Université de Montréal

Un protocole de communication pour applications transactionnelles distribuées

par

Wissam Hamzeh

Département d'informatique et
de recherche opérationnelle

Faculté des arts et sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de
Maître ès sciences (M. Sc.)
en informatique

juin 2002

Page d'identification du jury

Université de Montréal

Faculté des études supérieures

Ce mémoire intitulé :

Un protocole de communication pour applications transactionnelles distribuées

présenté par :

Wissam Hamzeh

a été évalué par un jury composé des personnes suivantes :

El Mostapha Aboulhamid

président du jury

Marc Feeley

directeur de recherche

Esma Aïmeur

membre du jury

Mémoire accepté le : 12 août 2002

Sommaire

Les applications transactionnelles sont très répandues et utilisées dans plusieurs contextes, à titre d'exemple, dans un système bancaire, un serveur web, dans les applications de télécommunication, etc. L'objectif initial de notre recherche est de trouver une solution permettant d'intégrer une application téléphonique transactionnelle en temps réel dans un environnement distribué.

La première voie choisie se base sur l'utilisation des systèmes à mémoire partagée distribuée. À cette fin, nous avons mené dans la première partie de notre travail un projet consistant à évaluer quelques systèmes à mémoire partagée distribuée et l'applicabilité d'une application transactionnelle à ces systèmes. Les résultats obtenus ont montré qu'un tel modèle ne satisfait pas le critère de performance visé.

Dans la deuxième partie de notre travail, nous avons conçu un protocole destiné aux applications transactionnelles distribuées. Les travaux déjà faits à ce sujet sont basés sur la centralisation d'un serveur frontal. Ce modèle souffre cependant d'un problème de congestion si le nombre des clients accroît. Notre protocole se fonde plutôt sur la distribution et la diffusion "multicast" ce qui induit une amélioration considérable de la performance du système. De plus, nous avons élaboré une technique de communication "Lightweight Ethernet Communication LEC" (LEC) basée sur la communication directe par la carte Ethernet. Nous avons réalisé et testé ce protocole sur un serveur parallèle. Les résultats obtenus sont encourageants et nous permettent de croire que l'applicabilité de ce protocole est envisageable dans un environnement industriel réel.

Mots clés : MPD, mémoire partagée distribuée, application transactionnelle, cluster, protocole distribué, serveur parallèle.

Summary

Transactional applications are widely used in many fields. As an example, in banking system, web server as well as in telecommunication. The initial objective of our research is to find a solution to enable the integration of a real time telecom application in a distributed environment. The first way we have chosen is based on utilizing a Software Distributed Shared Memory systems (SDSM). To this end, we have conducted in the first part of this work a project that consists on evaluating some SDSM systems and to test the applicability of a transactional application under these systems. The results obtained by our benchmarking process, did not show high performance.

In the second part of this work we have designed a protocol that fits the distributed transactional applications. Actually, the works that have been done in this field are mainly based on centralization of a front-end server. This model suffers from a congestion problem when the number of clients accessing the server increases. Our protocol is based on two concepts : the distribution and the multicast. In addition we have evolved a new communication technique known as "Lightweight Ethernet Communication" (LEC), which is based on the direct communication with the Ethernet Card. We have implemented and tested this protocol on a parallel server. The extracted results are encouraging and we believe that the applicability of this protocol is realizable in a real industrial environment.

Keywords : DSM, distributed shared memory, transactional application, cluster, distributed protocol, parallel server.

Remerciements

Je tiens à remercier toutes les personnes qui mont aidé à accomplir ce travail. Particulièrement, mon directeur de recherche Marc Feeley pour son orientation, ses précieux conseils et son aide pendant le séjour de ma maîtrise. Finalement je remercie ma famille et mes amis pour leur soutien inoubliable.

Table des matières

Table des figures

Chapitre 1

Introduction

1.1 Motivation

Dès les années 1980, le concept de système réparti se développe progressivement. Cette progression est due à la popularisation des PC ainsi qu'à l'évolution des réseaux. L'idée de créer un groupe de machines qui communiquent entre elles est largement utilisée pour plusieurs sortes d'applications, telles que les applications à calcul intensif, à temps réel, transactionnelle et d'autres. Également, la venue de l'Internet a joué un rôle important dans l'évolution des systèmes répartis. En effet, depuis que celui-ci s'est développé, une nécessité de faire coopérer des machines géographiquement distribuées est apparue, ce qui a rendu le concept de répartition, global au niveau de la planète. D'autre part, les systèmes répartis peuvent jouer un rôle important pour améliorer la performance de traitement en exploitant le parallélisme. Les systèmes répartis peuvent aussi être employés pour assurer la tolérance aux pannes (fault tolerance). Un ordinateur n'est après tout qu'un appareil mécanique et électrique qui peut être sujet à des défaillances matérielles de diverses natures. Le besoin du parallélisme dans un contexte distribué a entraîné des études approfondies et des recherches poussées dans ce domaine. Suite à ces recherches, il existe maintenant deux paradigmes principaux pour traiter le parallélisme dans les applications distribuées : le premier est par le passage des messages et le deuxième par les mémoires partagées distribuées. Le premier paradigme est adopté par les systèmes dont la communication entre ses

machines se fait par l'intermédiaire des messages, tandis que le deuxième est conçu pour donner l'illusion d'une mémoire globale formée par l'ensemble des mémoires physiquement distribuées afin de partager les données parmi les noeuds qui participent au calcul.

Dans les deux modèles décrits ci-dessus, le facteur prépondérant qui affecte la performance est le taux de communication effectué par unité de calcul. Malgré l'évolution des algorithmes répartis qui tentent de minimiser ce taux, les points de faiblesse restent souvent la latence du réseau ainsi que les surcoûts introduits par les protocoles de communications et le système d'exploitation.

Dans ce mémoire, nous nous intéressons à la conception et à la réalisation d'un système de traitement parallèle qui possède un protocole de communication performant, fiable et extensible pour traiter la tolérance aux pannes. Nous nous sommes particulièrement intéressés aux applications transactionnelles en temps réel qui demandent un taux d'accès assez élevé et pour lesquelles ce protocole est destiné.

1.2 Objectif

Ce mémoire est divisé en deux parties abordant le traitement parallèle des applications transactionnelles. La première partie est le résultat d'un travail effectué dans le cadre d'un projet subventionné par Ericsson, vise à étudier la faisabilité des systèmes à Mémoire Partagée Distribuée MPD, pour ce sujet ainsi qu'à présenter les résultats des benchmarks testés sur quelques systèmes MPD existants. Dans cette section, nous montrerons les limites des MPD pour répondre aux besoins des applications transactionnelles dans un contexte concurrent. Ceci sera démontré à partir d'un outil de simulation testé sur trois MPD sélectionnées. La deuxième partie du mémoire présentera la réalisation d'un serveur parallèle avec un protocole de communication fiable et spécialisé à ce genre d'applications.

La motivation pour réaliser ce travail, est que l'architecture classique client-serveur n'est plus en mesure de satisfaire le développement des "clusters" où le nombre des machines clients a grandement augmenté. Ce qui entraîne un problème majeur de congestion au niveau du serveur. Par exemple, il suffit de penser à un serveur web pour un site populaire où le nombre

d'accès se chiffre dans les milliers par heure. Dans ce mémoire, nous proposons une solution à ce problème en introduisant des techniques basées sur la répartition des serveurs tout en préservant la cohérence de la base de données reparties. Notre système est conçu essentiellement pour les applications transactionnelles distribuées, mais le protocole qu'il adopte est générique et peut servir à plusieurs sortes d'applications parallèles dont l'accès au groupe de serveurs coopératifs est indispensable. Dans ce protocole, nous avons élaboré une nouvelle architecture de communication évoluée pour le traitement parallèle satisfaisant les critères de performance et de scalabilité. Ce protocole est basé fondamentalement sur la communication directe de la carte Ethernet et le Multicast pour assurer un haut degré de performance. En plus ce protocole est extensible pour adopter un mécanisme tolérant aux pannes. Ce mémoire est organisé comme suit :

Le deuxième chapitre présente une vision globale sur les notions reliées à notre axe de recherche, le parallélisme et le réseau. Dans ce chapitre nous ferons un survol sur les différents sujets concernant les systèmes distribués ainsi que les protocoles de communications qui ont un lien avec notre travail. Dans le troisième chapitre nous aborderons le sujet de Mémoire Partagée Distribuée en détail. Ce chapitre explique notre travail dans ce domaine et évalue certains systèmes à Mémoire Partagée Distribuée dans le contexte d'une application transactionnelle téléphonique à temps réel. Le quatrième chapitre est le coeur de ce mémoire. C'est dans ce chapitre que nous expliquons la mise en oeuvre d'un protocole de communication destiné aux applications transactionnelles distribuées. Nous expliquons ensuite notre nouvelle architecture de traitement parallèle qui sera une solution pour le genre d'application traité dans le chapitre 3. Le chapitre 5 donne une conclusion sur le travail achevé dans ce mémoire ainsi qu'une idée des travaux futurs sur le protocole implanté.

Chapitre 2

Réseaux et parallélisme

La communication par ordinateur est devenue possible grâce à la connectivité fournie par les réseaux. L'importance des réseaux dans notre vie quotidienne est clairement apparente dans le développement rapide de l'Internet, qui comporte des milliers de sous-réseaux hétérogènes. Depuis quelques années, les chercheurs pensent de plus en plus à utiliser le réseau pour créer une entité de calcul coopérative. C'est dans ce contexte que le mot "cluster" (en français, grappe) est né et devenu populaire dans le monde du parallélisme. Un cluster est un ensemble de machines reliées par un réseau qui fonctionnent comme une seule ressource de calcul en utilisant des algorithmes de traitement parallèle qui assurent la coordination.

Ce chapitre présente les concepts de base et la terminologie utilisée dans le domaine du traitement parallèle, une introduction rapide et générale du modèle de système distribué et de la notion d'algorithme distribué. Nous procéderons aussi à la description de la technologie de commutation par paquets ainsi qu'à l'étude du mécanisme d'adressage physique utilisé par cette technologie pour ensuite faire un survol de quelques protocoles de communication populaires.

2.1 Système distribué

Un système distribué (ou réparti) peut être défini d'une façon formelle comme étant une structure logicielle et matérielle distribuée sur un réseau. Cette structure est modélisée par un graphe G connexe G = (X, U) où X est l'ensemble fini des entités (sites) et U celui des lignes de communication ou arêtes. Les constituants du réseau d'interconnexion sont les sites et le protocole de communication établi entre eux. Chaque site possède une mémoire locale et au moins un processeur. Les sites se communiquent entre eux par l'intermédiaire de canaux de communication. On distingue deux modèles de systèmes distribués soit "à passage de message" et "à mémoire partagée distribuée".

2.1.1 Système à passage de message

Ce modèle de système réparti est le plus vieux et le plus connu. Dans ce système, les processeurs qui sont distribués sur le réseau se communiquent par échange de message. Dans un tel modèle, les opérations effectuées par un processus peuvent être modélisées comme des événements qui sont :

- L'envoi d'un message m.
- La réception d'un message m.
- L'exécution d'une instruction qui n'implique ni une émission ni une réception mais plutôt une exécution interne dans le même processus.

Deux grands standards sont disponibles pour le passage de messages : Message Passing Interface (MPI) [MPI] et Parallel Virtual Machine (PVM) [PVM]. PVM est issu d'un projet de recherche universitaire alors que MPI est issu d'un consortium réunissant universitaires, utilisateurs et constructeurs. Le problème majeur du modèle de programmation par échange de messages est la difficulté d'optimisation des applications. Cela nécessite un effort considérable et de bonnes connaissances de la part du programmeur.

2.1.2 Système à mémoire partagée distribuée

L'abstraction de partage d'une mémoire sur un ensemble de sites physiquement distribués constitue un autre paradigme d'un système réparti. La mémoire partagée distribuée (MPD, ou en anglais DSM pour Distributed Shared Memory) logicielle donne l'illusion d'avoir une mémoire globale virtuelle formée par la somme des mémoires des noeuds qui sont distribués sur les réseaux. Dans ce modèle, le système gère les communications de bas niveaux entre les différents sites (par passage de message) sans que le programmeur s'en rende compte. Ceci facilite la tâche du programmeur puisque ce dernier n'a qu'à se soucier d'exprimer le parallélisme dans le programme, le partage des données se fait de la même façon que dans un programme séquentiel. L'enjeu dans les MPD c'est de garder une mémoire globale cohérente malgré les accès concurrents. De nombreux protocoles mettant en oeuvre une MPD sur une machine parallèle à mémoire répartie ont été proposés. IVY [Li89] est le premier système qui utilise un protocole de validation des données afin de garder la consistance de la mémoire, avec unité de cohérence, la page virtuelle du système. Dans le chapitre 3, les MPD seront traitées en détail du point du vue conceptuel et architectural.

2.2 Réseau

Un réseau est un ensemble de dispositifs matériels et logiciels permettant à deux machines (noeuds) ou plus de communiquer. Au plus bas niveau, les noeuds sont connectés par des moyens physiques que l'on appelle liens (ou liaisons). Ces derniers jouent le rôle de canaux de communication afin d'échanger les donné entre les noeuds. Il existe deux approches touchant les réseaux de communication pour assurer les connexions entre les noeuds. La première approche aborde les réseaux de "commutation de circuit" qui fonctionnent en établissant une connexion dédiée entre deux points. Ce genre de réseau est utilisé dans les applications téléphoniques. Un appel téléphonique fonctionne grâce à l'établissement d'un circuit point à point reliant l'appelant à celui de l'appelé. La deuxième approche est adoptée par les "réseaux à commutation par paquets".

2.2.1 Commutation par paquet

Les "réseaux à commutation par paquets" sont largement utilisés dans le monde de la télécommunication. La commutation par paquets "switched networks" est utilisée dans une telle quantité d'interconnexions que le terme "réseau" ne désignera plus que les réseaux de commutation par paquet. Dans ce modèle, le trafic émis sur le réseau est découpé en petits morceaux appelés paquets qui sont multiplexés sur les liens qui relient les machines. Un paquet ne contient généralement que quelques centaines d'octets et il contient des informations pertinentes permettant aux machines réceptrices d'identifier la machine destinataire avant de lui acheminer le paquet reçu. Ces informations sont stockées généralement au début du paquet dans une région qui s'appelle l'en-tête du paquet "packet header". Le principe de circulation des paquets sur les liaisons de connexions nous emmène à définir deux métriques importantes associées aux réseaux.

- *Le débit* : le débit d'une liaison entre deux noeuds sur un réseau est définie par le nombre maximum des bits qui peuvent être transmis sur cette liaison par unité de temps.
- *La latence* : cela représente le temps pris entre l'émission d'un paquet et sa réception par le destinataire.

2.2.2 La technologie Ethernet

"Ethernet" est le nom donné à une technologie très répandue de réseaux locaux à commutation par paquets inventée au début des années 1970 par Xerox. Dans cette technologie, chaque machine connectée sur le réseau est munie d'une interface Ethernet "NIC- Network Interface Card" généralement appelée carte transporteur ou simplement carte Ethernet. Cet équipement assure l'adaptation physique (niveaux électriques, codage, etc). Une Carte Ethernet contient des circuits électroniques qui permettent de détecter la présence des signaux sur le câble afin d'émettre et de recevoir des séquence de bits de ce dernier. En plus, elle convertit les signaux analogiques du câble en signaux numériques compréhensibles par la machine. La carte Ethernet incorpore un algorithme de détection des collisions. Une collision est établie lorsqu'une carte d'une machine veut envoyer un signal sur le lien physique partagé par plusieurs machines alors qu'un signal émi par une autre machine n'a pas encore atteint sa destination, ce qui fait que les deux signaux entrent en collision et deviennent incompréhensibles. Pour corriger un tel incident,

la carte utilise un algorithme CSMA/CD qui est basé sur la retransmission à délai aléatoire lors d'une collision. Les cartes Ethernet ont présentement une capacité de débit dans l'ordre de 100 Mb/s, d'autres architectures de cartes comme Myrinet (fabriquée par la société Myricom) peuvent atteindre 4000 Mb/s mais elles sont considérablement plus chères.

2.2.3 L'adressage Ethernet

L'interface Ethernet d'une machine utilise un mécanisme d'adressage qui empêche les paquets indésirables d'atteindre la machine. Dans le cas d'une topologie token ring, chaque interface reçoit une copie de tous les paquets même lorsqu'ils sont destinés à d'autres machines. Si les réseaux locaux n'avaient pas un tel mécanisme, chaque message circulant sur le réseau provoquerait une interruption processeur dans chaque poste raccordé, ce qui réduirait considérablement les performances de toutes les stations du réseau. Heureusement, l'interface filtre les paquets : elle ignore ceux qui sont destinés à d'autres machines et transmet à la machine juste ceux qui lui sont destinés. Les mécanismes d'adressage et le filtrage permettent d'éviter qu'une machine soit submergée de paquets entrants. Pour permettre à une machine de déterminer quels sont les paquets qui lui sont destinés, chaque machine connectée sur le réseau dispose d'un identificateur unique. L'IEEE a défini un format universel sur 48 bits qui constitue l'adresse Ethernet ou bien le "MAC address". Les constructeurs de matériaux Ethernet réservent des blocs d'adresses Ethernet et les affectent séquentiellement au fur et à mesure qu'ils produisent des interfaces Ethernet, ceci assure une unicité universelle des adresses Ethernet. Les adresses Ethernet sont parfois appelées "adresse physique" ; dans la littérature réseau, ceci parvient du fait que l'adresse représente une identification d'un équipement matériel. Il existe trois classes d'adresses Ethernet :

1. Adresse spécifique (unicast).
2. Adresse de diffusion (broadcast).
3. Adresse de diffusion de groupe (multicast).

La première adresse que l'on appelle adresse spécifique est l'adresse physique de la machine, elle est utilisée pour établir une communication un à un avec cette machine (unicast) tandis que l'adresse de diffusion est réservée pour l'émission à toutes les machines qui sont connectées sur le réseau. Finalement, la troisième adresse nommée adresse de diffusion de groupe constitue une forme limitée de diffusion où seulement un ensemble des machines désignées sur le réseau

reçoivent un paquet destiné à cette adresse. Les mécanismes de diffusion (broadcast) et de diffusion de groupes (multicast) sont abordés en détails dans les sections 2.3.3 et 4.8.1 .

2.2.4 Structure d'une trame Ethernet

Le mot *trame* est utilisé dans les réseaux Ethernet pour désigner l'unité de données transmise sur une ligne série. Les trames Ethernet ont des tailles variables comprises entre 64 et 1518 octets. La trame comme indiqué dans le tableau contient un préambule de 8 octets. Les champs adresses contiennent les adresses MAC qui définissent les adresses physiques source et destination, un identificateur de 2 octets (EtherType) qui identifie le type de données transportées c'est-à-dire la couche supérieure qui sera responsable au traitement de la trame. Les champs de données est suivi d'un contrôle d'erreurs de type CRC (Frame Check Sum) sur 4 octets.

Préambule 64 bits	Adresse destination 48 bits	Adresse source 48 bits	Type de trame 16 bits	Données 368-12000 bits	CRC 32 bits

Du point de vue de l'interconnexion de réseaux, le champ "type de trame" est essentiel, il signifie que les trames Ethernet s'auto-identifient. Le système d'exploitation d'une machine qui reçoit une trame examine ce champ afin de déterminer le protocole de communication qu'il doit traiter. Par exemple, le type de 0x87 représente une trame TCP/IP. Nous verrons dans les sections suivantes comment les protocoles, utilisent l'auto-identification des trames pour vérifier celles qui les concernent.

2.3 Protocoles

Un protocole est un ensemble de normes qui organisent la procédure de communication entre deux parties. Deux philosophes ne seront pas capables de partager leurs idées sans adopter une langue commune comprise par les deux. En télécommunication, le même principe s'applique. En effet, pour que deux machines connectées par des liens physiques puissent communiquer, il sera nécessaire qu'elles utilisent des standards et des règles communes qui assurent la compréhension

des informations échangées. Il existe des centaine des protocoles différents qui gèrent la communication entre des machines interconnectées.

2.3.1 Routeurs et passerelles

Dans un système de commutation par paquet, le "routage" (routing) fait référence aux processus de choix d'un chemin suivant lequel les paquets seront transmis et le terme routeur désigne la machine effectuant un tel choix. Un routeur permet le relayage entre deux réseaux d'espace d'adressage homogène (IP/IP, ISO/ISO,...) mais pas entre deux réseaux d'adressage hétérogène (ex. IP/ISO). Dans ce dernier cas la conversion de l'adressage nécessite l'utilisation d'un mécanisme de conversion particulier. Dans ces conditions l'organe d'interconnexion est une passerelle (Gateway). Une passerelle sert à réunir deux réseaux parfaitement hétérogène (ex. IP/X25). Dans les cas les plus extrêmes, la passerelle supprime toutes les informations spécifiques au réseau et emballe à nouveau les données dans des paquets adaptés pour l'autre réseau. Les routeurs orientent les paquets selon des informations contenues dans des tables de routage. Ils utilisent deux modes de routages :

1. le routage statique ou fixe, dans ce type de routage, les tables de routage sont introduites par l'administrateur de réseau.
2. le routage par le chemin le plus court, dans ce type de routage, les tables de routage indiquent pour chaque destination le coût le moins élevé. Périodiquement des échanges d'informations entre les routeurs permettent de maintenir ces tables à jour.

2.3.2 Organisation en couche des protocoles

La majorité des protocoles de communication sont organisés en plusieurs couches. Du point de vue conceptuel, on peut imaginer une telle organisation comme un empilement vertical de couches dont chacune est responsable de la gestion d'une partie du problème. Il y a plusieurs facteurs qui justifient cette organisation. Premièrement, il est vital dans tout logiciel de séparer les tâches c'est-à-dire de décomposer le problème en plusieurs sous-problèmes qui sont traités par des modules différents. Deuxièmement, la conception et l'architecture du réseau d'interconnexion

et les logiciels de communication sont interdépendants, ce qui nécessite la séparation des modules en couches en associant l'interaction logiciel-matériel aux bas niveaux et les traitements des données aux couches supérieures.

Prenons par exemple le processus de communication entre deux applications FTP sur le réseau, comme indiqué dans la figure 2.1, si l'on veut émettre un message à partir de la couche application d'une machine vers une autre, la démarche suivante s'impose :

1. le transfert de ce message de haut en bas à travers les couches logicielles de la machine.
2. le transfert des données sur le réseau.
3. le transfert de bas en haut à travers les couches logiciels de la machine réceptrice.

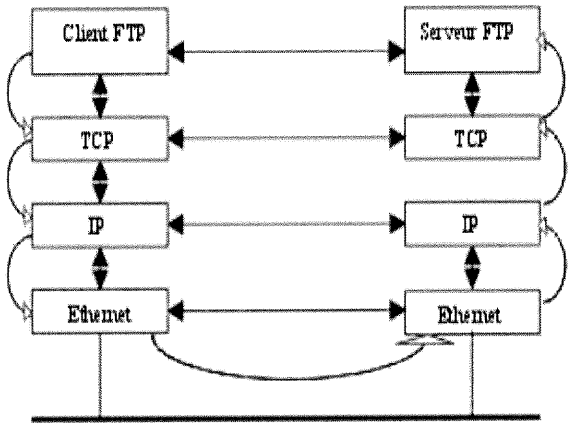

FIG. 2.1 – Structuration en couche des protocoles

Lors de la réception d'un message, chaque couche examine la validité des données du message et de son traitement d'après son type et son adresse de destination. Ainsi la couche peut décider de conserver le message ou de l'acheminer à une couche supérieure qui décide à son tour quel est le programme d'application destinataire de ce message. Dans la figure 2.1, la couche application représentée par l'application FTP est un programme usager, tandis que les trois couches en bas sont normalement implantées dans le noyau du système d'exploitation.

2.3.3 Protocole IP

Le protocole IP est un protocole sans connexion et non fiable. IP est considéré comme un support des protocoles TCP, UDP, ICMP et IGMP, tous ces protocoles envoient les données sous la forme d'un paquet IP. Le protocole IP est une partie de la conception d'interconnexion dont l'importance est telle que ces dernières sont parfois appelées "technologies fondées sur IP".

Protocole de livraison sans connexion

Un protocole primitif de communication est le protocole de livraison de paquets sans connexion. C'est un protocole de livraison de paquet non fiable. Le terme sans connexion signifie que le système ne garde aucun état d'information sur les paquets successifs, et ce service est dit non fiable car la livraison de paquet n'est pas garantie. Des paquets peuvent être perdus, dupliqués, retardés, altérés ou livrés dans le désordre. Le service sans connexion ne détecte pas ces conditions et n'en informe ni l'émetteur ni le destinataire. Il assure une livraison dite pour le mieux "Best effort", parce que les couches réseaux font tout ce qui est possible afin de délivrer les paquets.

Adresse IP

Il existe une certaine similitude entre le processus d'adressage Ethernet et celui de IP. Une "adresse Ethernet" IP est formée par des entiers qui sont soigneusement choisis afin d'assurer l'efficacité du routage. En particulier une adresse IP inclut l'identification du réseau auquel appartient la machine ainsi que celle de la machine particulière sur ce réseau. Chaque machine (avec une seule carte réseau) d'une interconnexion dispose d'une adresse IP unique sur 32 bits et cette adresse est utilisée pour toutes les communications avec cette machine.

Conceptuellement, les adresses IP sont divisées en cinq classes A, B, C, D, E qui sont identifiées à partir de trois bits de poids fort. Les adresses de la classe A sont utilisées pour les réseaux qui comportent plus de 216 machines et elles affectent 7 bits à l'identificateur de machine. Les réseaux qui ont entre 28 et 216 machines font partie des adresses de la classe B et elles affectent 14 bits à l'identificateur de réseau et 16 bits à l'identificateur de la machine. Ensuite, la

classe C alloue 21 bits à l'identificateur de réseau et 8 à l'identificateur de la machine. La classe D est consacrée pour le Multicast, cette classe des adresses ne contient pas l'identificateur de la machine parce qu'un message sera envoyé à tous les sites du réseau. Finalement, la dernière classe E est réservée pour une future utilisation. Le Tableau 1 présente l'intervalle de l'adressage de chaque classe :

Tableau 1

Classes	Intervalle
A	0.0.0.0 à 127.255.255.255
B	128.0.0.0 à 191.255.255.255
C	192.0.0.0 à 223.255.255.255
D	224.0.0.0 à 239.255.255.255
E	240.0.0.0 à 247.255.255.255

Il reste à mentionner que cette version de IP est notée IPv4. Cependant, vers la fin des années 90 une autre version de IP est apparue et se nomme IPv6. Pour répondre aux besoins du développement rapide de l'Internet et à la nécessité d'avoir un plus grand rang d'adresses IP, la version IPv6 permet l'adressage sur 128 bits. Cette nouvelle version de IP ne contient pas des classes d'adresses comme celle de IPv4, mais plutôt des bits de poids fort qui identifient le type d'adresse. Par exemple, si les 3 bits poids fort sont 001 ce sera une adresse unicast globale (Aggregatable Global Unicast). Cette adresse remplace les 3 classes A, B, C en IPv4, si les 8 bits de poids fort sont 11111111 (0xff), ceci signifie une adresse de diffusion de groupe (multicast). Dans le reste du mémoire, nous utiliserons le mot IP afin d'exprimer la version IPv4 du protocole IP.

Adresse de diffusion de groupe (Multicast)

Un autre avantage significatif du plan d'adressage IP est l'inclusion d'une adresse de diffusion qui fait référence à un groupe de machines d'un réseau. Les adresses de la classe D entre les bornes 224.0.0.0 à 239.255.255.255 sont les adresses de diffusion de groupe. Les 28 bits de poids faible de la classe D forment l'identificateur du groupe et l'adresse complète à 32-bit s'appelle

l'adresse de groupe. Pour faire une diffusion de groupe le système doit fournir une adresse physique en plus de l'adresse Multicast. L'adresse Ethernet de groupe est constituée de 48 bits, les 24 bits poids fort sont 01 :00 :5e. Le bit suivant est 0 et ce qui reste (23 bits poids faible) est copié à partir de l'adresse de l'adresse IP de multicast. Il existe quelques adresses de diffusion particulières en IP citons par exemple 224.0.0.1 pour la diffusion sur tous les sous-réseaux, et 224.0.0.2 pour la diffusion sur le groupe de routeurs. Enfin, la diffusion de groupe est un aspect très important et nous nous servons de cette technique afin de réaliser la communication entre les sites dans notre système. Nous aborderons cet aspect dans le chapitre 4.

Structure d'un datagramme IP

Il y a une grande similitude entre une liaison physique et une interconnexion IP en ce qui concerne le modèle de transfert de données. Une trame au niveau physique représentant l'unité de données a une en-tête qui contient une adresse physique source et destination et le type des données envoyées. Ainsi un datagramme IP représente l'unité de donnée pour une interconnexion IP est formé généralement par une en-tête qui contient des adresses source, destination et un type de données envoyées. Bien entendu les adresses qui sont utilisées dans un datagramme sont des adresses IP. La figure 2.2 représente un datagramme IP, il contient les champs suivants :

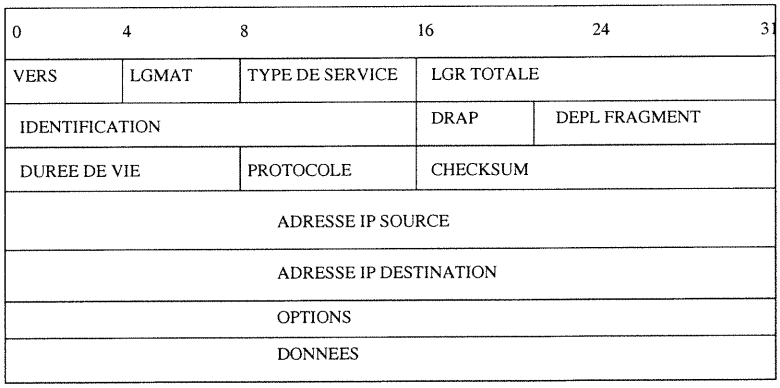

FIG. 2.2 – Structure d'un paquet IP

- Le premier champ sur 4 bits représente la version du protocole IP, dans le cas de IPv4 ce champ vaut 4.
- LGMAT est la longueur de l'en-tête il occupe également 4 bits et indique la longueur de

l'en-tête en mots de 32 bits.

- Le type de service sur 8 bits indique comment le datagramme doit être géré.
- LGR-TOTALE sur 16 bits est la taille du datagramme. Puisque ce champ est sur 16 bits la taille maximum d'un datagramme est de $216 = 65536$ octets.
- Le champ d'identification sur 16 bits est affecté par l'émetteur pour chaque datagramme.
- La durée de vie (TTL en anglais) sur 8 bits est utilisée par IP pour le routage, il est initialisé par l'émetteur et décrémenté de 1 par chaque routeur qui veut envoyer le paquet. Si le TTL est rendu à zéro le paquet sera éliminé.
- Le champ protocole sur 8 bits spécifie le type de données qui sont portées par le datagramme. Typiquement les valeurs 1 pour ICMP, 2 pour IGMP, 6 pour TCP et 17 pour UDP.
- Le champ checksum de 16 bits calculé sur l'en-tête IP est l'algorithme standard de l'Internet.
- L'adresse source et destination du IPv4.

2.3.4 Protocole ICMP

Dans une interconnexion IP, il n'existe pas de mécanisme permettant à l'émetteur de dire si un échec de transmission est dû à un dysfonctionnement local ou distant. En plus, le protocole IP ne contient aucune information pouvant aider l'émetteur à tester l'accessibilité ou à s'informer sur une panne. Pour permettre aux passerelles d'une interconnexion de se rendre compte des erreurs ou bien fournir des informations relatives sur des circonstances particulières, les concepteurs du protocole IP ont ajouté un mécanisme d'échange des rapports. ICMP (Internet Control Mechanism Protocol) est un mécanisme de compte rendu "error reporting mechanism". Ce protocole sert à informer l'émetteur initial d'une erreur rencontrée pendant le voyage du datagramme. Par la suite, l'émetteur doit à son tour associer ces erreurs à des programmes d'application qui vont y apporter les corrections nécessaires. Un message ICMP est encapsulé par le paquet IP qui est placé dans le champ de donnée d'une trame physique. Il est important de mentionner que ICMP n'est pas considéré comme étant un protocole de haut niveau, mais plutôt comme une partie essentielle d'IP.

Ainsi, une autre fonctionnalité de ICMP est de fournir des rapports afin de bien tester

l'accessibilité d'une machine sur le réseau. Le programme "ping" utilise ce mécanisme pour envoyer des messages de demande d'écho vers une destination particulière. Chaque machine qui reçoit un message de type demande d'écho, construit un message de réponse et le retourne à l'émetteur. Ceci constitue un outil précieux qui permet aux administrateurs du réseau de savoir le délai de temps pris par un paquet pour faire un aller-retour entre deux machines ainsi que pour obtenir des informations sur l'accessibilité de la machine destinataire.

2.3.5 Protocole UDP

Le protocole de datagramme utilisateur UDP (User Datagram Protocol) assure un mécanisme de base permettant aux programmes d'application d'envoyer des datagrammes à d'autres programmes d'application. Les messages UDP contiennent à la fois le numéro de port destinataire et le numéro de port source. Le protocole UDP adopte un service de livraison non fiable en mode sans connexion qui utilise IP pour acheminer les messages entre les machines sources et destinataires. Bien entendu le protocole UDP n'utilise pas les accusés de réception ni aucun autre moyen afin de garantir la réception d'un message, en plus il n'assure aucun séquencement pour les messages reçus. Un message envoyé par un client UDP peut être perdu sans que l'émetteur ne s'en aperçoive, de même que les datagrammes UDP peuvent être dupliqués et mal ordonnés lors de la réception.

Un logiciel qui utilise ce protocole comme moyen de communication pour acheminer les messages doit prendre soin de tous ces cas pour maintenir une fiabilité et une fonctionnalité correctes.

Un message UDP est encapsulé dans un paquet IP. Du point de vue conceptuel le datagramme UDP comporte : Une en-tête UDP et une zone de données. L'en-tête datagramme UDP contient les champs suivants :

1. Le numéro de port source.
2. Le numéro de port destination.
3. La longueur des données.
4. La somme de vérification (checksum est mentionné dans la section 2.3.3).

Le numéro de port source est facultatif puisqu'il indique le port où la réponse d'un message doit être envoyée. La couche UDP est conceptuellement défini au-dessus de la couche IP, ce qui signifie qu'un message UDP est contenu dans un datagramme IP placé dans une trame physique. Pour qu'un datagramme de type UDP destiné à une certaine application distante soit traité par ce dernier, il doit franchir la couche IP qui valide son type avant de le laisser passer vers la couche UDP supérieure. Tout ceci est effectué en vérifiant le champ "protocole" dans l'en-tête IP qui indique le type du protocole (17 pour UDP) des données encapsulées.

2.3.6 Protocole TCP

TCP (Transmission Control Protocol) est un protocole orienté connexion qui suppose un accord préalable entre les deux extrémités. Autrement dit, les programmes d'application client et serveur doivent accepter de participer à une connexion avant que cette dernière ne soit établie. Comme le protocole UDP présenté dans la section précédente se trouve au-dessous d'IP dans le modèle d'organisation en couches, le protocole TCP réside dans une couche supérieure d'IP comme l'indique la figure suivante :

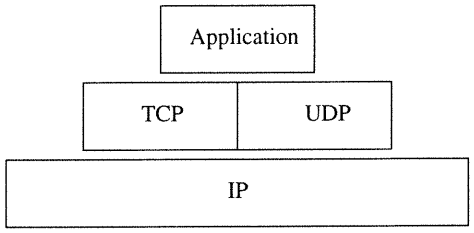

FIG. 2.3 – Hiérarchie des protocoles

Le service fourni par TCP pour une application est différent de celui offert par UDP. Tout d'abord, TCP fournit des connexions entre les clients et les serveurs, c'est-à-dire un client établit une connexion avec un serveur donné et échange les données avec lui pour ensuite terminer la connexion. TCP assure aussi la cohérence des données échangées. Quand un client TCP envoie des données vers l'autre extrémité, il exige un accusé de réception au retour. Si l'accusé n'est pas reçu, le client va automatiquement retransmettre les données en attendant une plus longue durée. Après un certain nombre de retransmissions échouées, il arrête ce processus en supposant que la connexion est brisée.

Un autre problème dans le réseau constitue le non ordonnancement. TCP s'occupe de ce problème en attribuant des numéros de série séquentiels à chaque segment (unité de transfert en TCP) envoyé. Par exemple, si deux segments étiquetés 1 et 2 respectivement sont reçus dans un ordre différent, le TCP receveur va réordonner ces segments selon la façon dont ils ont été envoyés avant de les retransmettre vers l'application correspondante.

Établissement d'une connexion

Pour créer une connexion TCP, les deux parties doivent s'entendre sur l'établissement de cette connexion. Le scénario suivant montre comment TCP implante ce mécanisme qui s'appelle le "three-hand-shaking" :

- Le serveur doit être prêt à accepter des connexions.
- Le client effectue un appel pour faire la connexion, et envoie un SYN (synchronize- segment TCP qui contient seulement les en-têtes TCP, IP et un nombre dans le champs de données) en informant le serveur du numéro de série initial.
- Le serveur acquitte le SYN du client en envoyant son propre SYN avec un numéro de série initial pour les segments envoyés par le serveur.
- Le client envoie un accusé de réception.

Fermeture d'une connexion TCP

Les détails de la fermeture d'une connexion sont toutefois plus subtils que ceux pour l'établissement. La figure 2.4 illustre le scénario. Le site émetteur envoie un segment FIN pour aviser l'autre site de son désir de mettre fin à la connexion. Le site receveur renvoie un accusé de réception et un autre message pour informer l'application de la demande de fermeture de connexion.

Enfin lorsque le programme d'application demande à TCP de libérer complètement la connexion, TCP émet le second segment FIN et le troisième message du site source est un accusé de réception.

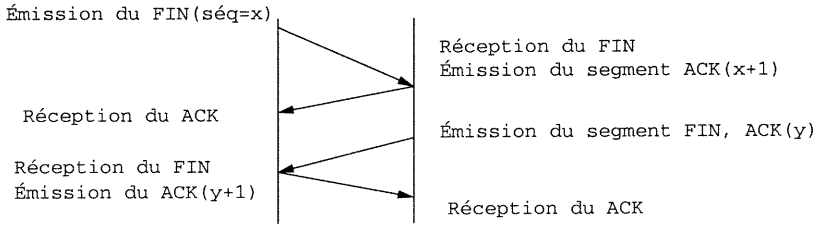

FIG. 2.4 – Fermeture d'une connexion TCP

2.3.7 Protocole IGMP

La diffusion (broadcast) exige que chaque machine sur le réseau reçoive le paquet diffusé ce qui veut dire que même les machines ne s'intéressant pas aux paquets transmis seront obligées de les recevoir. Une méthode pour résoudre ce problème, c'est par la transmission directe (unicast) d'un même paquet sur chaque machine qui veut recevoir les données. Cependant un problème sérieux de congestion sur le réseau est crée. La façon la plus efficace de contrer ce problème est l'utilisation de la diffusion de groupe où seule les machines dans le groupe désirant participer à la réception reçoivent une copie des données.

Le protocole IGMP (Internet Group Management Protocol) définit le mécanisme de diffusion de groupe sur un réseau IP. Les données sont envoyées à un ensemble d'usagers inscrits dans un groupe identifié par une adresse de destination de groupe "GDA". Selon IGMP, une machine sur le réseau peut décider à chaque moment de joindre ou de quitter le groupe de destination. Avec IGMP (figure 2.5) un émetteur envoie les données à diffuser sur le plus proche routeur. Dès que ce dernier reçoive les données, il vérifie si l'adresse IP de la classe D de groupe est existante. S'il ne trouve pas une telle adresse, le routeur ignore chaque paquet envoyé. Au moment où une machine décidera de joindre ce groupe, les étapes suivantes seront effectuées :

1. Le receveur envoie un message d'appartenance de type IGMP au routeur de son sous-réseau en déclarant son intérêt de joindre le groupe d'adresse 224.0.27.30.
2. Si le routeur a reconnu l'adresse, il commence à passer les paquets qui lui sont destinés, sinon il envoie un paquet aux autres routeurs pour chercher le groupe.
3. En communiquant avec les autres routeurs, le routeur dont il a reçu la demande de participation peut vérifier l'existence du groupe. Les routeurs de groupe utilisent le protocole

DVMRP pour échanger les messages d'appartenance de groupe.

4. Quand l'adresse de multicast est trouvée, les autres routeurs commencent à envoyer les paquets et à les dupliquer si nécessaire.

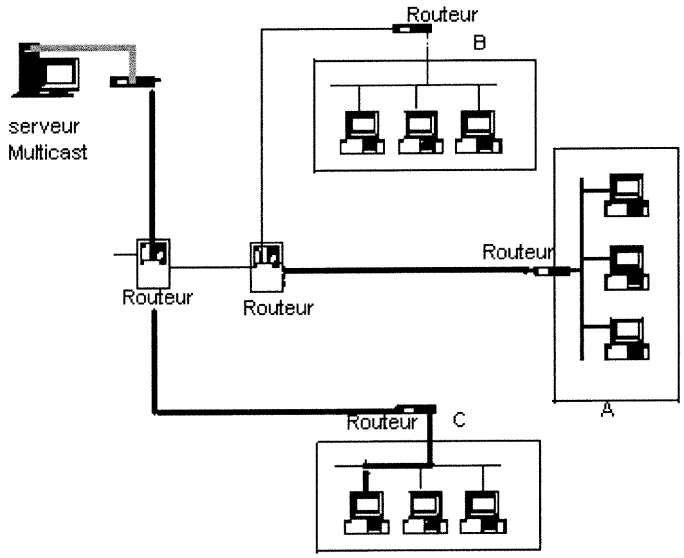

FIG. 2.5 – Protocole IGMP

Afin de garder l'information d'appartenance d'une machine dans le groupe, le routeur envoie périodiquement un paquet d'interrogation de type IGMP vers cette machine qui répond pendant un certain délai de temps. Les routeurs n'ont pas besoin de mémoriser l'état exact d'appartenance aux groupes, puisque la diffusion vers un groupe est effectuée en utilisant la diffusion matérielle. Il suffit que les routeurs sachent qu'il reste sur le réseau au moins une machine dans le groupe.

L'aspect important pour minimiser la congestion sur le réseau est identifié dans la figure 2.5.

5. Le paquet diffusé par le serveur Multicast n'est pas dupliqué au début de l'émission par le serveur, mais plutôt par le deuxième routeur qui identifie les groupes destinataires. Ce mécanisme évite une congestion considérable car le paquet est dupliqué juste au bon endroit où il devrait être dupliqué et non pas à partir de l'émetteur source.

2.4 Récapitulation

Dans ce chapitre nous avons abordé les notions sur les réseaux et le traitment parallèle, y compris les systèmes à passage de message et à memoire partagée distribuée. Ainsi, nous avons fait un survol sur l'architecture des réseaux d'interconnexion. Également, nous avons étudié les protocoles de communication sous IP au niveau conceptuel et architectural. Dans le chapitre suivant nous ferons une étude d'évaluation détaillée sur les systèmes à mémoire partagée distribuée en présentant nos résultats concernant l'applicabilité de tels systèmes aux applications transactionnelles distribuées.

Chapitre 3

Évaluation des systèmes à mémoire partagée distribuée

Le paradigme de la mémoire partagée distribuée MPD accroît en popularité depuis que le premier système logiciel IVY [Li89] est apparu en 1989. À partir de ce moment, de nombreuse études d'évaluation sont effectuées sur ce sujet. Par contre, souvent ces études tentent de montrer l'utilité des MPD par rapport aux applications qui nécessitent des calculs intensifs, telles que les applications dans les domaines de l'analyse numérique et l'imagerie. Dans ce chapitre, nous établirons un lien entre les applications transactionnelles distribuées et la MPD logicielle. Ce lien consistera à distribuer les accès transactionnels sur une grappe "cluster". Nous présenterons également une étude de faisabilité de l'implantation d'une application transactionnelle au-dessus d'une MPD logicielle. À cette fin, nous choisirons quelques MPD logicielles existantes, qui seront le sujet d'évaluation de ce chapitre. De plus nous simulerons une application téléphonique transactionnelle sur ces MPD choisies et montrerons les résultats d'évaluation obtenus.

3.1 Mémoire partagée distribuée

La mémoire partagée distribuée est un mot hybride qui combine deux notions de traitement parallèle : la mémoire partagée et le système distribué. Par définition, une MPD (Mémoire Partagée Distribuée) logicielle est un modèle qui fournit l'abstraction d'une mémoire partagée dans un système dont les mémoires sont physiquement distribuées sur le réseau. L'abstraction MPD possède plusieurs avantages en ce qui concerne le parallélisme et la distribution. Elle offre à l'utilisateur l'illusion d'avoir une mémoire virtuelle équivalente à la somme des mémoires physiques des sites, tout en gardant une cohérence parmi les mémoires physiquement distribuées. En plus, elle libère le programmeur de la tâche difficile de gérer manuellement le parallélisme et la communication entre les sites. C'est la responsabilité de la couche MPD de réaliser la distribution de mémoire et la synchronisation entre les différents processus distribués. Ces caractéristiques ont pour conséquence de simplifier beaucoup des difficultés associées au parallélisation pour une variété de domaines tels que l'analyse numérique et l'imagerie. Cependant, il ne faut pas oublier que la simplicité d'écrire des programmes parallèles sous une MPD, par rapport au modèle du passage par message, induit un certain coût à payer du côté performance pour réaliser la communication et la synchronisation. En effet, il est très difficile au système de pouvoir faire du recouvrement entre calculs et communications et d'atteindre les performances optimales obtenues avec les modèles à passage par message sans l'aide du programmeur. Toutefois, en prenant quelques précautions, il est possible d'implanter des applications distribuées efficaces utilisant la mémoire partagée. La première implémentation d'une MPD logicielle qui a vu le jour sur des réseaux de stations de travail est celle de IVY [Li89]. Dans une telle architecture, l'abstraction de mémoire partagée est implantée en utilisant les mécanismes de protection du système de mémoire virtuelle. Ce principe est connu sous le nom de Mémoire Virtuelle Partagée (MVP). À partir de ce moment, différentes techniques ont été mises au point pour favoriser la localité des accès aux données et limiter ainsi les échanges sur le réseau.

3.2 Classification des MPD logicielles

Les concepteurs des MPD ont été inspirés par le principe de la mémoire virtuelle du système d'exploitation centralisé aussi bien que par le principe de maintien de la cohérence de la mémoire cache dans les systèmes multiprocesseurs à mémoire partagée. Cependant ces derniers, souffrent d'un problème du "scalabilité" puisque le bus représente un goulot d'étranglement pour ce type de machines. D'autre part, le réseau des stations de travail devient de plus en plus populaire et puissant de sorte que l'accès aux données cachées dans une mémoire à distance puisse être, pour quelques réseaux sophistiqués, plus rapide qu'accédant à des données locales sur l'unité de disque dure. Toutes ces raisons ensemble ont donné la naissance à plusieurs systèmes implantant la MPD logicielle, citons les plus connus : Treadmarks [Keleh94], Midway [Bershad93], Munin [Carter91], CRL [Johnson95], Calypso [Barat95]. Bien que, le rôle de ces MPD soit essentiellement de mettre en application l'abstraction de la mémoire partagée au-dessus d'un réseau de stations de travail, ces systèmes n'ont pas suivi le même mécanisme quant à la conception et l'exécution. Nous classifions les MPD selon trois catégories principales "Algorithmique", "Implementation" et "Cohérence de mémoire".

3.2.1 Première classification (Algorithmique)

Algorithme d'un serveur central

Dans cet algorithme, toutes les demandes en provenance des différents sites passent par un serveur central. Le serveur est la seule entité interne dans le système qui connaît l'endroit réel du bloc demandé dans la mémoire distribuée. C'est sa responsabilité de faire l'ordonnancement des demandes afin de garder la cohérence des données distribuées. Le travail principal pour l'algorithme du serveur central, figure 3.1, est de servir les demandes de lectures issues d'autres noeuds, de mettre à jour des données lors des demandes d'écriture par des clients et de renvoyer des messages d'acquittement [Barat95]. Voici un scénario du fonctionnement de cet algorithme :

1. Le noeud A envoie une demande à S pour la mémoire sur le noeud B.
2. Le noeud S reçoit la demande du noeud A.

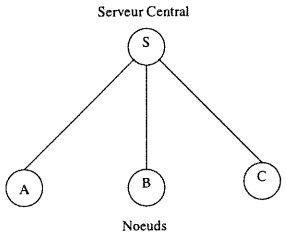

FIG. 3.1 – Modèle serveur central

3. Le noeud S fait suivre la demande au noeud B.
4. Le noeud B répond au noeud S.
5. Le noeud S fait suivre la réponse au noeud A.

L'avantage de cette méthode est qu'une cohérence forte des données distribuées est maintenue. Bien que l'algorithme du serveur central soit simple à mettre en application et assure le niveau élevé de la consistance de mémoire, il peut devenir un goulot d'étranglement. Pour surmonter ce problème, des données partagées peuvent être distribuées entre plusieurs serveurs. Dans ce cas, les clients doivent pouvoir localiser le serveur approprié pour chaque accès de données.

Algorithme de Migration

Dans cette approche il n'y a aucun serveur central, mais à la place tous les noeuds savent (basé sur la distribution statique des adresses) le processeur propriètaire du bloc de mémoire partagé sur le réseau. Un seul processeur à la fois peut posséder un bloc de données. Quand une demande est faite visant ces données, le propriétaire envoie la page correspondante au demandeur et invalide sa copie. Ainsi, seulement une copie d'un bloc existe à la fois dans le réseau entier et cette copie peut être lue ou écrite. Si un autre noeud a besoin d'une copie des données, figure 3.2, il envoie une demande au propriétaire et si ce dernier a toujours les données en question il les renvoie. Si le propriétaire a déjà envoyé les données à un autre noeud, il fait suivre la demande au nouveau propriétaire qui à son tour répète ce processus jusqu'à atteindre le site désiré.

L'inconvénient de ce schème, est qu'une demande peut être expédiée plusieurs fois avant

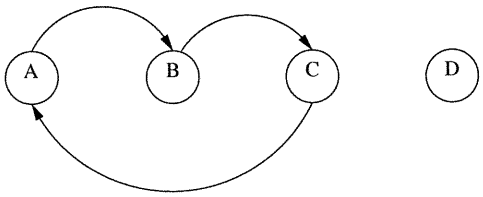

FIG. 3.2 – Modèle de migration de pages

d'atteindre le propriétaire courant. Cette faiblesse est très coûteuse pour des applications dont l'accès aux données est aléatoire et qui pourrait produire un taux élevé d'échange des données entre les noeuds. En plus, le partage d'une page entre plusieurs noeuds complexifie le problème, parce que si la migration est appliquée systématiquement, la page est déplacée de noeud en noeud et le coût de la copie n'est jamais amorti car très peu d'accès locaux sont effectués avant que la page soit déplacée à nouveau. Ce phénomène est communément appelé ping-pong [Cox90].

Duplication

Un autre inconvénient de l'algorithme de migration est que les threads d'une seule machine peuvent accéder à des données contenues dans le même bloc en même temps. La duplication peut réduire le coût moyen des lectures multiples. De plus, tous les noeuds savent (basé sur la distribution statique d'adresse) le processeur possèdant un bloc particulier de mémoire. Les demandes de lectures sont manipulées différemment aux écritures : en mode de lecture si un bloc est déjà partagé, alors le processeur demandant est ajouté à un ensemble de machines et le bloc est renvoyé. En mode d'écriture le bloc est renvoyé mais il sera exclusivement possédé par le processeur demandant.

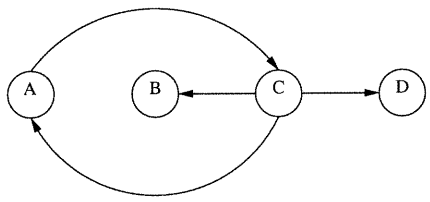

FIG. 3.3 – Modèle de duplication

Comme mentionné dans la figure 3.3, l'algorithme pour un accès de localité de données dans

la mémoire partagée distribuée est récapitulé ci-dessous :

1. Le noeud A détermine (suite à une faute de page) que l'adresse qu'il exige n'est pas locale et envoie une demande d'inscription de mémoire sur le noeud C.
2. Le noeud C reçoit la demande du noeud A, mais détermine qu'il a prêté des copies du bloc pour lecture aux noeuds B et D.
3. Le noeud C envoie un message d'invalidation à B et D.
4. Le noeud C répond avec le bloc demandé au noeud A et marque ce bloc comme exclusivement possédé par A.
5. Le noeud A reçoit la réponse de C.
6. Le noeud A peut maintenant lire et écrire le bloc.

Ce type de partage en lecture seule est souvent utilisé pour les données initiales des problèmes de calcul parallèle. Il est alors très intéressant de dupliquer ces données, le coût de la copie étant largement amorti par le nombre d'accès distants qu'il permet d'éviter. La consommation de mémoire que cette technique induit, représente un inconvénient inévitable. La duplication est donc une technique complémentaire de la migration pour accroître la localité des accès aux données.

3.2.2 Deuxième classification (Implémentation)

Le mécanisme d'implementation est considéré comme une des fonctions les plus importantes afin d'établir un système de MPD logicielle puisqu'il affecte la programmation et la valeur globale du système. Nous distinguons deux types d'implementation :

MPD basée sur la pagination

Ce genre de systèmes MPD se base fortement sur l'intervention du système d'exploitation. Il considère la page d'une mémoire (4Ko pour x86) comme étant l'unité de partage parmi les noeuds qui sont disponibles sur le réseau. En plus, il utilise les ressources et les signaux du système d'exploitation afin de détecter une faute d'une page sur une machine. Une faute de page est réalisée sur un noeud lorsque ce dernier tente d'accéder une unité de donnée qui n'est

pas disponible localement. Ces MPD logicielles diffèrent par la façon dont elles sont implantées. Certaines sont mises en application comme une partie intégrante dans le noyau du système d'exploitation alors que d'autres sont implantées par une librairie de fonctions qui fournit les services de gestion d'une mémoire partagée en mode utilisateur [Keleh94].

MPD basée sur l'objet

Au lieu d'utiliser la page de mémoire comme une unité de partage parmi les noeuds sur le réseau, ce modèle considère l'objet comme une unité de partage [Barat96]. Cette technique élimine la nécessité de l'intervention du système d'exploitation puisque les informations sont encapsulés dans l'objet partagé en question. L'élimination de faux de partage (décrit dans la prochaine section) souvent rencontré dans les systèmes basés sur la pagination, représente un autre avantage pour ce modèle de MPD logicielle. Certaines MPD existantes [Bershad93] modifient le compilateur pour aider à détecter une faute d'accès plus rapidement, cela a pour effet d'accroître la performance de la MPD au détriment de la portabilité du système.

3.2.3 Troisième classification (Cohérence de mémoire)

Le critère de la cohérence de mémoire dans les MPD est un facteur crucial qui affecte la qualité de ces systèmes. En effet, une MPD ne se contente pas de réaliser une mémoire distribuée mais doit aussi garantir la cohérence de cette mémoire distribuée. Autrement dit, quel que soit l'accès sur un bloc de mémoire par un processus p_i, le système doit retourner la valeur la plus récente reflétant toutes les modifications faites par les autres processus, tout en respectant l'ordre d'exécution de ces modifications. Dans la littérature des MPD plusieurs sortes de protocoles existent afin de maintenir la cohérence de mémoire distribuée. Nous allons en mentionner quelques-uns dans les sections suivantes.

Cohérence Séquentielle

La cohérence séquentielle a été proposée par Lamport en 1979 [Lamp79] pour définir un critère de correction pour des systèmes Multiprocesseurs à mémoire partagée. Un système est

dit séquentiellement cohérent si "le résultat de chaque exécution parallèle apparaît dans le système comme si l'exécution se faisait dans un ordre séquentiel sur un seul processeur". En d'autres termes, l'exécution d'un programme est séquentiellement cohérente si elle peut être produite par l'exécution de ce programme sur un système mono-processeur. Cependant, pour assurer un tel ordonnancement strict de toutes les modifications exécutées par les différents sites, Lamport propose aussi la création d'une horloge globale [Lamp78] du système afin d'assurer cette cohérence. De nombreux protocoles fondés sur la cohérence séquentielle ont été proposés dans le contexte des machines parallèles. Dans la plupart de ces protocoles, chaque processeur contient une copie de toute la mémoire partagée. Les opérations de lecture sont locales alors que les opérations d'écriture impliquent une synchronisation entre les processus et sont totalement ordonnées.

Cohérence Causale

Le modèle de cohérence causal [Ahmad91] représente un affaiblissement de la cohérence séquentielle parce qu'il fait une distinction entre les événements qui sont potentiellement causals et ceux qui ne le sont pas. Prenons un exemple de mémoire. Supposez que le processus P1 écrit X, puis le processus P2 lit la variable X et écrit à Y. La lecture de X et l'écriture de Y sont potentiellement causals parce que le calcul de Y a pu dépendre de la valeur de X lue par P2 (la valeur écrite par P1). D'autre part, si deux processus écrivent simultanément deux variables, ceux-ci ne sont pas reliés causalement. Quand il y a une lecture suivie plus tard par une écriture, les deux événements sont potentiellement causals. Les opérations qui ne sont pas reliées causalement sont considérées concurrentes. Pour qu'une mémoire soit considérée causale, il est nécessaire qu'elle obéisse à la condition suivante : toutes les écritures sont potentiellement causales doivent être vues par tous les processus dans le même ordre, tandis que les écritures concurrentes peuvent être vu par le système dans un ordre différent sur différentes machines.

Cohérence Processeur

La cohérence processeur a été définie en premier par Goodman [Good89] et a pour but d'alléger la "strictness" de la cohérence séquentielle qui est trop coûteuse à mettre en oeuvre.

"Un multiprocesseur vérifie la cohérence processeur si le résultat de n'importe quelle exécution est le même que si les opérations de chaque processeur individuel apparaissent (pour n'importe quel autre processeur) dans l'ordre séquentiel spécifié par son programme". Cette définition de Goodman de la cohérence processeur exige que tous les processus doivent voir toutes les écritures dans un même objet dans le même ordre et toutes les écritures d'un même processus dans le même ordre.

Cohérence Faible

La cohérence faible a été présentée par Dubois *et al* [Dubois86]. Ils définissent le concept d'ordonnancement faible dans lequel la cohérence objet est imposée uniquement à des points de synchronisation définis par l'utilisateur. Les synchronisations agissent comme des barrières. L'accès aux variables de synchronisation se fait d'une façon séquentielle. Considérons par exemple un processus qui met à jour une structure de données à l'intérieur d'une section critique. Il n'est pas nécessaire de s'assurer que tous les autres processus voient chaque mise à jour de façon séquentielle. Il suffit de s'assurer qu'ils voient la structure de données mise à jour à la fin de la section critique. Étant donné que tous les points de synchronisation sont identifiés (explicitement définis par le programmeur), alors il suffit de garantir la cohérence de ces points. Dans ce modèle, le concept de la section critique est indispensable pour définir les régions de la mémoire partagée par les différents processus.

Cohérence Release

La cohérence Release [Gharach90] a été mise en place afin d'améliorer la cohérence faible. Ce modèle a les mêmes caractéristiques que la cohérence faible, en plus il ajoute deux primitives pour assurer la synchronisation. Comme dans la cohérence faible, on distingue deux sortes de variables : celles qui ne sont pas partagées par les processus et celles qui le sont. Ces dernières sont protégées dans des sections critiques par des primitives acquire() et release() (figure 3.4).

- Acquire() : Entre la SC, obtient l'état à jour des variables protégées

- Release() : Quitte la SC, diffuse les changements des variables protégées pour les autres

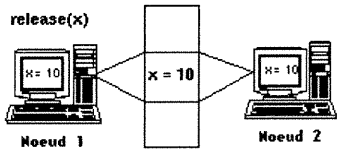

FIG. 3.4 – Cohérence release

processus.

Prenons comme exemple une implémentation d'un serveur central avec une cohérence Release dont le processus p1 acquiert la section critique :

"P1 Acquire" envoie un message pour le serveur de cohérence demandant le verrou L

Étape 1 : Reçoit L après les mises à jour d'un P quelconque

Étape 2 : P1 fait des opérations arbitraires sur les variables protégées par L

Étape 3 : P1 Release

Étape 4 : Envoie des mises à jour des données pour toutes les P qui veulent acquérir le verrou.

Lazy Release

Le modèle de cohérence "Lazy Release" [Keleh95] applique les mêmes primitives et la même technique que celles utilisées dans la cohérence release pour protéger la région partagée dans la mémoire, à l'exception de la propagation des modifications qui se fait lors du temps de "acquire" par le processus. À ce moment, le processeur qui arrive à la fonction "acquire()" décide de recevoir les modifications de la page dont il a besoin, selon la définition de la cohérence release (figure 3.5).

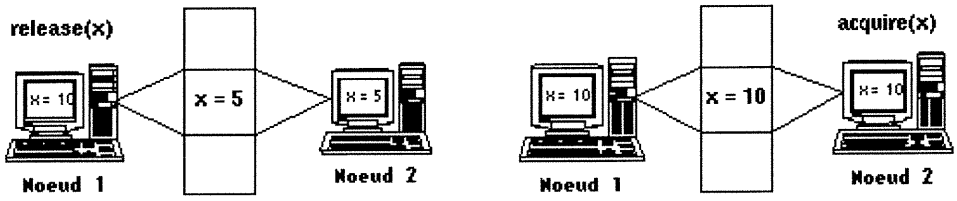

FIG. 3.5 – Cohérence lazy release

Faux partage

Jusqu'à maintenant, les techniques de gestion de mémoire que nous avons présentées ont un grain qui se limite à la page mémoire. Or, il arrive fréquemment que l'utilisateur possède des données partagées en lecture, d'autres partagées en écriture, et que toutes les deux soient stockées dans une même page mémoire. La figure 3.6, présente un cas de faux partage. Les noeuds 1 et 2 partagent une zone mémoire en lecture (partie gauche), et les noeud 1 et 3 partagent une zone en écriture (partie droite). Cependant, ces deux parties appartiennent à la même page mémoire. Le faux de partage [Bolosky89] est défini lorsque deux noeuds accèdent à des objets différents dans une même page partagée.

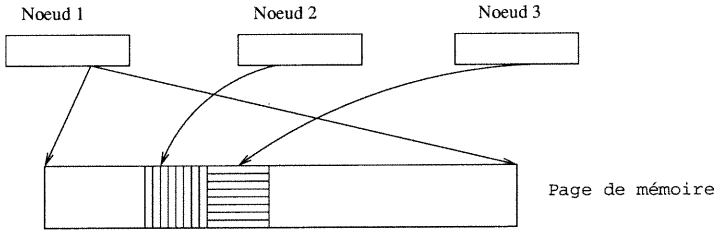

FIG. 3.6 – Faux partage d'une mémoire

3.3 Modèle transactionnel

3.3.1 Définition d'une transaction

Une transaction T est une "procédure atomique" d'un programme en exécution. Elle est composée généralement d'une suite d'opérations de lecture ou écriture qui accède et possiblement modifie des données. Le mot "atomique" signifie indivisible. En d'autres mots, une transaction s'effectue totalement ou ne s'effectue pas du tout. De plus, les étapes intermédiaires d'une transaction ne sont pas observables par une autre transaction. Pour assurer l'intégrité des données dans une base des données, il est indispensable que le système maintienne les propriétés suivantes :

Les propriétés ACID

Le mot ACID (A-C-I-D) est un acronyme dérivé de quatre mots qui constituent les propriétés de base d'une transaction dans les modèles transactionnels : Atomicité, Cohérence, Isolation, et Durabilité.

- Atomicité : "tout ou rien" toutes les opérations d'une transaction doivent être traitées comme une entité élémentaire.

- Cohérence : Si elle est effectuée seule, une transaction transforme la base de données en un état cohérent à un autre état cohérent.

- Isolation : Une transaction en cours ne peut révéler ses résultats intermédiaires aux autres transactions avant sa validation.

- Durabilité : Les modifications faites par un "commit" sont toujours conservées même après une défaillance.

Application transactionnelle cible

L'application que nous voulons simuler sur un système à MPD logicielle traite en réalité les appels téléphoniques à temps réel. Les appels sont multiplexés et techniquement résultent des accès sur la base des données des clients. Toutefois, on peut considérer que les accès transactionnels sont de type lire ou écriture. Cette application a trois propriétés concernant l'abstraction des données :

1. Chaque client est représenté par un objet qui encapsule les différentes informations reliées à son compte telles que : le taux d'appels effectué, informations personnelles etc.
2. La taille de chaque objet (au fond la granularité de l'application) est limitée à 1/2 Ko de mémoire. La taille totale de la base de données peut atteindre une dizaine de millions de comptes d'usagers, ce qui est équivalent à 5 Go d'espace mémoire.
3. Le modèle d'accès est mondiac, cela veut dire qu'un appel traite juste un seul objet à la fois. De plus le taux de calcul associé à chaque accès est minuscule, et les accès se font d'une façon complètement aléatoire. Finalement, la réalisation des quatre propriétés d'ACID forme un critère essentiel dans l'exécution de l'application.

3.4 Systèmes MPD sélectionnés

3.4.1 Calypso

Calypso [Barat95] est un logiciel prototype qui soutient le calcul sur un ensemble de postes de travail parallèles dans un réseau local (en anglais, NoW pour Network of Workstations). En fait, Calypso est classé dans la catégorie de "metacomputing". C'est un système de traitement parallèle qui fournit la balance de charge et la tolérance aux pannes tout en préservant l'abstraction partagée de mémoire sur un groupe de machines. Puisque notre intérêt réside dans l'aspect de la mémoire partagée distribuée, nous présenterons dans la prochaine section certaines des propriétés de ce système ainsi que nos essais initiaux et les résultats expérimentaux sur notre "cluster" local. Ces tests ont pour but d'extraire des informations appropriées à son comportement d'exécution dans un vrai réseau sous l'effet de changement et de la disponibilité de

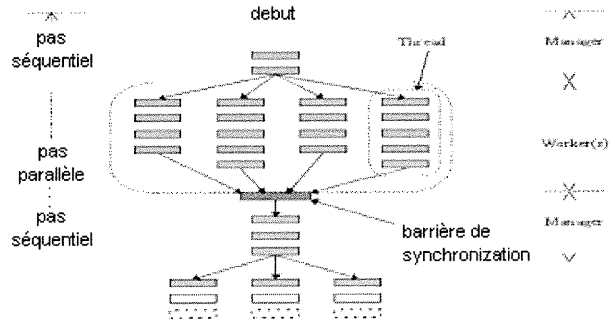

FIG. 3.7 – Modèle d'exécution parallèle en Calypso [Barat95]

ressources. Il existe deux versions de Calypso dépendant de la plate-forme du système d'exploitation utilisée : Linux et NT. Nous nous sommes particulièrement intéressés dans notre évaluation à la version sous linux. Quatre propriétés caractérisent Calypso au niveau architecturale, que nous allons décrire dans les sections qui suivent.

Algorithme d'un serveur central

Calypso met en application l'algorithme de serveur central classique largement utilisé dans l'environnement réparti, où le module "manager" est responsable de gérer toutes les tâches de scheduling et le partage de mémoire parmi les différents noeuds. Ainsi, le module "manager" est responsable de distribuer les travaux (threads) sur ces noeuds appelés "worker" (figure 3.7 tirée de l'article [Barat95])

Calypso utilise les barrières de synchronisation afin de gérer la cohérence des données. Ainsi aucune sorte de "lock" (verrou en français) distribué n'est utilisé pour gérer les accès concurrentiels. En fait, l'approche de ce système exige que tous les traitements parallèles soient distribués aux machines par les threads de calculs disjoints. Ce principe assure une cohérence séquentielle tant que le modèle de calcul respecte le principe MREW (Multiple Read Exclusive Write). Cependant le programmeur doit prendre soin des données dans la section parallèle afin de ne pas avoir de modification sur une même variable par deux threads parallèles. Cette déficience pourrait être allégée avec l'implantation d'un mécanisme qui créerait des sous-threads pour chaque

accès concurrentiel de type modifier. Par contre cette caractéristique n'est pas implantée dans la version courante de Calypso.

Séparation du parallélisme entre programmation et exécution

Le module central gère la notion du parallélisme d'une façon transparente au programmeur, ce qui veut dire que le parallélisme dans le programme est géré indépendamment du nombre de machines disponibles sur le réseau. Une machine client peut entrer dans le calcul à n'importe quel moment. Ainsi, elle peut quitter le calcul sans en affecter la nature et l'exactitude de calcul. C'est la responsabilité du système de "map" le nombre des threads de calcul au nombre des machines existantes.

Équilibrage de charge et tolérance aux pannes

Calypso distribue dynamiquement la charge dépendamment des machines qui participent aux calculs. Cependant, cette propriété n'est pas inclue dans nos tests d'évaluation. Le mécanisme mis en application pour supporter la tolérance aux pannes est efficace dans Calypso seulement du côté des clients. Il n'adopte pas un mécanisme de tolérance au niveau du serveur.

Mémoire partagée distribuée

L'architecture globale du système, figure 3.8, est basée sur un serveur central "manager", et un ensemble de clients "workers" qui se changent dynamiquement. Cela signifie qu'une machine *worker* peut impliquer ou quitter le calcul sans préavis au *manager*. Un programme sous calypso est composé d'étapes séquentielles et d'autres parallèles. Tout d'abord le *manager* exécute les étapes séquentielles du programme et assigne les tâches (threads) aux clients en gardant une table de statut concernant les statuts des différents segments assignés. Au début de l'exécution, le *manager* utilise une copie jumelle (twin) des pages partagées pour s'en servir aux demandes des *workers*.

Les *workers* savent déjà les pages qui contienent les variables dans les pages partagées. La première fois qu'un client accède une page partagée protégée, un signal de type SIGSEGV

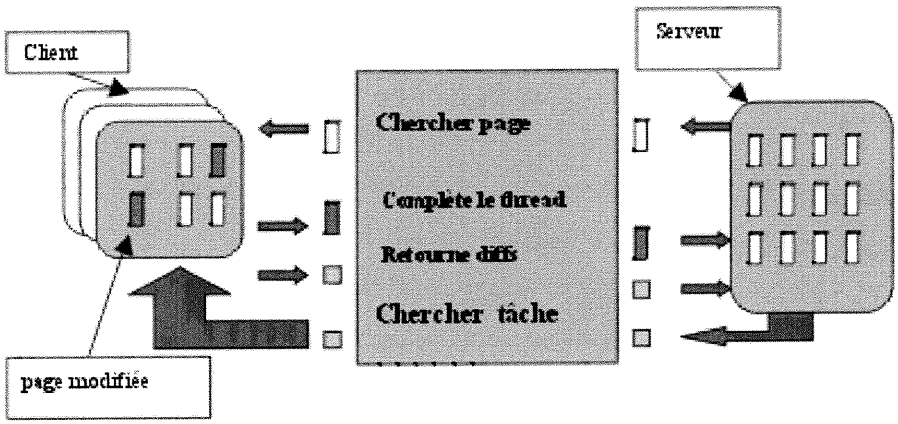

FIG. 3.8 – Mémoire partagée distribuée en Calypso

est généré. Suite à cette génération, le traiteur des signaux (signal handler) est activé pour chercher la page appropriée du serveur, l'installer dans l'espace de processus du *worker*, et enlève la protection "unprotect" de la page pour son futur usage. Toutefois, si le *worker* fait une modification à cette page, elle sera marquée "dirty". Quand il termine les tâches qui lui sont associées, le *worker* identifie toute sa mémoire modifiée "dirty" et envoie les modifications "diff" ("ou-exclusif" de la page initiale et la page modifiée) au *manager* qui à son tour accepte l'exécution des threads (jobs).

3.4.2 Charlotte

Charlotte [Barat96] constitue le deuxième système MPD que nous avons choisi dans notre étude d'évaluation. Charlotte est semblable à Calypso dans sa classification algorithmique puisqu'il met en application l'algorithme d'un serveur central pour maintenir le traitement parallèle. Cependant, les deux systèmes diffèrent de la structure architecturale et du point de vue d'exécution, parce que le principe de partage de Charlotte est basé sur l'objet comme unité du partage et non pas la page du système d'exploitation comme dans le cas de Calypso. L'idée

derrière Charlotte est de développer une MPD qui prolonge le cluster local à l'environnement web. Cela nécessite de créer un système permettant à n'importe quelle machine sur le web de participer dans un calcul continu et de tolérer l'échec imprévu d'un noeud sans affecter le comportement global du traitement. Ce système possède plusieurs bonnes caractéristiques qui valent la peine d'être mentionnées. Cependant, il comporte plusieurs déficiences que nous avons découvert durant notre phase d'évaluation. Puisque ce système est orienté vers l'Internet, le défi que celui-ci impose est récapitulé par les points suivants : Hétérogénéité, Portabilité et Sécurité. Dans la section suivante, nous allons décrire les caractéristiques de Charlotte ainsi que les problèmes associés à ce sujet.

Environnement d'exécution dynamique

Un programme parallèle sous Charlotte est écrit en java pour supporter les exigences imposées par le web. L'environnement d'exécution avec charlotte se fait d'une façon dynamique ; n'importe quelle machine peut participer au déroulement de calcul ou quitter à n'importe quel moment. Les tests effectués sur Charlotte ont démontré un fort traitement de l'échec des machines clients, sauf qu'une faute dans le serveur central peut empêcher le système de fonctionner.

Hétérogénéité

On dit qu'un système parallèle est hétérogène s'il est capable de s'exécuter en parallèle sur un ensemble de machines connectées sur le réseau et qui ont une architecture différente. La majorité des MPD logicielles ne prend pas ce facteur en considération car l'application parallèle s'exécute dans un réseau local dont les machines sont totalement homogènes (même architecture). Par la suite, il est convenable dans de tels systèmes de considérer la page d'une mémoire (4Ko pour x86) comme étant l'unité de partage entre les différents sites. Ce choix n'est pas le cas dans un cluster hétérogène dont chaque machine peut avoir une taille de page différente. En conséquence l'idée de créer une adresse virtuelle globale n'est pas valable. Contrairement aux MPD basées sur la pagination pour créer une adresse virtuelle partagée, Charlotte implante ce qu'on appelle l'adresse de partage d'objets de types. Par conséquent, l'unité du partage n'est plus la page mais plutôt un objet de la variable partagée (explication dans la section postérieure). Cette propriété

s'assure que n'importe quelle machine avec un browser d'applet peut être impliquée dans le calcul sans se soucier de son architecture (Pentium, Sparc..). En plus, cette propriété élimine le besoin de l'intervention du système d'exploitation pour générer des signaux (SIGSEGV) lors d'une faute de page.

Modèle de traitement

Charlotte met en application les mêmes techniques pour le balancement de charge et la tolérance de fautes appliquées dans Calypso, figure 3.7.

Un programme sous Charlotte est normalement composé par des étapes séquentielles et parallèles. Le *manager* est le module qui exécute les étapes séquentielles, alors que les étapes parallèles sont distribuées sur les différentes machines. Chaque thread de calcul est effectué par un *worker* de façon séquentielle et les modifications de ces calculs ne seront pas vues par les autres threads. Autrement dit, Charlotte comme Calypso n'assurent pas un accès concurrentiel pour les variables dans un thread et ne permettent pas la création des sous tâches dans la même thread pour gérer les accès parallèles. Cette façon de gérer les calculs assure une cohérence séquentielle à condition que la distribution des tâches soit précise par le programmeur et que chaque tâche distribuée respecte le principe MREW (Multiple Read Exclusive Write) afin de définir le type d'accès concurrentiel.

Technique de MPD dans Charlotte

La technique de MPD est basée sur l'abstraction des objets partagés [Castro96]. Pour chaque type primitif de Java, comme int, il y a une classe correspondante **Dint** (int distribué) de Charlotte qui encapsule la sémantique distribuée (Figure 3.9). La version en cours que nous étudions implante la mémoire partagée juste pour les types de base "int". Toutefois, mettre en application la mémoire partagée pour d'autres types de données tels que **Dfloat** n'est pas implanté. Chaque classe **Dint** définit un champ de valeur et un état de l'objet partagé : *non_valide*, *readable* ou *dirty*.

Un état de *non_valide* indique que l'objet ne contient pas la valeur significative; *readable*

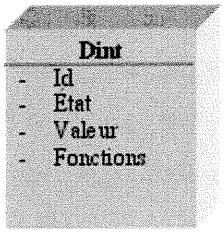

FIG. 3.9 – Un objet **Dint** qui correspond au type primitif **int**

indique que l'objet contient une valeur significative qui peut être employée dans une opération de "lecture" ; et un *dirty* indique que la valeur locale est significative et qu'elle a été modifiée. Puisque l'unité du partage est l'objet, l'accès à son état se fait par l'invocation de la méthode du statut de cette classe. Suite à un accès de lecture par un *worker*, un objet de donnée est déclaré invalide, le *worker* envoie une demande au *manager* et ensuite il le déclare valide lors de la réception. Il faut noter que tous les objets sont dans un état *non_valide* au début du pas parallèle. Pendant un accès d'écriture, l'objet est marqué dirty par le *worker*. À la fin du pas parallèle tous les objets dirty sont retournés au *manager*.

Puisque Java ne soutient pas le surchargement d'opérateur, Charlotte oblige le programmeur d'utiliser des fonctions membre get() et set() pour exprimer les opérations de lire et écrire des variables partagées. Par exemple, la suite d'instructions est utilisée pour lire et écrire une variable partagée i.

```
Dint i;
int sum = 0;
i.set(4);
sum = i.get();
```

Évidemment ce style de programmation va induire une forte dégradation de performance lorsque le nombre d'accès sur une variable partagée augmentera. De plus, le partage de cette variable conduit à un certain coût élevé lors d'échange entre les noeuds. De l'autre côté, la technique de parallel step reste une problématique puisqu'un accès de mise à jour sur une variable partagée dans deux threads dans le même pas parallèle, induit une valeur incohérente de cette variable.

3.4.3 TreadMarks

TreadMarks [Keleh94] est un système de Mémoire Partagée Distribuée logicielle, qui soutient le calcul parallèle sur un réseau de poste de travail en fournissant à l'application une abstraction de mémoire globale partagée. TreadMarks est classé comme étant une MPD basée sur la page de mémoire. En plus, il utilise l'approche des sections critiques pour protéger les variables partagées, en implantant le modèle de cohérence Lazy release [Keleh95] pour assurer la cohérence de la mémoire partagée. TreadMarks est une librairie de fonctions qui fournit les services de gestion d'une mémoire partagée en mode utilisateur. L'avantage d'une telle approche est sa portabilité sur l'ensemble des machines supportant le système d'exploitation Unix. Si la librairie consomme une partie de l'espace d'adressage du processus utilisateur, elle élimine les changements de contextes qui pourraient avoir lieu avec un démon (deamon) en mode utilisateur. Notre évaluation est basée sur nos essais et les investigations théoriques dans les articles qui décrivent ce système. TreadMarks [Keleh94] diffère des autres MPD que nous avons examinées dans plusieurs points cruciaux. Nous allons montrer les différentes caractéristiques dans TreadMarks

MPD basée sur la pagination

TreadMarks est un système à mémoire partagée distribuée, mis en application entièrement au niveau de l'utilisateur (user level) comme une bibliothèque "C" au dessus de Linux. Du point du vue conceptuel ce système utilise la page de mémoire comme une unité de partage entre les processus. Aucun changement n'est demandé au niveau du noyau pour faire rouler TreadMarks sur une grappe.

Cohérence des données

C'est le seul système MPD que nous avons examiné qui soutient la duplication des données sur les différents processeurs. Contrairement aux Calypso et Charlotte qui utilisent les serveurs centralisés pour gérer la mémoire partagée, TreadMarks réplique les données initiales de programme sur tous les noeuds en utilisant la fonction spéciale Tmk_distribute(). La cohérence relâchée de

type eager release consistency, implantée dans Munin [Carter91], a pu être améliorée dans TreadMarks en utilisant lazy release concistency [Keleh95] qui a permis de réduire le nombre de messages nécessaires pour maintenir la mémoire distribuée cohérente. TreadMarks adopte un mécanisme pour corriger l'anomalie du faux partage rencontré dans les systèmes MPD basés sur la pagination.

Synchronisation

Les primitives de synchronisation utilisées par TreadMarks sont les verrous et les barrières. Les "locks" sont utilisés pour maintenir une cohérence des variables protégées dans la Section Critique, tandis que les barrières sont utilisés comme un point de synchronisation pour tous les processus. Toutes les autres communications sont faites par les opérations de synchronisation. Tmk_lock_acquire(), Tmk_lock_release() sont les fonctions en TreadMarks, qui simulent les primitives acquire et release mentionnées dans la cohérence LRC pour protéger la section critique qui contient les variables partagées.

3.5 Processus d'évaluation et tests

3.5.1 Benchmark

Deux groupes de programmes sont exécutés pour tester les MPD choisies. Le premier groupe inclut des programmes à calculs intensifs, et le second comporte une simulation d'une application transactionnelle distribuée. Les benchmarks à calculs intensifs sont :

1. Multiplication de Matrice : ce programme calcule la multiplication de deux matrices.
2. Quadrature Adaptative : ce programme calcule l'aire intercepté par une fonction F(x) dans un intervalle donné [a,b]. La methode adaptative consiste à sub-diviser l'intervalle [a,b] en n sous intervalles pour ensuite calculer l'aire sur chaque sous-intervalle $[x_i, x_{i+1}], i = 0, ...n-1$, par la méthode de trapèze : $((F(x_i) + F(x_{i+1}))/2 * (x_{i+1} - x_i))$
3. Jacobi : est un programme qui applique la méthode de Jacobi pour résoudre une equation différentielle elliptique en partant d'une valeur initiale donnée.

La matrice A calculée par la méthode de Jacobi à partir d'une autre matrice B pour une seule itération, est donnée par le formule :

$a_{ij} = (b_{i-1,j} + b_{i+1,j} + b_{i,j-1} + b_{i,j+1})/4.$

4. Successive Over Relaxation (SOR) : est un programme qui utilise la methode "Red-Black Successive Over-Relaxation" pour résoudre le problème d'equation différentielle partielle.
5. Mandelbrot : est un programme qui nécessite un calcul intensif pour générer une image graphique.

Le deuxième groupe de benchmarks consiste à simuler une application transactionnelle pour une base de données partagée avec des accès concurrentiels. Les types d'accès pour chaque test sont décrits dans la prochaine section.

3.5.2 Environnement de test

Nous avons testé Calypso et charlotte sur un cluster (Ericsson) constitué de 11 machines sans disque 233Mhz avec 64 Mo RAM. Les machines sont connectées par des cartes Ethernet de 100 Mb/s. La version de TreadMarks est une version d'évaluation, l'exécution est limitée sur huit machines.

3.5.3 Résultats et discussions

Calypso

Nous constatons que les graphiques montrent une bonne performance pour le programme de la multiplication de matrice (MM), (figure 3.10) tandis que pour le deuxième programme, les résultats sont pauvres au niveau de l'accélération (en anglais, speedup) (figure 3.11). La raison est que le programme MM est un programme hautement parallélisable, Calypso distribue les tâches d'une façon équivalente sur toutes les machines alors une bonne performance est obtenue (accélération = 7.5 pour onze machines). Dans la quadrature adaptative l'accélération obtenue est de 1.8 pour onze machines. Ceci peut être expliqué par le fait que, le taux de calcul associé à chaque tâche parallèle n'est pas le même pour tous les clients. Ainsi ce taux de calcul est considérablement petit comparativement à la communication induit avec le serveur central.

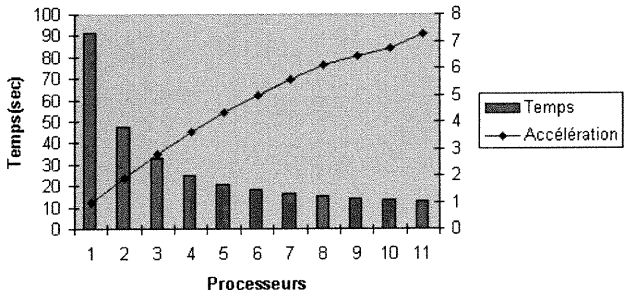

FIG. 3.10 – Multiplication de matrice 1024x1024 (Calypso)

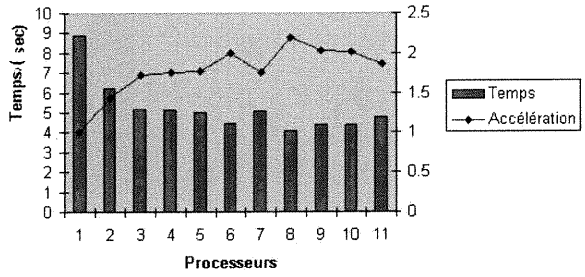

FIG. 3.11 – Quadrature Adaptative (Calypso)

Dans ce test nous visons à simuler l'exécution transactionnelle sur un cluster à MPD. Pour bien simuler l'application transactionnelle sous Calypso, il faut allouer une base de données partagée distribuée dont chaque enregistrement est de taille 1/2 Ko et simuler des accès concurrentiels sur cette base de données. La taille maximale de la BD distribuée allouée en Calypso est de 40000 enregistrements, ce qui vaut à 20 Mo de mémoire partagée distribuée. Les deux graphiques (figure 3.12 et figure 3.13) montrent les résultats obtenus de la simulation sous Calypso. Or, dans le premier graphique Calypso montre un gain de performance. Ceci est expliqué de la même manière que la MM. Puisque le taux de calcul associé pour chaque étape parallèle est constant pour tout les tâches parallèles (task). De plus, la performance plafonne à cause de l'amplification des accès concurrents ce qui produit plus de communication entre les clients et le serveur central. Ce qui fait que la performance est affectée par la latence du réseau. Dans le

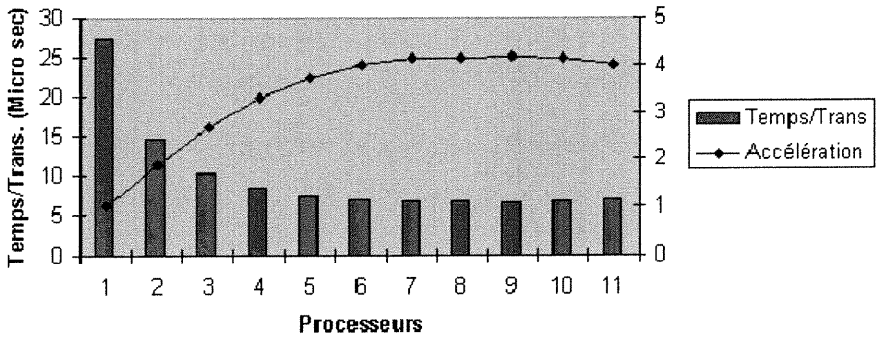

FIG. 3.12 – 4 million accès concurrents de type lecture avec le même taux de calcul sur une BD partagée de 40000 enregistrements (taille d'enreg. = 512 octet) (Calypso)

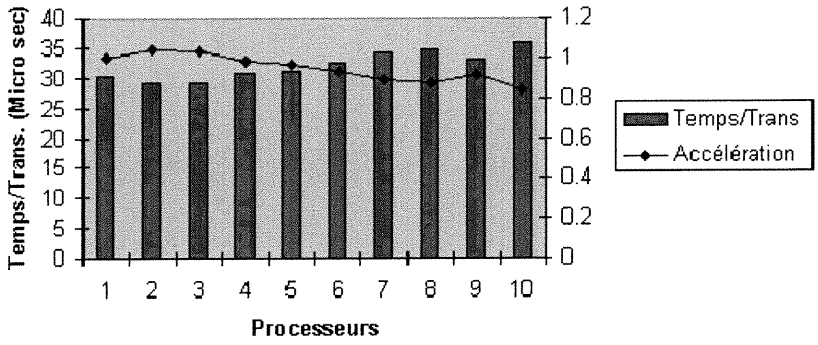

FIG. 3.13 – 4 million accès lecture/écriture sur une BD partagée de 40000 enregistrements (taille d'enreg. = 512 octet) (Calypso)

deuxième graphique, nous avons simulé des accès irréguliers sur la BD de type lecture et mis à jour. Les résultats montre que pour Calypso, l'accès à la base des données partagées n'ajoute pas de parallélisme sur l'exécution si l'accès ne contient pas un certain taux de calcul à effectuer. Ceci est nécessaire pour justifier le taux de communication entre les machines avec le serveur central.

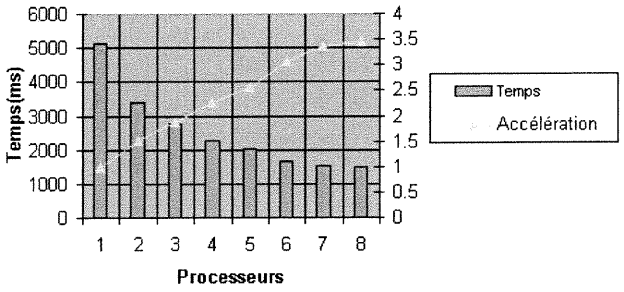

FIG. 3.14 – Multiplication de matrice 100x100 (Charlotte)

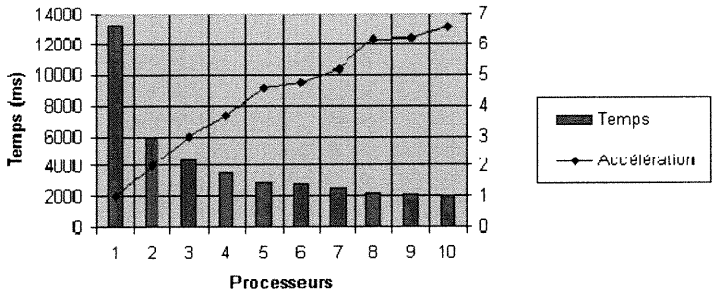

FIG. 3.15 – Mandelbrot (Charlotte)

Charlotte

Deux programmes parallèles sont testés avec Charlotte, la Multiplication de Matrice (MM) et le programme Mandelbrot. Les résultats montrent un "speedup" modeste pour ces deux applications mais un temps d'exécution très élevé. Cette faiblesse est causée principalement par deux facteurs :

1. Une latence associée à l'utilisation de JAVA comme langage d'implantation. Bien entendu, ce choix est justifié par l'utilisation de calcul parallèle sur le web. Mais de l'autre côté, il induit un coût très élevé pour l'accès aux variables partagées, puisque cet accès est fait à partir des fonctions membres telles que "get()" et "set()".

2. L'abstraction de partage par objet, comme implantée, est coûteuse car la taille de l'objet

partagé distribué est beaucoup plus grande que la taille d'un type de base "int" ou "float".

Un autre inconvénient de cette abstraction est que la taille maximale des données qui peuvent être allouées, est limitée à 200x200 **Dint**. Ces deux caractéristiques nous ont poussé de ne pas inclure Charlotte pour des tests transactionnels à cause de ses limitations du côté latence et l'allocation des données.

TreadMarks

La première phase de tests sur TreadMarks comportent le test des trois programmes MM, Jacobi et le SOR.

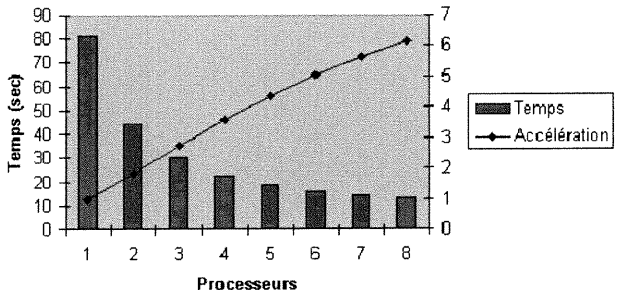

FIG. 3.16 – Multiplication de matrice 1024x1024 (TreadMarks)

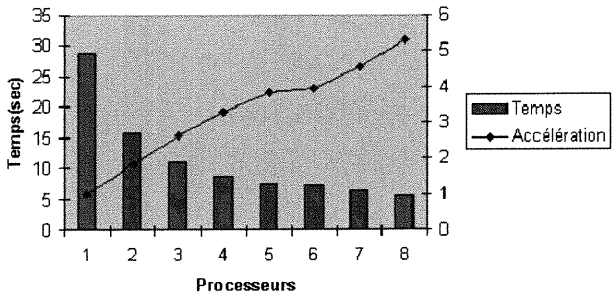

FIG. 3.17 – Jacobi 1024x1024 avec 200 itérations (TreadMarks)

Les trois benchmarks utilisés sont des applications numériques qui nécessitent un calcul

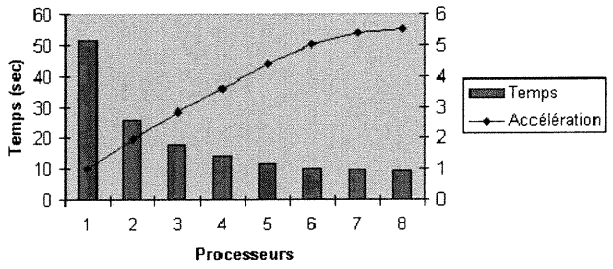

FIG. 3.18 – Successive Over Relaxation (SOR), 1500x1000 avec 300 itérations (TreadMarks)

intensif basé sur le calcul matriciel. Pour les trois programmes, TreadMarks a montré une très bonne performance ceci est dû principalement à la bonne façon de duplication des données ainsi qu'à l'implantation efficace de la cohérence "Lazy Release". Avec ces résultats nous pouvons conclure que TreadMarks atteint la meilleur performance pour ce type d'application de problème parmi les systèmes testés.

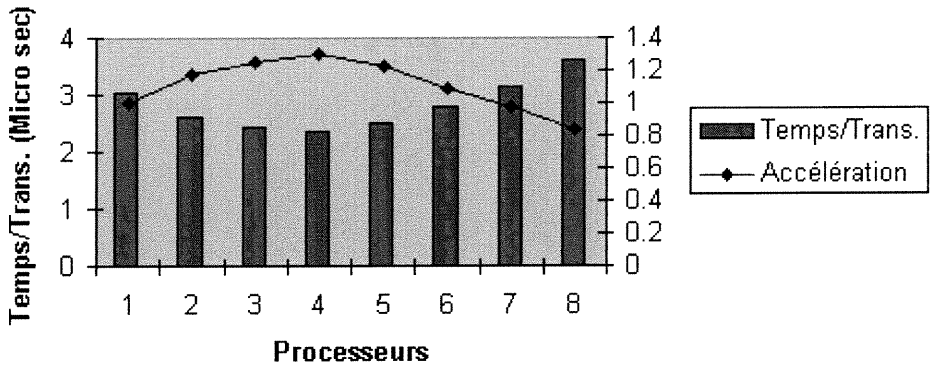

FIG. 3.19 – 4 million accès lecture/écriture sur une BD partagée de 100000 enregistrements (taille d'enreg. = 512 octet) (TreadMarks)

Après l'évaluation des benchmarks numériques, nous avons répété la même procédure que celle en Calypso, pour la simulation transactionnelle. Le but c'est d'allouer une base de données partagée et distribuée. En TreadMarks nous avons réussi à allouer jusqu'à 100,000 enregistrements de taille 1/2 Ko chaque (50 Mo en total). Notre application vise à simuler des accès

concurrentiels. Les accès sont générés d'une façon complètement aléatoire et irrégulière. Chaque accès est simplement de type lecture ou mis à jour sur un enregistrement. Le graphique montre un certain gain de performance pour trois machines mais plus le nombre de noeuds s'augmente plus il y a une dégradation de performance. Notre raisonnement concernant ces résultats est que l'application simulée est fin grain, chaque accès concurrent va induire une certaine communication pour assurer la synchronisation entre les différents processus comme décrit dans le modèle "Lazy Release". Dans les applications à "coarse grain" ce taux de communication est amorti par le taux de calculs associé à chaque accès concurrent, c'est le cas de MM et Jacobi, tandis que dans cette application le cas est inversé. Cette application demande beaucoup d'accès concurrents, et beaucoup d'échange des "locks" pour un taux minuscule de calcul pour chaque accès ce qui résulte une dégradation de performance considérable.

3.6 Récapitulation

Les trois systèmes évalués ont révélé les différentes propriétés des systèmes à mémoire partagée distribuée. Le grand défi dans ces systèmes est le traitement parallèle des applications fin-grain où la latence du réseau devient un facteur dominant. Cependant des systèmes tels que TreadMarks sont des solutions pertinentes pour améliorer la performance du traitement parallèle pour des applications à calcul intensif où le taux de communication est justifié par un taux de calcul considérable. Dans le prochain chapitre nous proposons un système de traitement parallèle performant qui sera notre solution pour les applications transactionnelles distribuées.

Chapitre 4

Protocole distribué destiné aux applications transactionnelles

Le modèle d'interaction client-serveur est le paradigme de communication le plus repandu pour les applications coopérantes sur les réseaux. Avec l'évolution des logiciels et le développement d'applications temps-réel, ce modèle montre une faiblesse au niveau de la vitesse de traitement. Il suffit de penser à un serveur web qui répond à des milliers de requêtes par seconde ou bien à un serveur destiné à la comptabilité des appels téléphoniques. Dans ce chapitre, nous expliquons notre système qui consiste en un serveur distribué dans lequel nous implantons un nouveau protocole de communication offrant un haut degré de performance pour des applications distribuées.

4.1 Objectif

Les résultats que nous avons obtenus dans la première partie montrent que les systèmes à mémoire partagée distribuée (MPD) ne constituent pas une solution adéquate pour les applications transactionnelles. Ceci est dû à plusieurs facteurs que nous avons analysés dans le chapitre précédent. La majorité des systèmes MPD sont fondés sur le concept de migration des pages en

demande, vers les machines qui en font la requête. Ce concept amène une dégradation de la performance pour le système, particulièrement si les requêtes sont distribuées de façon irrégulière. C'est normalement le cas pour bien des applications transactionnelles et surtout celle que nous visons. Un autre facteur, qui rend l'utilisation de tels systèmes coûteux, est la nécessité d'avoir une gestion de la concurrence sur la base des données ce qui exige l'utilisation et la gestion de sémaphores distribués. Dans ce chapitre nous expliquons notre système qui rectifie plusieurs de ces inconvénients. Nous avons élaboré une nouvelle architecture de communication basée sur la communication directe par Ethernet (en anglais, LEC pour Lightweight Ethernet Communication). En effet, les réseaux locaux offrent un potentiel important pour la communication, mais les protocoles ajoutent un coût additionnel qui se traduit par une latence élevée au niveau de la communication. Les protocoles conventionnels comme TCP et UDP ne sont pas toujours appropriés, du côté performance à cause de l'intervention du système d'exploitation quant à l'accès au réseau. Notre architecture est fondée sur la communication directe à partir de la carte réseau sans aucune intervention des couches réseaux intermédiaires comme c'est le cas pour TCP et UDP. Dans ce chapitre, nous proposons une solution pour les applications transactionnelles distribuées qui tire profit de cette architecture. Les résultats obtenus montrent un haut degré de performance et de parallélisme.

Malgré que l'architecture de notre système soit orientée particulièrement vers les applications transactionnelles distribuées, la communication rapide qu'adopte ce système lui permet d'être une solution pertinente pour d'autres applications qui nécessitent un accès en temps réel telles que les serveurs web et serveurs vidéo.

4.2 Pourquoi un système distribué ?

Une question peut se poser : Pourquoi concevoir un protocole tandis qu'on peut dédier une seule machine serveur pour répondre aux demandes des clients ? Les systèmes classiques centralisés qui utilisent le modèle client-serveur sont conçus généralement pour répondre à des besoins limités tels que le partage des données et des ressources sur un réseau local. Dans cette architecture les données sont centralisées dans le serveur. C'est donc à lui de calculer et de répondre aux requêtes. Cette architecture peut être valable pour certaines applications où le

nombre de clients est modéré. Par contre, si on considère le cas de l'application téléphonique en temps réel d'Ericsson qui est à l'origine de ce travail, cette architecture devient sûrement insuffisante car le taux d'accès peut atteindre plusieurs milliers de requêtes par seconde. De plus la taille de la base de données est dans l'ordre de 10 Go ce qui dépasse largement la capacité de mémoire RAM d'une seule machine (qui est limitée à 4 Go sur les machines 32 bits). En effet, le chargement de la base de données dans la mémoire rapide est essentiel pour les applications en temps réel et particulièrement celle d'Ericsson, afin d'assurer un traitement rapide des accès.

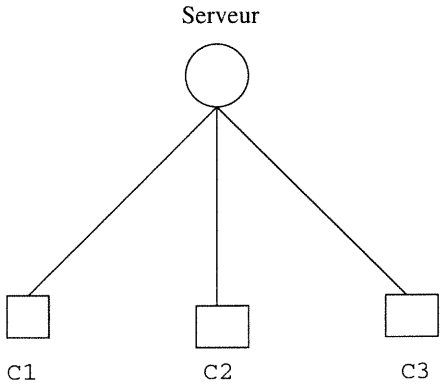

FIG. 4.1 – Architecture client-serveur

Un système distribué permet de régler les problèmes mentionnés ci-dessus. Concernant la taille des données dans un modèle distribué, les données peuvent être partitionnées, répliquées ou migrées parmi différentes machines ainsi davantage de données peuvent être stockées en mémoire rapide. De plus, le parallélisme permet de distribuer la charge sur plusieurs machines, éliminant ainsi le problème de goulot causé par une machine serveur centrale unique. Finalement un modèle distribué est plus apte à tolérer les pannes.

La distribution et le parallélisme sont deux caractéristiques fondamentales d'un système distribué et particulièrement dans notre système, car elles assurent plusieurs avantages sur les systèmes centralisés. Dans ce qui suit, nous expliquons comment notre protocole met ces notions en pratique.

4.3 Application transactionnelle

L'application que nous avons introduite dans le chapitre précédent et qui sera le sujet à traiter dans ce chapitre suppose l'existence d'une base de données de grande taille qui représente les informations des clients. Du point de vue conceptuel cette application exige un taux d'accès élevé sur la base de données, dont chaque accès traite uniquement un objet à la fois. D'une façon concrète, nous comptons réaliser cette application via un système distribué où chaque accès résultant d'un appel téléphonique est représenté par une requête de lecture ou de mise à jour. Le but est de déterminer le taux d'accès maximal auquel le système peut répondre (le nombre de transactions traitées par seconde) tout en préservant les propriétés d'ACID décrites dans le chapitre précédent. Pour atteindre ce but, il est important de définir la stratégie de distribution des données sur plusieurs sites avant de spécifier la façon de traiter des appels.

4.4 La stratégie de distribution et du parallélisme

L'idée de la duplication et de la migration des données, largement utilisée dans les MPD, peut être efficace dans certains cas où le taux d'échange d'une unité de données entre les sites est négligeable comparativement aux taux de calcul associé à cette unité. Ceci n'est pas le cas des applications transactionnelles dont l'accès aux données est irrégulier et le taux de calcul associé à chaque requête est minuscule. L'idée principale de notre système est qu'au lieu d'échanger une page de données lors de la réception d'une requête, il est plus avantageux que la requête en provenance de l'extérieur soit traitée directement par la machine contenant les données en question. Cette hypothèse implique que les données soient statiquement partitionnées sur plusieurs machines serveurs. Ainsi, chaque serveur sera responsable de gérer ses données et de répondre aux requêtes des clients qui lui sont destinées. Selon cette stratégie, les données sont statiquement allouées et disjointes, sans avoir le besoin de migration entre les sites. La figure 4.2 montre la distribution des données.

Deux avantages sont obtenus lors de l'implantation de cette stratégie de distribution :

1. Une plus grande taille de base des données est possible.
2. La gestion de la concurrence n'est pas un problème car les serveurs traitent les requêtes

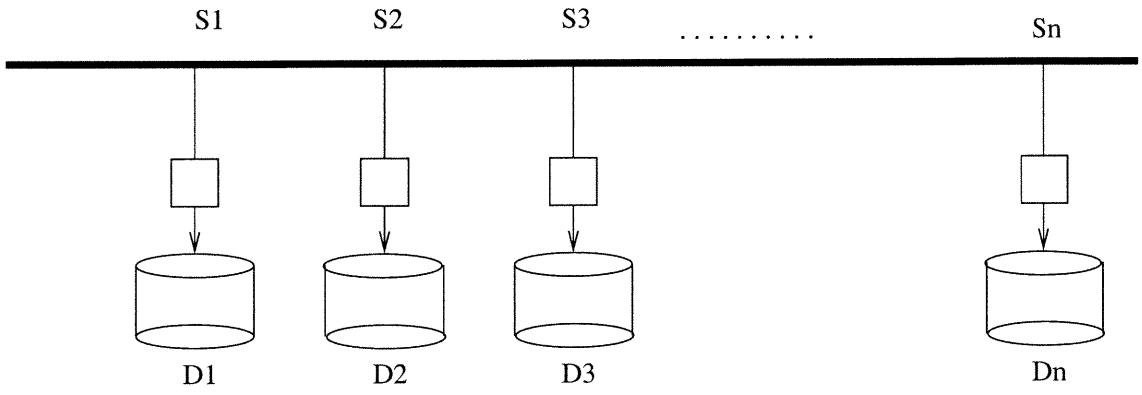

FIG. 4.2 – Partitionnement statique des données

d'une façon séquentielle sur leurs propres données.

Ces deux raisons rendent notre modèle attrayant. Mais cette distribution doit être accompagnée d'un mécanisme gardant un haut degré de parallélisme, qui ensuite permettra une meilleure performance. Notre système consiste en plusieurs machines serveurs formant ensemble un seul serveur logique. Si les machines serveurs balancent la charge nous pourrons atteindre une performance plus élevée qu'avec une seule machine serveur. Bien entendu, la distribution de la charge est fortement reliée à la distribution des données. Pour une application téléphonique réelle comme celle que nous étudions, certains clients peuvent accéder plus fréquemment leurs comptes que la moyenne. Pour assurer une distribution de charge acceptable sur toutes les machines, les données devraient être arrangées d'une façon adéquate en évitant le regroupement de toute les objets fortement accessibles sur une même machine. Avec telle méthode d'arrangement, nous pourrons implanter la distribution par l'utilisation de la fonction de hachage pour faire le partitionnement des données. Du point de vue conceptuel, un enregistrement dans la BD peut être de type différent dépendamment de la nature de l'application. Il peut contenir des informations concernant le compte d'un client pour les applications de compatibilité comme il peut représenter un URL pour un serveur web. Or, afin de simplifier l'accès aux données distribuées, il est indispensable que les objets de la base de données soient numérotés sans se soucier de leur type et que les requêtes accédant à ces objets soient étiquetées par des numéros de série identifiant l'enregistrement demandé. Si on suppose une telle numérotation des enregistrements, une fonction de hachage possible est simplement la fonction modulo. Chaque serveur S_s est responsable de la gestion des enregistrements de numéro "i" tel que s = (i mod N)+1.

Ainsi une requête qui vise un enregistrement "i" doit être traitée par le serveur "(i mod N)+1". Mais comment une requête émise par un client peut viser la bonne machine serveur pour son traitement ? Cette question définit le problème de localité d'accès aux données réparties.

4.5 Approche existante

Cette approche consiste à assigner une machine routeur, "frontal" [Pai98], [Aron00] et à lui attribuer la responsabilité d'orienter les requêtes en provenance des clients à la bonne machine serveur qui contient les données en question. Dans le contexte transactionnel, on peut imaginer qu'à chaque réception d'une requête le routeur vérifie l'étiquette représentant le numéro de l'objet à traiter, ainsi il applique une simple fonction de hachage pour retourner le numéro du serveur correspondant. Cette fonction est de la forme : H(i) = (i mod N) +1 ; où i est le numéro de l'enregistrement et N le nombre total des machines serveurs. La figure 4.3 illustre cette approche :

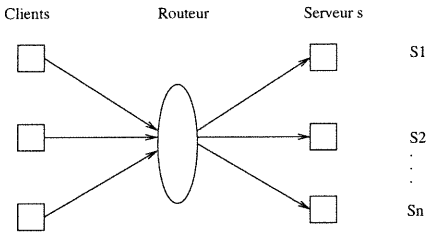

FIG. 4.3 – Routage des paquets par un routeur centralisé

Malgré que ce soit une solution pouvant résoudre le problème initial, elle souffre de deux inconvénients importants :

1. La centralisation du routeur peut facilement devenir un goulot si le nombre de clients impliqués dans les calculs accroît. Cette situation est très probable dans les applications téléphoniques où le nombre de clients est de l'ordre de quelques millions d'usagers. Aux heures de pointe, ceci affectera la performance du système au complet.
2. Chaque requête nécessite au moins trois messages de communication pour faire son aller-retour : Un message du client vers le routeur, un deuxième du routeur vers le serveur

correspondant et la réponse du serveur vers le client source.

Ce modèle d'interaction risque de produire une dégradation de performance dans le cas où un grand nombre de requêtes est atteint, comme dans le cas d'un serveur web pour un site populaire. Dans ce cas un accroissement dans le nombre d'accès simultanés à ce site pourrait induire une chute de performance au niveau de serveur. En conclusion cette approche n'offre pas les buts que nous visons pour un système distribué performant. Même si ce système résout partiellement le problème de distribution, il est affecté par la centralisation que nous essayons d'éviter.

4.6 Notre approche

Le modèle que nous proposons dans cette approche tire profit de la stratégie présentée précédemment. L'idée principale est basée sur la diffusion et non sur la centralisation (figure 4.4)

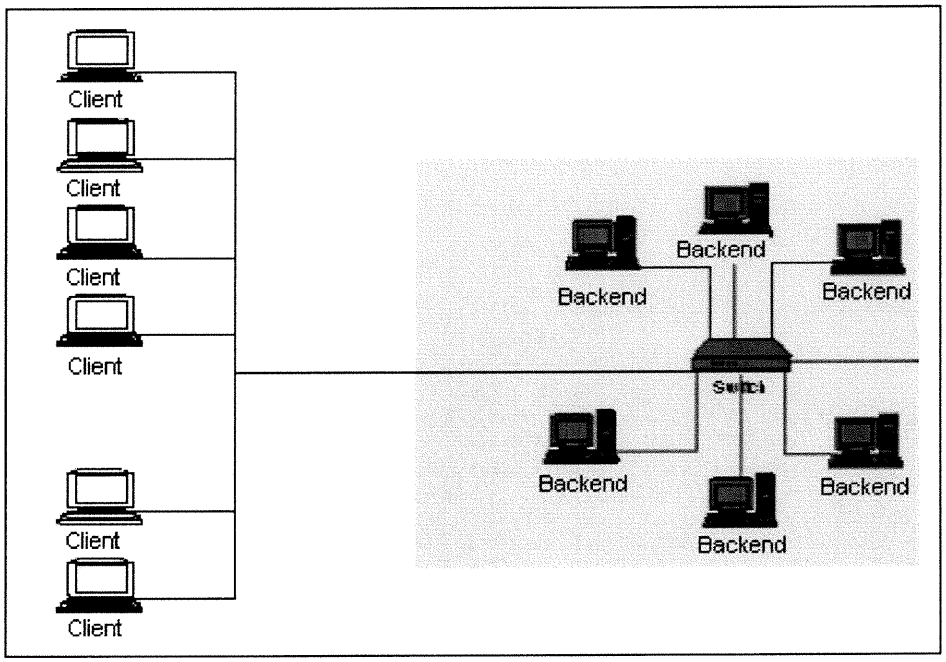

FIG. 4.4 – Architecture distribuée non-centralisée

Dans cette architecture nous disposons d'un groupe de serveurs (Backend) qui contiennent les données et qui répondent aux requêtes arrivées en provenance des machines clients. Cette architecture ne nécessite aucune machine centrale ou intermédiaire redirigeant les paquets en provenance des clients.

En fait, chaque requête est reçue par le Commutateur réseau "switch" qui la diffuse au groupe des serveurs. Cette requête est ensuite détectée par tous les serveurs appartenant au groupe de diffusion. En appliquant une simple fonction de hachage, chaque serveur vérifie si la requête le concerne ou non en fonction du numéro de l'objet à traiter. Puisque les données partitionnées statiquement sont disjointes, un seul serveur traitera la requête. Dépendant du type de la requête, le serveur concerné traite les données en question et renvoie un acquittement au client source. Cette mise en place garantira la cohérence des données sur les différents serveurs et par la suite, les trois propriétés d'ACID : atomicité, consistance, isolation, car notre système traite les applications de type Mondiac (une requête traite uniquement un objet) où les accès sont sérialisés par le réseau. Par la suite la cohérence des données est assurée sur chaque serveur. Le dernier point à mentionner est que l'architecture au niveau conceptuel est parfaite mais techniquement parlant, chaque réception d'un message conduira à une détection inutile de ce message par les autres machine serveurs. Ainsi, le coût supplémentaire du système d'exploitation (SE) emmènera une dégradation de performance. C'est pour cela qu'il est important d'élaborer une technique de communication au niveau des serveurs fondée sur la communication directe par la carte Ethernet et de tester sa performance.

4.7 La technique "Lightweight Ethernet Communication" (LEC)

Les protocoles de communication traditionnels tel que TCP/IP dépendent fortement du noyau du système d'exploitation pour implanter les primitives de communication *send* et *receive*. Cependant, le SE ajoute un coût supplémentaire considérable pour transférer l'exécution du mode usager au mode kernel à chaque réception ou transmission d'un message. L'utilisation de ce protocole n'est pas nécessairement le meilleur choix pour toutes les applications. En effet, à

l'intérieur d'un cluster, la technologie du réseau de communication est homogène même si les noeuds sont hétérogènes. De plus, l'architecture réseau des clusters est souvent simple, statique et sur de courtes distances. Il n'est donc pas nécessaire d'utiliser des protocoles de routage complexes et coûteux. Toutes ces raisons démontrent que l'utilisation de la pile TCP/IP n'est pas toujours rentable pour toutes les applications. Dans ce mémoire nous exploitons une technique "Lightweight Ethernet Communication" LEC, utilisée par le serveur distribué. Cette technique est basée sur la communication à travers la carte Ethernet sans aucune intervention ni du SE ni d'autres couches intermédiaires.

4.7.1 Etherboot

Etherboot [Etherboot] est un logiciel ayant pour objectif de créer une image ROM du système d'exploitation pour l'amorçage dans la carte Ethernet à partir du réseau. Ce logiciel est utilisé initialement pour amorcer un PC sans disque et peut servir dans une variété de situations tel que : un terminal X, un cluster de serveurs, routeurs, etc. Etherboot a gagné cette popularité à cause de l'efficacité qu'il offre sur les méthodes d'amorçage locales. Nous avons utilisé des routines internes de Etherboot pour les trois raisons suivantes :

1. Il roule comme une application indépendante (stand alone application) sans la nécessité d'avoir un système d'exploitation sur la machine hôte.
2. Il assure un accès complet à la carte Ethernet. Autrement dit, il communique directement avec la carte sans le besoin d'une couche intermédiaire.
3. Il supporte plusieurs marques de cartes Ethernet de différentes sortes.

Pour comprendre la façon de communiquer avec la carte Ethernet, il est important de savoir de quelle façon les gestionnaires de périphériques de carte "Drivers" sont conçus. En fait, il existe plusieurs sortes de carte Ethernet avec des architectures différentes. Dans la majorité des cas, elles sont programmées selon le principe I/O, c'est à dire que le contrôleur de la carte Ethernet lit les paquets reçus dans sa mémoire locale. Le gestionnaire de périphérique extrait les paquets octet par octet dans le même ordre qu'ils ont été transmis en les copiant dans la mémoire. Pour accomplir les différentes opérations de réception et de transmission, le gestionnaire de périphérique de la carte Ethernet fournit cinq routines. Les cinq routines sont les suivantes :

- Routine "probe" : cette fonction sert à déterminer si une carte Ethernet existe sur la machine. Si une carte est détectée, elle sauvegarde l'adresse I/O pour une utilisation future par d'autres fonctions. Elle cherche l'adresse MAC et la mémorise dans un tampon. Puis elle appelle la fonction "reset" pour initialiser la carte.
- Routine "reset" : cette fonction est appelée par la fonction "probe" pour initialiser l'état de la carte
- Routine "disable" : cette fonction est nécessaire pour désactiver la carte et permettre au système d'exploitation de la détecter. Si l'état de la carte est en mode actif, le système d'exploitation peut échouer lors de la détection.
- Routine "transmit" : cette fonction est utilisée pour envoyer une trame Ethernet. Cette routine est passée par pointeur dans la structure "nic". Elle ne connaît pas le type des données encapsulées dans la trame puisque ces donnés sont formées par les couches supérieures.
- Routine "poll" : cette routine vérifie si un paquet est reçu et prêt pour le traitement. S'il est prêt elle copie les données dans un tampon interne et retourne 1 sinon elle retourne 0.

Etherboot dispose de cinq méthodes qui implantent les cinq routines : eth_probe(), eth_reset(), eth_poll(), eth_transmit(), eth_disable().

Dans les sections qui suivent, nous verrons comment se servir de ces fonctions pour faire la communication directement par les cartes Ethernet.

4.7.2 Détection et transmission d'un message en "LEC"

Les réseaux de commutation par paquets offrent présentement un haut débit. Le taux de transfert des données par unité de temps est de l'ordre de 100 Mb/s pour la majorité des cartes physiques actuelles. Mais le taux d'utilisation réel dans les applications distribuées ne dépasse pas 20% de cette capacité, à moins de communiquer par des larges messages. Cette limitation est due essentiellement à l'utilisation des protocoles de communication traditionnels tel que TCP et UDP qui imposent que les accès au réseau passe par le noyau du système d'exploitation. Par ailleurs, l'utilisation de la fonction "recvfrom" en mode bloquant (figure 4.5) est la technique utilisée pour la réception d'un message dans un système UDP sous Unix. En fait, l'appel de cette fonction pour signaler la détection d'un message exige plusieurs étapes de communication entre l'application et le noyau du système d'exploitation. Sans se soucier de la manière dont

elle est implantée, l'utilisation de cette fonction implique un appel système pour transférer l'exécution au noyau du SE et puis revenir vers l'application après avoir copié les données du noyau à l'application. Pour concrétiser cette idée, le schéma suivant montre les différentes étapes à suivre pour la réception d'un message :

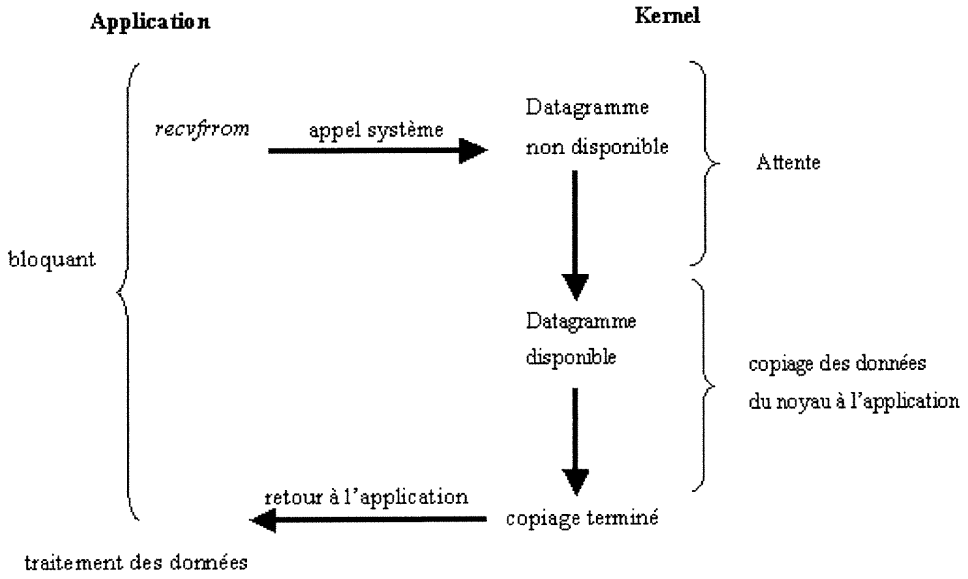

FIG. 4.5 – L'appel de recvfrom en UDP

Le coût associé par ce mécanisme est donné par la formule suivante : T = t(appel système)+ t(attente) + t(copiage) + t(retour), où t(appel système) est le coût associé à l'appel système, t(attente) est le temps pris par le paquet pour être traité et reçu par le serveur, t(copiage) est le temps requis pour copier les données du tampon de noyau à celui de l'application et finalement, le t(retour) est le temps nécessaire pour faire l'interruption du SE. Nous nous sommes intéressés à résoudre ce problème car pour la majorité des applications transactionnelles, le taux de traitement de chaque transaction est court comparativement aux taux de communication. L'idée principale du protocole LEC vise à éliminer toutes les couches intermédiaires et les interruptions entre le système d'exploitation et l'application. Le protocole utilise la carte Ethernet d'une machine pour détecter si le message reçu devrait être accepté ou rejeté (figure 4.6).

Cette stratégie de détection se fait par l'accès aux méthodes dans le gestionnaire de périphérique de la carte Ethernet. Ces méthodes sont implantées par le logiciel Etherboot qui assure l'accès

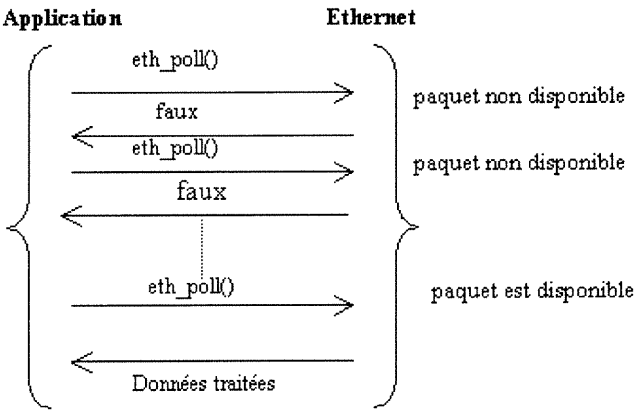

FIG. 4.6 – Mécanisme de polling dans le protocole LEC

physique à la carte Ethernet directement par les cinq fonctions de l'interface. Afin de détecter la présence d'un paquet dans la carte Ethernet, la fonction eth_poll() est appelée pour vérifier si un message est disponible. Si oui, les données sont traitées par la couche supérieure, sinon elles sont rejetées. Ce mécanisme se fait sans aucune interruption. Le coût gaspillé est celui pour vérifier si un message devait être accepté ou rejeté, mais comme ceci est fait au bas niveau de la carte Ethernet, le coût estimé est négligeable comparativement au coût associé à la latence du réseau. Dans le cas de transmission, le contenu du paquet est formé par la couche application. Les données sont encapsulées dans un datagramme de type UDP. Ensuite elles seront copiées dans la mémoire de la carte Ethernet pour la transmission en utilisant la fonction eth_transmit().

4.7.3 Comparaison avec "ping"

Afin de tester l'efficacité de cette technique de communication, nous avons effectué une expérience qui simule la communication entre un client et un serveur "écho". L'application est un programme client, qui envoie des paquets au serveur et ce dernier renvoie sans changement au client le paquet qu'il a reçu. Les paquets construits sont des datagrammes UDP de taille 64 octets. Les noeuds constituants notre "cluster" sont des machines mono-processeur avec une horloge de 1.2GHz, 256 Mo de RAM. Le système d'exploitation utilisé est MIMOSA (version 1.2). Les deux machines se communiquent directement par leur carte Ethernet qui est de type

"3Com 8257/PCI" assurant un débit de 100 Mb/s. Pour que les résultats soient réels, nous avons conduit plusieurs tests de performance en s'assurant à chaque test que le réseau ne soit pas chargé ou utilisé par d'autres applications. Voici la moyenne des résultats de nos expériences :

Nombre de paquets : **43,154 paquets**

Temps d'exécution : **1 seconde**

Temps de RTT = **23.17 micro secondes**

Pour faire la comparaison avec ICMP, nous avons démarré les deux même machines sous le SE Linux et lancé le programme ping entre le client et le serveur. Le programme ping mesure le temps pour envoyer un paquet du client au serveur et recevoir un deuxième paquet servant d'accusé de réception. Pour un paquet de type ICMP de même taille, les résultats obtenus sont :

Temps de RTT = **164 micro secondes**

Les résultats obtenus sont donc approximativement 7 fois plus rapide que lors de l'utilisation du protocole ICMP. Mentionnons que le protocole ICMP et plus particulièrement ping n'ajoute aucun coût additionnel sur l'échange des messages sauf celui ajouté par l'utilisation de la pile IP. L'importance de cette expérience de comparaison avec ping, est que la technique que nous avons adopté nous permet d'atteindre une communication plus rapide que celle dans les cas des protocoles sous IP.

4.8 Architecture globale du protocole

Après avoir présenté la stratégie de distribution et la technique de communication par Ethernet (LEC), nous procédons dans cette section à mettre toutes ces idées ensemble pour expliquer notre protocole de communication qui assure la fiabilité et la diffusion de groupe ainsi que la technique de communication directe par la carte Ethernet. Ce protocole nécessite l'existence d'un groupe de serveurs communiquant par leur carte Ethernet et d'un ensemble de clients qui envoient des requêtes à ce groupe. Pratiquement, par un client nous désignons n'importe quelle application : un browser, une application satellite, une application téléphonique, etc. Toutes ces applications sont des exemples valides qui représentent un client. Dans ce mémoire, nous nous

contenterons d'assimiler un client par une application UDP roulant sous Linux. L'architecture générale de l'implantation du protocole LEC (figure 4.7) du côté client et serveur est la suivante :

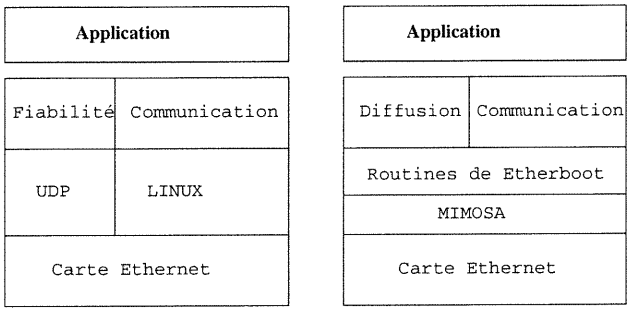

FIG. 4.7 – Architecture globale de l'implantation système

Bien entendu, un serveur désigne le groupe de serveurs traitant les requêtes des clients. Le serveur contient la couche application qui traite les données et une couche communication qui assure la transmission ainsi que la réception des datagrammes directement à partir de la carte Ethernet comme décrit dans la section précédente.

Le module de diffusion est responsable de la communication avec le routeur pour adopter la diffusion de groupe. MIMOSA est le système d'exploitation utilisé du côté serveur pour assurer la gestion des composants et les périphériques matérielles. Le côté client est constitué de trois modules : une couche application qui teste les résultats reçus, un module de communication assurant la communication avec les serveurs ainsi qu'un module de fiabilité qui assure la fiabilité des données transmises.

4.8.1 Diffusion de groupe

Notre choix de ne pas engager ni le système d'exploitation ni les protocoles traditionnels dans la communication au niveau serveur exige que nous implantions toutes les primitives de communications nécessaires. Tel qu'expliqué dans le chapitre 2, la diffusion de groupe est organisée par le protocole IGMP, qui implique une certaine communication entre le site récepteur et les routeurs. Pour que les serveurs soient capables de recevoir des paquets destinés à leur groupe, ils doivent être équipés par le mécanisme "join" basé sur la communication IGMP. Pour atteindre ce but, nous disposons d'une adresse Multicast convenable dans le tableau des adresses conte-

nant les adresses IP et MAC de toutes les machines du cluster. Par la suite, tous les serveurs appartenant à ce groupe recevront les messages destinés à cette adresse. Puisque le mécanisme de Multicast se fait matériellement il faut définir une adresse physique qui associe l'adresse IP choisi. Par exemple, si l'adresse IP de classe D choisie est 224.70.40.22, l'adresse physique associée sera $01 : 00 : 5e : 70 : 40 : 22$. Une fois les deux adresses définies, le serveur envoie une demande de type IGMP désirant joindre le groupe avec l'adresse 224.70.40.22. Le routeur cherche ce groupe en communiquant avec d'autres routeurs par le protocole DVRMP. Notre implantation demande que le serveur réponde toutes les cinq secondes à un message d'interrogation confirmant sa participation au groupe des serveurs.

Dès que le commutateur de réseau reçoit cette demande, il orientera les paquets qui lui sont destinés. Il faut mentionner que les étapes établies entre le serveur et le commutateur de réseau se font à l'insu du client. Le client ne fait qu'envoyer les requêtes sur l'adresse de diffusion de ce groupe et rien d'autre. C'est le travail du commutateur de réseau d'acheminer le paquet au groupe.

4.8.2 Fiabilité

Le concept de communication dans notre protocole est basé sur la transmission des données et la réception d'accusés de réception du serveur. En effet, les réseaux n'offrent aucun mécanisme assurant la fiabilité : les messages peuvent donc être perdus, retardés, transmis dans le désordre ou dupliqués. Alors, pour réaliser un protocole fiable, il faut résoudre ces problèmes de transmission. La fiabilité du protocole est fondée principalement sur le concept de retransmission des données. Le protocole dispose d'un temporisateur du côté client, à chaque fois que ce dernier émet une requête, il initialise un temporisateur puis attend de recevoir un accusé de réception. Si la temporisation expire avant que les données ne soient acquittées, il suppose que ce paquet a été perdu alors il le retransmet. À cette fin, il faut équiper le protocole de deux caractéristiques :

1. Temporisation et retransmission.
2. Numéro séquentiel pour les paquets envoyés, pour que le client puisse vérifier la validité de l'acquittement.

L'ajout de numéros séquentiels pour les transactions envoyées se fait simplement en incluant une étiquette dans le paquet transmis correspondant à un numéro séquentiel. Le serveur doit ensuite retourner cette valeur dans sa réponse. Ceci permet au client de vérifier s'il a reçu le bon acquittement attendu et d'ignorer toutes les autres qui ne le concernent pas. La deuxième caractéristique est implantée différemment dans notre protocole que dans les autres. La méthode classique utilisée afin de traiter les temporisations et retransmission est basée sur la transmission d'une requête et l'attente de N secondes correspondant à la valeur du temporisateur. Si aucune réponse n'est reçue, on retransmet la même requête une autre fois en attendant encore N secondes. Ce processus se répète plusieurs fois, s'il n'y a pas de réponse, on cesse d'envoyer. C'est ce qu'on appelle le "temporisation linéaire". Dans le cas qui nous intéresse, nous visons précisément un protocole fiable destiné à être utilisé sur une interconnexion IP. Cela veut dire que le trajet des paquets envoyés peut être varié. Par contre l'utilisation de la technique de la "temporisation linéaire" peut devenir problématique parce qu'on ne connaît pas a priori, la vitesse de chaque paquet pour faire son temps de boucle moyen "Round Trip Time" (RTT) sur un réseau internet. Plusieurs facteurs peuvent affecter le RTT tels que la congestion et la distance. En plus, le RTT entre un client et un serveur peut se changer rapidement si les conditions du réseau varient, d'où la nécessité de trouver un mécanisme qui utilise une temporisation et une retransmission prenant en considération les valeurs des RTT mesurées ainsi que leur changement avec le temps. Plusieurs travaux sont faits dans ce domaine, la majorité sont reliés au cas du TCP, mais l'idée principale s'applique à toute application sur le réseau. Nous avons implanté l'algorithme offert par Jacobson afin de gérer la fiabilité sur le réseau internet. L'algorithme prend en compte les variations de délai au sein de l'interconnexion en utilisant un "algorithme de transmission adaptatif" [Jacob88]. Cet algorithme consiste à calculer pour chaque transmission une valeur de temporisation différente "Retransmission TimeOut" (RTO) basée sur les anciens RTT mesurées. Pour calculer cette valeur, nous mesurons le RTT correspondant au temps écoulé par l'aller-retour du paquet. Chaque fois que le RTT actuel est calculé, on met à jour les deux autres variables srtt (estimateur du RTT) et le rttvar [Stevens98]. Le dernier est un estimateur de l'écart type du RTT. Après avoir calculé ces deux valeurs on peut estimer la valeur du prochain RTO en utilisant les formules suivantes :

$$
\begin{aligned}
\text{delta} &= \text{RTT} - \text{srtt} \\
\text{srtt} &= \text{srtt} + \tfrac{1}{8} * \text{delta} \\
\text{rttvar} &= \text{rttvar} + \tfrac{1}{4} * (|\text{delta}| - \text{rttvar}) \\
\text{RTO} &= \text{srtt} + 4 * \text{rttvar}
\end{aligned}
$$

Où delta est la différence entre la valeur du RTT mesuré pour ce paquet et la valeur de l'estimation initialisée à zéro au début. L'avantage de cet algorithme est la simplicité de calculer les variables sans avoir la nécessité d'utiliser la racine carrée. Un autre point important dans l'algorithme adaptatif mentionné par Jacobson est lorsque le temps de temporisateur expire un "backoff" exponentiel doit être utilisé. Par exemple, si le RTO expiré est de 2s et la réponse n'est pas reçue, le prochain RTO sera 2x2. Si ça se continue, le prochaine sera le double et ainsi de suite.

Malgré que les techniques utilisées dans cet algorithme résolvent les problèmes de calcul de RTO, un problème existe quant à la retransmission. Prenons un cas simple de retransmission : le client fabrique une requête et la transmet sur le réseau. La temporisation expire. Il retransmet la requête dans un autre datagramme. Comme les deux paquets transportent les mêmes données, l'émetteur n'a aucun moyen de déterminer si l'accusé correspond au datagramme initial ou retransmis. Et par la suite il ne connaîtra pas le vrai RTT correspondant au paquet envoyé, ce qui entraînera un calcul erroné du prochain RTO. Ce phénomène se nomme ambiguïté des accusés de réception "acknowledgment ambiguity". L'algorithme de Karn [Karn87] traite ce scénario avec le schéma (figure 4.8) :

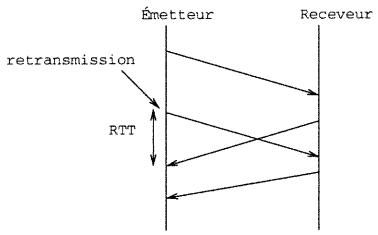

FIG. 4.8 – Algorithme de Karn

Une solution que nous adoptons est d'inclure une estampille dans chaque paquet envoyé, Ainsi chaque RTT sera calculé en utilisant la propre estampille de chaque paquet. Pour ensuite

résoudre l'ambiguïté.

4.8.3 Primitives et scénario de communication

Le scénario de communication entre le groupe des serveurs et les clients est construit fondamentalement sur le paradigme synchrone "request-reply". L'émetteur diffuse une requête sur le groupe des serveurs et le serveur convenable traite la transaction en retournant une réponse considérée par le client comme un acquittement.

Souvenons-nous que chaque requête est étiquetée par un numéro séquentiel qui l'identifie. Ce numéro (unique et différent pour chaque requête) est choisi initialement par le client au début de la connexion, est incrémenté au fur et à mesure que la communication évolue. La connexion entre les deux extrémités désigne l'accord entre le serveur et le client à échanger les données.

Cet accord est fait implicitement à la réception de la première requête. Le serveur dispose d'un tableau des processus associés aux connexions faites par les clients. Chaque entrée dans ce tableau représente une connexion en indiquant le nom de la machine avec le numéro de port qui forment ensemble une identification d'une connexion et le numéro du dernier message reçu du client. Au début de la communication le client envoie un paquet d'initialisation avec le groupe de serveur. Le serveur qui traite la requête initialise la connexion et envoie un acquittement en retour vers le client. À la fin de la connexion le client envoie une requête de type "fin" pour aviser le serveur avant d'interrompre la communication. Le serveur prendra ensuite l'initiative pour enlever la connexion de son tableau. La figure 4.9 montre le déroulement de messages durant l'exécution.

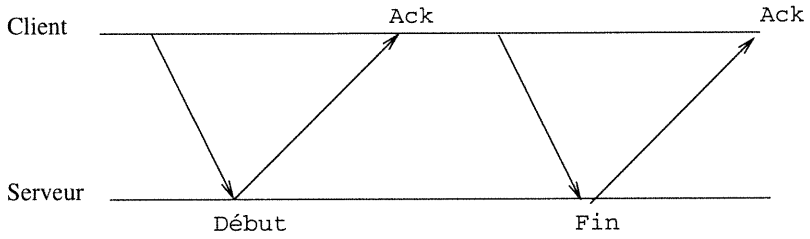

FIG. 4.9 – Modèle de communication en LEC

La raison pour laquelle nous avons choisi le mode connexion se réside dans la cohérence des

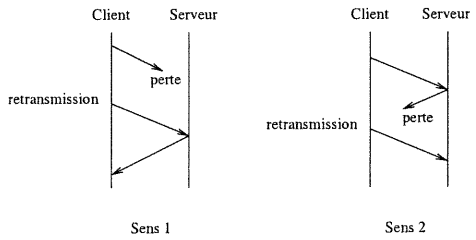

FIG. 4.10 – Le scénario de perte des paquets

données. En effet, il est très important de garder la trace des dernières requêtes exécutées par le serveur pour ne pas répéter les mêmes exécutions effectuées pour les mêmes requêtes. Cette anomalie peut parvenir dans deux cas :

1. La duplication des données.
2. Le traitement de retransmission lors de la fiabilité.

Pour le premier cas, il ne faut pas oublier que le protocole est au dessus d'une couche réseau sans connexion, donc un message peut être dupliqué en arrivant sur le serveur ce qui conduit à une double mise à jour pour la même requête. Dans le deuxième cas, lors de la retransmission il y a trois possibilités : une perte d'un message du sens client vers le serveur ou bien du sens inverse comme indiqué dans la figure 4.10.

Dans le premier sens, la retransmission ne cause pas d'incohérence dans les données mais le deuxième sens signifie que le serveur va traiter le même message deux fois. Afin de rectifier cette anomalie, nous ajoutons dans chaque entrée du tableau de connexion la dernière valeur du numéro séquentiel exécuté. Ensuite, à chaque réception d'une requête le serveur compare son étiquette avec la sienne qui correspond au numéro du dernier message exécuté. S'ils sont égaux, il envoie un acquittement sans modifier la base des données, sinon il procède à la modification et met à jour le dernier numéro séquentiel. Les primitives de communication au niveau client sont les fonctions send() et recvfrom() du protocole UDP, assurant une communication synchrone. Du côté serveur, les paquets sont construits champs par champs en encapsulant les données nécessaires pour la transmission. La fonction udp_transmit() a pour but de construire un paquet UDP ayant le même format qu'un paquet formé par le protocole UDP pour ensuite faire appel à la fonction eth_transmit() de la carte Ethernet afin de véhiculer le paquet sur le réseau. La fonction eth_transmit() de la carte Ethernet ne sait rien sur la nature des données envoyées. Il

faut donc minutieusement préparer les paquets pour qu'ils aient le même format de l'application réceptrice. De même, la routine eth_poll() de la carte Ethernet ne distingue pas le format des paquets reçus. C'est dans la couche application que le type de paquet sera détecté s'il s'agit d'un paquet UDP, ICMP, IGMP et ensuite le traitement de ce paquet prend place.

4.9 Performance

Dans cette section, nous voulons tester la performance du système en implantant le protocole que nous avons proposé. La plate-forme de test utilisée consiste en un "cluster" de 5 noeuds homogènes. Chaque noeud est une machine mono-processeur 1.2GHz, 256 Mo de mémoire RAM. Les machines sont équipées de cartes Ethernet 3Com8257/PCI de débit 100 Mb/s. Les machines serveurs utilisent le système d'exploitation MIMOSA version 1.2. Afin de mesurer la performance de notre système, nous avons simulé un benchmark pour montrer tout d'abord l'avantage obtenu par la technique LEC (Lightweight Ethernet Communication).

4.9.1 Serveur constitué d'une machine

Le premier benchmark est une application qui s'exécute sur plusieurs machines qui représentent les clients. Elle consiste à envoyer les requêtes de types UDP au serveur. Dans ce test, le serveur est une seule machine qui utilise LEC pour la communication.

La performance est définie par le nombre maximal de requêtes que le serveur peut traiter. Évidemment, on démarre les clients d'une façon successive pour voir la progression de la courbe et le point maximal atteint par le serveur. Les données affichées représentent la moyenne des résultats empiriques enregistrés. Pour extraire ces résultats nous avons fait des dizaines de tests dans un environnement contrôlé (il n'y a que notre application qui roule sur le réseau). Chaque test consiste à rouler l'application pendant 15 minutes.

Le graphique (figure 4.11) montre la capacité du serveur, en terme du nombre de requêtes traitées par seconde. Un client sous Linux sera capable d'envoyer 9443 requêtes/s. Les paquets envoyés sont de type UDP. La taille de la requête et la réponse correspondante est de 70 octet

Client	Requêtes/s
1	9443
2	18224
3	27192
4	30567
5	30492

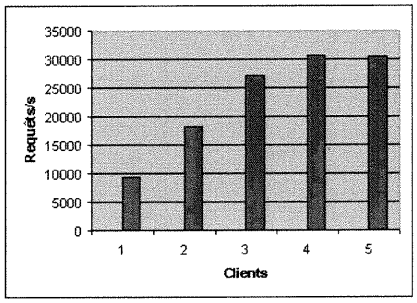

FIG. 4.11 – Simulation d'une application transactionnelle distribuée avec une machine serveur

chaque. D'un autre côté, la capacité du serveur atteinte est de 30492 requêtes/s, cette capacité inclut un certain traitement simple qui consiste à incrémenter un champs dans un enregistrement, suite à chaque requête reçue. Pendant 15 minutes d'exécution nous avons enregistré 40 paquets perdus et retransmis entre les clients et le groupe de serveur.

4.9.2 Serveur consituté de plusieurs machines

L'autre test effectué a pour but de mesurer la performance, mais cette fois avec un groupe de machines serveur, constituants le même "cluster" que nous avons décrit. Chaque requête (la même dans le premier test) est diffusée par un client sur le groupe de serveurs, mais seul un serveur va répondre à cette requête. Nous visons à mesurer la performance maximale du groupe tout entier. Nous avons suivi les même étapes décrites dans le test précédent. Nous démarrons les serveurs successivement et enregistrons la capacité maximale de chaque serveur. Ensuite, nous roulons l'application pendant 15 minutes dix fois et enregistrons les résultats régulièrement.

Serveur	Nombre de requêtes
1	30524
2	38416
3	42371
4	45088

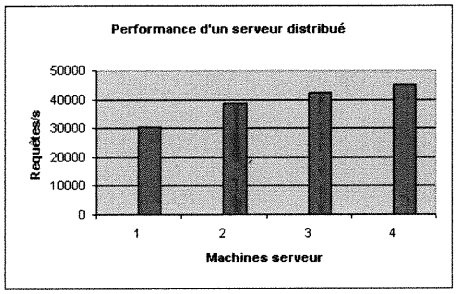

FIG. 4.12 – Performance d'un serveur distribué

La figure 4.12 montre une scalabilité obtenue au niveau du serveur. Ces résultats sont dus principalement à l'approche présentée dans ce chapitre. Le principe est fondé sur l'utilisation de la carte Ethernet directement sans l'implication du système d'exploitation, et la diffusion de groupe. Mentionnons que la scalabilité obtenue au niveau de notre serveur parallèle montre que le temps de réponse est plus élevé que dans le cas d'un serveur constitué d'une seule machine. Cependant, les comparaisons avec le programme ping ont révélé que le temps de réponse d'un serveur mono-machine utilisant notre technique de communication LEC est plus vite que la communication sous le protocole IP. Les résultats nous permettent de croire que notre serveur parallèle est performant comparativement à un serveur traitant le même genre d'application sous un protocole de communication IP. De plus, le protocole a montré un degré de parallélisme apréciable. Avec quatre serveurs nous avons pu atteindre une accélération de 1.5. Nous constatons aussi qu'à un certain point, la progression de la courbe plafonne à 45000requêtes/s ceci à cause de l'accroissement des paquets envoyés et la latence du réseau. Rappelons-nous que les réseaux Ethernet dépendent de deux facteurs : la bande passante et la latence (section 2.2.1). La bande passante d'un réseau Ethernet est limitée à 100 Mb/s. Vu ces deux contraintes notre application a échangé 50 Mb/s de données ce qui représente **50%** de la capacité totale offerte par

le réseau. Ce débit est élevé surtout si l'on considère que la présence de collision et l'opération d'acquisition du médium est élevé pour des petits paquets comme ceux que nous utilisons.

4.10 Analyse du protocole

Le protocole proposé dans ce mémoire, représente une contribution pertinente dans le domaine des systèmes répartis. Les points forts sont identifiés par la technique de communication rapide ainsi que la stratégie de distribution de données et l'utilisation de Multicast. Tous nos tests ont montré une performance très encourageante de notre serveur distribué qui applique ce protocole. Nous avons pu atteindre une capacité de traitement globale de 45000 requêtes/s. Cependant nous croyons que plusieurs autres tests peuvent être effectués pour montrer le perfectionnement de ce protocole dans plusieurs domaines réels. Citons par exemple, l'application de ce protocole à un serveur web. Les expériences avec le serveur web populaire APACHE montrent une capacité maximale de ce serveur approchant 2000 requêtes/s. Bien que, notre protocole ne soit pas basé sur TCP/IP (qui est utilisé par le protocole HTTP des serveurs web), il est envisageable d'appliquer les idées générales de notre protocole à ce domaine pour obtenir un serveur web parallèle plus rapide que APACHE. Or, le but initial de ce mémoire n'est pas de créer un serveur web, le fait qui nous incite de croire à une telle possibilité, est que pour une connexion TCP les communications entre deux machines sont 2 fois plus, que dans le cas de UDP (notre cas). Alors, même si nous supposons que cette amplification du nombre de messages peut réduire le temps de réponse de notre serveur, cela devait diminuer de 45000 requêtes/s à 22500 requêtes/s. Par conséquent, dans le pire cas il conservera une performance plus élevée que 2000 requêtes/s.

4.11 Contribution

L'importance de notre protocole réside dans le fait qu'il est orienté à résoudre un problème réel dans l'industrie de télécommunication introduit par l'application téléphonique cellulaire de la compagnie Ericsson. Notre contribution scientifique dans ce travail est apparue dans l'architecture et les techniques utilisées pour élaborer ce protocole. Grâce à ces facteurs, nous avons réussi

à résoudre le problème de limitation de la mémoire pour les bases des données de grande taille qui dépasse 4 Go (comme dans le cas d'Ericsson), permettant ainsi de créer un serveur parallèle performant et scalable. De plus, nous avons démontré dans ce mémoire par une étude empirique, que les systèmes parallèles existants investigués qui peuvent être adaptés à un contexte transactionnel distribué tels que les systèmes à mémoires partagées distribuées ne sont plus en mesure d'être une solution idéale. Alors que notre protocole, prouvé par les résultats et la performance obtenue, est une solution pertinente et adéquate.

Chapitre 5

Conclusion et perspective

Les réseaux d'interconnexion fournit une capacité de communication importante, mais les systèmes distribués n'exploitent qu'une proportion de cette capacité à cause du coût ajouté par les protocoles de communication et le système d'exploitation (SE). Par exemple, les systèmes à Mémoire Partagée Distribuée (MPD) sont reconnus par la simplicité offerte au niveau de programmation comparativement au modèle de passage par message. Malgré que les MPD fournissent un bon degré de parallélisme pour les applications à calcul intensif, ce n'est pas le cas pour les applications à fin grain telles que les applications transactionnelles. Cette déficience est due principalement à la latence du réseau ainsi qu'au coût du protocole de communication utilisé tel que TCP/IP.

Ce mémoire est divisé en deux parties. Dans la première partie (chapitre 3) nous avons abordé le sujet des MPD, en présentant les résultats d'une étude d'évaluation faite sur un échantillon des systèmes MPD existants. Cette évaluation a pour but d'étudier la faisabilité de tels systèmes à répondre aux besoins des applications transactionnelles en temps réel. Les résultats obtenus montrent que les systèmes MPD ne sont pas en mesure de réaliser ce but. Notre contribution est présentée dans le chapitre 4, nous avons proposé et implanté un protocole pour résoudre le problème de la localité d'accès dans les applications transactionnelles. L'objectif initial pour réaliser ce travail est de trouver une solution de répartition pour une application téléphonique en temps réel dans un contexte distribué. Vu que les systèmes à Mémoire Partagée Distribuée

ne sont pas en mesure de résoudre ce problème, la nécessité d'avoir un protocole performant distribué est indispensable.

Trois propriétés fondamentales caractérisent ce protocole. Premièrement, l'architecture qu'il adopte est basée sur la distribution statique des données sur un groupe de serveurs. Ces derniers agissent comme une seule entité de travail, sans aucune sorte de centralisation. Suite à cette stratégie de distribution une plus grande taille de base des données sera traitée. Deuxièmement, la communication rapide qu'il offre comparativement aux protocoles de communication traditionnels. Dans ce protocole nous avons élaboré une nouvelle technique de communication "Lightweight Ethernet Communication". Au contraire des autres protocoles de communication qui dépendent fortement du système d'exploitation, cette technique est fondée sur la communication directe par la carte Ethernet sans l'intervention du SE quant à la transmission et la réception des messages. Les tests montrent une bonne amélioration de communication entre deux sites utilisant cette technique. Troisièmement, le Multicast des requêtes. Grâce à cette technique le problème de centralisation est résolu. Aucune nécessité d'avoir un frontal. Bien que l'idée initiale de ce projet soit de réaliser une solution pour les applications transactionnelles distribuées, le protocole mis en place est générique et peut être utilisé dans une variété d'application où la localité d'accès et le taux de téléchargement sont indispensable, il s'agit d'un serveur web, un serveur multimédia ou bien dans le secteur de télécommunication pour applications en temps-réel.

Rappelons-nous que l'objectif initial de notre recherche était de trouver une solution performante pour mettre en oeuvre une application en temps réel sur un système distribué. Le protocole réalisé a atteint cet objectif et ouvre la porte vers d'autres projets. Les résultats mentionnés montrent qu'il est possible d'exploiter 50% de la capacité du réseau Ethernet, ceci est considérable comparativement à ce qu'offre beaucoup d'autres systèmes répartis. Une extension à ce travail est possible, nous avons mentionné la tolérance aux pannes. En fait, l'architecture que nous avons proposée peut probablement être adaptée à la redondance des données puisque les requêtes sont diffusées sur le groupe des serveurs. Or, il est envisageable de dédier un ou plusieurs serveurs de rechange qui peuvent prendre la charge en cas d'une panne du serveur principal. Un algorithme qui gère la duplication des données sur ces serveurs est cependant nécessaire pour réaliser cette étape.

Bibliographie

[Ahmad91] M. Ahmad, J. E. Burns et P. Hutto and G. Neiger, Causal Memory : Definitions, Implementation and Programming,*Technical Report number GIT-ICS-93/55, Georgia Institute of Technology, Atlanta, Georgia 1993.*

[Aron00] M. Aron, D. Sanders, P. Druschel, W. Zwaenepoel, Scalable Content-aware Request Distribution in Cluster-based Network Servers *2000 Annual Usenix Technical Conference, San Diego, Juin 2000.*

[Barat95] A. Baratloo, Calypso : A novel Software System for Fault Tolerant Parallel Processing on Distributed Platforms, *4th IEEE Intl. Symp. On High Performance Distributed Computing, Washington DC, pages 122-129, Août 1995.*

[Barat96] A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff, Charlotte : Metacomputing on the Web. International Journal on Future Generation, *Preliminary version appeared in Proc. of the 9th International Conference on Parallel and Distributed Computing Systems, Dijon, France, pages 181-188, Septembre 1996.*

[Bershad93] B. N. Bershad, M. J. Zekauskas et W. A. Sawdon, The Midway distributed shared memory system, *COMPCON '93. IEEE Computer Society Press, Los Alamitos, Calif., pages 528-537, Février 1993.*

[Bolosky89] W. J. Bolosky, R. P. Fitzgerald et M. L. Scott, Simple But Effective Techniques for NUMA Memory Management, *Proceedings of the 12th ACM Symposium on Operating System Principles, pages 142-153, Décembre 1989.*

[Carter91] J. B. Carter, J. K. Bennett et W. Zwaenepoel, Implementation and performance of Munin, *Proceedings of the 13th ACM Symposium on Operating System Principles, pages 152-164, Octobre 1991.*

[Castro96] M. Castro, M. Sequeira, M. Costa et P. Guedes, Efficient and Flexible Object Sharing, *Proceedings of the 1996 IEEE International Conference on Parallel Processing (ICPP), Bloomingdale IL, vol. 1, pages 128-137, Août 1996.*

[Cox90] A. L. Cox et R. J. Fowler, The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor : Experiences with PLATINUM, *Proceedings of the 12th ACM Symposium on Computer Architecture, pages 32-44, Décembre 1990.*

[Dubois86] M. Dubois, C. Scheurich et F. A. Briggs, Memory access buffering in multiprocessors, *Proceedings of the 13th Annual International Symposium on Computer Architecture, pages 434-442, Juin 1986.*

[Etherboot] http ://etherboot.sourceforge.net/distribution.html.

[Gharach90] K. Gharacholoo, D. Lenoski et J. Laudon, Memory consistency and event ordering in scalable shared memory Multiprocessors, *Proceedings of the 17th International Symposium on Computer Architecture, pages 15-26, Mai 1990.*

[Good89] J.R. Goodman, Cache consistency and sequential consistency, *Technichal report 61, IEEE Scalable Coherent Interface Working Group, Mars 1989.*

[Jacob88] V. Jacobson, Congestion avoidance and Control, *ACM SIGCOM Computer Communication Review, vol. 18, no. 4, pages 314-329, Août 1988.*

[Johnson95] K. L. Johnson, M. F. Kaashoek et D. A. Wallach, CRL : High-Performance All-Software Distributed Shared Memory, *Proceedings of the Fifteenth Symposium on Operating Systems Principles, pages 213-228, Décembre 1995.*

[Karn87] P. Karn et C. Partridge, Improving round trip estimates in reliable transport protocols, *Computer Communication Review, vol. 17, no. 5, pages 2-7, Août 1987.*

[Keleh94] P. Keleher, S. Dwarkadas, A. L. Cox and W. Zwaenepoel, TreadMarks : Distributed Shared Memory on Standard Workstations and Operating Systems, *Proceedings of the Winter 94 Usenix, pages 115-131, Janvier 1994.*

[Keleh95] P. Keleher, Lazy Release Consistency for Distributed Shared Memory, *PhD thesis, Houston, Texas, Janvier 1995.*

[Lamp78] L. Lamport, Time, Clocks and the ordering of events in a distributed system, *Communication of the ACM, vol. 21, no.7, pages 558-565, Juillet 1978.*

[Lamp79] L. Lamport, How to make a multiprocessor computer that correctly executes a multiprocess program, *IEEE transactions on Computers, vol. 28, no. 9, pages 690-691, Septembre 1979.*

[Li89] K. Li et P. Hudak, Memory coherence in shared virtual memory systems, *ACM Transactions on Computer Systems, vol. 7, no.4, pages 321-359, Novembre 1989.*

[MPI] http ://www-unix.mcs.anl.gov/mpi.

[Pai98] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel et E. Nahum, Locality-Aware Request Distribution in Cluster-based Network Servers, *8th Symposium on Architectural Support for Programming Languages and Operating Systems, pages 205-216, Octobre 1998.*

[PVM] http ://www.epm.ornl.gov/pvm/pvm_home.html.

[Stevens98] W. R. Stevens, "Unix Network Programming", vol. 1, Second Edition, Prentice Hall PTR, 1998.