui passent les clients. Comme souvent dans un serveur, on ne sait pas quand un client va émettre une requête. Dans un modèle de *Rendez-vous* tel que celui proposé par l'interface GM utilisée ici dans ORFA, le serveur doit préparer à l'avance ces zones en espérant en préparer suffisamment si le nombre potentiel de clients est grand. Cette caractéristique n'est pas spécifique à GM. En effet, aucune interface de programmation de réseau rapide ne permet de préparer un nombre illimité de zones de réception puisque les descripteurs associés doivent être stockés dans

la mémoire de la carte d'interface réseau, qui est très limitée. Une solution consiste à conserver les descripteurs de ces zones dans la mémoire de l'hôte tandis que la carte viendra les chercher. Mais la traversée du bus d'entrées-sorties nécessaire pour ce faire est très coûteuse, ce qui risque d'augmenter très sensiblement la latence.

Les cartes MYRINET proposent en général 2 Mo de mémoire embarquée. Elle est utilisée pour stocker les informations de routage, la table de traductions d'adresse pour la mémoire enregistrée et les descripteurs de requêtes d'émission et réception. GM permet la préparation de 250 zones de réception. Notre implantation du serveur les répartit en zones de réception des descripteurs de requête et en zones de réception des données à traiter. Le serveur ne peut potentiellement recevoir que 125 requêtes d'écriture simultanées. Que faire de la 126ème requête puisqu'aucune zone de réception n'est disponible pour la recevoir ?

En cas d'erreur (paquet perdu ou corrompu sur le réseau), la stratégie usuelle consiste à le réémettre. Mais, le fait qu'aucune zone de réception ne soit disponible est un problème spécifique au modèle de programmation de type *Rendez-vous*. La stratégie utilisée par GM consiste à réémettre là aussi le paquet. Ce choix peut s'appliquer relativement bien aux communications entre nœuds d'une application parallèle car le comportement des différents nœuds y est généralement prévisible. Les zones de réception peuvent alors être préparées à l'avance. Par contre, dans un modèle client-serveur comme le notre, ce n'est pas le cas. Avec un nombre de clients supérieur au nombre de zones de réception, les messages vont être continuellement réexpédiés jusqu'à être reçus. Cela engendre un gaspillage de bande passante utile puisque les paquets réémis parviennent jusqu'à la carte d'interface réseau du serveur qui va les jeter si aucune zone de réception n'est disponible.

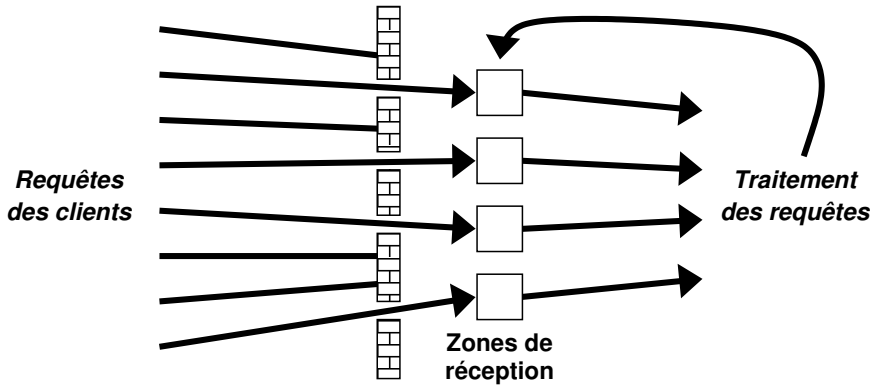


FIG. 6.5: *Gestion des multiples requêtes des clients dans le serveur où le nombre de zones de réception qui peuvent être préparées est limité. Les requêtes ne pouvant être reçues sont jetées.*

Dans notre expérience, 184 clients émettent des requêtes au serveur qui ne peut en recevoir que 125 (voir la figure 6.5). Il y a donc environ un tiers de messages (32,1 %) qui vont être jetés par la carte réseau, c'est-à-dire un gaspillage de un tiers de sa bande

passante utile. En fait, ces résultats doivent être modulés par la charge du serveur. En effet, si celui-ci parvient à traiter les requêtes assez rapidement, il pourra renouveler les zones de réception assez rapidement pour satisfaire les requêtes des autres clients en attente.

Dans notre expérience, la bande passante utile diminue de 30,4 % (ce qui signifie qu'un message est en moyenne expédié 1,44 fois avant d'être reçu). Puisque 32,1 % des messages ne peuvent théoriquement pas être reçus, cela signifie que le serveur n'est pas chargé au maximum parvient donc à recycler les zones de réception assez rapidement.

Ce problème est très lié au rapport de performance entre le réseau et l'hôte. En effet, si le réseau est beaucoup plus performant que le processeur central (ou ses entrées-sorties vers les périphériques de stockage), c'est la charge de l'hôte qui va limiter la réception des requêtes. La gaspillage de bande passante réseau sera alors d'autant plus grand que le processeur central sera saturé, mais également d'autant plus négligeable que la bande passante réseau sera grande. Par contre, si le réseau est beaucoup moins performant que le processeur central, le réseau sera le facteur limitant du serveur mais aucun gaspillage de bande passante ne sera observé. Il s'agit donc ici de bien équilibrer la puissance des différents composants matériels de la machine.

Il faut noter ici que le nombre de requêtes reçues par le serveur est plus fonction du nombre de processeurs dans la grappe que du nombre de nœuds. En effet, un même nœud SMP (*Symmetric MultiProcessing*[1]) peut émettre plusieurs requêtes simultanément si plusieurs threads ou processus s'exécutent simultanément (il est habituel de placer un processus MPI par processeur). Par ailleurs, avec l'avènement des interfaces asynchrones d'accès aux fichiers, il est désormais possible de soumettre de nombreuses requêtes simultanées dans un même processus. Dans ce cas, le serveur peut potentiellement recevoir un nombre énorme de requêtes (par exemple 40 000 sur une grappe de 1000 machines quadri-processeurs soumettant 10 requêtes par processus). Il est donc important de réagir efficacement à ce trafic surabondant en évitant le goulot d'étranglement que nous observons dans GM.

6.1.3 Protocole de gestion des messages inattendus

Nous pouvons envisager deux solutions pour éviter ce problème. La première consiste à modifier le protocole mis en place par l'interface de programmation du réseau. La seconde consiste à changer le protocole de haut niveau mis en place par l'application, ORFA ou ORFS dans notre étude, pour mieux tenir compte des faiblesses de l'interface sous-jacente dans le contexte étudié.

Nous avons montré dans la partie précédente que la gestion des messages inattendus mise en place par GM se révélait particulièrement mal adaptée à nos expérimentations alors qu'aucune faiblesse majeure n'a jamais été mise en évidence dans le cadre des communications entre les nœuds d'une application parallèle. Le trafic hiérarchisé

[1] Un compromis intéressant entre grappe de machines monoprocesseur et supercalculateur massivement parallèle consiste à utiliser une grappe de machines multiprocesseur (SMP).

utilisé par le stockage distribué, que le serveur soit parallélisé ou non, conduit nécessairement à un goulot d'étranglement au niveau du lien du serveur. Il peut donc être intéressant de voir dans quelle mesure on peut corriger le protocole de GM pour mieux satisfaire les besoins du stockage distribué sans pour autant négliger les besoins des applications parallèles. Ce problème des messages inattendus (*Unexpected messages*) est assez courant dans le domaine des applications parallèles. Il a par exemple été montré dans [BU04] que la puissance des cartes d'interface réseau existantes ne suffisait pas toujours à traiter efficacement les messages inattendus dans MPI. Les interfaces natives de programmation du réseau proposent à ce jour peu de support pour ces messages.

À défaut de pouvoir augmenter le nombre de zones de réception pouvant être préparées simultanément dans la carte d'interface (puisqu'il est limité par la quantité de mémoire embarquée dans la carte), il nous faut ici faire en sorte que le client ne réémette pas à l'aveugle les messages qui n'ont pas été reçus. Le client ne peut pas deviner à quel moment une zone de réception est disponible. Par ailleurs, un autre client peut utiliser cette zone avant lui. Il semble donc difficile de mettre en place un protocole efficace sans l'assistance du serveur.

On peut par exemple imaginer que l'interface réseau du serveur prévienne l'hôte qu'un message n'a pas pu être reçu. Dès qu'une zone de réception deviendra disponible, l'interface irait directement chercher les données sur le client. Ainsi, plus aucune bande passante n'est gaspillée, mais le protocole mis en place dans la carte réseau et son interface de programmation est plus complexe. Pour les petits messages, une solution intéressante consiste à déposer temporairement les messages dans une zone de mémoire intermédiaire sur le nœud cible. Lorsque la zone de réception réelle aura été préparée, une copie mémoire permettra d'obtenir le message immédiatement. Comme toujours, il faut trouver un compromis entre faible latence et utilisation du processeur central pour copier les données.

À défaut de support satisfaisant dans le protocole réseau natif, il est possible d'adapter l'application pour qu'elle traite elle-même le problème des messages inattendus. Dans les applications, cela s'est traduit par de nombreux travaux dans les implantations de MPI. Par exemple, l'implantation de MPI sur SISCI (voir en 2.2.4.a) impose d'utiliser des copies mémoire pour éviter de bloquer le nœud émetteur quand le récepteur n'est pas prêt à recevoir [HEH98]. Nous étudions maintenant précisément le déroulement des communications dans ORFA (ou ORFS).

Le protocole de communication entre le client et le serveur ORFA (voir la figure 6.6 et la partie 4.1.1) se décompose en l'envoi d'un descripteur de requête suivi d'éventuelles données à traiter (par exemple des données à écrire dans un fichier) puis le retour d'un descripteur de réponse suivi d'éventuelles données (par exemple les données lues dans un fichier). Le client n'a en fait besoin de n'être notifié que de l'arrivée de la réponse, pour pouvoir terminer le traitement de l'accès au fichier distant et libérer les ressources correspondantes. Par contre, le serveur doit être notifié de l'arrivée d'une nouvelle requête et également de la terminaison de l'envoi de la réponse pour libérer les ressources utilisées.

La notification doit inclure une notification pour le descripteur et une autre pour

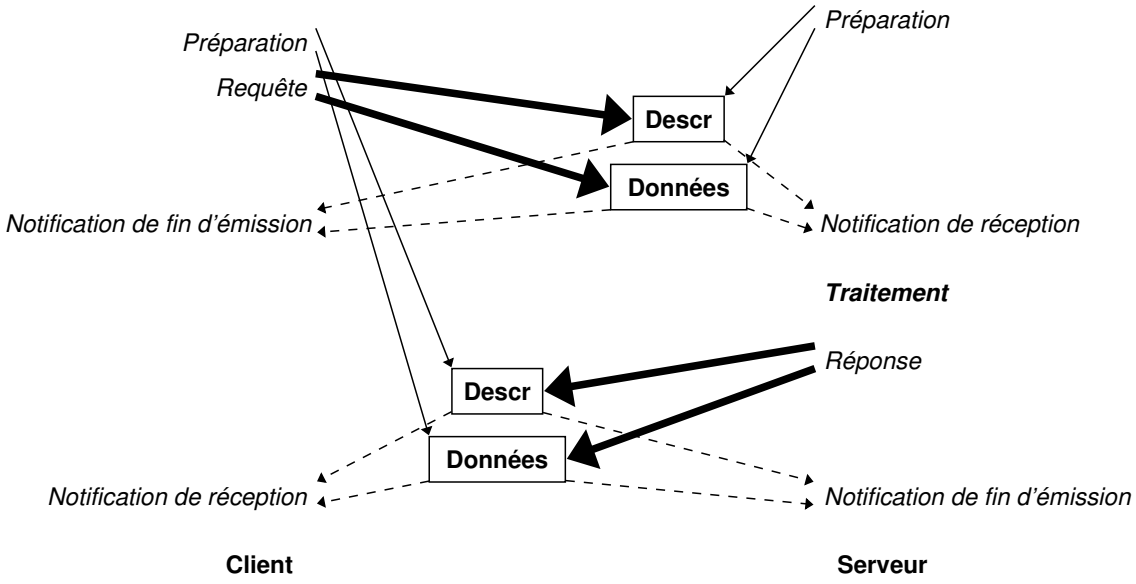


FIG. 6.6: *Fonctionnement du protocole* ORFA *sur le modèle du* Rendez-vous.

les données correspondantes (s'il y en a). Certains protocoles réseau assurent l'ordre d'arrivée des messages. Dans ce cas, si on envoie le descripteur après les données, la notification de l'arrivée du descripteur suffit. Mais en général, cet ordre n'est pas assuré pour tout type de message, il faut alors attendre les deux notifications.

Lorsque la notification n'est pas nécessaire sur le client et le serveur, le modèle du *Rendez-vous* peut être remplacé par des communications par RDMA (seul l'initiateur de l'accès mémoire à distance est notifié de sa terminaison). Ce modèle peut être mis en place au dessus de l'interface de passage de message GM en rajoutant des messages de contrôle dans le protocole ORFA. Mais cela augmenterait la charge du serveur qui est déjà un facteur limitant. Une solution plus intéressante et efficace consisterait à utiliser des communications par RDMA natif si l'interface de programmation le propose.

Ainsi, le client émet uniquement son descripteur de requête accompagné de l'identifiant de la zone mémoire où se trouvent les données à lire ou écrire. Le serveur vient chercher par RDMA les éventuelles données à traiter, traite la requête puis dépose les éventuelles données résultantes par RDMA dans la mémoire du client. Enfin, le serveur émet le descripteur de la réponse à la requête. Ainsi, les seules zones de réception qui doivent être préparées sont celles des descripteurs[2]. Celles contenant des données ne sont accédées que par RDMA initié par le serveur. Ce déport de l'initiative des transferts de données sur le serveur lui permet de maîtriser le flux entrant et donc d'éviter les goulots d'étranglement (voir la figure 6.7).

[2] Les zones mémoire accessibles par RDMA doivent également subir une préparation particulière mais leur nombre n'est généralement pas limité par l'interface de programmation ou la mémoire embarquée dans la carte d'interface réseau.

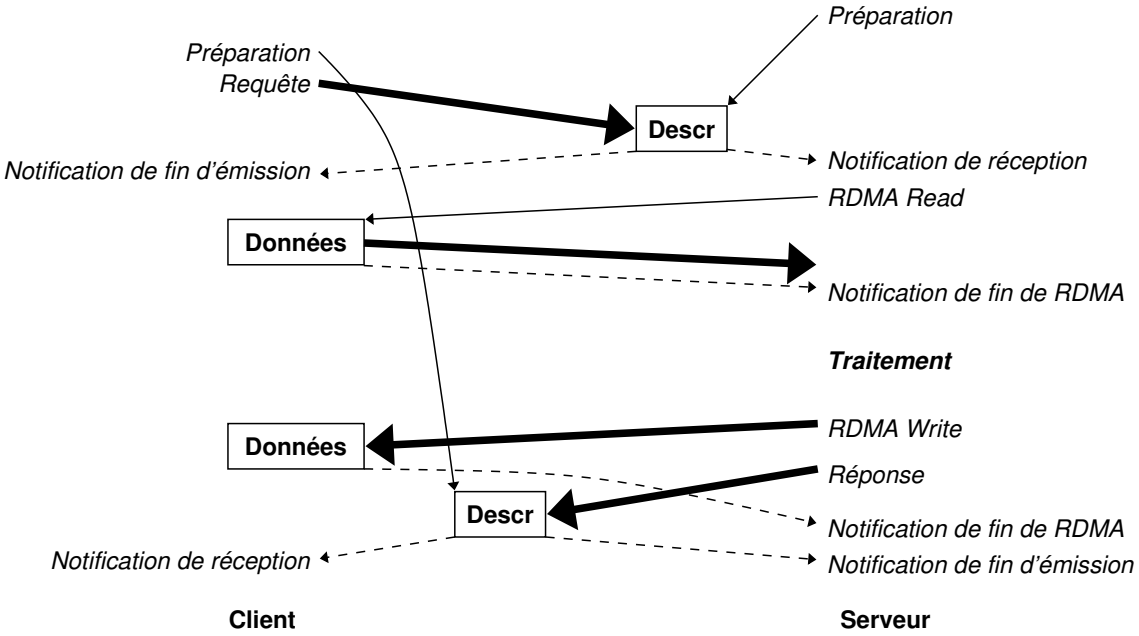


FIG. 6.7: *Fonctionnement du protocole* ORFA *avec des accès mémoire à distance.*

Nous n'avons pas pu tester cette mise en œuvre dans ORFA car le support des communications par RDMA dans GM est trop limité. Cependant, nous nous attendons à observer la disparition de la chûte des performances lorsque le nombre de clients augmente. Par contre, l'initiation des RDMA par le serveur peut engendrer une légère augmentation de latence et une très légère augmentation de la charge du serveur.

Il faut noter que le phénomène de gaspillage du débit réseau reste possible dans ce modèle si le serveur ne parvient pas à recevoir suffisamment vite les descripteurs des requêtes des clients. Cependant, étant donnée la taille de ces descripteurs, il faudra un nombre extrêmement élevé de client pour pouvoir observer une dégradation réelle du débit utile du serveur.

Il faut citer des travaux intéressants qui ont été menés dans DAFS pour permettre l'utilisation de RDMA initiés par le client [Mag02b]. Le principe est que le client tente de lire ou d'écrire directement dans le cache du serveur en espérant que les zones cibles soient toujours exportées. Si c'est le cas, le client n'a plus besoin de l'intervention du serveur pour accéder aux données. Sinon, une exception distante est provoquée et le client se rabat sur le protocole traditionnel guidé par le serveur.

6.2 Gestion des événements

Nous avons vu en partie 2.2.3.a que les réseaux haute performance utilisaient un modèle de programmation très particulier. Contrairement à la traditionnelle interface

SOCKET qui propose par défaut des primitives de communication bloquantes, les interfaces de programmation des réseaux rapides sont asynchrones. L'application soumet des requêtes qui sont traitées en arrière-plan par la carte d'interface pendant que l'application reprend ses calculs. Elle viendra plus tard tester la terminaison d'une requête et éventuellement bloquer en l'attendant.

Nous avons expliqué en partie 2.3.2 que l'utilisation d'un modèle de programmation si particulier pouvait rendre difficile l'interaction entre le réseau et le stockage. C'est ce que nous étudions tout d'abord en présentant les mécanismes de notification d'événements puis leur interaction avec l'interface standard d'entrées-sorties.

6.2.1 Notification des événements

Lorsqu'une application parallèle soumet plusieurs requêtes asynchrones de communication, elle peut souhaiter ensuite connaître individuellement leur état (1) ou bien savoir si l'une quelconque d'entre elles a terminé (2). La première méthode est utile lorsque l'application sait quelle requête va terminer en premier. La seconde sert lorsque plusieurs requêtes ont été soumises sans savoir laquelle va terminer en premier, notamment pour les requêtes de réception concernant différents nœuds.

Ces deux stratégies sont très souvent utilisées dans MPI mais les interfaces de programmation du réseau ne les proposent pas toujours. Il est possible d'implanter (1) en utilisant (2) mais cela peut imposer l'utilisation de threads, ce qui augmente la latence puisque les changements de contexte seront plus nombreux. Il est également possible d'implanter (2) en utilisant (1) mais cela nécessite des attentes actives qui gaspillent le temps processeur.

L'interface GM ne propose que la seconde méthode, c'est-à-dire que tous les événements de terminaison de requête d'émission ou réception sont renvoyés en espace utilisateur via une file unique par *port*. L'application ne peut donc récupérer que la première notification de cette file, c'est-à-dire l'événement correspondant à la requête qui a terminé en premier. Les autres événements ne peuvent être obtenus sans extraire tout d'abord ceux qui le précèdent dans la file. Nous détaillons maintenant comment ces stratégies sont utilisées dans les clients et serveurs ORFA et ORFS.

6.2.1.a Notification du côté serveur

Le rôle du serveur est de recevoir les requêtes des clients, de les traiter puis d'envoyer une réponse. Il s'articule donc autour d'une boucle principale d'attente d'événements. Lorsqu'une requête arrive du réseau, il sort de cette boucle pour la traiter puis y revient.

Les événements réseau attendus dans le serveur ORFA sur GM sont essentiellement les notifications de terminaison des réceptions. En effet, le modèle du *Rendez-vous* impose de préparer à l'avance des zones de réception puisqu'on ne peut pas prévoir quand un client va émettre une requête. Chacune des zones de réception correspond à une no-

tification attendue. L'attente d'événements se résume donc à une attente globale sur le port GM du serveur.

Chaque requête peut en fait potentiellement générer 4 événements (terminaisons de l'arrivée du descripteur de la requête ou des données associées, puis terminaisons de l'émission du descripteur de la réponse ou des données associées). La réactivité du serveur impose de ne jamais bloquer inutilement, c'est-à-dire lorsque d'autres travaux peuvent être effectués. Il est donc important de ne bloquer que lorsqu'aucune requête d'un client ni aucun événement réseau n'est en attente d'être traité. Par exemple, les données associées à un descripteur de requête peuvent ne pas être reçues immédiatement. Ou encore, l'émission de la réponse peut ne pas être traitée immédiatement par la carte d'interface. Il est donc important de ne jamais bloquer en attendant un événement particulier, mais au contraire, de recouvrir au maximum le traitement de toutes les communications réseau en maintenant un état pour décrire quelles terminaisons associées à chaque requête ont été signalées.

Ce modèle événementiel se satisfait parfaitement de l'interface GM puisqu'il ne nécessite qu'une primitive d'attente d'une terminaison quelconque. Nous verrons en partie 6.2.2 comment ceci s'articule avec le traitement des accès disques dans le serveur ORFA.

L'exploitation des machines multiprocesseur impose cependant de répartir la charge sur les différents processeurs. On peut donc imaginer de répartir la récupération des événements entre différents threads. Une solution consiste à répartir les événements réseau dans des files distinctes (*Completion Group*[3]) et à confier une file à chaque thread. GM ne supporte pas ce genre de stratégie puisque sa file de notification est unique. Il est par contre possible de laisser les différents threads relever en alternance des événements dans cette file unique.

6.2.1.b Notification du côté client

Lorsque plusieurs applications accèdent simultanément à un point de montage ORFS dans le système d'exploitation, elles vont soumettre des requêtes d'émission et réception dans le même port GM. Le même phénomène se présente en espace utilisateur si plusieurs threads d'un même processus accèdent simultanément aux fichiers distants par ORFA. Dans ce cas, il va falloir permettre à toutes ces files d'exécution d'attendre la terminaison de leur requête respective, ce qui n'est pas simple lorsque l'interface de programmation propose une file d'événements unique comme dans GM.

Une première solution consiste à utiliser une attente active alternant entre les différentes files d'exécution. Mais la consommation processeur est alors trop importante. Une meilleure stratégie consiste à charger un thread *Dispatcher* de récupérer tous les

[3] Certaines interfaces de programmation permettent de placer les requêtes de communication dans un groupe au moment de leur soumission. La notification peut alors être reçue dans ce groupe, ce qui permet par exemple d'attendre la terminaison d'une requête quelconque du groupe. Des exemples sont donnés dans les annexes A.2.2, A.2.3 et A.3.5.

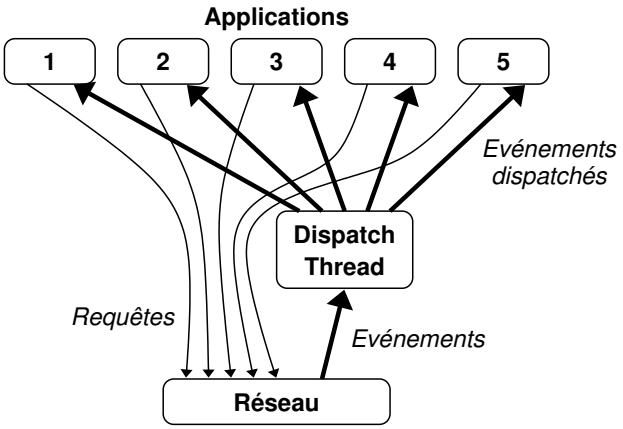


FIG. 6.8: *Répartition des événements réseau par un thread* Dispatcher.

événements dans la file de GM et de les donner (*Dispatch*) à la file d'exécution concernée (voir la figure 6.8). Cette méthode est relativement simple à mettre en place puisque les différentes files d'exécution sont endormies dans des files d'attente accessibles au *Dispatcher* (il s'agit soit de threads d'un même processus, soit de processus différents dans un contexte noyau). Mais cette stratégie impose un léger surcoût en latence du fait des changements de contexte entre les différents threads (plusieurs microsecondes dans ORFS).

La solution optimale consisterait probablement à laisser le thread qui a soumis la requête la plus vieille récupérer les événements. Si l'événement ne le concerne pas, il réveille le thread correspondant et lui donne l'événement. Sinon, il le traite puis laisse le thread suivant attendre les événements. Ainsi, le nombre de changements de contexte est réduit, en particulier si les événements arrivent dans l'ordre de la soumission des requêtes.

6.2.1.c Requêtes asynchrones du côte client

Nous n'avons parlé ici que des accès synchrones, c'est-à-dire lorsque la file d'exécution qui soumet la requête bloque en attendant sa terminaison avant de continuer son exécution. Nous avons expliqué en partie 2.3.1.b que les accès asynchrones au système de stockage étaient une fonctionnalité importante des systèmes d'exploitation moderne. Il nous a fallu également les mettre en œuvre dans ORFS pour voir si leur support imposait des contraintes différentes des accès synchrones.

Une même file d'exécution peut soumettre plusieurs requêtes asynchrones d'accès aux fichiers distants puis attendre la terminaison d'une d'entre elle, que ce soit une particulière ou une quelconque. Le traitement efficace des terminaisons de requêtes asynchrones dans le système consiste à placer un événement dans une file d'attente que l'application va venir tester. Ainsi, l'application n'a pas à parcourir la liste des requêtes

en cours pour savoir lesquelles sont terminées. Le stockage de cet événement doit donc se faire en tâche de fond lorsque la réponse à la requête arrive, même si l'application n'est pas encore en train d'attendre la terminaison. De plus, par soucis d'économie des ressources système, il nous faut être capable de relâcher immédiatement les ressources utilisées. Un thread dédié doit donc être utilisé pour effectuer ces travaux.

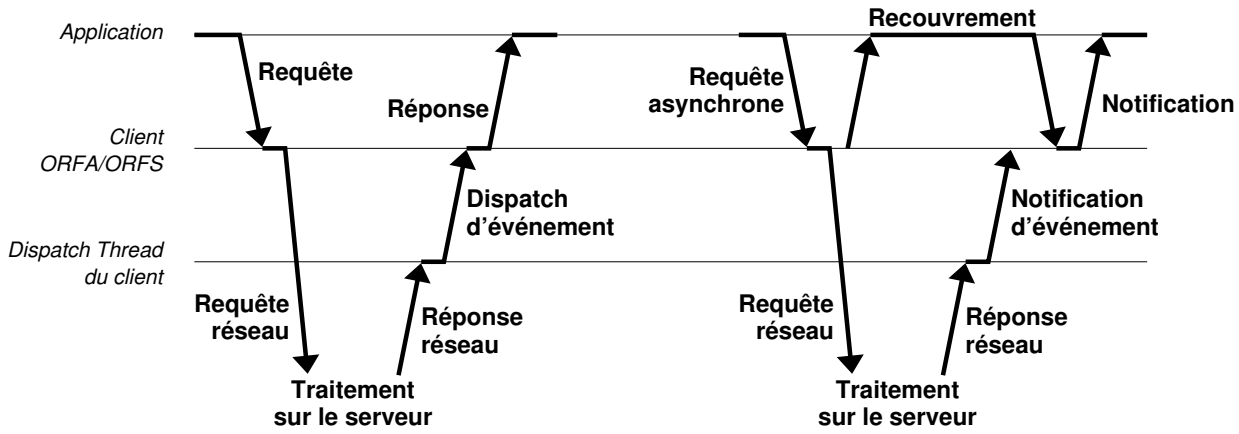


FIG. 6.9: *Traitement des requêtes synchrones (à gauche) et asynchrones (à droite) avec un thread* Dispatcher.

Dans notre modèle, l'interface de programmation de GM nous contraint déjà d'utiliser un thread *Dispatcher* pour distribuer les événements. Nous avons donc également placé le traitement de la terminaison des requêtes asynchrones dans ce thread (voir en figure 6.9).

6.2.1.d Synthèse des notifications

Les besoins en terme de notification d'événements réseau sont très différents sur le client et le serveur. Le serveur nécessite que les terminaisons de requête soient notifiées dans une file unique (ou éventuellement dans un *Completion Group*) tandis que le client se satisfait plutôt de notification distinctes pour chaque communication. L'interface GM ne fournit qu'une file unique d'événements mais nous avons pu mettre en place les autres stratégies en utilisant un thread *Dispatcher*.

L'utilisation dans ORFS d'une interface de programmation du réseau différente, notamment les VERBS INFINIBAND, aurait conduit à des problèmes similaires. Cette interface est plus souple puisqu'elle permet de préciser le *Completion Group* de chaque connexion (voir en annexe A.3.5). Mais il n'est pas non plus possible d'attendre la terminaison d'une requête particulière.

Une interface de programmation permettant ce genre d'attente simplifierait considérablement la mise en place du client puisqu'il suffirait que l'application ayant soumis une requête synchrone d'accès aux fichiers attende la terminaison de la réception asso-

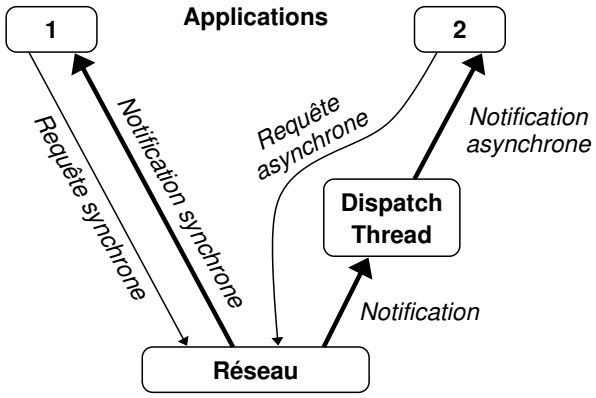


FIG. 6.10: *Notification de la terminaison des accès synchrones et asynchrones aux fichiers distants.*

ciée. Le thread *Dispatcher* ne serait alors plus utilisé que pour traiter les requêtes asynchrones (voir la figure 6.10). Dans ce cas, l'utilisation d'un *Completion Group* regroupant les communications correspondant aux requêtes asynchrones pourrait permettre de distinguer rapidement si une réponse doit être notifiée à l'application l'attendant (pour les requêtes synchrones) ou au thread dédié (pour les requêtes asynchrones).

6.2.2 Interaction avec l'interface standard d'entrées-sorties

Nous venons de présenter comment recevoir efficacement les événements du réseau. La mise en œuvre du serveur impose également d'intéragir avec les périphériques de stockage. Nous avons vu en 2.2.3.c que les stratégies de notification d'événements dans les réseaux rapides étaient très différentes des stratégies standard. Nous détaillons maintenant leur interaction avec les entrées-sorties vers les disques.

6.2.2.a Traitement efficace des entrées-sorties vers le disque et le réseau

Le traitement des requêtes des clients dans un serveur de fichiers consiste à accéder aux couches système de stockage. Lorsque les données cibles ne sont pas cachées dans le système d'exploitation, une entrée-sortie vers le périphérique de stockage doit être initiée. La lenteur de ces périphériques conduit le serveur à bloquer en attendant la fin du traitement. Le processeur central n'est plus utilisé pendant ce temps. Il est donc important de recouvrir les entrées-sorties vers le système de stockage de la même façon que les entrées-sorties vers le réseau sont recouvertes.

Les interfaces standard d'accès au stockage étaient jusque récemment uniquement bloquantes (voir en annexe A.1.1). Ce recouvrement a donc souvent été mis en place en utilisant des threads. Avec l'implantation de l'interface asynchrone d'accès aux fichiers avec LINUX AIO dans LINUX 2.6 (voir en annexe A.1.2), il est désormais possible de

soumettre des requêtes de lecture/écriture dans un fichier et de venir en chercher la notification plus tard. Le recouvrement est immédiat, et l'éventuel coût des changements de contexte entre threads peut être évité.

Algorithme 6.1 - Modèle événementiel parfait pour un serveur de stockage :

```
Faire
    Attendre un événement du réseau ou des disques
    Si Terminaison de réception de requête
        Soumettre une nouvelle réception de requête
        Soumettre l'accès disque
    Sinon Si Terminaison d'accès disque
            Soumettre l'émission de la réponse
    Sinon Si Terminaison d'émission de réponse
            Relâcher les ressources
```

L'efficacité du serveur de fichiers est alors liée à sa capacité à réagir rapidement aux notifications de terminaison de communications réseau et accès disque. Il faut en effet que le serveur puisse être notifié rapidement de ces deux types de terminaison. Pour ce faire, le serveur peut être mis en œuvre autour d'une boucle principale d'attente d'événements du réseau et des disques (voir l'algorithme 6.1). L'interface de notification d'événements de LINUX AIO a été conçue pour être utilisable pour d'autres types d'entrées-sorties que le stockage. En particulier les entrées-sorties réseau via l'interface SOCKET s'interfacent parfaitement avec l'interface LINUX AIO (mais le support correspondant n'a pas été totalement développé dans le noyau LINUX, par manque d'utilisateurs potentiels). Malheureusement, les réseaux haute performance des grappes n'utilisent pas du tout cette interface, d'une part parce que LINUX AIO a été développé trop récemment, et d'autre part car il ne répond pas à leurs besoins, notamment en terme de latence (à cause du coût des appels-système). Il est donc impossible d'être notifié des événements provenant du disque et des réseaux haute performance par la même stratégie.

Algorithme 6.2 - Modèle événementiel simulé par des attentes limitées alternant entre événements réseau et disque :

```
Faire
    Attendre un événement du réseau Pendant δ
    Si Terminaison de réception de requête
        Soumettre une nouvelle réception de requête
        Soumettre l'accès disque
    Sinon Si Terminaison d'émission de réponse
            Relâcher les ressources
    Attendre un événement disque Pendant δ
    Si Terminaison d'accès disque
        Soumettre l'émission de la réponse
```

Pour y remédier, nous avons utilisé la solution qui consiste à alterner des courtes attentes d'événements réseau avec des courtes attentes d'événements du stockage (voir l'algorithme 6.2). La durée δ des attentes doit être ajustée pour trouver le bon compromis entre une grande consommation processeur (si δ est petit) et une faible réactivité (si δ est grand). Nous avons également étudié une implantation où un thread est dédié à chacun des deux types d'événements. Mais les changements de contexte induits peuvent alors dégrader les performances. Par ailleurs, la répartition de la charge entre les deux threads ne sera pas forcément équitable puisque les capacités de traitement sont souvent beaucoup plus importantes dans les cartes réseau que dans les contrôleurs de disque.

6.2.2.b Adaptation des événements réseau à l'interface standard

Nous détaillons ci-après pourquoi les interfaces des réseaux rapides ne sont pas compatibles avec l'interface standard UNIX de programmation des entrées-sorties, et en particulier LINUX AIO.

Un des concepts de base de UNIX est que **tout est fichier**. Que ce soit des fichiers réels stockés sur disque ou des connexions réseau par l'interface SOCKET, tout se manipule de la même façon par les applications et le système d'exploitation (voir en annexe A.1.1). Les requêtes asynchrones manipulées dans LINUX AIO sont alors caractérisées par un descripteur de fichier et un type d'opération (notamment lecture, écriture ou attente d'événement). Les événements récupérés par l'application contiennent un identifiant de requête et leur état.

L'utilisation du réseau haute performance dans ce modèle nécessite donc d'être capable de notifier les terminaisons de communication comme les événements AIO. La soumission des communications peut continuer à utiliser l'interface dédiée du réseau haute performance puisque le seul problème est l'interaction des stratégies de notification. Il suffit donc de faire en sorte que le pilote de la carte réseau génère des événements AIO.

Le problème est que ces événements AIO doivent être générés dans le système d'exploitation, ce qui est contraire à la conception actuelle des mécanismes de notification dans les réseaux haute performance. En effet, la réduction de la latence passe par la suppression du système d'exploitation du chemin critique, c'est-à-dire l'absence d'appel-système (leur coût est d'environ 400 ns sur un processeur cadencé à 2 GHz). Les notifications sont donc prioritairement mises en place en espace utilisateur, le système d'exploitation n'intervenant que lorsque l'application doit s'endormir en attendant un événement.

Il serait intéressant de proposer au niveau de l'interface de programmation réseau un mode de fonctionnement traversant toujours le système d'exploitation lors des notifications des terminaisons des communications réseau (voir la figure 6.11). Les notifications traditionnelles sont directement données à l'application en espace utilisateur tandis que les notifications AIO doivent être données au moteur AIO dans le noyau qui les passera ensuite à l'application quand elle les demandera. Cela nous permettrait

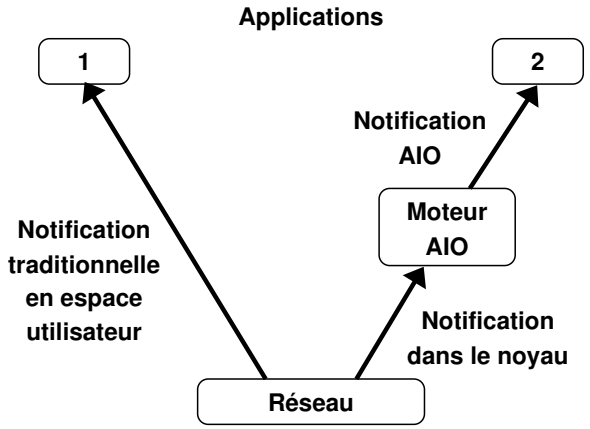


FIG. 6.11: *Notification de la terminaison des communications réseau via le système* AIO *du noyau.*

alors de traiter de la même façon les notifications réseau et disque.

Pour prouver le concept, nous avons mis en place cette stratégie dans BIP [4]. La notification est effectuée dans l'interface `epoll`, qui est une stratégie proche de LINUX AIO mais dédiée au réseau (voir en partie 2.2.1.a). Comme précédemment, des problèmes liés à BIP ne nous ont pas permis de pousser ces développements plus en avant.

6.3 Synthèse

Nous avons présenté dans ce chapitre les problèmes liés à l'utilisation des réseaux haute performance dans un modèle client-serveur. Après avoir présenté au chapitre 5 les modifications à apporter à l'interface réseau et au système d'exploitation pour permettre un transfert efficace de données dans le cadre du stockage distribué, nous nous sommes intéressés ici au problème du contrôle. Nous avons d'une part détaillé les problèmes liés aux messages inattendus et d'autre part les notifications des événements.

Nous avons débuté notre étude par la question Q3 : **le trafic généré par l'accès distant aux fichiers est-il bien géré par le contrôle de communications des réseaux des grappes ?**

Les mécanismes de contrôle proposés par l'interface GM des réseaux MYRINET montrent leurs limites lorsque de nombreux clients émettent des requêtes vers le serveur. Le nombre de zones de réception étant limité, les messages rejetés engendrent un grand gaspillage de bande passante. Nous avons montré que ce problème pouvait être évité en déportant la **maîtrise du protocole applicatif du côté du serveur**. Ainsi, plutôt que de

[4] Nous n'avons pas effectué ce travail dans GM car son implantation dans le noyau est plus complexe et s'y prétait moins que celle de BIP. Il était donc plus simple d'étudier la validité de nos idées dans ce contexte simple.

laisser les clients renvoyer leurs requêtes rejetées indéfiniment, le serveur peut initier la lecture des données sur le client par RDMA lorsque suffisamment de zones de réception ont été préparées.

Nous avons ensuite étudié en 6.2 les notifications d'événements pour voir dans quelle mesure le modèle asynchrone spécifique des réseaux haute performance pouvait être utilisé efficacement dans notre modèle client-serveur de stockage distribué. Il s'agissait donc de répondre à la question Q4 : **peut-on mixer les notifications d'événements des réseaux rapides avec les événements traditionnels du stockage ?**

Nous avons tout d'abord montré qu'il serait très intéressant de disposer de **différentes méthodes de notification de terminaison des communications** : attente d'une requête particulière, d'une quelconque, ou d'une quelconque parmi un groupe. Nous avons ensuite présenté l'interaction de ces stratégies spécifiques avec l'interface standard d'entrées-sorties asynchrones dans LINUX. La notification des terminaisons des communications asynchrones dans cette interface, et en particulier dans LINUX AIO, nécessite de **renoncer aux notifications rapides en espace utilisateur**. Mais il est alors possible d'attendre de la même manière simultanément aussi bien les notifications d'événements réseau que d'événements disque.

Ces résultats montrent que les fonctionnalités disponibles actuellement pour l'accès aux fichiers distants dans les réseaux haute performance restent limitées. Cependant, nous avons montré qu'il était possible de contourner certaines de ces limitations en mettant en place le support nécessaire dans l'application utilisant le réseau. Par exemple, le client ORFA permet d'attendre la terminaison d'une communication particulière alors que GM ne peut attendre qu'une terminaison quelconque.

Il faut noter qu'il est assez courant d'implanter ce genre de stratégies en dehors de l'interface de programmation réseau. Par exemple, les implantations de MPI doivent gérer ce genre de problèmes de notification et de contrôle des communications. On est donc en droit de se demander si certaines de ces fonctionnalités ne devraient pas être directement mises en place dans l'interface de programmation du réseau. Leur optimisation serait alors probablement plus facile. C'est l'objet du chapitre suivant.

Chapitre 7

Propositions pour une interface de communication adaptée au stockage distribué

Nous avons montré au chapitre 4 que les réseaux haute performance pouvaient apporter un gain important pour l'accès au stockage distant. Mais nous avons également constaté différents problèmes. Les problèmes liés au transfert de données, et notamment à l'enregistrement mémoire, ont été détaillés au chapitre 5. Nous avons montré qu'il était nécessaire de modifier le système d'exploitation et l'interface de programmation GM des réseaux MYRINET pour les faire intéragir efficacement. Les problèmes liés au contrôle ont ensuite été exposés au chapitre 6 où nous avons d'une part étudié le contrôle des communications et d'autre part la gestion des événements dans un modèle client-serveur tel que celui des systèmes de stockage distribué auxquels nous nous intéressons.

Il nous faut maintenant tenter de répondre à notre question principale : **dans quelle mesure les réseaux haute performance peuvent-ils être utilisés efficacement dans un système de stockage distribué de type client-serveur où les applications manipulent les données par l'intermédiaire de l'interface standard de programmation des accès aux fichiers ?**

Nous nous intéressons pour cela à une interface très récente de programmation des réseaux haute performance, MX (*Myrinet Express*). Après une rapide présentation de ses caractéristiques principales, les différentes fonctionnalités avancées qu'elle propose sont détaillées ainsi que leur apport vis-à-vis des différents problèmes que nous avons rencontrés au cours de notre étude. Nous présentons enfin l'intégration de nos travaux dans l'interface de programmation de MX dans le noyau et montrons l'efficacité du modèle obtenu.

7.1 Stockage distribué avec l'interface Myrinet Express

MX est la nouvelle interface logicielle des réseaux MYRINET. MYRICOM souhaite répondre aux différents problèmes qui ont été rencontrés avec l'interface précédente, GM. Sa conception prend en compte un certain nombre d'idées qui ont été developpées ces dix dernières années, en particulier parmi les nombreux travaux qui ont utilisé MYRINET comme plate-forme de développement.

Nous présentons les caractéristiques de MX ainsi que quelques évaluations de performances avant de détailler les apports réels dans le cadre du stockage distribué des fonctionnalités avancées qu'elle propose.

7.1.1 Présentation de MX

Myrinet Express se caractérise par une interface de programmation particulièrement flexible, notamment pour les applications parallèles de type MPI. Ses principales caractéristiques sont :

- Pas d'enregistrement mémoire explicite, ce qui évite d'imposer aux applications d'avoir à enregistrer toutes les zones mémoire qu'elles vont mettre en jeu dans des communications.
- Communications sur le modèle du *Rendez-vous* avec *Matching* générique sur 64 bits. Chaque message et réception est caractérisé par un *tag* 64 bits. Un message sera reçu dans une zone de réception dont le tag est identique (à un masque près).
- Nombre de requêtes soumises en émission ou réception illimité.
- Différentes stratégies d'attente ou test de terminaison. L'application peut attendre la terminaison d'une communication particulière, ou bien celle d'une communication quelconque.
- Primitives de communication vectorielles.
- Communication RDMA (lecture ou écriture dans la mémoire d'une application distante).
- Communications collectives (*Barrier*, *Broadcast*).

L'interface de MX est détaillée dans [MX05] et en annexe A.3.2. Elle est très proche de l'interface de programmation de MPI (voir en annexe A.2.1). L'intégration de différentes fonctionnalités avancées dans l'interface de programmation du réseau permet de les mettre en place plus efficacement. L'implantation de MPI au dessus de MX se révèle particulièrement simple[1], et donc efficace[2].

De plus, le support natif des communications collectives permettra à l'avenir de se passer de la traditionnelle émulation par des communications point-à-point que propose dans MPI la couche d'abstraction de ADI (*Abstract Device Interface* [ADI]). Il a

[1] L'implantation de CH_MX regroupe environ 15 000 de lignes de code C tandis que CH_GM en regroupe près de 29 000.

[2] Le surcoût en latence de MPICH-MX sur MX est inférieur à 0,5 μs tandis que la bande passante observée est identique.

par exemple été montré dans [BPS01] que la latence d'une barrière[3] MPI sur 16 nœuds pouvait être divisée par 1,8 lorsque son support était déporté dans la carte d'interface réseau. Le support des interfaces de type VIA ou DAPL ne devrait pas présenter de difficultés lui non-plus puisque les communications par RDMA seront bientôt également disponibles nativement dans MX.

La mise en œuvre de MX utilise par ailleurs différents mécanismes avancés permettant d'obtenir des meilleures performances qu'avec GM :

- Utilisation des PIO (*Programmable Input-Output*) pour obtenir une faible latence (voir en 2.2.3.b).
- Utilisation des adresses physiques dans la carte d'interface réseau.
- Recouvrement des communications sans intervention de l'application. Un thread réagit aux événements réseau sans que l'application n'ait besoin d'appeler explicitement la bibliothèque MX.
- Support efficace des messages inattendus.
- Routage dispersif adaptatif pour répartir le trafic sur le réseau et réagir aux éventuelles congestions.

Nous présentons maintenant une comparaison simple des performances de MX et GM.

7.1.2 Évaluation des performances

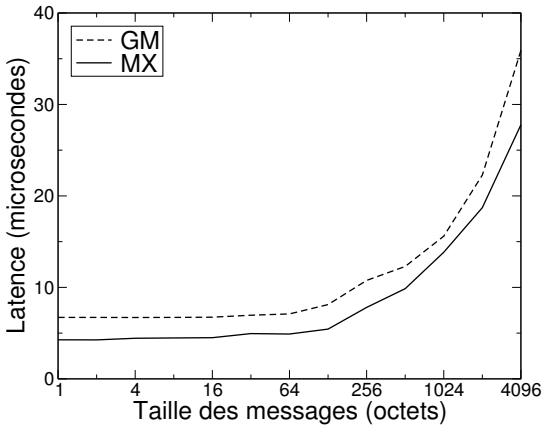
Les expérimentations que nous avons menées sur MX ont été réalisées sur des machines équipées de deux processeurs PENTIUM 4 XEON cadencés à 2,6 GHz et de 2 Go de mémoire vive. Le réseau MYRINET utilisé reliait des cartes PCIXD [4] dont le processeur LANAI est cadencé à 225 MHz.

La figure 7.1 présente les mesures de latence et bande passante de MX et GM obtenues avec un programme de *Ping-Pong* dédiée. La latence de MX est beaucoup plus basse de celle de GM puisqu'elle atteint 4 μs pour des messages de taille nulle, contre 6,7 μs. Ce gain est notamment lié à l'utilisation des PIO (*Programmable Input/Output*) pour les petits messages (jusqu'à 128 octets). Il s'agit d'utiliser le processeur central pour copier les données entre la carte d'interface et la mémoire de l'application. Ainsi, le transfert est rapide et contrôlé par l'hôte ce qui permet des notifications rapides.

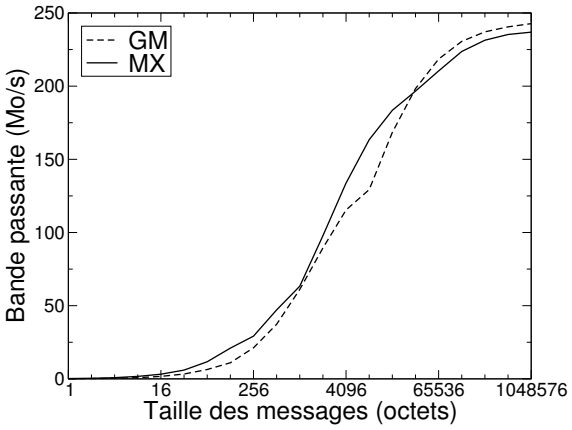
Les bandes passantes de MX et GM sont assez proches. Celle de MX est légèrement supérieure pour les messages de taille inférieure à 32 ko, et très légèrement inférieure ensuite. Les différentes irrégularités sur la figure 7.1(c) correspondent aux seuils des différents modes de fonctionnement de GM et MX. Là où GM repose essentiellement sur l'enregistrement mémoire et les DMA quelle que soit la taille des messages (s en octets), MX utilise 3 modes différents :

[3] Une barrière est une communication collective de synchronisation. Elle est levée lorsque tous les nœuds ont atteint ce point de synchronisation.

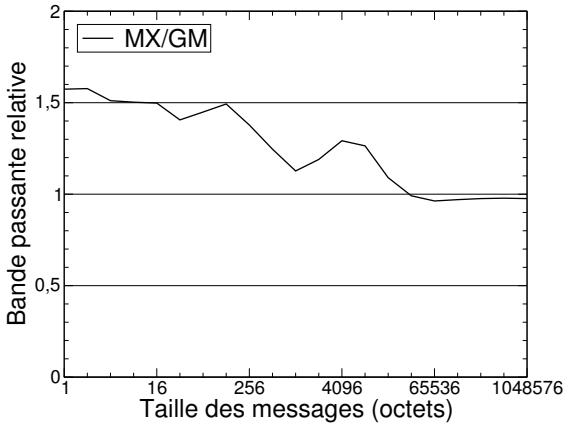
[4] MX ne supporte pas les modèles de cartes qui ont précédé les PCIXD



(a) Latence.



(b) Bande passante.



(c) Bande passante relative de MX.

FIG. 7.1: *Comparaison des performances de* GM *et* MX. *Les mesures ont été effectuées en utilisant le programme de* Ping-Pong *dédié à chacune des deux interfaces.*

Petits messages ($s \leq 128$) : Les données sont transférées par PIO entre la carte et la mémoire de l'application.

Moyens messages ($128 < s \leq 32768$) : Les données sont copiées dans une zone intermédiaire (pré-enregistrée dans la carte d'interface) puis transférées par DMA.

Gros messages ($32768 < s$) : Un appel-système est effectué pour verrouiller les pages pendant la communication (enregistrement mémoire). Un DMA transfère les données entre le réseau et la mémoire de l'application.

Nous venons de voir que le transfert des petits messages est beaucoup plus efficace qu'avec GM. Les moyens messages utilisent une copie de la même façon que les applications utilisant GM doivent trouver un compromis entre copies mémoire et enregistrement (voir en partie 5.1.1). L'intégration de cette fonctionnalité dans l'interface MX apporte jusqu'à 25 % de gain en bande passante.

Le mode de fonctionnement de MX pour les gros messages est proche de celui de GM mais l'enregistrement mémoire n'est ici pas explicite. L'application peut utiliser librement son espace mémoire, MX se charge de l'enregistrer systématiquement à la volée. Le traditionnel modèle *OS-bypass* (voir en partie 2.2.2) n'est donc plus respecté. Cependant, le coût d'un appel-système est de 400 ns sur les machines actuelles tandis que le transfert de 64 ko à 250 Mo/s dure 262 μs. Le gain de quelques microsecondes procuré par la stratégie *OS-bypass* ne justifie donc pas l'obligation pour l'application d'enregistrer explicitement toutes les zones mémoire mises en jeu dans les communications. Il se révèle beaucoup plus aisé pour l'utilisateur de perdre ces quelques microsecondes (négligeables devant la durée total du transfert) et de ne plus avoir à faire ce travail d'enregistrement mémoire lui-même.

Le fait que GM présente une meilleure bande passante dans ce cas doit donc être relativisé car le test étudié ici a la particularité de réutiliser les mêmes zones mémoire à chaque communication (voir en partie 5.1.2). Par contre, la bande passante de MX est ici indépendante de la réutilisation des zones mémoire puisque l'enregistrement est fait à chaque communication.

Nous avons vu en partie 5.1.1 que le coût de l'enregistrement mémoire était très élevé dans GM, l'enregistrement à la volée pourrait donc être nocif aux performances MX. En fait, son coût était élevé dans GM car le *firmware* de la carte n'a pas été conçu pour le gérer efficacement. Sa mise en œuvre dans MX est bien plus efficace (environ 2 μs pour un enregistrement ou un désenregistrement). Par ailleurs, des techniques du type cache d'enregistrement mémoire seront ajoutées par la suite dans MX afin de réduire ce coût dans le cas où les zones mémoire seront réutilisées plusieurs fois. La bande passante observée ici sera alors le pire cas pour MX alors que celle de GM est ici le meilleur cas. L'utilisateur pourra alors profiter beaucoup facilement des performances de MX sans avoir à optimiser sa gestion mémoire comme avec GM.

7.1.3 Réponses de MX aux besoins du stockage distribué

Nous venons de voir qu'il était possible de bénéficier des performances de MX sans que l'application n'ait à respecter les contraintes de l'enregistrement mémoire. Les interfaces de programmation basées sur l'enregistrement mémoire (en particulier GM sur MYRINET ou VERBS sur INFINIBAND) sont en effet assez peu flexibles et imposent au développeur un rigoureux travail d'optimisation de l'utilisation mémoire au niveau applicatif.

Les caractéristiques de MX que nous avons listées en 7.1.1 montrent que l'interface que nous proposons est beaucoup plus flexible. D'une part, elle fournit des fonctionnalités avancées et d'autre part dispose d'une mise en œuvre efficace. Les évaluations de performances précédentes ont été menées avec un programme de test dédié (*Ping-Pong*). Mais il s'avère que MX s'adapte en fait très bien aux applications réelles. Nous avons expliqué que la mise en œuvre de MPI sur MX était très simple car la plupart des fonctionnalités nécessaires à MPI sont implantées nativement et efficacement dans MX.

Nous étudions maintenant les apports de ces fonctionnalités avancées pour le problème du stockage distribué, en nous concentrant sur les problèmes rencontrés aux chapitres 5 et 6.

7.1.3.a Contrôle des communications

Nous avons montré en partie 6.1 que le contrôle des communications, et en particulier la gestion des messages inattendus, avait un impact critique sur les performances dans les modèles client-serveur. Nous avons expliqué en partie 6.1.3 qu'à défaut de disposer d'une interface de programmation gérant efficacement ce problème, il fallait adapter le protocole de niveau applicatif pour déporter la maîtrise du trafic du côté serveur en utilisant par exemple des communications RDMA.

Ce problème n'est en fait pas uniquement lié au modèle client-serveur et à son trafic hiérarchique : les limites de la gestion des messages inattendus dans GM se sont également révélées problématiques dans MPI. MX intègre donc un support efficace de ces messages. Les petits et moyens messages inattendus sont reçus dans une zone temporaire et copiés dans la zone de destination lorsqu'elle est donnée par l'application. Lorsqu'un gros message inattendu arrive, l'interface de programmation marque son arrivée sur le récepteur jusqu'à ce que l'application donne la zone de réception associée. Le récepteur va alors chercher les données par RDMA sur l'émetteur.

Il s'agit en fait d'une stratégie proche du protocole applicatif initié par le serveur que nous avons présenté en 6.1.3. Cependant, le modèle du *Rendez-vous* est conservé. MX se charge d'utiliser implicitement des RDMA si nécessaire. La mise en œuvre du protocole de haut niveau ORFA au dessus de MX s'est révélée particulièrement simple. Nous nous attendons donc à une disparition de la chûte de bande passante observée avec GM lorsqu'un nombre conséquent de clients émettent des requêtes d'écriture vers le serveur (voir la figure 6.4).

Il faut par ailleurs citer le fait que MX ne limite pas le nombre de communications

pouvant être soumises simultanément. La mémoire de la carte d'interface limite, comme dans n'importe quelle interface de programmation, le nombre de descripteurs de communication disponibles dans la carte. Mais MX met en place un dialogue complexe entre l'hôte et la carte de façon à garder les descripteurs nécessaires dans la carte tandis que la liste complète est maintenue dans la mémoire de l'hôte. Ainsi, le serveur ORFA peut potentiellement préparer un très grand nombre de requêtes de réception, tandis que MX se chargera de les donner à la carte régulièrement. Le problème des messages inattendus peut également être évité ainsi.

Ce modèle est assez proche du modèle proposé en 6.1.3 puisque MX va utiliser en interne des RDMA pour aller chercher les données sur le client lorsque celui-ci voudra lui envoyer un gros message.

7.1.3.b Stratégies d'attente d'événements

Nous avons montré en partie 6.2.1 que les besoins en notification d'événements dans notre modèle client-serveur de stockage distribué étaient assez variés (attente d'une requête spécifique, d'une requête quelconque, ou d'une quelconque parmi un groupe). MX propose là aussi les fonctionnalités les plus utiles à MPI, c'est-à-dire l'attente ou le test de terminaison d'une requête précise ou d'une requête quelconque. Cette interface de notification s'adapte également très bien aux besoins d'un modèle client-serveur de stockage distribué. Là où l'interface de programmation GM obligeait le client ORFA à se baser sur un thread *Dispatcher* (voir en 6.2.1), la mise en œuvre sur MX est particulèrement aisée.

Il faut noter que MX ne permet pas de notification parmi un groupe de requêtes (*Completion Group*) mais cela n'a pas posé de problème majeur lors de l'implantation de ORFS. Cette fonctionnalité peut être émulée en attendant la terminaison d'une requête quelconque au lieu d'une requête d'un groupe (ce qui impose un léger surcoût puisque certaines terminaisons seront inutilement notifiées).

Concernant l'interaction entre les stratégies de notification dans les réseaux rapides et les mécanismes d'entrées-sorties dans LINUX (voir en partie 6.2.2), MX utilise les stratégies habituelles de notification des événements réseau, c'est-à-dire en espace utilisateur sauf quand l'application doit dormir. Il n'est donc pas possible d'intégrer les événements MX aux mécanismes de LINUX AIO afin d'obtenir une gestion conjointe efficace des entrées-sorties vers le réseau et les disques.

7.1.3.c Réductions des copies

Nous avons montré en partie 4.2.6 que la charge du serveur est un facteur limitant et que les réductions de copie mémoire permettent d'améliorer notablement les performances. MX utilise une copie mémoire intermédiaire pour transférer les petits et moyens messages. On est donc en droit de se demander si cette copie va avoir un impact notable sur les performances.

Tout d'abord, il faut noter que les interfaces basées sur l'enregistrement mémoire conduisent souvent à utiliser des copies intermédiaires pour les petits messages afin d'éviter le coût de l'enregistrement mémoire (voir en partie 5.1.1). La copie intermédiaire utilisée par MX n'est donc pas une nouveauté. Mais son utilisation est ici systématique pour les messages de taille inférieure ou égale à 32 ko.

Si la consommation processeur de cette copie devient trop importante, il est possible d'obliger MX à utiliser une stratégie zéro-copie pour envoyer et recevoir les données, quelle que soit leur taille. Pour ce faire, une primitive de communication *synchrone* est utilisée (voir `mx_issend` en annexe A.3.2). Elle ne notifie pas l'émetteur avant que les données n'aient été réellement reçues. La différence avec le mode traditionnel de communication réside dans le cas d'un petit ou moyen message inattendu. En effet, un tel message émis normalement sera déposé dans une zone intermédiaire du récepteur et notifié immédiatement à l'émetteur avant même que le récepteur n'ait préparé la zone finale de réception correspondante. En mode synchrone, l'envoi est toujours traité comme un gros message. L'émetteur prévient le récepteur et ce dernier vient chercher les données par RDMA quand la zone de réception a été préparée. La latence est donc supérieure mais les transferts sont zéro-copie.

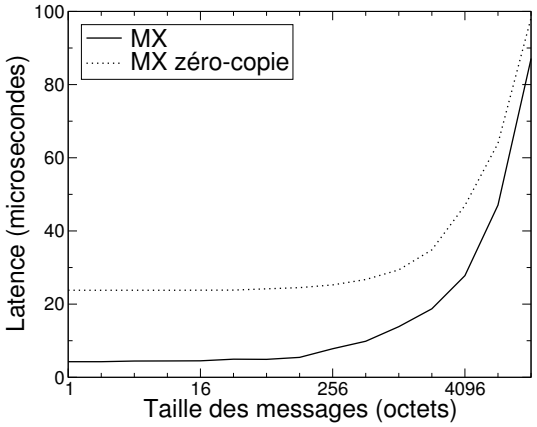


FIG. 7.2: *Latence des communications zéro-copie dans* MX. *Les mesures ont été effectuées en utilisant le programme de* Ping-Pong *dédié à* MX *et en utilisant les communications normales ou zéro-copie.*

La figure 7.2 présente la latence des communications zéro-copie dans MX. On observe un surcoût en latence d'environ 20 μs pour les messages de taille nulle, qui est peu à peu compensé par le coût de la copie mémoire quand la taille augmente. Comme nous avons vu en 4.2.3 que la latence a moins d'impact sur les performances de l'accès aux fichiers distants que l'utilisation d'un cache du côté client, cette stratégie peut donc être intéressante si la charge du serveur est trop grande.

Nous avons également évoqué en 4.2.6 les copies mémoire intervenant du côté du serveur pour déplacer les données entre le processus serveur et le cache de fichiers du

système. Les stratégies du type `sendfile` ou `recvfile` (voir en 2.2.4.b) permettent d'éviter cette copie. Ce type de primitives de communications n'est pas disponible dans MX mais nous nous attendons à ce que leur mise en place ne pose pas de problème particulier puisque nous l'avons précédemment réalisé dans BIP.

7.1.3.d Communications vectorielles

Parmi les fonctionnalités dont le support est prévu dans MX se trouvent les primitives vectorielles de communication. Elles répondent au besoin des applications parallèles de modifier l'organisation des données en mémoire (par exemple en transposant des blocs de matrice à la volée). Dans le cadre du stockage distribué, les communications vectorielles répondent également au besoin des applications de faire des accès vectoriels aux fichiers distants (voir en partie 2.3.1.a). Cependant l'utilisation réelle de ce type de communication varie beaucoup selon que les accès sont directs ou via le cache.

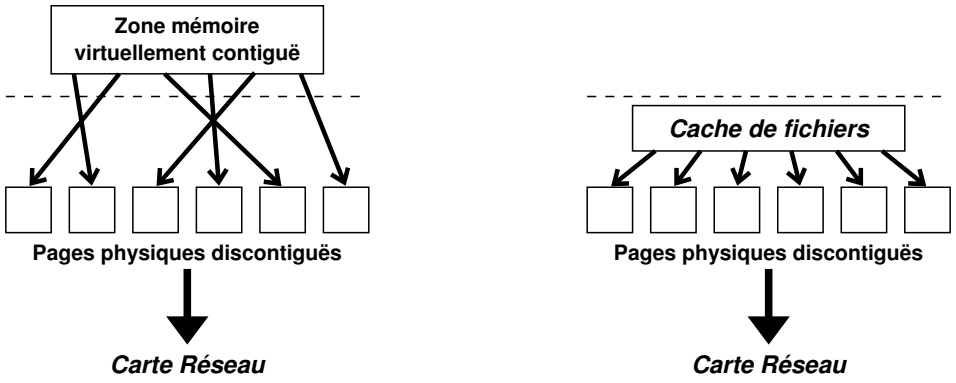
Dans le cas d'accès directs, les vecteurs passés par l'application pour accéder aux fichiers sont passés à l'identique à la couche réseau pour transférer les données correspondantes. Le support des communications vectorielles dans MX permet donc une mise en œuvre aisée dans ORFS. En attendant que ce support soit effectivement disponible, ORFS doit mettre en œuvre les accès vectoriels par émulation. La stratégie la plus simple consiste à regrouper les données dans une zone contiguë par une copie mémoire. Ce mécanisme peut être intégré à la copie utilisée par MX pour les petits et gros messages. Mais pour les gros messages, la consommation en temps processeur serait trop importante.

Il est préférable d'utiliser une stratégie basée par exemple sur une communication réseau pour chaque segment mémoire mis en jeu [CCL+03, WWP03b]. Avec une interface telle que GM, cette stratégie pose le problème du nombre de communications simultanément en cours de traitement (32 émissions au maximum). Par contre, dans MX ce nombre est virtuellement illimité. Ce type de stratégie est donc facilement applicable.

Les accès vectoriels aux fichiers distants à travers le cache se traduisent par la mise à jour de certaines pages du cache. Les différentes pages du cache n'étant pas mappées dans la mémoire du système, le fait qu'un accès soit vectoriel ou non ne modifie rien au problème, il s'agira toujours d'un ensemble de pages distinctes.

Les communications mises en jeu dans ce contexte sont contrôlées par la stratégie du système d'exploitation pour maintenir son cache à jour. Comme nous l'avons vu en partie 5.2.3, la stratégie simple de LINUX consiste à les mettrre à jour une par une. Mais LINUX 2.6 propose désormais des stratégies concernant un ensemble de pages, en particulier pour gérer les lectures à l'avance (*Read-Ahead*) ou les écritures différées (*Write-Back Caching*). Dans ce cas, l'utilisation des communications vectorielles se révèle particulièrement utile puisque chaque page du cache devient un segment mémoire du vecteur.

Cependant, nous pouvons envisager de mettre en place des communications depuis ces pages sans attendre le support des communications vectorielles dans MX. En effet,



(a) Zone contiguë en mémoire virtuelle.

(b) Pages du cache.

FIG. 7.3: *Traitement des pages non-contiguës en mémoire physique.*

puisqu'une grande zone contiguë en espace virtuel utilisateur est manipulée comme un ensemble de pages non-contiguës dans la carte d'interface (voir la figure 7.3(a)), il est envisageable de détourner légèrement l'interface de MX pour passer directement un ensemble de pages du cache (voir la figure 7.3(b)).

7.1.4 Synthèse des apports de MX pour le stockage distribué

Nous avons présenté dans cette partie les caractéristiques principales de MX, ses performances puis les réponses que cette nouvelle interface de programmation des réseaux MYRINET apporte au problème du stockage distribué. Si le but premier de MX était de fournir de meilleures performances que GM avec une interface de programmation plus souple et adaptée à MPI, il se révèle en fait particulièrement adapté à une utilisation dans un modèle client-serveur de stockage distribué.

Nous avons mis en place le protocole ORFS sur MX sans difficulté, en particulier grâce aux primitives d'attente d'une terminaison particulière ou quelconque et à la gestion efficace des messages inattendus. Ces fonctionnalités avancées ont été implantées dans MX essentiellement pour répondre aux besoins de MPI mais s'appliquent très bien à nos travaux. Le *Matching* permet quant à lui une reconnaissance aisée des données qui accompagnent le descripteur d'une requête ou d'une réponse. Enfin, le support des communications vectorielles permettra une mise en place efficace des accès vectoriels aux fichiers ou des mises à jour simultanées de plusieurs pages du cache.

Par contre, en ce qui concerne les transferts de données, MX ne fournissait initialement pas de moyen efficace de gérer les différents types de mémoire mis en jeu, en particulier le cache de fichiers du système d'exploitation. Nous présentons ci-après les travaux que nous avons intégrés à MX pour gérer efficacement ces problèmes particuliers d'adressage mémoire.

7.2 Propositions pour une interface noyau dans MX

Les interfaces de programmation des réseaux rapides étant essentiellement conçues pour les communications dans les applications parallèles, leurs primitives sont prioritairement exposées aux applications en espace utilisateur. MX était donc initialement disponible en espace utilisateur. Notre travail a donc débuté par l'exposition de l'interface de programmation dans le noyau afin de pouvoir par exemple l'utiliser dans le client ORFS implanté dans le noyau LINUX.

QUADRICS a choisi de proposer deux interfaces très différentes dans le noyau et en espace utilisateur. Les KCOMM (*Kernel Communications*) sont en effet basées sur le principe de services (des types logiques de communication incompatibles entre eux, voir en annexe A.3.4) tandis que les interfaces utilisateur sont basées sur des accès mémoire à distance (ELANLIB) ou sur du passage de messages (TPORTS).

Comme dans l'interface VERBS de INFINIBAND (voir en annexe A.3.5), nous avons choisi de conserver la même interface MX dans le noyau. Ainsi, d'une part, une application implantée en espace utilisateur peut communiquer avec une application noyau (ce qui peut être le cas dans un modèle client-serveur de stockage distribué, voir en partie 2.3.2.c). D'autre part, nous nous attendions à ce que l'interface de passage de messages de MX soit tout à fait adaptée à nos travaux (voir en 7.1.4).

Nous avons repris toute la bibliothèque utilisateur de communication MX et l'avons modifiée pour qu'elle soit utilisable à l'identique dans le noyau et en espace utilisateur. Cette mise en œuvre a été réalisée dans les noyaux LINUX et FREEBSD en faisant attention à ce que le cœur de l'implantation soit suffisamment générique pour ne pas désavantager les communications en espace utilisateur ou en espace noyau.

Cependant, ce cœur, et en particulier les mécanismes mis en jeu lors de l'enregistrement des pages, avait été conçu pour des pages utilisateur tandis que les communications depuis le noyau peuvent potentiellement utiliser des zones utilisateur ou noyau. Il nous a donc fallu étendre l'interface noyau de MX pour supporter ces différents cas.

L'interface de programmation flexible que nous proposons ici est désormais disponible dans la suite officielle MX. Ces travaux ont été menés en étroite collaboration avec MYRICOM.

7.2.1 Adressage mémoire souple

Nous résumons tout d'abord les différents types de zones mémoire qui peuvent être mis en jeu. Une implantation en espace utilisateur n'a accès qu'à des zones utilisateur (voir en 5.1.2). Ce type d'accès est donc parfaitement supporté par MX puisque les zones mises en jeu sont identiques au cas d'une application parallèle.

Par contre, une implantation dans le noyau a accès aux :

Zones de l'espace d'adressage noyau : par exemple pour échanger les messages de contrôle du protocole de l'application ;

Zones de l'espace utilisateur des processus : par exemple pour mettre en place des accès zéro-copie aux fichiers distants (voir en 5.2.2).

Pages physiques du système : par exemple pour accéder au cache des fichiers (voir en 5.2.3) ;

De plus, il nous faut être capable de distinguer les différents espaces d'adressage utilisateur des différents processus afin d'éviter des collisions (voir en 5.2.2.a). Enfin, une même adresse virtuelle peut aussi bien correspondre à une zone de mémoire virtuelle du noyau que d'un processus utilisateur (voir également en annexe D.1).

Notre interface noyau de MX doit donc permettre de distinguer ces différents types d'adressage mémoire. Il s'agit de demander à l'application de nous préciser quel type d'adresses elle passe à MX afin que la gestion mémoire soit adaptée. Cette assistance de l'application n'est pas une contrainte forte sur le développeur puisque ce dernier sait parfaitement quel type de mémoire il utilise. La nécessité d'enregistrer les pages dans une interface comme GM ou les VERBS d'INFINIBAND demande beaucoup plus de travail en particulier un travail d'optimisation de l'utilisation mémoire.

Nous avons ajouté à l'interface traditionnelle de MX des primitives supplémentaires dans le noyau. Il s'agit de permettre à toutes les primitives adressant de la mémoire de préciser le type de mémoire. Par exemple, la primitive originale d'émission `mx_isend` considère dans le noyau que la mémoire qu'on lui passe est de la mémoire du noyau. Par contre, notre version flexible `mx_kisend` dispose d'un paramètre supplémentaire `pin_type` précisant le type de mémoire (voir en annexe A.3.2).

Nous avons enfin modifié les fonctions d'enregistrement mémoire afin d'adapter le comportement en fonction du type `pin_type`.

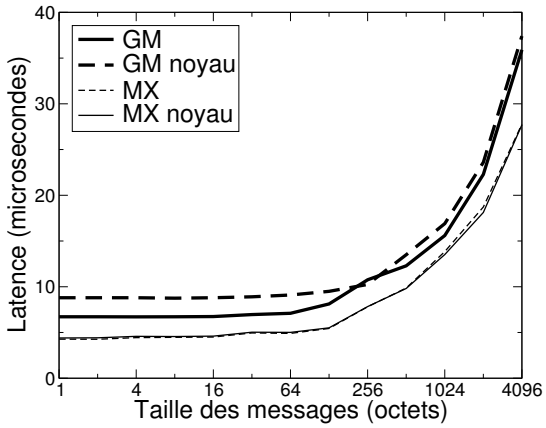
Mémoire virtuelle noyau (`MX_PIN_KERNEL`) : MX doit verrouiller les pages puis traduire leurs adresses virtuelles avant de les passer à la carte d'interface.

Mémoire virtuelle utilisateur (`MX_PIN_USER`) : MX doit verrouiller les pages utilisateur du processus courant, les traduire puis les passer à la carte. `MX_PIN_USER` peut être remplacé par un pointeur vers un espace d'adressage d'un processus quelconque afin d'utiliser sa mémoire en dehors de son contexte.

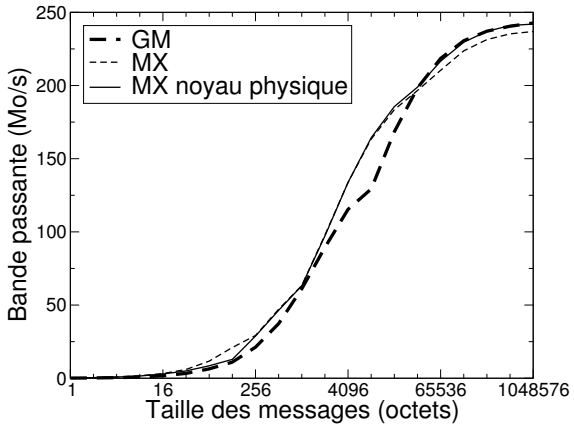
Cadres physiques (`MX_PIN_PHYSICAL`) : MX passe directement les adresses physiques à la carte d'interface. Aucun verrouillage n'est requis puisqu'une application n'est pas censée manipuler des adresses physiques sans s'être précédemment assurée que les pages correspondantes étaient verrouillées.

Nous avons essentiellement parlé ici de l'enregistrement mémoire utilisé en interne dans MX mais il nous a également fallu adapter les copies dans le cas d'un petit ou moyen message.

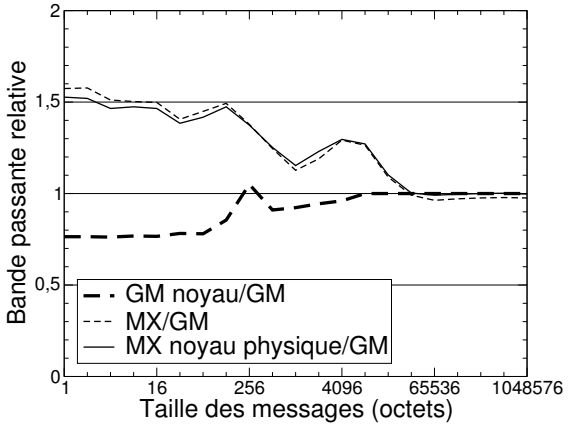
La figure 7.4 présente une comparaison des performances de MX et GM en espace utilisateur et noyau. On constate que la mise en œuvre de GM dans le noyau est moins performante que celle en espace utilisateur (la latence est supérieure de 2 μs). Ceci est dû à une conception initiale pour des communications en espace utilisateur, ce qui a imposé un surcoût quand le support des communications dans le noyau a été ajouté.



(a) Latence.



(b) Bande passante.



(c) Bande passante relative.

FIG. 7.4: *Comparaison des performances de* GM *et* MX *dans le noyau.*

Par contre, MX ne présente pas de différence notable puisque nous avons pris soin de concevoir un cœur suffisamment générique.

Les bandes passantes de MX et GM ne dépendent pas de la mise en œuvre dans le noyau ou en espace utilisateur. Mais notre interface MX flexible dans le noyau nous permet de traiter plus efficacement les communications par adresse physique (puisque le verrouillage et la traduction d'adresse sont inutiles). Ainsi, la bande passante de MX pour des communications par adresse physique atteint 242 Mo/s contre 237 dans les autres.

7.2.2 Application à ORFS

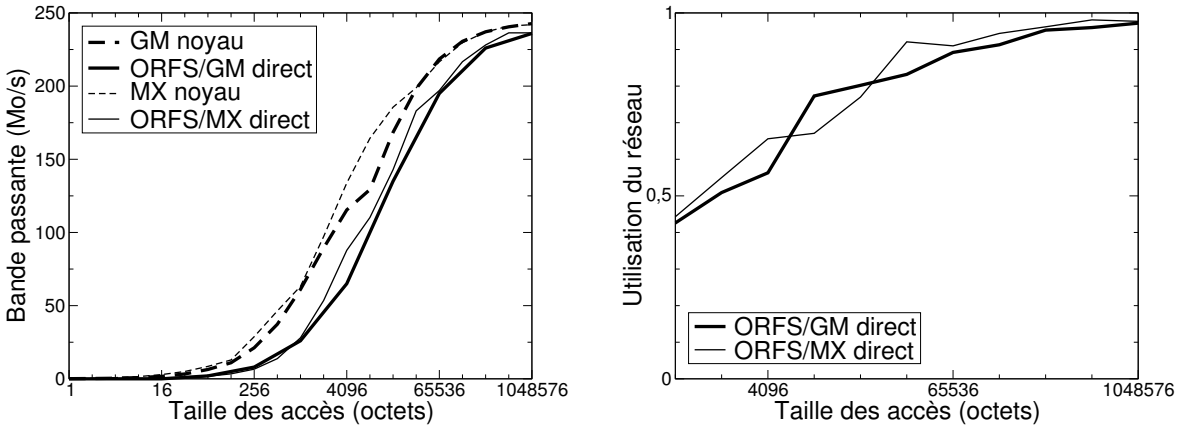
Nous avons utilisé l'interface noyau de MX pour mettre en œuvre les transferts de données dans le client ORFS dans le noyau LINUX. Notre interface pour gérer l'adressage mémoire s'est révélée particulièrement facile à utiliser pour gérer les accès depuis le cache de fichiers (par leur adresse physique, `MX_PIN_PHYSICAL`) et les accès directs (`MX_PIN_USER`). Les descripteurs de requêtes et de réponses étant alloués par le client dans l'espace d'adressage noyau, les communications traditionnelles de MX (ou `MX_PIN_KERNEL`) étaient adaptées pour les transférer.

La possibilité d'indiquer l'espace d'adressage d'un processus particulier au lieu de `MX_PIN_USER` permet par ailleurs de soumettre des requêtes au serveur distant hors du contexte de ce processus. Par exemple, si un processus soumet un accès asynchrone aux fichiers distants alors qu'une ressource système nécessaire est occupée, il faudra programmer le traitement de l'accès et rendre la main au processus. Le traitement différé sera exécuté plus tard en tâche de fond dans le contexte du thread ORFS chargé de traiter les accès asynchrones. Ce thread noyau ne fait pas partie du processus cible mais l'interface de MX lui permet tout de même d'utiliser l'espace d'adressage de ce processus dans les communications réseau nécessaires au traitement de l'accès.

La figure 7.5 présente les performances des accès directs avec ORFS sur MX. La supériorité des performances brutes de MX sur celles de GM se reflète dans les performances de ORFS. Cependant, une fois de plus, nous rappelons que les performances de GM et d'ORFS sur GM sont tributaires de la réutilisation des mêmes zones mémoire dans l'application.

Mais il faut surtout insister ici sur la facilité d'utilisation de l'interface noyau de MX. En effet, MX est directement utilisable par ORFS alors que GM imposait de mettre en place un cache d'enregistrement (GMKRC), de modifier le noyau pour mettre à jour ce cache (VMA SPY) et d'augmenter la taille des pointeurs dans le *firmware* de la carte MYRINET (voir en 5.2.2).

La figure 7.6 présente ensuite les performances des accès à travers le cache. Comme nous l'avions vu en partie 5.2.3.b, ces performances sont influencées par la copie mémoire entre l'application et le cache, et la stratégie utilisée par le système pour maintenir son cache à jour vis-à-vis du serveur distant. Là encore, nous ne nous sommes intéressés qu'aux requêtes concernant une seule page du cache, ce qui explique la limitation du



(a) Bande passante. (b) Bande passante relative.

FIG. 7.5: *Performance des accès distants directs avec* ORFS *sur* MX *et* GM.

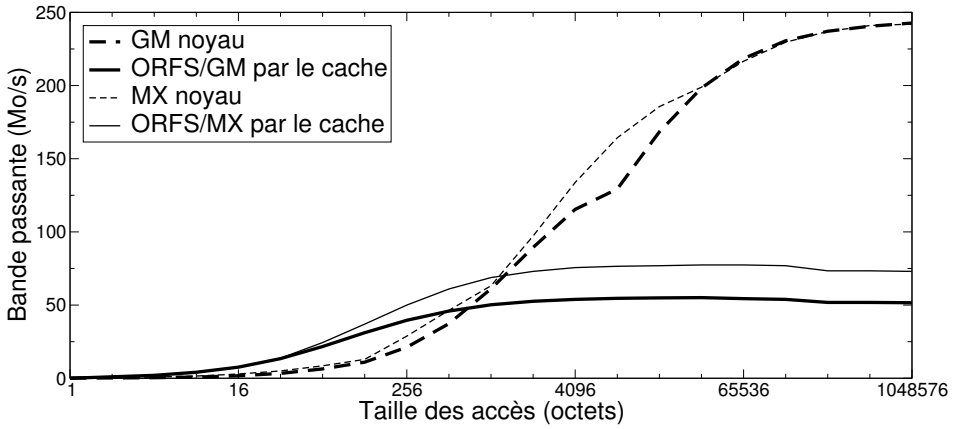


FIG. 7.6: *Performance des accès distants avec* ORFS *sur* MX *et* GM *à travers le cache.*

débit.

Les performances des accès ORFS par le cache sur MX présentent une bande passante 40 % supérieure à celle sur GM. Le temps d'un accès est de 54,2 μs sur MX, contre 76 sur GM. Pourtant, la différence de latence pour l'envoi de la requête (environ 64 octets) est de 3 μs tandis que celle du retour des données lues est de 10 μs. Nous pouvons donc en déduire que seuls 13 sur les 22 microsecondes de gain qu'apporte MX sont liées à ses performances brutes. Les 9 μs restant sont dus à la gestion dans MX des fonctionnalités avancées, en particulier les stratégies de notification d'événements qui permettent de se passer d'un thread *Dispatcher* (voir sur la figure 6.8). L'utilisation de notre interface flexible pour préciser le type d'adressage mémoire dans MX se révèle au moins aussi efficace que les primitives basées sur l'adresse physique que nous avions ajoutées à GM en partie 5.2.3.b. Ces résultats ont été publiés dans [E].

7.2.3 Optimisation des transferts des moyens messages

L'assistance de l'application que nous avons mise en place dans l'interface noyau de MX nous permet par ailleurs d'optimiser le traitement des communications selon le type de mémoire utilisé. Nous l'avons expliqué en ce qui concerne le verrouillage et la traduction des adresses. Mais il est également possible d'en profiter pour supprimer les copies intermédiaires inutiles.

En effet, nous avons expliqué en partie 7.1.1 que MX utilise des adresses physiques dans la carte d'interface. La copie intermédiaire pour les moyens messages est utilisée pour placer les données dans une zone dont MX connait les adresses physiques. Cette copie est inutile dans le cas où l'application décrit les zones mémoire directement par leurs adresses physiques.

Pour prouver le concept, nous avons mis en place dans MX une émission par adresse physique sans copie intermédiaire. Cette stratégie ne s'applique qu'aux messages de taille moyenne. La bande passante observée est présentée sur la figure 7.7. On observe un gain de 15 Mo/s pour les messages de taille 32 ko (environ 8 %). Le temps d'accès est en réduit de 165 à 153 μs, ce qui correspond assez bien aux 15 μs nécessaires pour copier 32 ko sur nos machines.

Nous avons également prédit l'impact de la suppression de la copie du côté récepteur. L'impact atteindrait également 15 Mo/s pour les messages de taille 32 ko, ce qui permet d'atteindre 233 des 250 Mo/s disponibles sur le lien. Cette bande passante est d'ailleurs bien plus élevée que celle obtenue pour les gros messages. On pourrait donc envisager ici d'augmenter le seuil séparant le traitement des moyens et gros messages lorsqu'on manipule les zones mémoire par leur adresse physique. Cependant, il faudrait là aussi trouver un compromis entre performance et consommation de cycles processeur.

Il nous a été impossible de mettre en place réellement cette seconde suppression car MX ne place pas les descripteurs de réception dans la carte d'interface. Soit les messages sont reçus dans des zones temporaires que la carte connaît (petits et moyens mes-

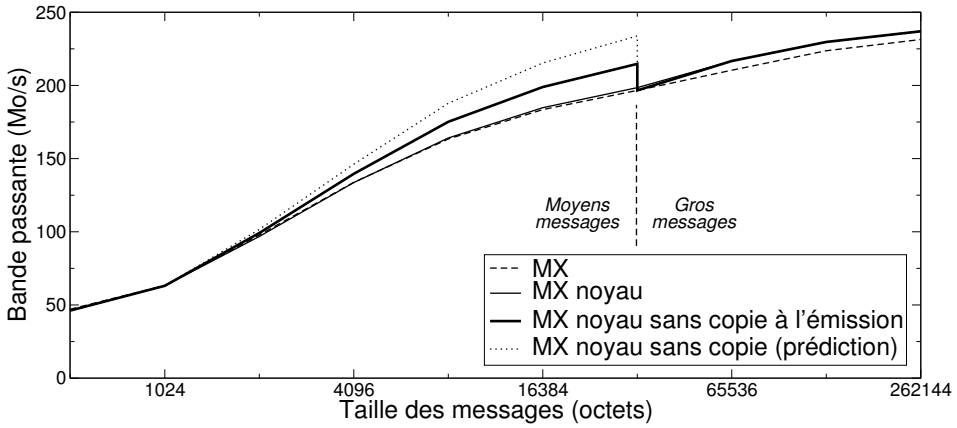


FIG. 7.7: *Optimisation de la bande passante de* MX *pour des messages de taille moyenne dans le noyau par suppression de copie. La suppression de la copie en réception n'a pu être que simulée.*

sages), soit l'événement est remonté dans l'hôte pour qu'il aille chercher les données par RDMA (gros messages). Il n'est donc pas possible de préciser à la carte où un message particulier devra être reçu. Ce problème devrait disparaître lorsque MX effectuera le *Matching* des réceptions dans la carte d'interface. Les descripteurs de réception y seront alors stockés et il pourrait nous être possible de préciser directement l'adresse physique de la zone de réception.

Ces optimisations nous permettent d'améliorer très sensiblement la bande passante et de réduire la consommation processeur dans le cas des accès aux fichiers distants via le cache. Cependant, nous sommes limités au cas où la zone mémoire est physiquement contiguë. Or, l'allocation des pages physiques dans LINUX ne se fait jamais de manière contiguë (sauf pour certaines structures du système d'exploitation, voir le *mapping linéaire* en annexe D.1). En particulier, deux pages du cache décrivant des segments successifs d'un même fichier ne sont généralement pas contiguës en mémoire physique. Notre optimisation ne s'appliquera en fait que pour des transferts de la taille d'une page (4 ko sur architecture IA-32). Dans ce cas, l'apport sera d'environ 6 Mo/s soit 5 % pour chacune des copies supprimées.

Par ailleurs, nous avons expliqué en partie 7.1.3.d que MX utilisait des adresses physiques dans la carte d'interface pour décrire les zones virtuellement contiguës passées par l'application. En combinant notre stratégie zéro-copie pour les communications par adresse physique et les structures de données utilisées par MX pour décrire des pages physiquement non-contiguës pour les gros messages, nous pouvons envisager de généraliser notre optimisation. Mais les modifications trop intrusives que cela aurait demandé ne nous ont pas permis de mettre en place cette idée pour le moment.

7.2.4 Adaptation à d'autres applications dans le noyau

Nous avons conçu l'interface noyau de MX en nous concentrant sur les besoins du stockage distribué. Nous présentons maintenant une étude de son utilisation dans un contexte très différent, celui d'un protocole SOCKET zéro-copie.

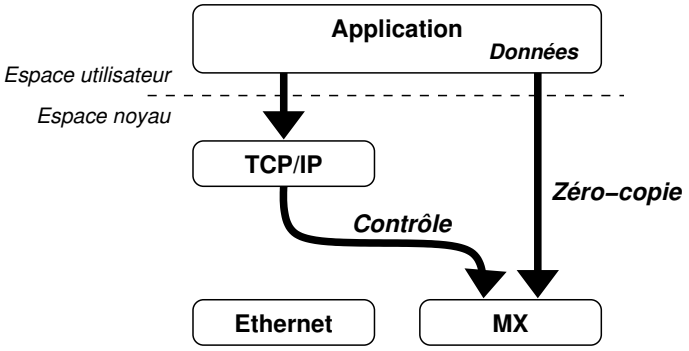
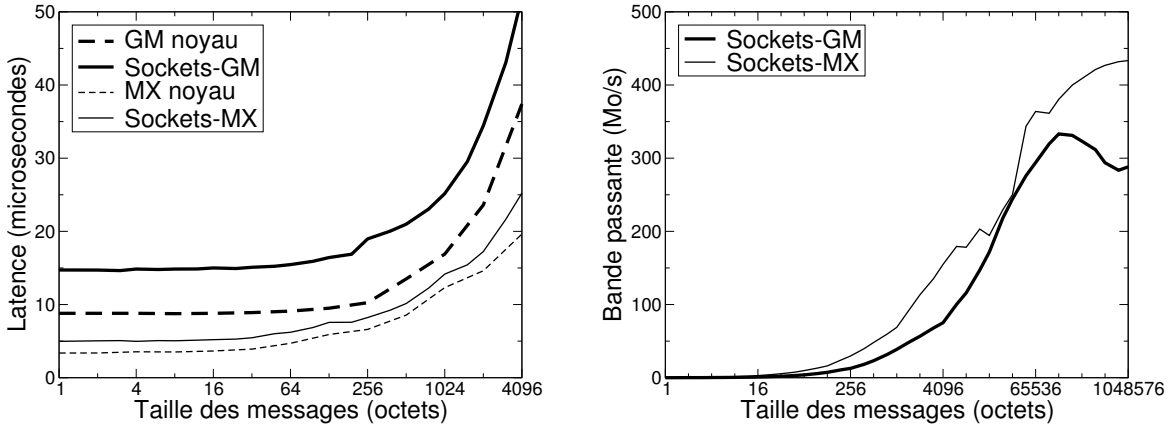


FIG. 7.8: *Implantation de* SOCKETS-MX.

La figure 7.8 présente la mise en œuvre de SOCKETS-MX. Il s'agit de permettre aux applications existantes de bénéficier du réseau haute performance MYRINET en utilisant l'interface SOCKET. SOCKETS-MX ajoute un nouveau protocole SOCKET au noyau LINUX pour transférer directement les données sur le réseau MYRINET en contournant la pile TCP/IP. Le recouvrement efficace des communications dans MX réduit significativement le coût de l'accès au réseau par rapport à la pile TCP/IP traditionnelle (qui impose la fragmentation et le calcul de sommes de contrôle). En fait TCP/IP est connu pour engendrer 50 % du coût total d'une communication [Sum00].

Le contrôle utilise les traditionnelles couches TCP/IP du système tandis que les données sont transférées directement par zéro-copie depuis la mémoire de l'application, sans subir une copie intermédiaire dans les *Socket Buffers* (voir en partie 2.2.1.a). SOCKETS-MX nécessite d'une part de transférer depuis le noyau des données situées dans l'espace utilisateur des applications, mais aussi ses messages de contrôle dans la mémoire du noyau. L'utilisation de notre interface noyau se révèle particulièrement adaptée.

La figure 7.9 présente une évaluation des performances de SOCKETS-MX et son implantation similaire sur GM, SOCKETS-GM. Les mesures de latence montrent que les performances brutes de MX peuvent être observées depuis une application réelle. La latence des petits messages est de 5 μs sur SOCKETS-MX, 15 sur SOCKETS-GM (et beaucoup plus sur un réseau GIGABIT ETHERNET classique). Le surcoût de SOCKETS-MX sur MX n'est que d'une microseconde alors qu'il implique un appel-système (400 ns sur nos machines). Par contre, le surcoût de GM est de 6 microsecondes, en particulier parce que la seule méthode d'attente d'événements de GM impose d'utiliser un thread *Dispatcher* (voir la figure 6.8).



(a) Latence.

(b) Bande passante.

FIG. 7.9: *Comparaison des performances de* SOCKETS-MX *et* SOCKETS-GM. *Les performances de* SOCKETS-MX *et* SOCKETS-GM *ont été mesurées avec un* Ping-Pong NETPIPE *[Net]. Les cartes* MYRINET PCIXE *utilisées peuvent soutenir 500 Mo/s dans chaque direction en utilisant deux liens.*

La bande passante de SOCKETS-MX est toujours supérieure à celle de SOCKETS-GM. Le gain atteint notamment 100 % pour l'émission de 4 ko et 50 % pour 1 Mo. La différence est tout particulièrement liée aux problèmes d'enregistrement mémoire. SOCKETS-GM rencontre en effet les mêmes problèmes que ORFS dans le cas des transferts directs puisque les données doivent être transférées directement vers l'espace mémoire de l'application (voir en partie 5.2.2). Par contre, SOCKETS-MX se révèle efficace puisque notre interface flexible de gestion de la mémoire est adaptée à ses besoins.

7.2.5 Synthèse

Nous avons présenté dans cette partie les apports d'une interface moderne de programmation des réseaux haute performance, *Myrinet Express*. MX intègre un certain nombre de fonctionnalités avancées afin de faciliter une mise en place efficace de MPI. Les concepteurs de cette interface ont trouvé un compromis intéressant entre les performances brutes de l'interface (qui peuvent souffrir de sa complexité) et l'utilité des fonctionnalités qui y sont proposées pour facilité la mise en place des applications. Nous avons montré que **ces fonctionnalités s'adaptent également particulièrement bien aux problèmes du contrôle des communications** que nous avons présentés en partie 6.

Tout d'abord, la **gestion efficace des messages inattendus** dans MX permet d'éviter les problèmes de gaspillage de bande passante du lien du serveur lorsque celui-ci ne peut pas recevoir les requêtes suffisamment vite. Ensuite, les **méthodes variées de notifications d'événements** permettent d'attendre la terminaison d'une communication

quelconque ou celle d'une communication particulière. La mise en œuvre d'un modèle client-serveur tel que celui de ORFA se révèle particulièrement aisée et efficace. Par contre, l'interaction de l'interface de notification de MX avec les mécanismes d'entrées-sorties standard tels que LINUX AIO reste impossible.

MX évite par ailleurs à l'application d'avoir à enregistrer explicitement les zones mémoire qu'elle va mettre en jeu dans les communications. Cela permet d'utiliser des applications classiques sans les modifier et sans avoir à intercepter leurs appels pour enregistrer la mémoire à la volée.

Cependant, MX ne proposait pas d'interface noyau lorsque nous avons débuté nos travaux. Nous avons donc tout d'abord implanté la même interface de programmation dans le noyau qu'en espace utilisateur. Nous avons ensuite ajouté des primitives supplémentaires pour **permettre à une application s'exécutant dans le noyau de préciser si les zones mémoire mises en jeu dans une communication sont des zones de l'espace noyau, de l'espace utilisateur d'un processus ou des zones de la mémoire physique**.

Les premières évaluations de performances nous ont montré que les performances obtenues avec l'interface noyau de MX étaient aussi bonnes qu'en espace utilisateur. Mais l'assistance de l'application pour préciser le type de mémoire mise en jeu nous a également permis de montrer que cela ouvrait la voie à des optimisations spécifiques. Pour prouver le concept, nous avons supprimé, à titre expérimental, une copie intermédiaire lors de l'émission de moyens messages (32 ko) par adresse physique, ce qui permet d'augmenter la bande passante de 8 %.

	Limites de GM	Apports de MX
Latence noyau brute	8 μs (6 en espace utilisateur)	4 μs (4 en espace utilisateur)
Accès aux fichiers distants (par le cache)	Nécessite interface physique Recompilation du firmware Thread *Dispatcher*	40 % de gain par rapport à GM
Accès aux fichiers distants (direct)	Enregistrement mémoire complexe Nécessite *patch* noyau Recompilation du firmware Thread *Dispatcher*	Au moins aussi bon que GM
Latence SOCKET zéro-copie	15 μs (surcoût de 7) Thread *Dispatcher*	5 μs (surcoût de 1)
Bande-passante SOCKET zéro-copie	Enregistrement mémoire complexe Moins de 70 % du lien	Jusqu'à 100 % de gain par rapport à GM

TAB. 7.1: *Résumé des limites de* GM *et des apports de* MX *dans le cadre des applications de stockage distribué et du protocole* SOCKET *zéro-copie.*

La table 7.1 résume les limites de GM et les apports de MX pour des applications réelles dans le noyau LINUX. Les expérimentations de ORFS sur MX ont confirmé nos

attentes puisque d'une part notre interface noyau s'est révélée beaucoup plus simple à utiliser que celle de GM, et d'autre part les performances observées sont bien meilleures, en particulier pour les accès aux fichiers distants à travers le cache. L'étude d'une autre application dans le noyau, SOCKETS-MX, qui consiste à permettre aux applications utilisant les Sockets de bénéficier des performances du réseau MYRINET, a montré un apport similaire de nos travaux dans un contexte très différent de celui du stockage. Ces résultats ont été publiés dans [D].

Nous nous attendons à ce que MX soit également particulièrement adapté à une utilisation dans le cadre d'un accès aux données distantes au niveau bloc (*Network Block Device*, voir en partie 2.1.1.e) . En effet, les besoins de cette application sont très proches de ceux de l'accès aux fichiers distants à travers le cache.

Nous avons donc montré dans cette partie que l' **intégration fine des fonctionnalités nécessaires à des communications MPI efficaces** accompagnée d'une **interface de programmation gérant de manière souple et assistée par l'application les différents types de mémoire** permet d'**utiliser efficacement le réseau haute performance** des grappes pour le stockage distribué.

Chapitre 8

Conclusion et perspectives

Le développement des grappes de stations interconnectées par des réseaux haute performance a permis l'exécution parallèle performante d'applications de calcul scientifique. Les besoins de ces applications en communications rapides ont abouti au développement de ces réseaux tandis que les besoins en stockage se sont plutôt traduits par des travaux sur le modèle d'accès au stockage distant, avec notamment la parallélisation du traitement des requêtes dans différents serveurs.

Nous avons présenté dans cette thèse une autre approche consistant à utiliser efficacement le réseau haute performance pour améliorer l'accès au stockage distant. Il s'agit donc d'optimisations supplémentaires qui peuvent être appliquées aux systèmes de stockage existants, par exemple aux modèles parallèles.

8.1 Contributions

Notre protocole expérimental ORFA nous a permis de mettre en évidence au chapitre 4 les apports de l'utilisation des réseaux haute performance pour le stockage distribué. La grande bande passante permet de transférer très efficacement les données lorsque l'application effectue des larges accès aux fichiers. En revanche, la faible latence des communications ne permet pas de remettre en cause la nécessité du cache de fichiers côté client. La réduction du temps d'accès au serveur distant est significative, mais les stratégies mettant en jeu un cache du côté client ou des lectures à l'avance (*Read-Ahead*) ont un impact bien plus important sur les performances vues de l'application.

Cette étude préliminaire nous a ensuite conduit à étudier les transferts de données sur le réseau dans le cadre du stockage distribué, puis le contrôle des communications. Nous avons mené au chapitre 5 une étude détaillée des besoins en transfert de données depuis les couches système d'accès au stockage afin de voir dans quelle mesure ils étaient compatibles avec le modèle de programmation des réseaux haute performance. Nous avons mis en œuvre les accès directs aux fichiers distants et les accès à travers le cache avec l'interface de programmation GM des réseaux MYRINET. Cette mise en œuvre dans le noyau LINUX a présenté de nombreux problèmes, en particulier à cause des mécanismes d'enregistrement mémoire imposés par le réseau rapide. Tout d'abord, l'obtention d'accès directs efficaces aux fichiers distants a nécessité d'une

part l'adaptation de l'interface de programmation réseau au multiplexage des canaux de communication et d'autre part la modification du système d'exploitation pour permettre à notre client d'être informé des modifications de l'espace d'adressage des applications (VMA SPY). Ensuite, le support efficace des accès à travers le cache de fichiers du système d'exploitation nous a mené à remettre en cause le modèle d'enregistrement mémoire actuellement souvent utilisé dans les grappes. Nous avons montré que ce modèle n'était pas adapté à ce contexte et avons mis en place une interface de programmation basée sur l'adresse physique. Cette interface simplifie la mise en place de ce type d'accès et en améliore les performances.

Ces expérimentations nous ont permis de montrer qu'il était possible d'utiliser très efficacement le réseau des grappes pour accéder aux fichiers distants. Mais elles ont surtout révélé de manière très claire les lacunes des systèmes d'exploitation et des interfaces de programmation des réseaux haute performance.

L'étude du contrôle des communications au chapitre 6 nous a conduit à examiner des problèmes liés au modèle client-serveur. Ce modèle présente des besoins très différents des traditionnelles applications parallèles réparties sur une grappe. Tout d'abord, le trafic hiérarchisé engendre un goulot d'étranglement au niveau des serveurs. Nous avons montré que la gestion des messages inattendus avait un impact important sur les performances du serveur. À défaut d'une gestion efficace dans la couche protocolaire, nous avons proposé l'utilisation d'accès mémoire à distance controlés par le serveur pour éviter que les clients ne gaspillent la bande passante réseau par manque de coordination avec le serveur. Dans un second temps, nous nous sommes intéressés aux notifications d'événements dans le modèle client-serveur de stockage distribué et avons montré qu'il était nécessaire de pouvoir attendre la terminaison d'une communication particulière ou quelconque.

Ces deux résultats rejoignent en fait les besoins des applications parallèles, ces fonctionnalités étant souvent mises en place dans la couche MPI. Mais nous avons également mis en évidence des problèmes spécifiquement liés au serveur de stockage en montrant que les notifications d'événements dans les réseaux haute performance ne pouvaient pas être traitées conjointement à celles des entrées-sorties standard. Nous avons proposé un mécanisme de notification réseau dans le système d'exploitation afin de permettre le traitement unifié de toutes les entrées-sorties et donc une amélioration de la réactivité et de l'efficacité du serveur.

À la lumière de ces résultats, nous avons présenté au chapitre 7 des propositions pour améliorer l'utilisation des réseaux haute performance dans le cadre du stockage distribué. Nous avons tout d'abord montré que l'intégration dans les couches protocolaires bas niveau de fonctionnalités avancées utilisées dans MPI (messages inattendus, attentes d'événements et communications vectorielles) permettait un contrôle beaucoup plus efficace des communications pour le stockage. La disponibilité de ce type de fonctionnalité dans la nouvelle interface de programmation MX des réseaux MYRINET permet un développement beaucoup plus aisé et efficace. Par contre, le traitement des transferts de données restait problématique et nous a amené à proposer une interface flexible d'adressage de la mémoire dans le noyau. Nous avons intégré cette idée

dans MX et montré que ce support natif de différents types de mémoire permettait de gérer efficacement les transferts de données dans différents types d'accès aux fichiers distants, mais également dans d'autres applications, comme par exemple les *Sockets* sur MX.

8.2 Perspectives

Ces résultats ouvrent de nombreuses pistes intéressantes que nous souhaitons explorer dans l'avenir.

Application de MX aux systèmes de fichiers distribués de production Tout d'abord, la mise œuvre de nos travaux dans un système de stockage distribué de production permettra de vérifier notre approche et d'affiner notre réponse aux besoins en stockage des applications parallèles. Nous mesurerons l'impact combiné des stratégies traditionnelles de parallélisation du serveur de fichiers et de nos travaux sur l'interaction entre le réseau et le stockage par une mise en œuvre de LUSTRE ou de PVFS dans le noyau LINUX sur MX.

Nous avons étudié la mise en œuvre de LUSTRE sur GM et avons constaté que le problème majeur était la nécessité de copier les données dans des zones intermédiaires pré-enregistrées pour éviter le problème de l'enregistrement mémoire dans le noyau. Nous nous attendons à ce que notre interface de MX dans le noyau résolve totalement ce problème et se révèle particulièrement efficace.

Étude des accès au niveau bloc Les accès au niveau des périphériques blocs doivent être étudiés pour compléter l'ensemble des différents types d'accès aux fichiers distants. Il s'agira de mettre en œuvre un *Network Bloc Device* sur MX. Nous nous attendons à ce que ce travail ne présente aucun problème d'interaction entre l'interface de programmation du réseau et les couches système d'accès aux périphériques blocs. En effet, les besoins de ce type d'implantation sont très proches de ceux de l'accès aux fichiers à travers le cache. L'interface de MX dans le noyau doit donc se révéler tout à fait adaptée à ce genre de problème.

Poursuite des travaux dans MX Nous allons poursuivre notre collaboration avec la société MYRICOM pour travailler sur les fonctionnalités en cours de développement dans MX. Il s'agira notamment de tirer profit des primitives vectorielles de communication. Ce type de communication nous permet en effet d'envisager un gain important en performances pour les accès distants depuis le cache ou au niveau bloc en agrégeant plusieurs pages non-contiguës.

Par ailleurs, la flexibilité de l'interface noyau permet d'utiliser les informations fournies par l'application pour décrire précisément les zones mémoire mises en jeu dans les communications. Cette assistance peut permettre des optimisations spécifiques au

contexte noyau, en particulier des stratégies zéro-copie telles que celle que nous avons présentée en 7.2.3.

Support générique pour les réseaux haute performance dans le noyau Les problèmes de notification de modification d'espace d'adressage que nous avons exposés en partie 5.2.2.b ont montré les lacunes du système d'exploitation pour répondre efficacement aux besoins des communications zéro-copie dans les réseaux haute performance. Ces problèmes ne sont pas exposés à l'application dans MX puisque l'enregistrement mémoire est effectué en interne. Mais la mise en œuvre interne peut également utiliser un cache d'enregistrement et par conséquent nécessiter les mêmes techniques de notification des modifications d'adressage que celles utilisées par notre cache d'enregistrement GMKRC.

Nous avons mené des discussions avec les développeurs du noyau LINUX, de QUADRICS et d'OPENIB à propos du support qui devrait être intégré aux noyaux standard pour faciliter la mise en œuvre de ces communications zéro-copie. QUADRICS préconise l'intégration de leur patch permettant de maintenir la MMU de la carte ELAN à jour. Nous avons signalé l'existence de notre approche VMA SPY dont les buts sont très similaires mais la mise en œuvre assez différente (voir en annexe E) sans pouvoir conclure à la supériorité d'une des deux propositions.

En revanche, les développeurs d'OPENIB préfèrent ne pas modifier le système d'exploitation mais contraindre l'application, par exemple en empêchant le *Copy-on-Write*. Cette approche nous semble peu adaptée aux problèmes que MYRICOM et QUADRICS ont rencontrés depuis une dizaine d'années. Aucun consensus ne s'est dégagé pour le moment mais nous allons continuer les discussions avec les développeurs du noyau afin qu'une stratégie proche de ce que QUADRICS propose ou de notre infrastructure VMA SPY soit intégrée dans LINUX.

Généralisation de l'approche à l'ETHERNET 10G L'avènement des cartes ETHERNET 10 Gigabit/s proposant des communications par RDMA ou TOE (*TCP Offload Engine*) va augmenter les besoins en support de communications zéro-copie. En effet, de tels débits de données ne permettent pas de conserver les couches protocolaires TCP/IP actuelles car les processeurs des machines sont incapables d'assurer la charge de traitement qu'elles imposent. Ces cartes réseau proposent donc des stratégies de communications zéro-copie, ce qui soulève des problèmes très similaires à ceux que nous avons rencontrés dans les réseaux haute performance.

La généralisation de ce type de matériel dans les serveurs d'ici quelques années imposera une standardisation de son support logiciel. Cela devrait notamment se traduire par des fonctionnalités du noyau LINUX assistant ce type de communications et celles des réseaux haute performance. Nous allons étudier ces travaux pour voir dans quelle mesure nos propositions peuvent s'y appliquer et ainsi faciliter l'interaction de ces technologies réseau avec les systèmes de stockage.

Application aux grilles de calcul Nous nous sommes concentrés dans cette thèse sur le problème du stockage distribué efficace pour répondre aux besoins des applications parallèles s'exécutant sur les grappes. Nous avons proposé une approche complémentaire aux traditionnelles stratégies de parallélisation en nous basant sur l'utilisation efficace du réseau par l'adaptation de son interface de programmation.

Comme nous l'avons constaté en partie 7.2.4, nos travaux s'appliquent également particulièrement bien à un protocole SOCKET zéro-copie. Ce modèle permet aux applications classiques de bénéficier des performances élévées du réseau sous-jacent en conservant l'interface SOCKET. Ces travaux permettent d'envisager l'utilisation de ce protocole sur les grilles de calcul où les différentes grappes de stations reliées par des réseaux haute performance sont interconnectées par des réseaux TCP/IP haut débit courte ou longue distance. L'utilisation d'un protocole unique pour communiquer à l'intérieur d'une grappe ou entre deux grappes de la grille permet d'envisager de mettre en place des applications parallèles pour les grilles beaucoup plus aisément puisqu'il ne sera plus nécessaire de modifier le protocole selon la localisation de l'application.

D'une manière générale, nos travaux permettent une meilleure intégration des couches protocolaires des réseaux haute performance dans le système d'exploitation. Cela permet leur utilisation efficace dans les applications standard, que ce soit les protocoles du type SOCKET ou le stockage distribué. L'étape suivante consisterait à mettre en place une transition transparente entre les réseaux haute performance des grappes et les réseaux TCP/IP longue distance qui les relient entre elles. Le transport serait ainsi assuré d'utiliser au mieux le matériel sous-jacent de bout en bout. Ce genre d'innovation couplée à une interaction efficace avec les systèmes de stockage pourrait permettre d'améliorer les transferts de données entre les différents dispositifs de stockage qui sont considérés comme étant le prochain goulot d'étranglement dans les grilles de calcul.

Annexe A

Interfaces logicielles de programmation

A.1 Entrées-sorties dans LINUX

A.1.1 Interface standard UNIX

L'interface standard d'entrées-sorties UNIX permet de manipuler les différents périphériques, fichiers ou *sockets* sous forme de descripteurs (identifiants entiers).

```
int open(const char *pathname, int flags, mode_t mode);
int socket(int domain, int type, int protocol);
int close(int fd);

ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

Les accès se font généralement de manière séquentielle. Cependant, certains descripteurs (en particulier les fichiers) supportent également des accès aléatoires.

```
off_t lseek(int fd, off_t offset, int whence);

ssize_t pread(int fd, void *buf, size_t count, off_t offset);
ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset);
```

Des primitives vectorielles permettent par ailleurs de manipuler des zones mémoire non-contiguës.

```
int readv(int fd, const struct iovec *vector, size_t count);
int writev(int fd, const struct iovec *vector, size_t count);

struct iovec {
  void *iov_base;   /* Starting address */
  size_t iov_len;   /* Number of bytes */
};
```

Les systèmes modernes permettent également des accès zéro-copie. Pour cela, il faut ouvrir le fichier avec le paramètre `O_DIRECT` pour indiquer au noyau de ne pas cacher les données (voir les détails d'implémentation dans LINUX en annexe B). Les routines de lecture-écriture se traduisent alors par des transferts directs entre le périphérique de stockage et la mémoire de l'application. Cela impose de respecter la granularité du bloc disque (512 octets en général).

A.1.2 Interface asynchrone LINUX

Depuis LINUX 2.5, les applications peuvent soumettre au niveau utilisateur des requêtes d'entrées-sorties nativement asynchrones. L'absence de réelle demande des utilisateurs a conduit les développeurs à ne pas implanter le support des sockets dans cette interface.

```
long io_submit (aio_context_t ctx_id, long nr, struct iocb **iocbpp);

enum {
  IOCB_CMD_PREAD,
  IOCB_CMD_PWRITE,
  IOCB_CMD_FSYNC,
  IOCB_CMD_POLL,
};

struct iocb {
  uint16_t aio_lio_opcode; /* see IOCB_CMD_ above */
  int16_t  aio_reqprio;
  uint32_t aio_fildes;
  uint64_t aio_buf;
  uint64_t aio_nbytes;
  int64_t  aio_offset;
};
```

Ces requêtes sont placées dans un *contexte* d'entrées-sorties et traitées en tâche de fond par le système d'exploitation.

```
long io_setup (unsigned nr_events, aio_context_t *ctxp);
long io_destroy (aio_context_t ctx);
```

L'application peut ensuite tester ou attendre la terminaison d'une ou plusieurs requêtes dans un contexte.

```
long io_getevents (aio_context_t ctx_id, long min_nr, long nr,
                   struct io_event *events, struct timespec *timeout);

struct io_event {
  uint64_t data;          /* the data field from the iocb */
  uint64_t obj;           /* what iocb this event came from */
  int64_t  res;           /* result code for this event */
  int64_t  res2;          /* secondary result */
};
```

A.2 Entrées-sorties de haut niveau dans les grappes

A.2.1 Message Passing Interface (MPI)

L'interface MPI permet aux différents nœuds d'une application parallèle de communiquer sans avoir besoin d'ouvrir préalablement des connexions, en précisant directement le destinataire (par son identifiant, un entier). MPI fournit différentes primitives d'émission ou réception (bloquante ou non, synchrone ou non, ...).

MPI suit le paradigme du *Rendez-vous*, c'est-à-dire qu'à chaque requête d'émission correspond une requête de réception. La reconnaissance de la requête de réception correspondante se fait par le biais d'un *tag*. On parle de *Matching*.

```
int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm,
              MPI_Request *request);
int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm,
              MPI_Request *request);
```

Les requêtes sont associées à un identifiant de type `MPI_Request` qui permet à l'application d'être notifiée de leur terminaison. Il est possible de tester ou attendre la terminaison d'une requête particulière, d'une requête quelconque, de certaines ou de toutes les requêtes en cours.

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);
int MPI_Wait(MPI_Request *request, MPI_Status *status);
int MPI_Testany(int count, MPI_Request *array_of_requests, int *index, int *flag, MPI_Status *status);
int MPI_Waitany(int count, MPI_Request *array_of_requests, int *index, MPI_Status *status);
int MPI_Testall(int count, MPI_Request *array_of_requests, int *flag, MPI_Status *array_of_statuses);
int MPI_Waitall(int count, MPI_Request *array_of_requests, MPI_Status *array_of_statuses);
int MPI_Testsome(int incount, MPI_Request *array_of_requests, int *outcount, int *array_of_indices,
                 MPI_Status *array_of_statuses);
int MPI_Waitsome(int incount, MPI_Request *array_of_requests, int *outcount, int *array_of_indices,
                 MPI_Status *array_of_statuses);
```

MPI fournit enfin des primitives collectives de type *Barrière* et *Broadcast*.

```
int MPI_Barrier(MPI_Comm comm);
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);
```

L'extension MPI2 a ensuite introduit des primitives d'accès mémoire à distance (RDMA) basées sur des fenêtres (*Window*) que l'application déclare préalablement. Dans ce cas, seul le nœud initiateur de la lecture ou de l'écriture distante sera notifié de la terminaison de la requête.

```
int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm,
                   MPI_Win *win);
int MPI_Win_free(MPI_Win *win);
int MPI_Put(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank,
            MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win win);
int MPI_Get(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank,
            MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win win);
```

Par ailleurs, les problèmes d'enregistrement mémoire ont conduit à ajouter des routines d'allocation mémoire dans l'interface. MPI2 garantit que la mémoire ainsi allouée sera prête à l'emploi dans les communications. Ceci permet d'éviter les éventuels surcoûts dus à l'enregistrement mémoire ou à des copies dans les bibliothèques de bas niveau.

```
int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void *baseptr);
int MPI_Free_mem(void *base);
```

Enfin, l'extension MPI-IO a généralisé l'interface MPI aux accès disque. L'application manipule des identifiants de requête de type `MPI_Request` indépendamment du fait qu'il s'agisse d'une requête réseau ou disque.

```
int MPI_File_open(MPI_Comm comm, char *filename, int amode, MPI_Info info, MPI_File *fh);
int MPI_File_close(MPI_File *fh);

int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype, MPI_Datatype filetype,
                      char *datarep, MPI_Info info);

int MPI_File_iread_at(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype,
                      MPI_Request *request);
int MPI_File_iwrite_at(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype,
                       MPI_Request *request);
int MPI_File_iread_shared(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
                          MPI_Request *request);
int MPI_File_iwrite_shared(MPI_File fh, void *buf, int count, MPI_Datatype datatype,
                           MPI_Request *request);
```

A.2.2 Virtual Interface Architecture (VIA)

La diversité des interfaces de programmation des différents réseaux a conduit au développement de l'interface VIA censée les uniformiser. VIA se base sur des connexions entre des `ViHandle`.

```
VIP_RETURN VipCreateVi (VIP_NIC_HANDLE NicHandle, VIP_VI_ATTRIBUTES *ViAttribs,
                        VIP_CQ_HANDLE SendCQHandle, VIP_CQ_HANDLE RecvCQHandle,
                        VIP_VI_HANDLE *ViHandle);
VIP_RETURN VipDestroyVi (VIP_VI_HANDLE ViHandle);
```

L'application doit créer un `ViHandle` local pour chaque nœud avec lequel elle souhaite communiquer. Deux `ViHandle` devant communiquer entre eux doivent ensuite être connectés grâce à une interface proche de SOCKET.

```
VIP_RETURN VipConnectRequest (VIP_VI_HANDLE ViHandle,
                              VIP_NET_ADDRESS *LocalAddr, VIP_NET_ADDRESS *RemoteAddr,
                              VIP_ULONG Timeout, VIP_VI_ATTRIBUTES *RemoteViAttribs);
VIP_RETURN VipConnectWait (VIP_NIC_HANDLE NicHandle, VIP_NET_ADDRESS *LocalAddr,
                           VIP_ULONG Timeout, VIP_NET_ADDRESS *RemoteAddr,
                           VIP_VI_ATTRIBUTES *RemoteViAttribs, VIP_CONN_HANDLE *ConnHandle);
VIP_RETURN VipConnectAccept (VIP_CONN_HANDLE ConnHandle, VIP_VI_HANDLE ViHandle);
```

L'application peut ensuite poster des requêtes asynchrones d'envoi ou réception dans une connexion.

```
VIP_RETURN VipPostSend (VIP_VI_HANDLE ViHandle, VIP_DESCRIPTOR *DescriptorPtr,
                        VIP_MEM_HANDLE MemoryHandle);
VIP_RETURN VipPostRecv (VIP_VI_HANDLE ViHandle, VIP_DESCRIPTOR *DescriptorPtr,
                        VIP_MEM_HANDLE MemoryHandle);
```

Cependant, les zones de mémoire manipulées lors des communications doivent être explicitement passées à la bibliothèque VIA qui va les *préparer,* par exemple en les enregistrant dans la bibliothèque de communication sous-jacente.

```
VIP_RETURN VipRegisterMem (VIP_NIC_HANDLE NicHandle, VIP_PVOID VirtualAddress, VIP_ULONG Length,
                           VIP_MEM_ATTRIBUTES *MemAttribs, VIP_MEM_HANDLE *MemoryHandle);
VIP_RETURN VipDeregisterMem (VIP_NIC_HANDLE NicHandle, VIP_PVOID VirtualAddress,
                             VIP_MEM_HANDLE MemoryHandle);
```

Les requêtes postées dans une connexion sont traitées dans l'ordre de leur soumission. L'attente de leur terminaison consiste donc à attendre la terminaison de la plus vieille requête.

```
VIP_RETURN VipSendWait (VIP_VI_HANDLE ViHandle, VIP_ULONG Timeout, VIP_DESCRIPTOR **DescriptorPtr);
VIP_RETURN VipRecvWait (VIP_VI_HANDLE ViHandle, VIP_ULONG Timeout, VIP_DESCRIPTOR **DescriptorPtr);
```

Si des `CQHandle` d'envoi et/ou de réception sont passés à la création du `ViHandle`, la méthode précédente n'est plus disponible. Toutes les notifications de terminaison seront regroupées dans des *Completion Queues*, ce qui permet d'attendre la terminaison d'une requête quelconque dans une file unique.

```
VIP_RETURN VipCreateCQ (VIP_NIC_HANDLE NicHandle, VIP_ULONG EntryCount, VIP_CQ_HANDLE *CQHandle);
VIP_RETURN VipDestroyCQ (VIP_CQ_HANDLE CQHandle);
VIP_RETURN VipCQWait (VIP_CQ_HANDLE CQHandle, VIP_ULONG Timeout, VIP_VI_HANDLE *ViHandle,
                      VIP_BOOLEAN *RecvQueue);
```

A.2.3 Direct Access File System (DAFS)

DAFS propose une interface de programmation des accès aux fichiers distants calquée sur les interfaces réseau de type VIA. Le montage et l'ouverture des fichiers sont proches des méthodes traditionnelles.

```
DAP_ERROR dap_mount(DAP_CHAR * server_name, DAP_DIRECTORY_HANDLE * base_dir_handle);
DAP_ERROR dap_unmount(DAP_DIRECTORY_HANDLE dir_handle);

DAP_ERROR dap_open_file(DAP_DIRECTORY_HANDLE dir_handle, DAP_CRED_HANDLE cred_handle,
                        const DAP_CHAR * path, DAP_FLAGS flags,
                        DAP_CREATE_MODE mode, DAP_FILE_HANDLE * file_handle);
DAP_ERROR dap_close_file(DAP_FILE_HANDLE file_handle);
```

Les accès aux données sont complètement asynchrones.

```
DAP_ERROR dap_async_read(DAP_FILE_HANDLE file_handle, DAP_OFFSET file_offset,
                         DAP_COUNT io_count, DAP_MEM_DESC * mem_desc,
                         DAP_CG_HANDLE cg_handle, DAP_IO_RESULT * io_desc);
DAP_ERROR dap_async_write(DAP_FILE_HANDLE file_handle, DAP_OFFSET file_offset,
                          DAP_COUNT io_count, DAP_MEM_DESC * mem_desc,
                          DAP_CG_HANDLE cg_handle, DAP_IO_RESULT * io_desc);
```

Les requêtes sont représentées par des `DAP_IO_RESULT` dont la terminaison peut être testée ou attendue.

```
DAP_ERROR dap_io_done(DAP_IO_RESULT * io_desc);
DAP_ERROR dap_io_wait(DAP_TIMEOUT timeout, DAP_IO_RESULT * io_desc);
```

Il est possible de regrouper les requêtes dans des *Completion Groups* pour tester ou attendre la terminaison d'une requête parmi un ensemble.

```
DAP_ERROR dap_create_cg(DAP_CG_HANDLE * cg_handle, DAP_COUNT cg_entries);
DAP_ERROR dap_destroy_cg(DAP_CG_HANDLE cg_handle);
DAP_ERROR dap_cg_done(DAP_CG_HANDLE cg_handle, DAP_IO_RESULT ** io_desc);
DAP_ERROR dap_cg_wait(DAP_CG_HANDLE cg_handle, DAP_TIMEOUT timeout, DAP_IO_RESULT ** io_desc);
```

Comme VIA, DAFS charge l'application de préparer les zones mémoire qui vont être utilisées pour les accès distants. Il faut donc enregistrer la mémoire à l'avance.

```
DAP_ERROR dap_register_mem(DAP_PVOID *buffer, DAP_LENGTH length, DAP_MEM_HANDLE * mem_handle);
DAP_ERROR dap_deregister_mem(DAP_MEM_HANDLE);
```

A.3 Bibliothèques réseau de bas niveau dans les grappes

A.3.1 MYRINET GM

L'interface GM des réseaux MYRINET utilise des *ports* pour permettre aux applications de soumettre des requêtes de communication.

```
gm_status_t gm_open (struct gm_port **p, unsigned int unit, unsigned int port,
                     const char *port_name, enum gm_api_version version);
void gm_close (struct gm_port *p);
```

Les zones mémoire utilisées dans les communications doivent être préalablement enregistrées. L'identifiant des zones enregistrées est normalement son adresse virtuelle, mais il est possible de le préciser (`pvma`). Il est par ailleurs possible d'allouer de la mémoire directement enregistrée (GM se charge d'optimiser les enregistrements nécessaires).

```
gm_status_t gm_deregister_memory (struct gm_port *p, void *ptr, gm_size_t length);
gm_status_t gm_register_memory (struct gm_port *p, void *ptr, gm_size_t length);
gm_status_t gm_register_memory_ex (struct gm_port *p, void *ptr, gm_size_t length, void *pvma);
void gm_dma_free (struct gm_port *p, void *addr);
void *gm_dma_malloc (struct gm_port *p, gm_size_t length);
```

Les primitives de communication permettent de soumettre des requêtes asynchrones d'émission ou réception dans des zones contiguës en mémoire virtuelle.

```
void gm_provide_receive_buffer_with_tag (struct gm_port *p, void *ptr, unsigned int size,
                                         unsigned int priority, unsigned int tag);
void gm_send_with_callback (struct gm_port *p, void *message, unsigned int size,
                            gm_size_t len, unsigned int priority,
                            unsigned int target_node_id, unsigned int target_port_id,
                            gm_send_completion_callback_t callback, void *context);
```

Les requêtes soumises n'ont pas d'identifiant visible pour l'application. Leurs notifications remontent à l'application par une file d'attente unique. On ne peut pas tester ou attendre la terminaison d'une requête particulière. Seul le premier événement de la file est accessible.

```
union gm_recv_event *gm_receive (struct gm_port *p);
union gm_recv_event *gm_blocking_receive (struct gm_port *p);
```

Des primitives d'accès mémoire à distance ont été ajoutées dans GM2.

```
void gm_put (struct gm_port *p, void *local_buffer, gm_remote_ptr_t remote_buffer, gm_size_t len,
             enum gm_priority priority, unsigned int target_node_id, unsigned int target_port_id,
             gm_send_completion_callback_t callback, void *context);
void gm_get (struct gm_port *p, gm_remote_ptr_t remote_buffer, void *local_buffer, gm_size_t len,
             enum gm_priority priority, unsigned int target_node_id, unsigned int target_port_id,
             gm_send_completion_callback_t callback, void *context);
```

Dans nos travaux, nous avons ajouté des primitives de communication basées sur les adresses physiques dans l'interface noyau de GM. Cette interface n'est pas disponible dans la distribution officielle de GM.

```
void gm_provide_physical_receive_buffer_with_tag (struct gm_port *p, unsigned long pa,
                                                  unsigned int size, unsigned int priority,
                                                  unsigned int tag);
void gm_send_physical_with_callback (struct gm_port *p, unsigned long pa, unsigned int size,
                                     gm_size_t len, unsigned int priority,
                                     unsigned int target_node_id, unsigned int target_port_id,
                                     gm_send_completion_callback_t callback, void *context);
```

A.3.2 MYRINET Express (MX)

Le nouveau pilote MX des réseaux MYRINET utilise des *endpoints* pour permettre à l'application de communiquer avec la carte d'interface.

```
mx_return_t mx_open_endpoint(uint32_t board_number, uint32_t endpoint_id, uint32_t endpoint_key,
                             mx_param_t *params_array, uint32_t params_count,
                             mx_endpoint_t *endpoint);
mx_return_t mx_close_endpoint(mx_endpoint_t endpoint);
```

Les primitives de communication sont très proches de MPI. L'émission peut être synchrone ou non. L'utilisation de multiples zones mémoire non-contiguës est possible (communications vectorielles).

```
mx_return_t mx_irecv(mx_endpoint_t endpoint, mx_segment_t *segments_list, uint32_t segments_count,
                     uint64_t match_info, uint64_t match_mask, void *context,
                     mx_request_t *request);
mx_return_t mx_isend(mx_endpoint_t endpoint, mx_segment_t *segments_list, uint32_t segments_count,
                     mx_endpoint_addr_t dest_endpoint, uint64_t match_info, void *context,
                     mx_request_t *request);
mx_return_t mx_issend(mx_endpoint_t endpoint, mx_segment_t *segments_list, uint32_t segments_count,
                      mx_endpoint_addr_t dest_endpoint, uint64_t match_info, void *context,
                      mx_request_t *request);
```

Il est possible de tester ou d'attendre la terminaison d'une requête particulière (de type `mx_request_t`) ou bien d'une requête quelconque concernant l'*endpoint* utilisé.

```
mx_return_t mx_test(mx_endpoint_t endpoint, mx_request_t *request,
                    mx_status_t *status, uint32_t *result);
mx_return_t mx_wait(mx_endpoint_t endpoint, mx_request_t *request, uint32_t timeout,
                    mx_status_t *status, uint32_t *result);
mx_return_t mx_ipeek(mx_endpoint_t endpoint,
                     mx_request_t *request, uint32_t *result);
mx_return_t mx_peek(mx_endpoint_t endpoint, uint32_t timeout,
                    mx_request_t *request, uint32_t *result);
```

MX permet par ailleurs des accès mémoire à distance en utilisant des fenêtres préalablement déclarées (RDMA *Window*).

```
mx_return_t mx_create_rdma_window(mx_endpoint_t endpoint, mx_segment_t *segment_list,
                                  uint32_t segment_count, mx_rdma_window_flag_t flags,
                                  uint32_t *window_handle);
mx_return_t mx_destroy_rdma_window(mx_endpoint_t endpoint, uint32_t window_handle);

mx_return_t mx_rdma_read(mx_endpoint_t endpoint, mx_segment_t *segments_list,
                         uint32_t segments_count, mx_endpoint_addr_t target,
                         uint32_t window_handle, uint32_t offset, void *context,
                         mx_request_t *request);
mx_return_t mx_rdma_write(mx_endpoint_t endpoint, mx_segment_t *segments_list,
                          uint32_t segments_count, mx_endpoint_addr_t destination,
                          uint32_t window_handle, uint32_t offset, void *context,
                          mx_request_t *request);
```

Les primitives de communication de MX ne fonctionnent dans le noyau qu'avec de la mémoire noyau. Dans le cadre de nos travaux, nous avons ajouté des variantes de ces primitives utilisant un argument supplémentaire pour préciser le type d'adressage mémoire utilisé.

L'application indique alors si les segments utilisés sont situés en mémoire virtuelle noyau, en espace utilisateur ou bien représentés par leurs adresses physiques. Il est également possible de manipuler des adresses virtuelles utilisateur dans le contexte d'une autre application.

Cette nouvelle interface est disponible dans la distribution officielle de MX.

```
/* define memory types that are passed to communication routines */
#define MX_PIN_KERNEL           (1UL << 1)
#define MX_PIN_PHYSICAL         (1UL << 2)
#define MX_PIN_USER             mx_klib_memory_context()
/* MX_PIN_USER might be replaced with a specific address space pointer */

mx_return_t mx_kisend(mx_endpoint_t endpoint, mx_ksegment_t *segments_list,
                      uint32_t segments_count, uint32_t pin_type,
                      mx_endpoint_addr_t dest_endpoint, uint64_t match_info,
                      void *context, mx_request_t *request);
mx_return_t mx_krdma_read(mx_endpoint_t endpoint, mx_ksegment_t *segments_list,
                          uint32_t segments_count, uint32_t pin_type,
                          mx_endpoint_addr_t target, uint32_t window_handle,
                          uint32_t offset, void *context,
                          mx_request_t *request);
```

A.3.3 SCI SISCI

L'interface SISCI des réseaux SCI dialogue avec la carte d'interface par l'intermédiaire d'un *Virtual Device*.

```
void SCIOpen(sci_desc_t *sd, unsigned int flags, sci_error_t  *error);
void SCIClose(sci_desc_t sd, unsigned int flags, sci_error_t *error);
```

Une application peut créer des segments de mémoire et les rendre accessibles aux autres nœuds.

```
void SCICreateSegment(sci_desc_t sd, sci_local_segment_t *segment, unsigned int segmentId,
                      unsigned int size, sci_cb_local_segment_t callback, void *callbackArg,
                      unsigned int flags,sci_error_t *error);
void SCIPrepareSegment(sci_local_segment_t segment, unsigned int localAdapterNo,
                       unsigned int flags, sci_error_t *error);
void SCISetSegmentAvailable(sci_local_segment_t segment, unsigned int localAdapterNo,
                            unsigned int flags, sci_error_t *error);
```

Les autres applications peuvent se connecter aux segments distants et les mapper (projeter) dans leur espace d'adressage. Déréférencer une adresse dans cette zone provoque alors un accès mémoire à distance de manière complètement transparente.

```
void SCIConnectSegment(sci_desc_t sd, sci_remote_segment_t *segment, unsigned int nodeId,
                       unsigned int segmentId, unsigned int localAdapterNo,
                       sci_cb_remote_segment_t callback, void *callbackArg,
                       unsigned int timeout, unsigned int flags, sci_error_t *error);
void *SCIMapRemoteSegment(sci_remote_segment_t segment, sci_map_t *map, unsigned int offset,
                          unsigned int size, void *addr, unsigned int flags, sci_error_t *error);
```

Il est également possible de demander des accès mémoire à distance explicites par des *DMA Queue* dans lesquelles on ajoute des descripteurs de transferts. On poste ensuite la queue pour qu'elle soit traitée en arrière plan.

```
void SCICreateDMAQueue(sci_desc_t sd, sci_dma_queue_t *dq, unsigned int localAdapterNo,
                       unsigned int maxEntries, unsigned int flags, sci_error_t *error);
void SCIEnqueueDMATransfer(sci_dma_queue_t dq, sci_local_segment_t localSegment,
                           sci_remote_segment_t remoteSegment, unsigned int localOffset,
                           unsigned int remoteOffset, unsigned int size,
                           unsigned int flags, sci_error_t *error);
void SCIPostDMAQueue(sci_dma_queue_t dq, sci_cb_dma_t callback, void *callbackArg,
                     unsigned int flags, sci_error_t *error);
```

La notification de terminaison du traitement d'une *DMA Queue* se fait ensuite en testant son état ou en bloquant.

```
sci_dma_queue_state_t SCIDMAQueueState(sci_dma_queue_t dq);
sci_dma_queue_state_t SCIWaitForDMAQueue(sci_dma_queue_t dq, unsigned int timeout,
                                         unsigned int flags, sci_error_t *error);
```

A.3.4 QUADRICS ELANLIB, TPORTS et KCOMM

A.3.4.a ELANLIB

L'utilisation des cartes ELAN passe par un ELAN_STATE.

```
ELAN_STATE *elan_init (ELAN_FLAGS flags);
```

Les opérations standard sont des accès mémoire à distance. On commence par allouer un descripteur de *PutGet* (ELAN_PGCTRL) puis on le passe à une primitive de lecture ou écriture. La communication se fait de manière vectorielle, mais les segments sont tous de même taille et leur nombre est identique du côté receveur et émetteur.

```
ELAN_PGCTRL *elan_putgetInit (ELAN_STATE *state, void *qMem, size_t smallPutSize,
                              size_t splitPutSize, size_t splitGetSize,
                              uint32_t throttle, ELAN_FLAGS flags);
ELAN_EVENT *elan_getv (ELAN_PGCTRL *pgctrl, void **source, void **dest, size_t size,
                       uint32_t count, uint32_t srcvp);
ELAN_EVENT *elan_putv (ELAN_PGCTRL *pgctrl, void **source, void **dest, size_t size,
                       uint32_t count, uint32_t destvp);
```

On peut ensuite tester ou attendre la terminaison de la requête.

```
extern int elan_poll (ELAN_EVENT *event, long waitType);
extern void elan_wait (ELAN_EVENT *event, long waitType);
```

A.3.4.b TPORTS

L'interface TPORTS permet des communications selon le modèle du *Rendez-vous* au dessus de ELANLIB. Ces communications utilisent un ELAN_TPORT.

```
ELAN_TPORT *elan_tportInit (ELAN_STATE *state, ELAN_QUEUE *q, u_int nSlots,
                            size_t smallMsgSize, size_t bigMsgSize, size_t stripeMsgSize,
                            long waitType, long retryCount, int *shmKey, long shmFifoDepth,
                            size_t shmFragSize, ELAN_FLAGS flags);
```

L'émission d'un message renvoie un identifiant ELAN_EVENT dont on peut ensuite tester ou attendre la terminaison.

```
ELAN_EVENT *elan_tportTxStart (ELAN_TPORT *usrcp, ELAN_FLAGS tflags, u_int destvp,
                               int sender, int tag, void *base, size_t size);
int elan_tportTxDone (ELAN_EVENT *t);
void elan_tportTxWait (ELAN_EVENT *t);
```

La réception suit le même modèle.

```
ELAN_EVENT *elan_tportRxStart (ELAN_TPORT *p, ELAN_FLAGS rflags, int senderMask, int senderSel,
                               int tagMask, int tagSel, void *base, size_t size);
int elan_tportRxDone (ELAN_EVENT *r);
void *elan_tportRxWait (ELAN_EVENT *r, int *sender, int *tag, size_t *size);
int elan_tportRxPoll (ELAN_TPORT *up, int senderMask, int senderSel,
                      int tagMask, int tagSel, int *sender, int *tag, size_t *size);
```

A.3.4.c KCOMM

Les communications dans le noyau utilisent un EP_SYS

```
EP_SYS *ep_system(void);
```

Les zones mémoire doivent être mappées dans l'espace virtuel de la carte avant utilisation. On réserve d'abord un segment de cet espace (EP_NMD), puis on effectue le mapping en précisant si la zone est en espace noyau ou dans espace utilisateur d'un processus.

```
EP_NMH *ep_dvma_reserve (EP_SYS *sys, unsigned npages, unsigned perm)
void ep_dvma_load (EP_SYS *sys, void *map, caddr_t vaddr, unsigned len, EP_NMH *nmh,
                   unsigned index, EP_RAILMASK *hints, EP_NMD *subset)
```

Les communications sont associées à des services. On crée un *Receiver* EP_RCVR pour un service puis on y poste des segments de réception. On poste également une fonction *callback* à appeler quand la réception aura eu lieu, ainsi qu'un argument à lui passer.

```
EP_RCVR *ep_alloc_rcvr(EP_SYS *sys, unsigned int service, unsigned int nenvs);
int ep_queue_receive(EP_RCVR *rcvr, EP_RXH *handler, void *arg, EP_NMD *nmd, int attr);
```

L'envoi de message utilise un *Transmitter* EP_XMTR. Le message est envoyé dans un service et sera reçu par le EP_RCVR correspondant. On passe une liste de segments EP_NMD ainsi qu'un *callback*.

```
EP_XMTR *ep_alloc_xmtr(EP_SYS *sys);
EP_STATUS ep_transmit_message(EP_XMTR *xmtr, unsigned int dest, unsigned int service,
                              int attr, EP_TXH *handler, void *arg, EP_PAYLOAD *payload,
                              EP_NMD *nmd, int nFrags);
```

A.3.5 OPENIB VERBS

L'accès au réseau INFINIBAND par l'interface VERBS de OPENIB utilise un context (`ipv_context`) dans lequel on doit créer un *Protection Domain* (`ipb_pd`).

```
struct ibv_context *ibv_open_device(struct ibv_device *device);
struct ibv_pd *ibv_alloc_pd(struct ibv_context *context);
int ib_modify_qp(struct ib_qp *qp, struct ib_qp_attr *qp_attr, int qp_attr_mask);
```

Les requêtes d'émission ou de réception seront soumises dans un *Queue Pair* (`ibv_qp`) qui contient des *Completion Queues* (`ibv_cq`) où les notifications de terminaison seront placées. Ce *Queue Pair* doit être modifié pour être connecté à un autre afin de préciser le destinataire des messages.

```
struct ibv_cq *ibv_create_cq(struct ibv_context *context, int cqe, void *cq_context);
struct ibv_qp *ibv_create_qp(struct ibv_pd *pd, struct ibv_qp_init_attr *qp_init_attr);
```

Avant d'être utilisé dans les communications, les zones mémoire doivent être enregistrées.

```
struct ibv_mr *ibv_reg_mr(struct ibv_pd *pd, void *addr, size_t length,
                          enum ibv_access_flags access);
```

Les requêtes d'émission ou réception sont soumises dans le *Queue Pair*. Les communications s'effectuent alors avec le *Queue Pair* auquel on s'est connecté.

```
int ibv_post_send(struct ibv_qp *qp, struct ibv_send_wr *wr, struct ibv_send_wr **bad_wr);
int ibv_post_recv(struct ibv_qp *qp, struct ibv_recv_wr *wr, struct ibv_recv_wr **bad_wr)
```

La notification de terminaison des requêtes se fait en testant ou attendant des événements sur une *Completion Queue*.

```
int ibv_get_cq_event(struct ibv_context *context, int comp_num,
                             struct ibv_cq **cq, void **cq_context);
int ibv_poll_cq(struct ibv_cq *cq, int num_entries, struct ibv_wc *wc);
```

Dans le noyau, il est également possible d'enregistrer de la mémoire en précisant qu'il s'agit de zone en espace utilisateur ou bien d'adresses physiques.

```
struct ib_mr *ib_reg_user_mr(struct ib_pd *pd,
                             struct ib_umem *region,
                             int mr_access_flags,
                             const void __user *udata,
                             int udatalen);
struct ib_mr *ib_reg_phys_mr(struct ib_pd *pd,
                             struct ib_phys_buf *phys_buf_array,
                             int num_phys_buf,
                             int mr_access_flags,
                             u64 *iova_start);
```

Annexe B

Les accès aux fichiers dans LINUX

B.1 Structures de données

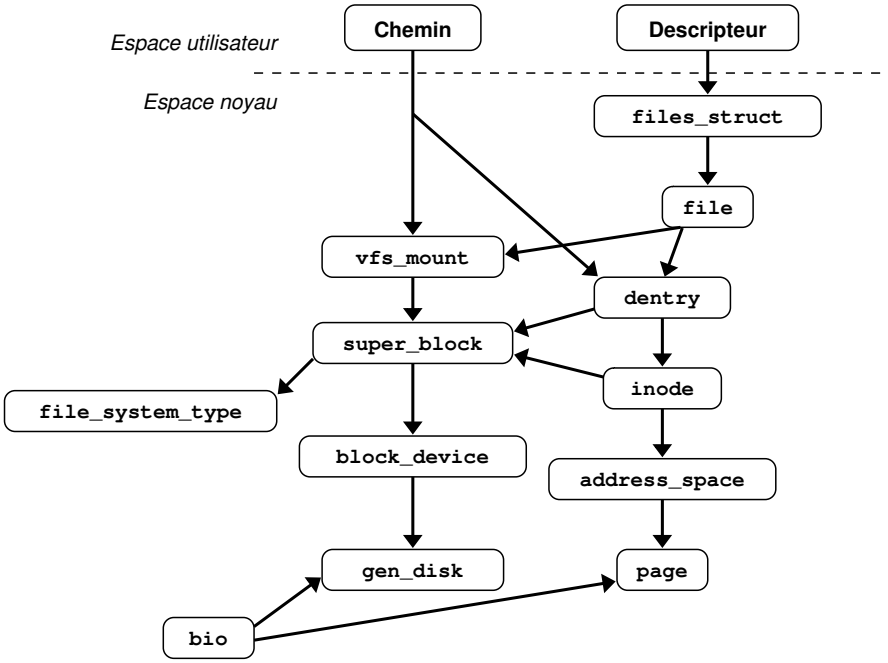


FIG. B.1: *Structures de données utilisées dans le* Virtual File System *de* LINUX *2.6.*

La figure B.1 représente les différentes structures de données manipulées lors de l'accès aux fichiers dans LINUX 2.6. On pourra se référer à [BC03, Lov04, LXR] pour une présentation plus détaillée.

L'application peut manipuler les fichiers par leur *chemin d'accès* ou par un *descripteur* (un entier). Le chemin d'accès est décomposé en une succession de *liens* (les `dentry`, *Directory Entry*) reliant la racine `/` du système de fichiers au fichier cible. Un fichier ouvert est décrit par une structure `file`. Au sein d'un même processus, la liste des fichiers ouverts est maintenue dans la structure `files_struct` qui contient un tableau

de pointeurs vers des `file`. Le descripteur manipulé par l'application est l'indice du fichier dans ce tableau. Lors d'un partage de fichier suite à un `fork`, le tableau est dupliqué mais le `file` est partagé.

La structure `file` qui décrit un fichier ouvert pointe vers le `dentry` qui a été ouvert. Ainsi, la manipulation des fichiers par descripteur ou par chemin d'accès peut être centralisée au niveau des `dentry`. Mais les `dentry` ne sont que des liens entre les vrais objets du système de fichiers, les *I-nœuds* (`inode`).

Le système de fichiers réels est décrit par un objet `super_block` qui correspond, pour les systèmes de fichiers locaux, au premier bloc physique sur la partition du disque. Le type de système de fichier manipulé (EXT3, VFAT, ...) est décrit par une structure `file_system_type`. Le disque physique stockant les données est décrit par une structure `block_device` et un objet de bas niveau `gen_disk`.

Les fonctionnalités modernes de montage, notamment le fait de pouvoir dupliquer un point de montage en deux endroits distincts, ont nécessité de préciser légèrement le modèle précédent. L'objet `dentry` ne permettant plus de préciser de manière sûre le chemin manipulé, il est en fait accompagné d'un `vfs_mount` qui repère l'instance de point de montage dans laquelle se trouve le chemin. Il peut donc y avoir plusieurs `vfs_mount` pointant vers le même `super_block`. L'ensemble de tous les `vfs_mount` forme un arbre qui, avec les `dentry`, constitue l'*espace de nommage* du système.

Les *I-nœuds* contenus dans le système de fichiers contiennent en fait des données. Ces données sont représentées par un espace d'adressage dédié au fichier (`address_space`). Cet espace peut être mappé (projeté) dans la mémoire du système lorsque certaines parties du fichier ont été chargées dans le cache du système. Dans ce cas, l'`address_space` pointe vers les pages physiques (`page`) contenant ces données.

Les accès aux données réelles sur un disque sont décrites par une structure `bio` (*Block Input/Output*) qui contient le descripteur du disque (`gen_disk`) et les `page` cibles (là où les données du disque doivent être lues ou écrites).

Dans le cas d'accès directs (ouverture du fichier avec le paramètre `O_DIRECT`), les structures `bio` sont également utilisées pour transférer les données entre le disque des `page` de l'espace utilisateur d'un processus.

B.2 Principe de fonctionnement

Le VFS de LINUX dispose d'un certain nombre de caches permettant une amélioration notable des performances en évitant les accès disque répétitifs. Les objets utilisés sont toujours conservés en mémoire. Ainsi, un fichier ouvert conservera son `dentry` et son `inode` dans le système. Par contre, les autres objets ne sont conservés que si le système n'a pas besoin de récupérer de la mémoire physique.

Le VFS dispose donc tout d'abord d'un cache de métadonnées afin d'accélérer le parcours de chemin d'accès aux fichiers. Le parcours d'un chemin consiste à le découper en `dentry` dont on cherche la présence dans le cache, puis sur le disque si nécessaire.

L'existence d'un `dentry` est en fait lié à la présence d'un lien du même nom dans le répertoire courant. Les `inode` pointés par les `dentry` sont cachés de façon similaire.

Les données contenues dans les fichiers sont également cachées dans le système d'exploitation. Cependant, ce cache est intégré au *Page-Cache* global du système car les données des fichiers peuvent être mappées dans la mémoire des processus. Ces données sont donc manipulées sous forme de pages (`page`) qui correspondent à un ensemble de blocs consécutifs sur le disque. Ce cache est beaucoup plus complexe que les caches de métadonnées car il doit proposer des fonctionnalitées avancées telles que le *Read-Ahead* (lecture à l'avance) ou le *Write-Back Caching* (écritures regroupées différées qui sont traitées conjointement au *Swap*). Le remplissage du cache peut être effectué page par page, ou bien plusieurs pages à la fois depuis les noyaux 2.6.

B.3 Mise en place d'un nouveau système de fichiers

La définition d'un nouveau type de système de fichiers dans le VFS de LINUX, par exemple pour notre client ORFS (voir en partie 5.2.1), nécessite de définir un ensemble de routines que le VFS va appeler pour effectuer les travaux spécifiques à ce système.

Les principaux objets (`file`, `dentry`, `inode`, `address_space`, ...) disposent donc d'un ensemble de routines spécifiques (`file_operations`, `dentry_operations`, ...). Si l'une d'entre elles n'est pas définie, le VFS se rabat sur les routines par défaut. Par exemple, pour préciser la façon de lire un `inode` sur le disque, on peut définir la routine `read_inode` parmi les `super_operations` du `super_block` du système de fichiers. Ainsi, toute l'organisation logique des données sur le disque est mise en œuvre dans ces routines spécifiques.

Il est également possible de modifier les paramètres des objets `gen_disk` pour définir des disques virtuels, notamment pour mettre en place un *Network Block Device*. Dans ce cas, on définit la méthode qui sera appelée pour traiter les `bio` (en la passant à `blk_init_queue`).

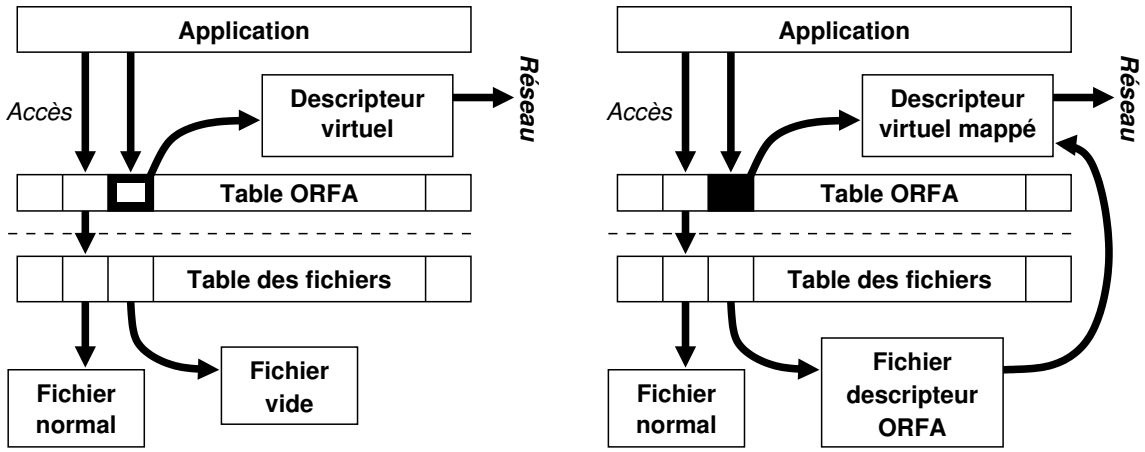
Annexe C

Accès transparent aux fichiers distants en espace utilisateur

Nous présentons ici les détails du support transparent des accès distants dans le client ORFA. L'architecture générale de ORFA a été présentée en partie 4.1.

C.1 Descripteurs virtuels de fichiers distants

Accéder aux fichiers distants la même façon qu'aux fichiers locaux impose de simuler dans le client ORFA le comportement des descripteurs de fichiers qui est traditionnellement mis en œuvre dans le système d'exploitation (le VFS de LINUX). La bibliothèque client ORFA met donc en place des descripteurs virtuels pour les fichiers distants, sous la forme de structure de données stockées en mémoire.



(a) Descripteur virtuel initialement stocké en mémoire.

(b) Descripteur virtuel mappé.

FIG. C.1: *Descripteurs virtuels dans le client* ORFA.

Ces descripteurs sont accédés par l'application via l'interface standard de programmation, c'est-à-dire par un identifiant entier représentant l'indice du fichier dans la table des fichiers ouverts du processus (voir en annexe B). Le client ORFA maintient donc en mémoire une table des fichiers ouverts indiquant si un fichier est local ou non. La cohérence entre la table des fichiers stockée dans le système et celle d'ORFA est assurée en ouvrant un faux fichier local pour chaque fichier distant ORFA. Ainsi, chaque descripteur virtuel a son identifiant entier réservé par ce faux fichier (voir la figure C.1(a)).

Simuler le comportement des descripteurs de fichiers impose de partager les fichiers après un `fork` (*File Sharing Semantics*) et de permettre la conservation de fichiers ouverts après `exec`. Le partage des descripteurs virtuels après un `fork` nécessite un mécanisme de mémoire partagée, ce qui peut être mis en place en mappant en mémoire un fichier. Le client ORFA utilise donc le faux fichier qui réservait l'identifiant pour y stocker la structure du descripteur puis le mapper en mémoire (voir la figure C.1(b)). Ainsi, lors d'un `fork`, le mapping est automatiquement partagé entre les deux processus engendrés.

Dans le cas d'`exec`, le mapping est perdu mais un fichier conservé ouvert (en l'absence de drapeau *Close-on-exec*) pourra être remappé aussitôt. Des variables d'environnement sont utilisées pour indiquer pendant `exec` quels fichiers correspondent à des descripteurs virtuels ORFA.

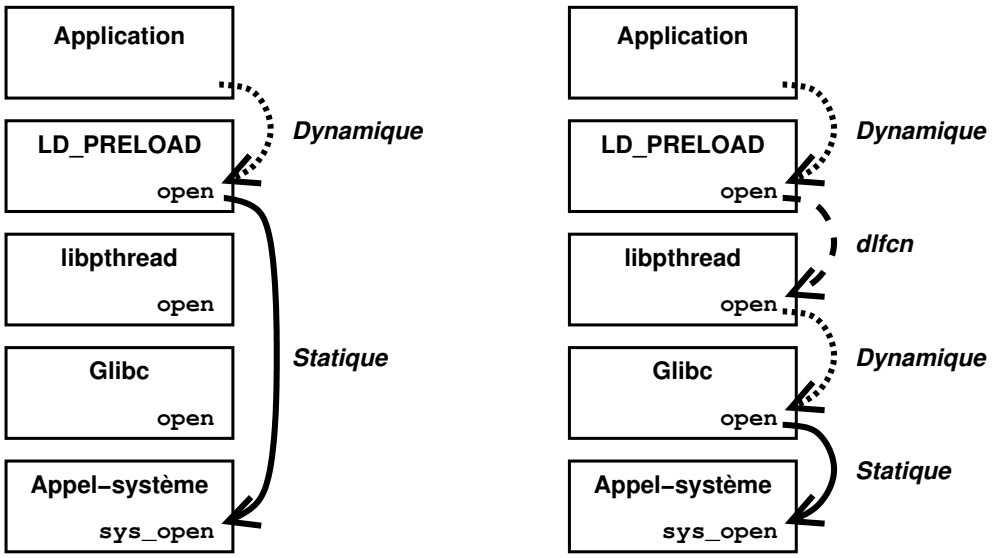
Nous avons cependant choisi de ne pas stocker les descripteurs dans ce fichier par défaut car le mapping mémoire impose un surcoût non-négligeable. Le descripteur est donc conservé en mémoire jusqu'à ce qu'un `fork` ou `exec` soit demandé par l'application.

C.2 Transparence vis-à-vis des autres bibliothèques

Le client ORFA intercepte les appels de l'application car il est chargé avant toutes les autres bibliothèques système (grâce à la variable d'environnement `LD_PRELOAD`). Il modifie donc le comportement de certains routines, mais doit utiliser les anciennes lorsqu'un fichier local est accédé.

Beaucoup de fonctions de la bibliothèque standard sont mises en place comme de simples *wrappers* effectuant un appel-système. Conserver le comportement standard pour les fichiers locaux consiste alors simplement à copier le code effectuant cet appel-système. Cependant, cette méthode a l'inconvénient de ne pas être compatible avec d'autres bibliothèques modifiant également le comportement des accès aux fichiers puisque leur code ne pourra pas être copié en cours d'exécution. C'est notamment le cas de la bibliothèque `libpthread`. Une telle méthode d'interception conduirait à ce que la bibliothèque ORFA oublie `libpthread` (voir la figure C.2(a)).

Une vraie compatibilité avec ces autres bibliothèques impose que le client ORFA n'utilise pas l'appel-système sous-jacent pour gérer les fichiers locaux, mais l'implantation sous-jacente, quelque soit la bibliothèque qui la met en œuvre. Pour ce faire, la bibliothèque `dlfcn` propose des routines cherchant l'occurence suivante d'un symbole



(a) Interception de `open` par redéfinition des appels-système.

(b) Interception de `open` par édition dynamique de liens avec `dlfcn`.

FIG. C.2: *Interception de l'appel* `open` *par une bibliothèque préchargée par la variable d'environnement* `LD_PRELOAD`.

dans un espace d'adressage (grâce à `dlsym` avec le paramètre `RTLD_NEXT`). Elle permet donc de retrouver la routine qui aurait été exécutée en l'absence de ORFA. Ainsi, la compatibilité est assurée puisque le traitement des fichiers locaux à travers ORFA sera exactement le même qu'en son absence (voir la figure C.2(b)).

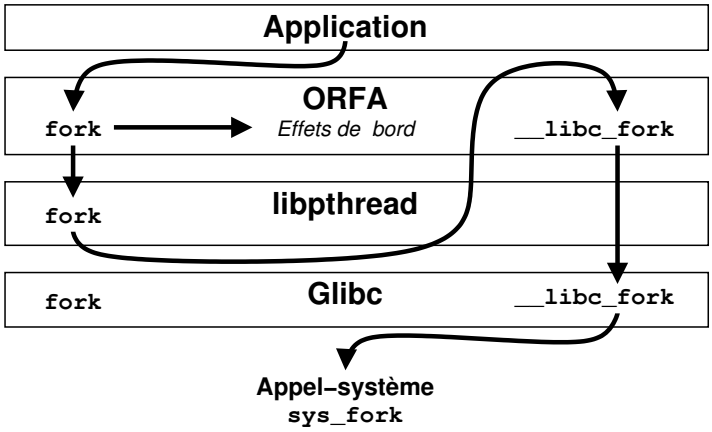


FIG. C.3: *Support des appels imbriqués dans le client* ORFA.

Dans certains cas, le client ORFA applique des effets de bord, c'est-à-dire qu'il conserve le comportement normal et y ajoute quelques traitements spécifiques. C'est notamment le cas des appels non-directement liés aux fichiers. Par exemple, `fork` impose de partager les descripteurs de fichiers virtuels et donc d'adapter ces descripteurs au partage. La présence d'une autre librairie modifiant `fork`, par exemple `libpthread`, pose une nouveau problème puisqu'elle le redéfinit en utilisant le *wrapper* d'appel-système `__libc_fork` de la bibliothèque standard. Comme le client ORFA intercepte à la fois `fork` et `__libc_fork`, il faut s'assurer que les effets de bord ne seront pas appliqués deux fois (voir la figure C.3).

Annexe D

L'adressage mémoire dans LINUX

D.1 Organisation de la mémoire

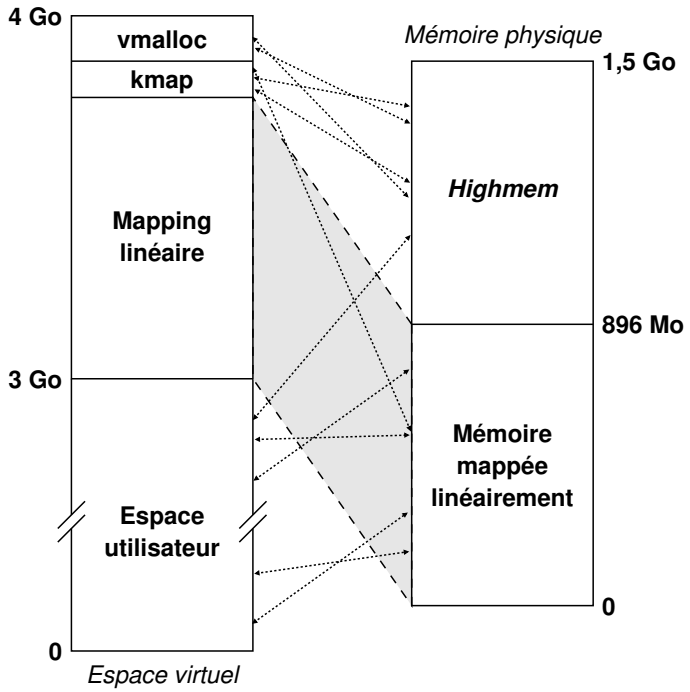


FIG. D.1: *Organisation de la mémoire virtuelle dans* LINUX *sur une architecture 32 bits disposant de 1,5 Go de mémoire physique.*

La figure D.1 présente l'organisation mémoire de LINUX. On pourra se référer à [BC03, LXR] pour une présentation plus détaillée.

L'espace d'adressage est divisé en deux parties principales, l'espace noyau, qui est partagé entre tous les processus mais n'est accessible que depuis un contexte noyau, et l'espace utilisateur qui est propre à chaque processus. L'espace utilisateur est situé entre 0 et la constante `PAGE_OFFSET`, l'espace noyau au dessus. Sur architecture 32 bits,

`PAGE_OFFSET` vaut 3 Go, ce qui laisse 1 Go pour l'espace noyau.

L'espace utilisateur des processus est décrit par la structure `mm_struct` et composé de pages quelconques de la mémoire physique. Il se divise en un ensemble de zones appelées VMA (*Virtual Memory Area*) qui regroupent les pages consécutives ayant les mêmes propriétés. Par exemple, le tas et la pile ont chacun une VMA dédiée, chaque mapping de fichier également. Chaque VMA est caractérisée par une protection (lecture, écriture, exécution, ...), le fait de pouvoir être partagée avec d'autres processus ou non, et des routines spécifiques déterminant comment traiter les pages (par exemple lors de leur initialisation).

L'espace noyau est subdivisé en :

Mapping linéaire de la mémoire physique : La partie basse de l'espace mémoire du noyau est constituée d'un mapping (projection) linéaire de la mémoire physique. Ce mapping peut ne contenir qu'une partie de la mémoire physique si celle-ci est supérieure à la taille de mémoire virtuelle allouée au mapping (896 Mo sur architecture 32 bits). C'est dans cet espace que sont stockées le code du noyau (mais pas les modules) et les principales structures de données (allouées avec `kmalloc`).

Espace de mapping temporaire : Lorsque le mapping linéaire est trop petit pour contenir toute la mémoire physique, il faut pouvoir mapper temporairement les autres pages. Ces autres pages physiques appartiennent à la zone *Highmem*. Un espace est réservé pour en mapper temporairement un nombre fixé (512 ou 1024 pages physiques en général).

Espace `vmalloc` : Un espace particulier est réservé à la fin de l'espace noyau pour des grandes allocations de mémoire virtuelle. Il est notamment utilisé pour stocker les modules insérés dans le noyau, mapper la mémoire des périphériques et faire de larges allocations de mémoire non-physiquement contiguë (avec `vmalloc`).

Même si certaines architectures l'autorisent, il n'est théoriquement pas possible d'accéder directement à l'espace utilisateur depuis l'espace noyau, c'est-à-dire en déréférençant des adresses (le cas contraire, l'accès au noyau depuis l'espace utilisateur, est interdit par manque de privilèges pour des raisons de sécurité).

La limite d'1 Go de l'espace noyau commençant à être contraignante, notamment lorsque la mémoire physique ne tient pas dans le mapping linéaire, des versions spéciales du noyau LINUX ont été développées pour proposer 4 Go d'espace d'adressage utilisateur et autant pour le noyau sur architecture 32 bits. Ces patchs *4G/4G* interdisent réellement l'accès direct à l'espace utilisateur depuis le noyau puisqu'il est impossible de distinguer les adresses des deux espaces (elles peuvent varier de 0 à 4 Go).

Beaucoup d'autres systèmes d'exploitation utilisent une organisation mémoire similaire à LINUX. Mais MACOS X et les WINDOWS récents ont la particularité d'intégrer une organisation *4G/4G* en standard.

Mais avec l'avènement des machines 64 bits, les tailles respectives des espaces utilisateur et noyau ont pu être considérablement accrues. Par exemple, sur architecture POWERPC 64, l'espace utilisateur mesure 12 millions de Go tandis que le mapping linéaire du noyau en mesure 4 millions.

D.2 Mapping

Le noyau a potentiellement accès à toute la mémoire du système. Cependant, cette mémoire peut appartenir à l'espace d'adressage d'un processus (et donc ne pas être déréférençable) ou ne pas être utilisée du tout (par exemple une page du cache de fichiers). Pour y accéder, il faut *mapper* les pages correspondantes en mémoire virtuelle, c'est-à-dire les faire correspondre à des adresses virtuelles.

Ainsi, lorsqu'une application veut écrire dans un fichier, on mappe la page du cache qui contient la zone destination du fichier, on écrit les données dedans puis on la démappe. Le VFS manipulant le cache de fichiers sous la forme de pages physiques (`page`, voir en B), elles peuvent être directement mappées.

L'accès à une page utilisateur depuis le noyau nécessite également de mapper les pages correspondantes dans l'espace noyau pendant la durée de l'accès. Mais il faudra tout d'abord connaître les pages physiques correspondantes et s'assurer qu'elles ne seront pas swappées entre temps, par exemple en utilisant `get_user_pages` qui va les verrouiller.

Pour faire ce mapping, le noyau LINUX dispose d'un ensemble de pages virtuelles inutilisées, l'espace de mapping temporaire `kmap`. Il est possible d'y mapper temporairement une page physique (avec `kmap`). Pour éviter qu'un travail important ne souffre d'une famine dans l'espace `kmap`, le noyau fournit également une interface de mapping qui ne peut pas échouer (`kmap_atomic`). Cette interface n'est utilisable que si les travaux à effectuer pendant la validité du mapping sont *atomiques*, c'est-à-dire très courts et ne pouvant pas être interrompus (par exemple en dormant pour attendre un événement).

D.3 Traduction d'adresse

Le processeur ne manipule que des adresses virtuelles. La traduction en adresse physique est effectuée de manière transparente dans le processeur par la MMU (*Memory Management Unit*) et son cache, le TLB (*Translation Lookaside Buffer*).

Il arrive également que le système d'exploitation ait besoin de traduire des adresses virtuelles. En effet, le système est responsable de la gestion des pages physiques, leur utilisation pour allouer la mémoire des processus, gérer le swap, ... La méthode traditionnelle pour traduire une adresse consiste à reproduire logiciellement la MMU.

La mémoire virtuelle est mise en place à travers une table de pages. Le noyau LINUX utilise depuis 2.6.11 quatre niveaux nommés PGD, PUD et PMD (*Page Global*, *Upper* et *Middle Directory*) et PTE (*Page Table Entry*). Cette table est stockée en mémoire et accédée par la MMU lorsque le processeur accède à la mémoire. Le système d'exploitation présente donc des routines logicielles pour parcourir cette table et récupérer l'adresse physique correspondant à une adresse virtuelle (`pgd_offset`, `pte_offset_map`, ...).

Cependant cette méthode est coûteuse et n'est pas toujours nécessaire. En effet, la traduction des adresses est facile lorsque l'adresse virtuelle se trouve dans le mapping li-

néaire du noyau. Dans ce cas, il suffit de lui retrancher `PAGE_OFFSET` (par exemple avec `virt_to_phys`). Toutes les structures habituelles du noyau (allouées par `kmalloc`) peuvent donc être traduites ainsi. Pour les autres adresses, en particulier la mémoire utilisateur et l'espace `vmalloc`, le parcours de la table de pages est indispensable. Et il ne faut pas oublier de s'assurer que la page ne pourra être swappée (ce qui invalide son adresse physique).

D.4 Adresses de bus

Les adresses virtuelles ne sont manipulées que sur le processeur central. Les périphériques accédant la mémoire centrale par DMA ne connaissent que des adresses physiques, ou plus précisément, des adresses de bus, c'est-à-dire une adresse que le bus d'entrées-sorties (par exemple PCI) est capable de traiter.

Le système d'exploitation doit donc être capable de traduire une adresse virtuelle (ou physique) en adresse de bus. En fait, sur les architectures simples, en particulier IA-32, les adresses physiques sont identiques aux adresses de bus. Sur d'autres, par exemple ALPHA, il suffit d'ajouter une constante.

Par contre, sur les architectures avancées telles que IA-64 ou POWERPC 64, une IOMMU (MMU pour les entrées-sorties) est placée sur le contrôleur de bus d'entrées-sorties pour traiter les adresses de bus et donc permettre aux périphériques d'accéder à la mémoire centrale. Sur ces architectures, l'obtention d'une adresse de bus impose de mapper l'adresse physique correspondante dans l'IOMMU (par exemple avec `pci_map_single`).

Annexe E

Notification des modifications d'espace d'adressage dans le noyau LINUX avec VMA SPY

E.1 Architecture VMA SPY

Nous avons conçu l'infrastructure VMA SPY pour notifier les changements d'espace d'adressage à un système externe. Les événements qui nous intéressent tout particulièrement ici sont la suppression d'une partie de l'espace d'adressage.

Un changement de mapping peut également se produire après un `fork`. En effet, la technique de *Copy-on-Write* est utilisée pour éviter des duplications inutiles de pages. La duplication réelle d'une page est effectuée lorsque le père ou le fils du `fork` la modifie. Celui qui la modifie obtient une copie de la page initiale et laisse l'original à l'autre. Cette modification d'adressage n'étant pas prévisible, il vaut mieux l'anticiper.

VMA SPY permet donc de définir des espions notifiant des suppression de mapping et de la duplication de zones mémoire au moment d'un `fork`. Ces mécanismes s'articulent autour des VMA (*Virtual Memory Area*, voir en annexe D). c'est-à-dire les zones mémoire qui subdivisent l'espace d'adressage d'un processus (voir [BC03, LXR] pour plus de détails). Le patch VMA SPY ajoute des appels à des routines notificatrices dans les parties du code qui modifient l'adressage.

```
/* called when area is partially or totally unmapped */
void notify_vma_spy_unmap(struct vm_area_struct * area, unsigned long start, unsigned long end);

/* called when area is duplicated */
void notify_vma_spy_fork(struct vm_area_struct * area);
```

La définition des espions utilise un type `vma_spy_type` où les fonctions de traitement de `fork` et de la suppression de mapping sont définies. Ce type maintient par ailleurs une liste de zones espionnées.

```
struct vma_spy_type {
  void (*unmap) (struct vm_area_struct * area, unsigned long start, unsigned long end, void * data);
  void (*fork) (struct vm_area_struct * area, void * data);
  void * data;
```

```
  struct list_head vma_list;
  spinlock_t vma_lock;
};
```

Une fois l'espion `vma_spy_type` défini, on peut l'enregistrer auprès d'une zone mémoire VMA.

```
/* add a vma_spy to the list in a vma and to the list in a vma_spy_type */
struct vma_spy * register_vma_spy(struct vm_area_struct * area, struct vma_spy_type * type);

/* return a vma_spy if a type is already spying a vma */
struct vma_spy * is_vma_spy_registered(struct vm_area_struct * area, struct vma_spy_type * type);
```

Chaque couple (zone, espion) est représenté par une structure `vma_spy`.

```
struct vma_spy {
  struct list_head spy_list;
  struct list_head vma_list;
  struct vm_area_struct * vma;
  struct vma_spy_type * type;
};
```

La structure `vm_area_struct` décrivant les VMA a dû être modifiée pour maintenir la liste des espions qui la surveillent.

```
struct vm_area_struct {
  ...
  /* Driver that watch this vma may register their callbacks here */
  struct list_head vm_spy_list;
  spinlock_t vm_spy_lock;
  ...
};
```

Par ailleurs, une VMA peut être coupée en deux lorsqu'une de ses parties est modifiée, ou fusionnée avec une de ses voisines si leurs caractéristiques le permettent. Dans ce cas, il faut dupliquer ou fusionner la liste d'espions pour que les zones cibles continuent à être espionnées convenablement. Les parties du code du noyau qui découpent ou fusionnent des VMA ont donc également été modifiées pour notifier l'infrastructure VMA SPY.

```
/* called when old is splitted into old and new */
void notify_vma_spy_split(struct vm_area_struct * old, struct vm_area_struct * new);

/* called when old is merged into new */
void notify_vma_spy_merge(struct vm_area_struct * old, struct vm_area_struct * new);
```

E.2 Utilisation dans GMKRC

L'infrastructure VMA SPY a été conçue pour maintenir le cache d'enregistrement mémoire GMKRC à jour.

Un espion `vma_spy_type` est donc défini pour chaque cache d'enregistrement. À chaque enregistrement mémoire, on regardera si la VMA correspondante est déjà espionnée. Si ce n'est pas le cas, on ajoute l'espion sur cette zone.

Lorsque la zone est modifiée ou dupliquée, GMKRC en est notifié et peut donc réagir en fonction, c'est-à-dire désenregistrer les zones correspondantes pour assurer la validité future du cache.

E.3 Autres propositions et support noyau

Le noyau LINUX ne propose aucun support pour les communications zéro-copie sur les réseaux haute performance. Le patch VMA SPY a été conçu pour y remédier. Nous avons vu en 2.2.4.c que QUADRICS propose également une modification du système d'exploitation LINUX pour mettre à jour la MMU embarquée sur les cartes d'interface réseau ELAN. Ce patch définit une infrastructure basée sur une structure IOPROC comparable à nos espions VMA SPY mais sa mise en œuvre est différente.

Les développeurs d'OPENIB ne proposent pas de modifier le noyau mais préfèrent limiter les possibilités de l'application pour éviter les problèmes. Ils ont cependant proposé d'ajouter des états de VMA, par exemple pour faciliter le support de `fork` en empêchant le *Copy-on-Write* (les pages seraient alors immédiatement dupliquées au moment du `fork` et pas lors de leur première modification) mais le coût pourrait alors devenir prohibitif.

Bibliographie

[ADI] The implementation of the second generation MPICH ADI. `http://www-unix.mcs.anl.gov/mpi/mpich/workingnote/adi2impl/note.html`.

[ADN+96] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. *ACM Trans. Comput. Syst.*, 14(1) :41–79, 1996.

[Amd67] G. Amdahl. The Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the AFIPS Conference*, volume 30, pages 483–485, 1967.

[ANL+04] Rafael B. Avila, Philippe O. A. Navaux, Pierre Lombard, Adrien Lebre, and Yves Denneulin. Performance Evaluation of a Prototype Distributed NFS Server. In *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'04)*, pages 100–105, Foz do Iguaçu, PR - Brazil, October 2004.

[Bac86] M. J. Bach. *The Design of the UNIX Operating System*. Prentice Hall, 1986.

[BAP00] Mohammad Banikazemi, Bulent Abali, and Dhabaleswar K. Panda. Comparison and Evaluation of Design Choices for Implementing the Virtual Interface Architecture (VIA). In *Communication, Architecture, and Applications for Network-Based Parallel Computing*, pages 145–161, 2000.

[BBS02] Peter J. Braam, Ron Brightwell, and Phil Schwan. Portals and Networking for the Lustre File System. In *Proceedings of the 2002 IEEE International Conference on Cluster Computing*, Chicago, September 2002.

[BC03] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel, Second Edition*. O'Reilly, 2003.

[BCF+95] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet : A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1) :29–36, 1995.

[BdRC93] R. Bordawekar, J. M. del Rosario, and A. Choudhary. Design and Evaluation of primitives for Parallel I/O. In *Supercomputing '93 : Proceedings of*

the 1993 ACM/IEEE conference on Supercomputing, pages 452–461, New York, NY, USA, 1993. ACM Press.

[BGM99] Amnon Barak, Ilia Gilderman, and Igor Metrik. Performance of the Communication Layers of TCP/IP with the Myrinet Gigabit LAN, 1999.

[BM98] Gaurav Banga and Jeffrey C. Mogul. Scalable kernel performance for Internet servers under realistic loads. In *Proceedings of the 1998 USENIX Annual Technical Conference*, New Orleans, LA, June 1998.

[BM00] R. Brightwell and A. Maccabe. Scalability limitations of VIA-based technologies in supporting MPI. In *Proceedings of the Fourth MPI Developer's and User's Conference*, March 2000.

[BMD99] Gaurav Banga, Jeffrey C. Mogul, and Peter Druschel. A Scalable and Explicit Event Delivery Mechanism for UNIX. In *USENIX Annual Technical Conference*, pages 253–265, June 1999.

[BN99] Peter J. Braam and Philip A. Nelson. Removing Bottlenecks in Distributed Filesystems : Coda & InterMezzo as examples. In *Proceedings of the Linux Expo*, May 1999.

[BPPM03] Suparna Bhattacharya, Steven Pratt, Badari Pulavarty, and Janet Morgan. Asynchronous I/O Support in Linux 2.5. In *Proceedings of the Linux Symposium*, pages 371–386, Ottawa, Canada, July 2003.

[BPS01] D. Buntinas, D. K. Panda, and P. Sadayappan. Fast NIC-level Barrier over Myrinet/GM. In *Proceedings of Int'l Parallel and Distributed Processing Symposium (IPDPS)*, 2001.

[Bra99] Peter Braam Braam. The InterMezzo File System, 1999.

[BRLM02] Ron Brightwell, Rolf Riesen, Bill Lawry, and A. B. Maccabe. Portals 3.0 : Protocol Building Blocks for Low Overhead Communication. In *Workshop on Communication Architecture for Clusters (in conjunction with IPDPS)*, April 2002.

[Bru99] Jose Carlos Brustoloni. Interoperation of Copy Avoidance in Network and File I/O. In *INFOCOM (2)*, pages 534–542, 1999.

[BTSM04] Suparna Bhattacharya, John Tran, Mike Sullivan, and Chris Mason. Linux AIO Performance and Robustness for Entreprise Workloads. In *Proceedings of the Linux Symposium*, pages 63–77, Ottawa, Canada, July 2004.

[BU04] Ron Brightwell and Keith D. Underwood. An Analysis of NIC Resource Usage for Offloading MPI. In *Proceedings of the 2004 Workshop on Communication Architecture for Clusters*, Santa Fe, New Mexico, April 2004.

[CACR95] Phyllis E. Crandall, Ruth A. Aydt, Andrew A. Chien, and Daniel A. Reed. Input/Output Characteristics of Scalable Parallel Applications. In *Proceedings of Supercomputing '95*, San Diego, CA, 1995. IEEE Computer Society Press.

[CCL+03] A. Ching, A. Choudhary, W. Keng Liao, R. Ross, and W. Gropp. Efficient structured data access in parallel file systems. In *IEEE International Conference on Cluster Computing*, 2003.

[CKP+93] David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus E. Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP : Towards a Realistic Model of Parallel Computation. In *Principles Practice of Parallel Programming*, pages 1–12, 1993.

[CLRT00] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS : A Parallel File System for Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, 2000. USENIX Association.

[Clu02] Cluster File Systems, Inc. Lustre : A Scalable, High Performance File System, November 2002. `http://www.lustre.org`.

[CRU03] Olivier Cozette, Cyril Randriamaro, and Gil Utard. READ^2 : Put disks at network level. In *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, Tokyo, Japan, May 2003. IEEE Computer Society.

[DBLP97] Cezary Dubnicki, Angelos Bilas, Kai Li, and James Philbin. Design and Implementation of Virtual Memory-Mapped Communication on Myrinet. Technical Report TR-570-97, 1997.

[DCK+03] M. DeBergalis, P. Corbett, S. Kleiman, A. Lent, D. Noveck, T. Talpey, and M. Wittle. The Direct Access File System. In *Proceedings of USENIX Conference on File and Storage Technologies 2003*, San Francisco, CA, March 2003.

[DM03] Ulrich Drepper and Ingo Molnar. The Native POSIX Thread Library for Linux, 2003. `http://people.redhat.com/drepper/nptl-design.pdf`.

[DN03] Vincent Danjean and Raymond Namyst. Controling Kernel Scheduling from User Space : an Approach to Enhancing Applications' Reactivity to I/O Events. In *Proceedings of the 2003 International Conference on High Performance Computing (HiPC '03)*, volume 2913 of *Lect. Notes in Comp. Science*, pages 490–499, Hyderabad, India, December 2003. Springer-Verlag.

[Dol] Dolphin Interconnect Solutions Inc. `http://www.dolphinics.com`.

[Fab98] Theodore Faber. Optimizing Throughput in a Workstation-based Network File System over a High Bandwidth Local Area Network. In *ACM SIGOPS Operating System Review*, volume 32(1), pages 29–40, January 1998.

[For94] Message Passing Interface Forum. MPI : A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

[FTP85] RFC 959 : File Tranfer Protocol (FTP), October 1985. `http://www.ietf.org/rfc/rfc959.txt`.

[FUS] Filesystem in Userspace. `http://fuse.sourceforge.net/`.

[Geo01] P. Geoffray. OPIOM : Off-Processor I/O with Myrinet. *PPL - Parallel Processing Letters*, 11(2-3) :237–250, 2001.

[GGKK94] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing*. Benjamin Cummings Addison-Wesley Publishing Company, 1994.

[GM03] Myricom, Inc. *GM : A message-passing system for Myrinet networks*, 2003. `http://www.myri.com/scs/GM-2/doc/html/`.

[Gno] GnomeVFS - Filesystem Abstraction library. `http://developer.gnome.org/doc/API/gnome-vfs/`.

[Goo] Richard Gooch. I/O Event Handling Under Linux. `http://www.atnf.csiro.au/people/rgooch/linux/docs/io-events.html`.

[Gus92] D. B. Gustavson. The Scalable Coherent Interface and related standard projects. *IEEE Micro*, 12(1) :10–22, February 1992.

[HEH98] Bjarne Geir Herland, Michael Eberl, and Hermann Hellwagner. A common messaging layer for MPI and PVM over SCI. In *HPCN Europe*, pages 576–587, 1998.

[HKM+88] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michel J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1) :51–81, 1988.

[HSCL97] J. Hall, R. Sabatino, S. Crosby, and I. Leslie. Counting the Cycles : A Comparative Study of NFS Performance over High Speed Networks. In *Proceedings of the 1997 IEEE Conference on Local Computer Network (LCN 1997)*, volume 22, pages 8–19, Minneapolis, MN, November 1997. IEEE Computer Society Press.

[Inf01] Infiniband architecture specifications. InfiniBand Trade Association, 2001. `http://www.infinibandta/org`.

[IP81] RFC 791 : Internet Protocol, September 1981. `http://www.ietf.org/rfc/rfc791.txt`.

[ISSW97] M. Ibel, K. Schauser, C. Scheiman, and M. Weis. High-Performance Cluster Computing Using SCI. In *Hot Interconnects*, July 1997.

[JKN+01] Philippe Joubert, Robert B. King, Rich Neves, Mark Russinovich, and John M. Tracey. High-Performance Memory-Based Web Servers : Kernel and User-Space Performance. In *Proceedings of the USENIX Annual Technical Conference, Boston, MA*, pages 175–188, 2001.

[Jur95] C. Jurgens. Fibre Channel : A Connection to the Future. *IEEE Computer*, 28(8) :88–90, August 1995.

[Keg] D. Kegel. The C10K Problem. `http://www.kegel.com/c10k.html`.

[Kle86] S. R. Kleiman. Vnodes : An Architecture for Multiple File System Types in Sun UNIX. In *Proceedings of the Summer 1986 USENIX Technical Conference*, pages 238–247, June 1986.

[Lah02] Benjamin Lahaise. Design Notes on Asynchronous I/O (AIO) for Linux, 2002. `http://lse.sourceforge.net/io/aionotes.txt`.

[LCY+03] Jiuxing Liu, Balasubramanian Chandrasekaran, Weikuan Yu, Jiesheng Wu, Darius Butinas, Sushmitha Kini, and Dhabaleswar K. Panda. Microbenchmark Performance Comparison of High-Speed Cluster Interconnects. In *Hot Interconnects 11*, August 2003.

[LD02] Pierre Lombard and Yves Denneulin. nfsp : A Distributed NFS Server for Clusters of Workstations. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS'02)*, Marriott Marina, Fort Lauderdale, FL, April 2002.

[LDVL03] Pierre Lombard, Yves Denneulin, Olivier Valentin, and Adrien Lebre. Improving the Performances of a Distributed NFS Implementation. In *Proceedings of the 5th International Conference on Parallel Processing and Applied Mathematics (PPAM03)*, pages 405–412, Czestochowa, Poland, September 2003.

[Lem00] J. Lemon. Kqueue : A Generic and Scalable Event Notification Facility for FreeBSD. BSDCon 2000 Conference, 2000.

[Lev86] P. H. Levine. The Apollo DOMAIN Distributed File System. In Y. Parker et al., editor, *NATO ASI series*, volume F28, pages 241–260. Springer Verlag, Distributed Operating Systems : Theory and Practice edition, August 1986.

[Lig01] Walt Ligon. Next Generaton Parallel Virtual File System. In *Proceedings of the 2001 IEEE International Conference on Cluster Computing*, Newport Beach, CA, October 2001.

[Lov04] Robert Love. *Linux Kernel Development*. Developer's Library, Sams Publishing, 2004.

[LPC98] Mario Lauria, Scott Pakin, and Andrew A. Chien. Efficient layering for high speed communication : Fast messages 2.x. In *HPDC*, pages 10–20, 1998.

[LR99] W. Ligon and R. Ross. An Overview of the Parallel Virtual File System. In *Proceedings of the 1999 Extreme Linux Workshop*, June 1999.

[LXR] Linux cross-reference. `http://lxr.linux.no/`.

[MAF+02] K. Magoutis, S. Addetia, A. Fedorova, M. Seltzer, J. Chase, A. Gallatin, R. Kisley, R. Wickremesinghe, and E. Gabber. Structure and Performance of the Direct Access File System. In *Proceedings of USENIX 2002 Annual Technical Conference*, pages 1–14, Monterey, CA, June 2002.

[MAFS03] K. Magoutis, S. Addetia, A. Fedorova, and M. I. Seltzer. Making the Most out of Direct-Access Network Attached Storage. In *Proceedings of USENIX Conference on File and Storage Technologies 2003*, San Francisco, CA, March 2003.

[Mag02a] K. Magoutis. Design and Implementation of a Direct Access File system (DAFS) Kernel Server for FreeBSD. In *Proceedings of USENIX BSDCon 2002 Conference*, San Francisco, CA, February 2002.

[Mag02b] K. Magoutis. The Optimistic Direct Access File System : Design and Network Interface Support. In *Proceedings of Workshop Novel Uses of System Area Network 2002*, Cambridge, MA, February 2002.

[MC99] Richard P. Martin and David E. Culler. NFS Sensitivity to High Performance Networks. In *Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, Atlanta, GE, 1999.

[MGH+96] Olivier Maquelin, Guang R. Gao, Herbert H. J. Hum, Kevin B. Theobald, and Xin-Min Tian. Polling Watchdog : Combining Polling and Interrupts for Efficient Message Handling. In *ISCA*, pages 179–188, 1996.

[MPIa] Message Passing Interface. `http://www-unix.mcs.anl.gov/mpi/`.

[MPIb] MPI-IO : I/O Extensions to the Message-Passing Interface. `http://www.mpi-forum.org/docs/mpi-20-html/node172.htm`.

[MX05] Myricom, Inc. *Myrinet Express (MX) : A High Performance, Low-Level, Message-Passing Interface for Myrinet*, 2005. `http://www.myri.com/scs/MX/doc/mx.pdf`.

[Myr] Myricom Inc. `http://www.myri.com`.

[NBK99] Erich M. Nahum, Tsipora P. Barzilai, and Dilip D. Kandlur. Performance Issues in WWW Servers. In *Measurement and Modeling of Computer Systems*, pages 216–217, 1999.

[Net] NetPIPE : A Network Protocol Independent Performance Evaluator. `http://www.scl.ameslab.gov/netpipe/`.

[NFS95] RFC 1813 : NFS Version 3 Protocol Specification, June 1995. `http://www.ietf.org/rfc/rfc1813.txt`.

[NFS00] RFC 3010 : NFS Version 4 Protocol, December 2000. `http://www.ietf.org/rfc/rfc3010.txt`.

[NKP+96] Nils Nieuwejaar, David Kotz, Apratim Purakayastha, Carla Schlatter Ellis, and Michael Best. File-Access Characteristics of Parallel Scientific Workloads. *IEEE Transactions on Parallel and Distributed Systems*, 7(10) :1075–1089, 1996.

[NWO88] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, 6(1), 1988.

[Ope] OpenIB Alliance. `http://www.openib.org`.

[Ous96] J. K. Ousterhout. Why Threads Are a Bad Idea (for most purposes). In *USENIX Technical Conference*, January 1996. `http://www.softpanorama.org/People/Ousterhout/Threads/`.

[PcFH+02] Fabrizio Petrini, Wu chun Feng, Adolfy Hoisie, Salvador Coll, and Eitan Frachtenberg. The Quadrics Network : High-Performance Clustering Technology. *IEEE Micro*, 22(1) :46–57, 2002.

[PDZ00] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite : a unified I/O buffering and caching system. *ACM Transactions on Computer Systems*, 18(1) :37–66, 2000.

[PFHC03] Fabrizio Petrini, Eitan Frachtenberg, Adolfy Hoisie, and Salvador Coll. Performance Evaluation of the Quadrics Interconnection Network. *Journal of Cluster Computing*, 6(2) :125–142, April 2003.

[Pfi01] Gregory F. Pfister. Aspects of the InfiniBand Architecture. In *Proceedings of the 2001 IEEE International Conference on Cluster Computing*, pages 369–371, Newport Beach, CA, October 2001.

[PT97] L. Prylli and B. Tourancheau. Protocol Design for High Performance Networking : a Myrinet Experience. Technical Report 97-22, LIP-ENS Lyon, 69364 Lyon, France, 1997.

[PVF03] PVFS2 Development Team. *Parallel Virtual File System, Version 2*, September 2003. `http://www.pvfs.org/pvfs2/pvfs2-guide.html`.

[Qua] Quadrics. `http://www.quadrics.com`.

[RI04] Murali Rangarajan and Liviu Iftode. Building a User-level Direct Access File System over Infiniband. In *3rd Workshop on Novel Uses of System Area Networks (SAN-3)*, February 2004.

[RPC88] RFC 1057 : RPC : Remote Procedure Call Protocol Specification, June 1988. `http://www.ietf.org/rfc/rfc1057.txt`.

[Sam] Samba - Opening Windows to a Wider World. `http://www.samba.org/`.

[SASB99] Evan Speight, Hazim Abdel-Shafi, and John K. Bennett. Realizing the Performance Potential of the Virtual Interface Architecture. In *International Conference on Supercomputing*, pages 184–192, 1999.

[Sch99] Peter Braam School. File Systems for Clusters from a Protocol Perspective. In *Second Extreme Linux Topics Workshop*, June 1999.

[Sch03] Philip Schwan. Lustre : Building a File System for 1,000 Node Clusters. In *Proceedings of 2003 Linux Symposium*, pages 401–408, Ottawa, Canada, July 2003.

[SGK+85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the Summer 1985 USENIX Conference*, Berkeley, CA, 1985.

[SH02] Frank Schmuck and Roger Haskin. GPFS : A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the Conference on File and Storage Technologies (FAST'02)*, pages 231–244, Monterey, CA, January 2002. USENIX, Berkeley, CA.

[SR97] E. Smirni and D. A. Reed. Workload Characterization of Input/Output Intensive Parallel Applications. In *Proceedings of the Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, volume 1245, pages 169–180. Springer-Verlag, 1997.

[SRO96] Steven R. Soltis, Thomas M. Ruwart, and Matthew T. O'Keefe. The Global File System. In *Proceedings of the Fifth NASA Goddard Conference on Mass Storage Systems*, pages 319–342, College Park, MD, 1996. IEEE Computer Society Press.

[SSB+95] 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.

[Sum00] Shinji Sumimoto. *A Study of High Performance Communication for Parallel Computers Using a Commodity Network*. PhD thesis, Keio University, 2000.

[Sys01] Nishan Systems. iSCSI Technical White Paper, 2001.

[TCP81] RFC 793 : Transmission Control Protocol, September 1981. `http://www.ietf.org/rfc/rfc793.txt`.

[TGL99] Rajeev Thakur, William Gropp, and Ewing Lusk. On Implementing MPI-IO Portably and with High Performance. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 23–32, 1999.

[TOHI98] H. Tezuka, F. O'Carroll, A. Hori, and Y. Ishikawa. Pin-down cache : A Virtual Memory Management Technique for Zero-copy Communication. In *12th International Parallel Processing Symposium*, pages 308–315, April 1998.

[UDP80] RFC 768 : User Datagram Protocol, August 1980. `http://www.ietf.org/rfc/rfc768.txt`.

[vEBBV95] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net : A User-Level Network Interface for Parallel and Distributed Computing. In *15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 40–53, December 1995.

[vECGS92] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages : a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Int'l Symp. on Computer Architecture*, Gold Coast, Australia, May 1992.

[VRC+03] V. Velusamy, C. Rao, S. Chakravarthi, J. Neelamegam, W. Chen, S. Verma, and A. Skjellum. Programming The Infiniband Network Architecture For High Performance Message Passing. In *Proceedings of the ISCA 16th International Conference on Parallel and Distributed Computing Systems*, Reno, Nevada, August 2003.

[WBvE97] Matt Welsh, Anindya Basu, and Thorsten von Eicken. Incorporating Memory Management into User-Level Network Interfaces. In *Proceedings of Hot Interconnects V*, Stanford, August 1997.

[WLS+85] D. Walsh, B. Lyon, G. Sager, J. M. Chand, D. Goldberg, S. Kleiman, T. Lyon, R. Sandberg, and P. Weiss. Overview of the SUN Network File System. In *Proceedings of the Winter Usenix Conference*, Dallas, TX, 1985.

[WP02] Jiesheng Wu and Dhabaleswar K. Panda. MPI-IO over DAFS over VIA : Implementation and Performance Evaluation. In *Workshop on Communication Architecture for Clusters (in conjunction with IPDPS)*, April 2002.

[WWP03a] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar Panda. PVFS over InfiniBand : Design and Performance Evaluation. In *International Conference on Parallel Processing (ICPP 03)*, October 2003.

[WWP03b] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar Panda. Supporting Efficient Noncontiguous Access in PVFS over InfiniBand. Technical Report OSU-CISRC-05/03-TR, May 2003.

[WWPR04] Jiesheng Wu, Pete Wyckoff, Dhabaleswar Panda, and Rob Ross. Unifier : Unifying Cache Management and Communication Buffer Management for PVFS over InfiniBand. In *IEEE International Symposium on Cluster Computing and the Grid (CCGrid2004)*, April 2004.

[XDR87] RFC 1014 : XDR : eXternal Data Representation Standard, June 1987. `http://www.ietf.org/rfc/rfc1014.txt`.

[ZBJ+02] Yuanyuan Zhou, Angelos Bilas, Suresh Jagannathan, Cezary Dubnicki, James F. Philbin, and Kai Li. Experiences with VI communication for database storage. In *29th annual International Symposium on Computer Architecture*, pages 257–268. IEEE Computer Society, 2002.

Liste des publications

Conférences internationales avec publication des actes et comité de lecture

[A] Brice Goglin, Loïc Prylli, and Olivier Glück. Optimizations of Client's side communications in a Distributed File System within a Myrinet Cluster. In *Proceedings of the IEEE Workshop on High-Speed Local Networks (HSLN), held in conjunction with the 29th IEEE LCN Conference*, pages 726–733, Tampa, Florida, November 2004. IEEE Computer Society Press.

[B] Brice Goglin and Loïc Prylli. Transparent Remote File Access through a Shared Library Client. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'04)*, volume 3, pages 1131–1137, Las Vegas, Nevada, June 2004. CSREA Press.

[C] Brice Goglin and Loïc Prylli. Performance Analysis of Remote File System Access over a High-Speed Local Network. In *Proceedings of the Workshop on Communication Architecture for Clusters (CAC'04), held in conjunction with the 18th IEEE IPDPS Conference*, Santa Fe, New Mexico, April 2004. IEEE Computer Society Press.

[D] Brice Goglin, Olivier Glück, and Pascale Vicat-Blanc Primet. An Efficient Network API for in-Kernel Applications in Clusters. In *Proceedings of the IEEE International Conference on Cluster Computing*, Boston, Massachussets, September 2005. IEEE Computer Society Press.

Conférences nationales avec publication des actes et comité de lecture

[E] Brice Goglin, Olivier Glück, Pascale Vicat-Blanc Primet, and Jean-Christophe Mignot. Accès optimisés aux fichiers distants dans les grappes disposant d'un réseau rapide. In *Actes de RenPar'16, CFSE'4, SympAAA'2005*, Le Croisic, Presqu'île de Guérande, France, April 2005.

Rapports de recherche

[F] Brice Goglin, Olivier Glück, and Pascale Vicat-Blanc Primet. An Efficient Network API for in-Kernel Applications in Clusters. Research Report RR2005-18, LIP, ENS Lyon, Lyon, France, April 2005. Also available as Research Report RR-5561, INRIA Rhône-Alpes.

[G] Brice Goglin, Olivier Glück, and Pascale Vicat-Blanc Primet. Accès optimisés aux fichiers distants dans les grappes disposant d'un réseau rapide. Research Report RR2004-56, LIP, ENS Lyon, Lyon, France, December 2004. Also available as Research Report RR-5458, INRIA Rhône-Alpes.

[H] Brice Goglin, Loïc Prylli, and Olivier Glück. Optimizations of Client's side communications in a Distributed File System within a Myrinet Cluster. Research Report RR2004-24, LIP, ENS Lyon, Lyon, France, April 2004. Also available as Research Report RR-5174, INRIA Rhône-Alpes.

[I] Brice Goglin and Loïc Prylli. Transparent Remote File Access through a Shared Library Client. Research Report RR2003-56, LIP, ENS Lyon, Lyon, France, December 2003. Also available as Research Report RR-5056, INRIA Rhône-Alpes.

[J] Brice Goglin and Loïc Prylli. Performance Analysis of Remote File System Access over High Bandwidth Local Network. Research Report RR2003-22, LIP, ENS Lyon, Lyon, France, April 2003. Also available as Research Report RR-4795, INRIA Rhône-Alpes.

[K] Brice Goglin. Un protocole pour l'accès aux systèmes de fichiers distants dans les grappes. Mémoire de DEA, ENS Lyon, Lyon, France, July 2002.

Rapports techniques

[L] Brice Goglin and Loïc Prylli. Design and Implementation of ORFA. Technical Report TR2003-01, LIP, ENS Lyon, Lyon, France, September 2003.

Index

Abstract :

This work aims at studying the exploitation of high-speed networks of clusters for distributed storage.

Parallel applications running on clusters require both high-performance communications between nodes and efficient access to the storage system. Many studies on network technologies led to the design of dedicated architectures for clusters with very fast communications between computing nodes. Efficient distributed storage in clusters have been essentially developed by adding parallelization mechanisms so that the server(s) may sustain an increased workload.

In this work, we propose to improve the performance of distributed storage systems in clusters by efficiently using the underlying high-performance network to access distant storage systems. The main question we are addressing is : do high-speed networks of clusters fit the requirements of a transparent, efficient and high-performance access to remote storage ?

We show that storage requirements are very different from those of parallel computation. High-speed networks of clusters were designed to optimize communications between different nodes of a parallel application. We study their utilization in a very different context, storage in clusters, where client-server models are generally used to access remote storage (for instance NFS, PVFS or LUSTRE).

Our experimental study based on the usage of the GM programming interface of MYRINET high-speed networks for distributed storage did raised several interesting problems. Firstly, the specific memory utilization in the storage access system layers does not easily fit the traditional memory model of high-speed networks. Secondly, client-server models that are used for distributed storage have specific requirements on message control and event processing, which are not handled by existing interfaces.

We propose different solutions to solve communication control problems at the file-system level. We show that a modification of the network programming interface is required. Data transfer issues need an adaptation of the operating system. We detail several propositions for network programming interfaces which make their utilization easier in the context of distributed storage. The integration of a flexible processing of data transfer in the new programming interface MYRINET/MX is finally presented. Performance evaluations show that its usage in the context of both storage and other types of applications is easy and efficient.

Keywords :

Distributed Storage, Remote File Access, Cluster, High-Speed Network, MYRINET, Zero-copy, Memory Registration, Communication Control, Event Notification, Application Programming Interface.

Résumé :

L'objectif de ce travail est d'étudier l'exploitation des réseaux haute performance des grappes dans le cadre du stockage distribué.

Les applications parallèles s'exécutant sur les grappes nécessitent à la fois des communications performantes entre les différents nœuds et des accès efficaces au système de stockage. Les travaux menés sur les technologies réseau ont abouti à la conception d'architectures dédiées aux grappes qui permettent des communications très rapides entre les nœuds. Les travaux visant à obtenir un stockage distribué efficace dans les grappes se sont pour leur part principalement focalisés sur des mécanismes de parallélisation pour augmenter la charge de travail supportée par le (ou les) serveur.

Nous proposons dans ce travail d'améliorer les performances du stockage distribué dans les grappes en utilisant au mieux le réseau haute performance sous-jacent pour accéder au stockage distant. La question générale que nous soulevons est : est-ce que les réseaux rapides des grappes sont adaptés à un accès transparent, efficace et performant au stockage distant ?

Nous montrons que les besoins du stockage sont très différents de ceux du calcul parallèle. Les réseaux des grappes ont été conçus pour optimiser les communications entre les différents nœuds d'une application parallèle. Nous étudions leur utilisation dans le cadre, très différent, du stockage dans les grappes, qui s'appuie généralement sur un modèle client/serveur d'accès aux fichiers distants (par exemple NFS, PVFS ou LUSTRE).

Une étude expérimentale reposant sur l'utilisation de GM, l'interface de programmation du réseau rapide MYRINET, dans le contexte du stockage distribué révèle différents freins. Tout d'abord, l'utilisation mémoire particulière dans les couches système d'accès au stockage s'intègre difficilement dans l'habituelle gestion mémoire des réseaux rapides. Ensuite, les modèles client-serveur utilisés dans le stockage distribué présentent des besoins spécifiques pour la gestion des messages et des événements réseau, besoins non couverts par les interfaces actuelles.

Nous proposons différentes solutions pour résoudre, au niveau du système de fichiers les problèmes liés au contrôle du réseau mais montrons qu'il est nécessaire de modifier l'interface de programmation réseau et le système d'explotation pour venir à bout des difficultés liées au transfert de données. Nous détaillons des propositions à mettre en œuvre dans les interfaces de programmation du réseau pour faciliter leur utilisation dans le cadre du stockage. L'intégration dans une nouvelle interface de programmation, MYRINET/MX, d'une gestion souple des transferts de données est présentée. Les premiers résultats montrent que son utilisation dans le cadre du stockage distribué, mais aussi dans d'autres applications, se révèle aisée et efficace.

Mots-clés :

Stockage distribué, accès distant aux fichiers, grappe de calcul, réseau haute performance, MYRINET, zéro-copie, enregistrement mémoire, contrôle des communications, notification d'événements, interface de programmation.