THESE

présentée par

Vincent ROCA

pour obtenir le titre de DOCTEUR

de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

(Arrêté ministériel du 30 mars 1992)

Spécialité: Informatique

==========

ARCHITECTURE HAUTES PERFORMANCES POUR SYSTEMES DE COMMUNICATION

==========

Date de soutenance:	Vendredi 19 janvier 1996	
Composition du jury:	J.P. VERJUS	Président
	E. BIERSACK	Rapporteur
	S. FDIDA	Rapporteur
	C. DIOT	Directeur de thèse
	M. HABERT	Examinateur
	C. HUITEMA	Examinateur
	X. ROUSSET DE PINA	Examinateur

Thèse préparée au sein du laboratoire LSR dans le cadre
d'un contrat CIFRE avec la société Bull S.A. - Echirolles

A Catherine

Remerciements

On n'est pas grand chose tout seul, pas d'accord ? Et oui, alors c'est mon tour de remercier tous ceux et celles à qui je dois tout, et ça fait du monde.

Bon, ben je commence par les chefs, de toutes façons je n'ai pas le choix ;-). J'ai eu l'immense privilège d'avoir deux supers chefs durant ces trois années. J'ai nommé Christophe Diot, toujours prêt à lire et à barbouiller de rouge[1] ma prose, et qui de plus a réussi l'exploit de me faire comprendre ce qu'est la recherche. Il aura tout de même fallu cinq ans, depuis le DEA ! Et Michel Habert, mon chef Bulleur, débordant d'idées et de dévouement. Tous les deux, je ne vous remercierai jamais assez.

Les membres du jury maintenant : Jean-Pierre Verjus, merci d'avoir accepté de présider mon jury. Ernst Biersack et Serge Fdida : chapeau, c'est vous qui avez fait le plus dur ! Christian Huitéma, j'apprécie grandement que vous soyez dans le jury. Xavier Rousset, merci pour ton soutien.

Merci à tous ceux qui m'ont permis, par leurs commentaires, d'améliorer ces travaux et ce mémoire.

Et merci à tous mes ex-collègues Bulleurs qui m'ont toujours aidé sous une forme ou une autre : Aimé, Jean-Dominique, Michèle, Kaïs, Frédéric, Philippe, Olivier, Jean, Denis, Thierry, Imed, MarieAn, Jean-Luc, Nouar, Sylvie, et j'en passe.

Guy Mazaré, Gérard Michel et Jacques Mossière, merci de m'avoir facilité mes démarches administratives. Et pour tous les futurs thésards qui hésitent quant au choix du labo : venez au LSR, les pots y sont nombreux et copieux. Merci Jacques !

Un grand merci à mes mécènes, en l'occurence Bull S.A., l'ANRT, et l'ENSIMAG.

Et puis longue vie à la section Archi, aux montagnes, aux montagnard(e)s, à la Yaute et à ses habitants, à Mountain Wilderness, au CAF, à la FRAPNA... Bon, je m'arrête.

Et last but not least, je remercie mes parents de m'avoir mis au monde - c'était une très bonne idée -, et je pourrais en dire autant de Beau-papa et Belle-maman. Non non, ce n'est pas du fayottage, je vous assure ! N'oublions pas François et Aude, la Moumoune, et enfin Catherine à qui je dédie ce mémoire.

1. Je me souviens qu'au départ c'était de l'encre violette. Est-ce parce que le rouge impressionne davantage que tu as changé de couleur d'encre ?

Table des matières

ACRONYMES . . . . . **VIII**

RÉSUMÉ ET ABSTRACT . . . . . **X**

1 INTRODUCTION **13**

2 ETAT DE L'ART **17**

2.1 Tendances dans le domaine des réseaux 17

2.1.1 Les couches basses : des LANs traditionnels aux réseaux gigabits/s 17

2.1.2 Les applications. 19

2.1.2.1 Les deux générations d'applications 19

2.1.2.2 Les besoins des applications 21

2.1.3 Caractéristiques des stations de travail actuelles 23

2.1.3.1 L'aspect architecture de la machine hôte. 23

2.1.3.2 L'aspect système 26

2.1.3.3 L'aspect protocole de communication. 26

2.2 Approches pour la conception de systèmes de communication hautes performances 29

2.2.1 Adaptation mutuelle des composants du système de communication 30

2.2.2 Optimisation des mécanismes de contrôle des protocoles 32

2.2.3 Optimisation des manipulations de données et des accès mémoire . 34

2.2.3.1 Moins de manipulations de données 34

2.2.3.2 Meilleure utilisation des caches 36
2.2.4 Accroissement des traitements par le parallélisme 37

3 NOTRE PLATE-FORME EXPÉRIMENTALE 41
3.1 Présentation de nos travaux 41
3.1.1 Les points abordés par ces travaux 41
3.1.2 Présentation de notre plate-forme expérimentale 42
3.2 La machine hôte 43
3.2.1 La famille POWER de microprocesseurs 43
3.2.2 Le DPX/20 modèle 420. 44
3.2.3 Le système d'exploitation AIX 44
3.3 BSD versus Streams 45
3.3.1 L' environnement BSD 45
3.3.2 L'environnement Streams 47
3.3.3 Résumé 50
3.4 Implémentation classique d'une pile TCP/IP sous BSD et Streams 51
3.4.1 Une pile TCP/IP classique sous BSD 51
3.4.2 Une pile TCP/IP classique sous Streams. 52
3.4.3 Limites à priori de l'implémentation Streams à la lumière de BSD . 54
3.5 Les alternatives. 55
3.5.1 Une implémentation démultiplexée sous Streams. 55
3.5.1.1 Architecture de l'implémentation démultiplexée. 55
3.5.1.2 Bénéfices de cette approche. 57
3.5.1.3 Techniques supplémentaires 61
3.5.2 Une implémentation au niveau utilisateur de TCP 63
3.5.3 Une méthode d'accès ILP 64
3.5.3.1 Le cas du flux sortant. 65
3.5.3.2 Le cas du flux entrant. 67
3.6 Les outils d'évaluation de performances 68
3.6.1 Un environnement intégré d'évaluation de performances 68
3.6.2 Les autres outils. 69
3.6.2.1 L'outil de trace. 69
3.6.2.2 L'outil de comptage d'instructions 70
3.6.2.3 L'outil d'analyse des contentions d'accès aux verrous . . 70

4 ETUDE DE PERFORMANCES 73
4.1 Expériences de haut niveau 73
4.1.1 Comparaison des différentes architectures 74
4.1.1.1 Les piles TCP/IP BSD et Streams classiques. 74
4.1.1.2 La pile TCP/IP Streams démultiplexée 75
4.1.1.3 Implémentation au niveau utilisateur de TCP 77
4.1.2 Evaluation des architectures sur des multiprocesseurs 78
4.1.2.1 Performances et taux d'accélération 78
4.1.2.2 Contentions d'accès aux verrous 80
4.1.2.3 Bilan. 83

4.1.3 Evaluation d'ILP 83
4.1.3.1 Bénéfices liés à l'intégration copie/checksum 83
4.1.3.2 Impact de la méthode d'accès ILP 84
4.1.3.3 Discussion 86
4.2 Expériences de bas niveau 87
4.2.1 Temps de traitement 87
4.2.1.1 Distribution des temps de traitement sous AIX/3.2.5 . . . 87
4.2.1.2 Distribution des temps de traitement sous AIX/4.1.2 . . . 88
4.2.1.3 Discussion 90
4.2.2 Instructions déroulées 93
4.2.2.1 Distribution des instructions déroulées sous AIX/4.1.2. . 93
4.2.2.2 Discussion 95
4.2.2.3 Comparaison avec d'autres expériences de comptage d'instructions 96
4.2.3 Ratios instructions/cycle 97
4.2.3.1 Ratios instructions/cycle moyens 98
4.2.3.2 Ratios instructions/cycle durant les manipulations de données 99
4.2.3.3 Discussion 100

5 DISCUSSION 103
5.1 Aspect architecture de la machine hôte 103
5.1.1 Limites du parallélisme 103
5.1.2 Limites de ILP 105
5.2 Aspect système d'exploitation 106
5.2.1 Limites des environnements actuels 106
5.2.1.1 Limites de Streams 106
5.2.1.2 Limites de BSD 108
5.2.2 Caractéristiques d'un environnement de système de communication hautes performances 108
5.2.3 Architecture de la pile de communication 112
5.2.3.1 Implémentations démultiplexées 112
5.2.3.2 Implémentations de niveau utilisateur 113
5.3 Aspect protocole de communication 114

6 CONCLUSION 117

BIBLIOGRAPHIE 121

ANNEXE A: PROPOSITIONS D'AMÉLIORATION DE STREAMS ET DE SON USAGE 131

Acronymes

AAL	ATM Adaptation Layer
ADU	Application Data Unit ([Clark90])
ALF	Application Level Framing ([Clark90])
API	Application Program Interface (interface avec les services réseau offerts aux applications)
ARP	Address Resolution Protocol
ARPANET	Advanced Research Projects Agency network (premier réseau international)
ATM	Asynchronous Transfer Mode
BOOTP	Bootstrap Protocol
BPF	BSD Packet Filter
BSD	Berkeley Software Distribution
CBQ	Class Based Queueing (méthode d'accès par classes de priorités au réseau [Wakeman95])
CC	Communication Channel (cf. pile TCP/IP démultiplexée sous Streams)
CDLI	Common Data Link Interface (interface d'abstraction aux drivers réseau)
CISC	Complex Instruction Set Computer
CLNP	ConnectionLess Network Protocol
CRC	Cyclic Redundancy Code
CPU	Central Processing Unit
DLPI	Data Link Provider Interface (interface d'abstraction aux drivers réseau utilisée avec Streams)
DMA	Direct Memory Access
DNS	Domain Name Server
E/S	Entrée/Sortie
ES	Eléments de synchronisation (utilisé pour implémenter les niveaux de synchronisation Streams)
FDDI	Fiber Distributed Data Interface
FTP	File Transfer Protocol
HIPPI	High Performance Parallel Interface
ICMP	Internet Control Message Protocol
IEEE	Institute of Electrical and Electronics Engineers
IESG	Internet Engineering Steering Group
IETF	Internet Engineering Task Force
IGMP	Internet Group Management Protocol
ILP	Integrated Layer Processing ([Clark90])
IP	Internet Protocol
IPC	Inter-Process Communication (pipe, socket, mémoire partagée, sémaphore, etc.)
IPS	Internet Protocol Suite (désignée sous le nom de pile TCP/IP dans ce mémoire)
ISO	International Organization for Standardization
ISOC	Internet Society
LAN	Local Area Network
LLC	Logical Link Control
MAC	Medium Access Control

MAN	Metropolitan Area Network
MBLK	Message Block (buffer utilisé par Streams)
MBONE	Multicast Backbone
MBUF	Memory Buffer (buffer utilisé par BSD)
MCA	Micro-Channel Architecture (technologie IBM utilisée sur les machines DPX/20)
MIME	Multipurpose Internet Mail Extension
MSL	Maximum Segment Lifetime
MSS	Maximum Segment Size (taille maximale d'un segment TCP, égale à : (P)MTU - <taille des en-têtes TCP/IP>)
MTU	Maximum Transmission Unit (taille maximale des trames définie par la couche de liaison de données)
NDIS	Network Driver Interface Specification (interface d'abstraction aux drivers réseau)
NFS	Network File System
NNTP	Network News Transfer Protocol (Usenet news)
NPI	Network Protocol Interface (interface transport/réseau utilisée avec Streams)
NSFNET	National Science Foundation Network (successeur de l'ARPANET)
OS	Operating System (ou système d'exploitation)
OSF	Open Software Foundation
OSI	Open System Interconnection (modèle réseau défini par l'ISO)
PDU	Protocol Data Unit
PIO	Programmed I/O (technologie alternative au DMA)
PMTU	Path Maximum Transmission Unit (MTU pour un chemin donné [RFC1191])
POSIX	Portable Operating System Interface (standards définissant UNIX)
QoS	Quality of Service
RFC	Request For Comments
RISC	Reduced Instruction Set Computer
RPC	Remote Procedure Call
RTP	Real-time Tranport Protocol
RTT	Round Trip Time (temps d'aller-retour entre deux machines)
SMTP	Simple Mail Transfer Protocol
SNMP	Simple Network Management Protocol
TCP	Transmission Control Protocol
TFTP	Trivial File Transfer Protocol (version simpliste de FTP au dessus d'UDP, souvent utilisée lors du démarrage de stations sans disques)
TIMOD	Transport Interface Module (associé à la bibliothèque TLI/XTI, utilisé avec Streams)
TLB	Translation Lookaside Buffer (crée une translation pages virtuelles/segments mémoire)
TLI	Transport Layer Interface (API concurrente de l'API Socket, très proche de XTI)
TOS	Type of Service (champ de l'en-tête TCP)
TPDU	Transport Protocol Data Unit (message envoyé par l'application à la couche transport)
TPI	Transport Protocol Interface (interface application/transport utilisée avec Streams)
UDP	User Datagram Protocol
VC	Virtual Circuit (ATM)
VCI	Virtual Circuit Identifier (ATM)
VP	Virtual Path (association de plusieurs VCs, ATM)
VPI	Virtual Path Identifier (ATM)
WAN	Wide Area Network
WWW	World Wide Web
XDR	eXternal Data Representation
XTI	X/Open Transport Layer Interface (normalisation de TLI par l'X/Open)
XTP	eXpress Transport Protocol (le T signifiait "Transfer" jusqu'à la version 3.6 [XTP94])

Résumé

Nos travaux abordent le problème de l'efficacité des techniques d'implémentation hautes performances du système de communication. Car si les principes généraux sont désormais bien connus, il n'en va pas toujours de même de leur application. Des résultats obtenus sur notre plate-forme expérimentale, nous tirons des conclusions quant aux aspects machine hôte, système, et protocoles.

Pour l'aspect machine hôte nous montrons que la parallélisation des piles de communication est décevante et grandement limitée par les contentions d'origine système. Nous montrons également comment la technique ILP, destinée à limiter les accès mémoire, peut être intégrée à la méthode d'accès d'une pile TCP/IP. Les bénéfices de cette technique restent largement dépendants des conditions d'utilisation, de la nature de l'application, et des manipulations de données.

Du point de vue système, l'étude de deux environnements d'exécution de protocoles opposés, BSD et Streams, a permis d'identifier leur faiblesses, et notamment pourquoi l'environnement Streams actuel n'est pas adapté aux hautes performances. Nous en déduisons un ensemble de principes qu'un environnement performant se doit de respecter.

L'architecture de la pile de communication est essentielle. Nous montrons qu'une architecture démultiplexée, avec des chemins de données directs entre applications et drivers réseaux, permet un excellent support du contrôle de flux local et des mécanismes systèmes de QoS. En revanche, les implémentations de niveau utilisateur souffrent de nombreuses limitations. Elles sont cependant indispensables à certaines techniques.

Du point de vue protocoles, nous montrons que la présence d'options dans les en-têtes de paquets n'est pas contraire à l'obtention de bonnes performances. A la notion trop rigide d'en-têtes de taille fixe nous substituons celle d'en-têtes de taille prédictible.

Enfin, il ressort deux notions clés de ces travaux, la simplicité et la flexibilité, dont dépendent les performances et les fonctionnalités du système de communication.

Mots clés : protocoles de communication, systèmes de communication, ingénierie des protocoles, BSD, Streams, hautes performances, ILP, parallélisme, qualité de service.

Abstract

Our work investigates the efficiency of high performance implementation techniques for the communication system. Indeed, if the general principles are now well known, it is not always the case of their application. From the results collected on our experimental platform, we draw conclusions on the host machine, system, and protocol aspects.

For the host machine aspect, we show that the parallelization of the communication stack is disappointing and largely limited by system lock contentions. We also show how the ILP technique, meant to limit memory accesses, can be integrated in the access method of a TCP/IP stack. The benefits of this technique largely depend on the conditions of use, the kind of application, and the data manipulations.

From the system point of view, the study of two opposed protocol environments, BSD and Streams, made it possible to identify their limitations, and in particular why the current Streams environment is not suited to high performance. We deduce a set of principles that a high performance environment should follow.

The architecture of the communication stack is essential. We show that a demultiplexed architecture, with direct data paths between the applications and the network drivers, enables a good support of local flow control and system QoS mechanisms. On the contrary, user-level implementations suffer many limitations. Yet they remain essential for certain techniques.

From the protocol point of view, we show that the presence of options in packet headers is not opposed to high performance. To this too rigid notion of fixed size headers, we substitute that of foreseeable size headers.

Finally, this work highlights two key notions, simplicity and flexibility, upon which depend the performance and functionalities of the communication system.

Keywords : communication protocols, communication systems, protocol engineering, BSD, Streams, high performance, ILP, parallelism, quality of service.

CHAPITRE 1

Introduction

Plusieurs années sont passées depuis la naissance de l'ARPANET en 1969. Le succès de ce réseau a conduit à l'Internet, un ensemble de plus de trois millions d'hôtes (1993) de par le monde qui connaît une croissance exponentielle. Associé à ce succès est celui du système d'exploitation UNIX dont l'origine remonte là aussi au début des années 1970. La distribution libre de l'UNIX Berkeley, intégrant une pile de communication TCP/IP, le standard de l'Internet, a contribué à cette tendance.

Un effet de bord de ce succès est que le système de communication, un ensemble complexe de protocoles, de fonctionnalités systèmes et de support matériel, a été développé il y a dix ou vingt ans. Ainsi TCP et Ethernet, deux briques essentielles de l'Internet, ont tous deux été conçus en 1974. A cette époque des débits de quelques centaines de kilo-bits/s étaient courants, les taux de pertes de paquets élevés, et les applications conçues pour tolérer des délais d'acheminement largement variables. Cet héritage se retrouve dans le code réseau de Berkeley qui a été développé sur des machines VAX à une époque où un VAX-11/780 disposant de 4 méga-octets de mémoire était un gros système. Plus généralement, les systèmes informatiques étaient alors conçus avant tout pour les traitements, les communications restant secondaires.

Il n'est donc pas surprenant que le système de communication actuel se révèle inapproprié aux nouveaux réseaux physiques, applications et stations de travail. Les réseaux sont désormais caractérisés par de hauts débits allant jusqu'au giga-bits/s, de faibles taux d'erreurs, voir des garanties de qualités de service (QoS). Les nouvelles applications, en particulier les applications multimédias, ont souvent des contraintes temps-réel et se contentent difficilement des services réseaux

traditionnels de type “meilleur effort”. Enfin les stations de travail actuelles n’ont plus aucun rapport avec celles des années 1980 : ainsi, en dépit de techniques de cache évoluées, les processeurs sont désormais davantage limités par les accès mémoires que par les traitements. Toutes ces raisons rendent une évolution majeure du système de communication nécessaire.

Nos travaux, menés sur la période décembre 1992 - janvier 1996, s’inscrivent dans cette lignée. Notre but est d’*améliorer le système de communication des stations de travail actuelles*, notamment du point de vue des performances brutes (débit et latence).

Plusieurs approches ont été proposées au cours des dernières années, chacune d’elles visant à améliorer un aspect particulier : mécanismes des protocoles, architecture de la pile de communication, support de la QoS, adéquation avec la machine hôte, etc. Nos travaux nous ont conduits à *examiner les conditions d’application et l’efficacité de quelques-unes de ces approches*. En effet, si les principes généraux commencent à être bien connus, il n’en va pas toujours de même de leur intégration au système de communication. Une raison est que ces approches remettent souvent en cause l’architecture du système de communication, et des contraintes diverses peuvent alors grandement limiter les bénéfices que l’on peut en retirer. Le cas d’ILP (Integrated Layer Processing), qui vise à limiter les accès mémoire par une intégration des fonctions de manipulation de données, est à cet égard significatif. L’aspect *applicabilité des techniques hautes performances* revêt donc une importance capitale qui sera tout particulièrement soulignée.

La plate-forme expérimentale que nous avons conçue afin de valider nos idées est suffisamment flexible et complète pour permettre une large gamme d’essais. Ceci est essentiel car de nombreux facteurs contribuent à l’obtention des performances du système de communication. Les points sur lesquels nous nous sommes penchés sont :

- la machine hôte dont les caractéristiques et les limitations influencent le système de communication dans sa totalité. Une grande importance a été apportée au rendement effectif du processeur, aux possibilités de traitements parallèles offertes par les machines multiprocesseurs, et à l’impact des copies de données et de l’approche ILP.

- le système d’exploitation et l’environnement d’implémentation de la pile de protocoles, responsables de 80% des traitements. Nous nous sommes intéressés aux deux environnements majeurs des systèmes ouverts, BSD et Streams, qui reposent sur deux paradigmes de communication radicalement opposés : appels de fonctions pour BSD et passages de messages pour Streams. Nous voulons apprécier l’ensemble des conséquences que peut avoir le choix de l’un ou de l’autre.

- l’architecture de la pile de communication qui influence plusieurs caractéristiques essentielles, telles que la complexité du chemin de données, les possibilités de contrôle de flux, le type de parallélisme, etc. Pour cela nous avons étudié une large palette d’architectures, depuis les piles classiques sous BSD ou Streams, jusqu’aux architectures démultiplexées ou implantations de niveau utilisateur.

- les protocoles qui sont au coeur du système de communication.

Signalons que ces travaux ont pour cible les stations de travail et serveurs actuels, à la fois monoprocesseurs et multiprocesseurs symétriques (SMP), utilisant un système d'exploitation (OS) UNIX. Le choix d'un système d'exploitation ouvert a des conséquences majeures : cela signifie qu'un certain niveau de compatibilité est requis, et de façon plus générale, que le système de communication subisse une évolution plutôt qu'une révolution. Par conséquent nous avons décidé que les applications existantes et les interfaces avec ces applications (API) devraient être préservées. Cependant il reste possible, afin de supporter de nouvelles applications, de proposer des extensions au système de communication dans la mesure où cela ne remet pas en cause la compatibilité ascendante.

En utilisant ces règles nous avons l'intention de *permettre au plus grand nombre possible d'applications de bénéficier de gains de performances*. Dans le même temps certaines applications critiques pourront bénéficier de traitements particuliers.

Organisation de ce document

Ce document est divisé en la présente introduction, quatre chapitres, et une conclusion :

Le chapitre deux propose un état de l'art dans le domaine des réseaux hautes-performances. Il commence par un survol des évolutions des technologies réseaux et des besoins des applications. A la lumière de cette étude, nous présentons les limites du système de communication actuel. Enfin une classification selon quatre axes des principaux concepts et techniques destinés à résoudre ces limites est proposée. Nous introduisons ainsi quelques-unes des techniques mises en oeuvre dans notre plate-forme expérimentale.

Le chapitre trois est consacré à la description des différents composants de notre plate-forme. Il présente tout d'abord le système de communication classique d'une station UNIX, depuis les processeurs RISCs jusqu'aux piles TCP/IP en environnement BSD et Streams. Nous décrivons ensuite les trois techniques que nous proposons comme alternative : une pile de communication démultiplexée, une implémentation de niveau utilisateur de TCP, et enfin une méthode d'accès ILP. Ce chapitre s'achève sur la description des outils d'analyse de performances que nous avons conçus et/ou utilisés durant les tests.

Le chapitre quatre est une étude de performances de notre plate-forme expérimentale. Elle nous permet d'évaluer l'opportunité, dans des conditions réelles, de chacune des alternatives proposées. Deux approches ont été suivies : des expérimentations de haut niveau (débits et temps d'aller-retour moyens mesurés au niveau applicatif), et des expérimentations de bas niveau (temps de traitements, instructions déroulées, et ratios instructions/cycle). Ces approches sont complémentaires dans la mesure où la première permet une analyse globale du système de communication, en prenant en compte son comportement dynamique. A l'opposé l'approche de bas niveau permet une étude détaillée mais statique, révélant certains phénomènes cachés à l'étude de haut niveau.

Le chapitre cinq tire les conclusions de l'expérience théorique et pratique que nous avons obtenue et propose diverses solutions afin d'améliorer encore le système de communication. Cette discussion suit trois axes : l'aspect architecture de la machine hôte, l'aspect système et l'aspect

protocoles de communication.

Le chapitre six conclut ce mémoire. Après une synthèse des résultats obtenus, nous mettons en évidence deux points clés du système de communication, la simplicité et la flexibilité.

Notons que ce document utilise des anglicismes lorsqu'il n'existe pas de traduction précise. Ainsi nous parlerons de thread, de scheduling, de driver, et de buffer.

CHAPITRE 2

Etat de l'art

Ce deuxième chapitre d'état de l'art s'intéresse au système de communication des stations de travail actuelles. Il débute par une présentation des évolutions qu'ont connues ces dernières années les couches basses des réseaux informatiques et les applications. Ceci nous permet ensuite de mettre en évidence les limites tant matérielles que logicielles du système de communication. Enfin nous proposons une classification selon quatre axes des différentes approches destinées à résoudre ces problèmes.

2.1 TENDANCES DANS LE DOMAINE DES RÉSEAUX

2.1.1 Les couches basses : des LANs traditionnels aux réseaux gigabits/s

Depuis l'apparition des systèmes de communication à la fin des années 1950, les techniques se sont peu à peu améliorées et les équipements sont devenus à la fois plus rapides et moins coûteux. Le rapport prix/performances a longtemps progressé d'environ 20% par an. Mais actuellement il est prévu que le coût des lignes de communication longues distances soit réduit non de quelques pourcents mais d'un facteur 100 à 1000 en quelques années seulement [IBM93]. Les principales causes en sont :

- l'adoption des techniques numériques au sein des réseaux de télécommunication publiques,

- et l'arrivée des technologies optiques.

Le nouvel environnement réseaux est désormais caractérisé par :

De hautes performances

Les réseaux Ethernet 10 Mbits/s et Token Ring 16 Mbits/s sont désormais légion. Le standard FDDI à 100 Mbits/s et les nouvelles versions d'Ethernet (100BaseT et 100BaseVG) tendent à se généraliser. ATM utilisé en association avec Sonet/SDH va au delà avec des débits de 155 méga-bits/s à 622 méga-bits/s. Des débits de l'ordre du giga-bits/s seront bientôt disponibles et cette tendance ne peut que se poursuivre.

Un faible taux d'erreurs

Le taux d'erreurs associé aux nouvelles technologies s'est grandement amélioré. Une ligne téléphonique analogique accédée depuis un modem peut avoir un taux d'erreurs de l'ordre de 10^{-5}, soit une erreur tous les 100 000 bits. Avec une fibre optique ce taux est ramené à 10^{-11}, soit un million de fois moins. Le type d'erreurs a également changé. Alors qu'elles apparaissaient sous forme d'erreurs simples ou doubles sur les lignes analogiques, les erreurs tendent à apparaître par séries avec les techniques digitales.

De nouveaux services

C'est une autre tendance importante : des garanties de qualité de service (QoS) sont introduites dans l'infrastructure de communication elle-même. Ainsi FDDI-II, extension à FDDI, offre un service synchrone en plus du service traditionnel.

ATM fournit lui aussi des garanties qui le rendent adapté à tout type de trafic : voix, image, vidéo et données traditionnelles. Ainsi chaque canal virtuel possède un jeu de paramètres de QoS spécifiant le débit minimum disponible et le délai maximum permis. Les cellules contiennent quant à elles un bit de priorité utilisé en cas de congestion afin de sélectionner les cellules devant être détruites. Les couches d'adaptation (AAL) fournissent différentes classes de services au dessus de ces caractéristiques de bas niveau :

- CBR (Constant Bit Rate) pour la transmission de la voix et l'émulation de circuits,

- VBR (Variable Bit Rate) qui se subdivise en une version temps-réel (voix avec suppression des blancs) et une version non temps-réel (données transactionnelles),

- UBR (Unspecified Bit Rate) afin de tirer profit des créneaux de transmission laissés libres par le trafic CBR/VBR, et

- ABR (Available Bit Rate), similaire à la classe UBR, mais incluant un mécanisme de rétroaction afin d'adapter en temps réel le trafic ABR au débit disponible. Les pertes de cellules sont donc limitées ce qui rend ce service bien adapté aux applications élastiques (Section 2.1.2.1).

La catégorie des réseaux sans fils doit être mise à part. En effet, le medium électromagnétique diffère en de nombreux points des médias traditionnels. Ainsi ces réseaux sont caractérisés par des débits faibles, de quelques kilo-bits/s à quelques méga-bits/s, variables avec le temps, et des types d'erreurs différents. Les réseaux sans fils suivent actuellement un processus de standardisation au sein de l'IEEE sous la référence 802.11. Ils sont récemment devenus économiquement viables et des produits existent déjà sur le marché.

2.1.2 Les applications

2.1.2.1 Les deux générations d'applications

Deux classes d'applications doivent être distinguées : les applications traditionnelles et les applications de nouvelle génération.

Les applications traditionnelles

Cette classe inclut essentiellement les applications qui sont apparues durant les années 1970s, après la création du réseau ARPANET. Elles peuvent être classées en [RFC1633] :

- applications interactives avec transfert par vagues, ou "burst" (telnet, rlogin, X, NFS),
- applications interactives avec transfert de masse (FTP), et
- applications asynchrones avec transfert de masse (e.mail, FAX).

Un certain nombre de points caractérisent ces applications. Tout d'abord elles *tolèrent aisément d'importants taux de pertes de paquets et délais de transfert*. Elles n'imposent aucune limite inférieure au service réseau et se contentent de ce qu'elles obtiennent. C'est la raison pour laquelle ces applications sont aussi appelées *élastiques*. Elles laissent à l'utilisateur la responsabilité d'apprécier si le service offert est suffisant[1]. Si ce n'est pas le cas, la seule solution consiste alors à réessayer plus tard.

Deuxièmement, ces applications sont caractérisées par le fait qu'elles utilisent essentiellement un mode de *communication point-à-point* et reposent soit sur un service totalement fiable (mode connexion), soit sur un service sans garanties (mode non-connecté ou datagramme).

1. [RFC1144] montre qu'un délai de réponse supérieur à 100 ou 200 ms est perçu comme mauvais dans le cas d'une application interactive.

L'architecture du système de communication actuel (couches réseau et liaison de données) reflète cette génération d'applications : seule une classe de services dite de "meilleur effort" existe. Elle ne fournit aucun mécanisme de contrôle d'admission, et aucune garantie de délai d'acheminement ni d'arrivée à destination des datagrammes.

Les applications de nouvelle génération

Une nouvelle génération d'applications est apparue. Leur caractéristique principale est d'avoir des *contraintes temps-réel* : les données qui ne sont pas arrivées passé un certain délai deviennent inutiles. De nombreuses applications multimedias, en particulier celles qui doivent jouer localement une séquence (voix ou vidéo) qui a été collectée à distance, sont de ce type. Leur implémentation définit habituellement un décalage entre le moment où les données sont reçues et celui où elles sont utilisées. Cette caractéristique permet une tolérance, plus ou moins élevée suivant l'importance du décalage, vis-à-vis des variations du délai d'acheminement (gigue). Cette technique est néanmoins insuffisante dans le cas de très fortes variations comme cela existe sur les réseaux longues distances.

Une seconde caractéristique courante est le besoin d'un *transfert multi-points* (distribution TV, vidéo-conférence, tableau-blanc, etc.). Cette caractéristique crée de nombreux problèmes. Ainsi certains algorithmes reposent sur des informations de rétroaction. La taille du groupe de récepteurs étant à priori indéterminée, l'émetteur peut aisément être engorgé lorsque les récepteurs sont nombreux. Ceci est vrai tant au niveau des protocoles (Section 2.1.3.3) que des mécanismes d'adaptation des applications (Section 2.2.1).

Enfin un mécanisme de contrôle d'erreurs fin allant au delà des modèles connecté ou datagramme est requis. Le type exact de mécanisme est lié à la nature de l'application : si une fraction d'image peut aisément être traitée dans le désordre, ce sera moins vrai avec un signal audio. Une autre motivation pour ce choix est que les mécanismes de contrôle d'erreurs sont souvent contradictoires avec les besoins temps réel des applications (Section 2.1.3.3).

Nous venons de voir que cette catégorie d'applications a des besoins temps réel. Deux approches sont possibles afin d'y faire face :

- réduire les contraintes temps réel : cela passe par une adaptation de l'application au service réseau (Section 2.2.1), soit en s'adaptant aux valeurs courantes de délai et de gigue, soit en s'adaptant au débit disponible.

- inclure des garanties de qualité de service (QoS) dans le système de communication : cette technique requiert la création de nouveaux mécanismes au sein du réseau afin d'aller au delà du modèle de "meilleur effort" actuel.

Ces deux approches sont valables et complémentaires. La première permet aux applications temps réel d'utiliser l'infrastructure de réseau actuelle et de s'adapter à ses imperfections. Mais cette adaptation a des limites que seule l'introduction de mécanismes de QoS peuvent résoudre.

2.1.2.2 Les besoins des applications

Les besoins des applications vis-à-vis du service réseau

[Shenker94] décrit qualitativement comment les différents types d'applications se satisfont du service réseau fourni. A des fins de simplicité seul le paramètre "débit disponible" est considéré.

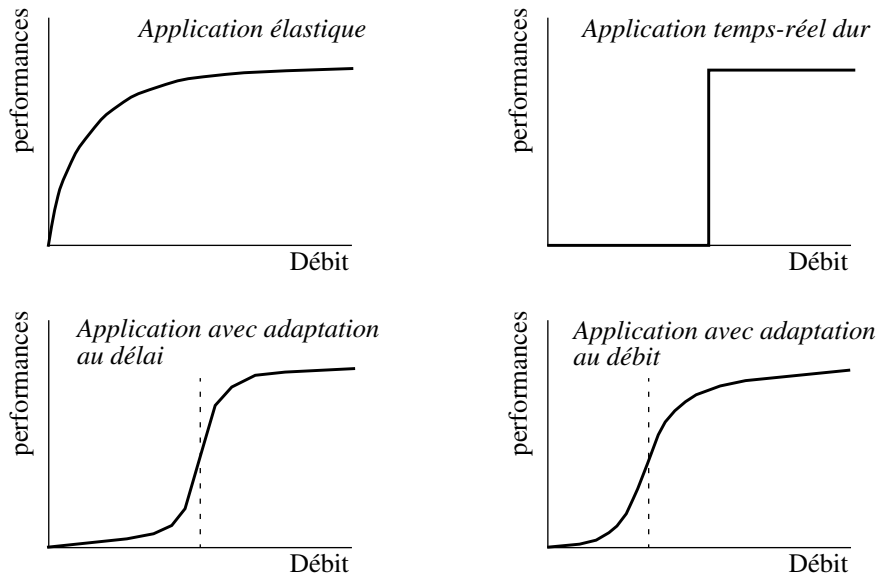

Figure 1: Aspect des fonctions d'utilité selon le type d'application.

La Figure 1 décrit la forme des fonctions d'utilité (performances de l'application en fonction du débit disponible) pour chaque type d'applications :

1. Applications élastiques (ou traditionnelles) : la courbe est concave ce qui signifie que l'application retire un bénéfice, qui tend à diminuer, d'un accroissement du débit.

2. Applications temps réel dur : tant que le service réseau est supérieur au seuil requis par l'application, ses performances sont constantes, mais dès que ce service tombe en dessous du seuil, l'application ne peut plus fonctionner.

3. Applications avec adaptation au délai : la courbe est globalement similaire à celle des applications temps réel dur excepté pour la chute qui est adoucie du fait des possibilités d'adaptation. Le point d'inflexion correspond à la consommation moyenne du signal.

4. Applications avec adaptation au débit : parce que le délai de transmission est généralement acceptable (c'est le but de l'adaptation qui s'efforce d'éviter de surcharger le réseau), les performances de l'application ne dépendent que de la qualité du signal. Le point d'inflexion correspond du débit requis par le signal de qualité minimale acceptable.

Le comportement de la courbe autour de zéro détermine si l'application peut se satisfaire d'un

faible niveau de service réseau (courbes concaves) ou non (courbes convexes). L'abscisse du point d'inflexion dans ce dernier cas détermine le service moyen requis par l'application. Enfin la pente de la courbe au point d'inflexion détermine à quel point l'application peut s'adapter : plus elle est faible, meilleure sera l'adaptabilité.

Les besoins des applications en terme de QoS

L'analyse des applications a souligné l'importance d'un service réseau respectant des aspects de QoS. Ceci est requis à la fois par l'application dont les algorithmes nécessitent un niveau minimum, mais aussi par les utilisateurs qui, indépendamment du bon fonctionnement de l'application, sont les seuls juges pour décider si le service global de l'application est satisfaisant ou non.

La définition de la QoS passe habituellement par la fourniture d'un certain nombre de paramètres [Campbell94]. Il s'agit tout d'abord d'un jeu complexe de paramètres *quantitatifs* relatifs aux niveaux de débit, pertes, délai et gigue désirés. Un deuxième jeu permet de raffiner *qualitativement* ces désirs en introduisant en particulier la distinction entre garanties dures et souples. Une difficulté de cette approche est de fournir un jeu pertinent de valeurs numériques, les paramètres manipulés ayant peu de signification pour l'utilisateur final.

Une autre approche a été proposée dans [Diaz95]. Il s'agit d'un cadre général de définition de la QoS qui repose sur trois notions (Figure 2) :

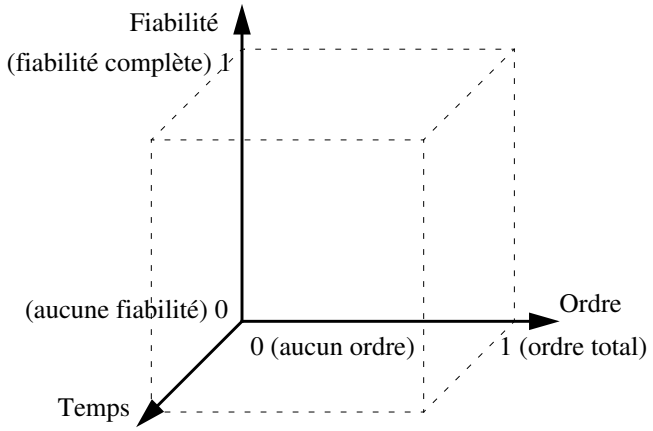

Figure 2: Spécification de la QoS selon les aspects fiabilité, ordre et temps.

- *Fiabilité* : cet axe détermine le niveau de fiabilité du transfert d'informations requis par l'application : complète (cas de TCP), nulle (cas de UDP) ou intermédiaire (par exemple en définissant une échelle de priorités utilisée en cas d'engorgement).

- *Ordre* : cet axe détermine si les données doivent être délivrées à l'application réceptrice avec un ordre total (défini par l'émetteur), partiel (le flux de données est subdivisé en sous-flux indépendants les uns des autres), ou dans le désordre (ordre d'arrivé).

- *Temps* : cet axe définit quelles sont les limites de délai d'acheminement acceptables.

Ces trois notions sont présentées comme étant trois paramètres fondamentaux et constituent à ce titre un cadre formel de définition et de gestion de la QoS. Ils sont d'un grand intérêt pour spécifier les désirs de l'application et/ou utilisateur avec le maximum de simplicité et de généralité.

2.1.3 Caractéristiques des stations de travail actuelles

Nous venons de voir que les couches basses des réseaux évoluent vers plus de performances et des garanties de QoS. Les applications quant à elles sont de plus en plus exigeantes. Par conséquent il est du ressort des composants intermédiaires, à savoir la station de travail et son système de communication, de suivre cette tendance et d'intégrer ces nouveaux besoins. C'est ce que nous examinons ici.

Ainsi qu'il a été mentionné en introduction, nous nous intéressons aux stations de travail du marché utilisant un système ouvert de type UNIX (cela reste applicable à WindowsNT qui partage beaucoup de similarités techniques [WindowsNT93]). Les cas des cartes réseau intelligentes qui incluent l'essentiel du système de communication (souvent vues comme une façon de décharger la machine hôte) et des implantations matérielles des protocoles de communication ne sont pas traités ici. Ils ne sont pas représentatifs de la classe de machines considérée.

Trois aspects clés du système de communication peuvent être identifiés, soit en suivant une logique d'abstraction croissante :

- l'aspect *architecture de la machine hôte* regroupe les composants matériels tels le processeur, la mémoire, les caches, le bus système, la carte d'adaptation réseau, etc,

- l'aspect *système* regroupe le système d'exploitation, l'environnement d'implémentation des protocoles, ainsi que la pile de communication elle-même. Il s'agit essentiellement de composants logiciels, et

- l'aspect *protocole* qui regroupe les fonctionnalités et mécanismes des protocoles de communication. Ici seuls sont considérés les concepts, non l'implémentation qui fait partie de l'aspect système.

L'adéquation des stations de travail face aux évolutions est maintenant discutée suivant cette classification.

2.1.3.1 L'aspect architecture de la machine hôte

Le point clé de la machine hôte est que les technologies des divers composants (processeur, mémoire, caches, bus etc.) évoluent à différentes vitesses. Cette situation conduit à des déséquilibres importants qui se déplacent au cours du temps. Il y a une vingtaine d'années le réseau était

le principal goulot d'étranglement, ce qui a incité à définir des protocoles optimisés pour économiser les bits d'E/S. Ce n'est plus le cas et il est courant de définir des protocoles dont les champs d'en-tête sont alignés sur des mots de 64-bits afin d'en favoriser le traitement. De la même façon on peut supposer que les goulots d'étranglement seront différents dans quelques décennies.

Les traitements sont limités par les accès mémoire

En dépit de techniques élaborées de caches, la technologie des mémoires n'a pas suivi la rapide montée en puissance des processeurs. [Mosberger95] présente une analyse détaillée durant des transferts TCP/IP des événements de bas niveau d'un processeur ALPHA 21064 cadencé à 175 MHz (taux d'échecs lors d'accès aux caches instructions et données, taux d'instructions de lecture, écriture, opérations arithmétiques entières, etc.). Une première conclusion est que, quelle que soit la taille des TSDUs, *ce processeur n'est jamais actif plus de 37% du temps*. Ceci est très loin du taux optimal de deux instructions par cycle.

Une comparaison du comportement du processeur lors du traitement de grosses TSDUs (20 kilo-octets) ou petites TSDUs (1 octet) montre que les économies de manipulation de données possibles avec les petites TSDUs sont compensées par un nombre accru d'échecs d'accès au cache instructions. La deuxième conclusion est que *les traitements sont limités soit par les accès aux instructions, soit par les accès aux données, jamais par le processeur*.

Limitations de la technologie DMA, ou Direct Memory Access

Deux techniques existent pour déplacer des données entre la mémoire de l'hôte et de l'adaptateur réseau :

- le PIO (Programmed I/O) où la copie est prise en charge par le processeur, et
- le DMA (Direct Memory Access) où cette tâche est exécutée par un contrôleur spécialisé.

Le DMA est habituellement préféré au PIO. Une première raison est que le DMA ne requiert qu'un seul transfert sur le bus de la machine alors qu'il en faut deux avec le PIO (lecture puis écriture). Une deuxième raison est que le processeur effectue des traitements utiles parallèlement au DMA. En fait ceci n'est vrai que dans le cas où le processeur utilise ses caches instructions et données. Les accès à la mémoire centrale nécessitent en effet le partage du bus avec le DMA.

Mais l'utilisation du DMA a aussi ses contraintes. Tout d'abord, c'est une technique asynchrone et une interruption est générée à son issue. Ces interruptions sont coûteuses. [Druschel94] rapporte que la prise en charge d'une interruption ajoute 75 µs aux 200 µs requises pour le traitement d'un paquet UDP entrant. La Section 4.2 montre des résultats similaires.

Deuxièmement les buffers systèmes utilisés doivent être verrouillés en mémoire afin d'éviter qu'ils puissent être vidés sur le disque durant le DMA. Comme les buffers systèmes ne possèdent pas par défaut cette propriété, le verrouillage avant DMA puis déverrouillage après DMA doivent être faits explicitement. L'adresse de ces buffers doit également être translatée dans l'espace d'adressage du

bus afin d'être visible par le contrôleur de DMA. Ces opérations de verrouillage et translation d'adresses sont coûteuses.

Troisièmement une restriction du DMA implique que la zone mémoire système d'où ou vers laquelle se fait le transfert soit contiguë. Sur certains systèmes cette zone doit être contiguë en mémoire physique [Druschel94] alors que sur d'autres il est suffisant qu'elle doit être contiguë en mémoire virtuelle[2]. Mais une trame Ethernet sortante est rarement composée d'un unique buffer du fait des ajouts successifs des en-têtes des protocoles.

Enfin, l'utilisation d'un DMA a de grandes conséquences sur le cache de données du processeur. Afin de garantir une cohérence entre le cache de données et la mémoire centrale, avant tout transfert vers ou depuis l'adaptateur, le système d'exploitation invalide de façon sélective certaines lignes du cache. Cette opération est coûteuse peut représenter jusqu'à 48% du temps de traitement de l'interruption lors de la réception d'une trame Ethernet de 1514 octets [Thekkath93a].

Un exemple d'adaptateur Ethernet

L'adaptateur Ethernet hautes-performances d'IBM qui équipe les machines à base de bus Micro-Channel (MCA) RS/6000 (IBM) ou DPX/20 (Bull S.A.) utilise un DMA pour les transferts de données [Tracey94]. Il inclut 16 kilo-octets de mémoire statique, un processeur 16-bits cadencé à 12,5 MHz, et un contrôleur de DMA.

Le driver Ethernet associé emploie des buffers systèmes permanents de transmission et de réception alignés sur des frontières de pages. Ces buffers sont verrouillés en mémoire et leur adresse translatée dans l'espace d'adressage MCA. Une trame Ethernet sortante est d'abord copiée dans le buffer du driver et ensuite transférée par DMA vers le buffer de l'adaptateur. La situation est similaire coté réception (Figure 3). Effectuer une copie de données supplémentaire est dans ce cas plus économique que le verrouillage et la translation d'adresses d'une chaîne arbitraire de buffers [IBM92; p. 3-31].

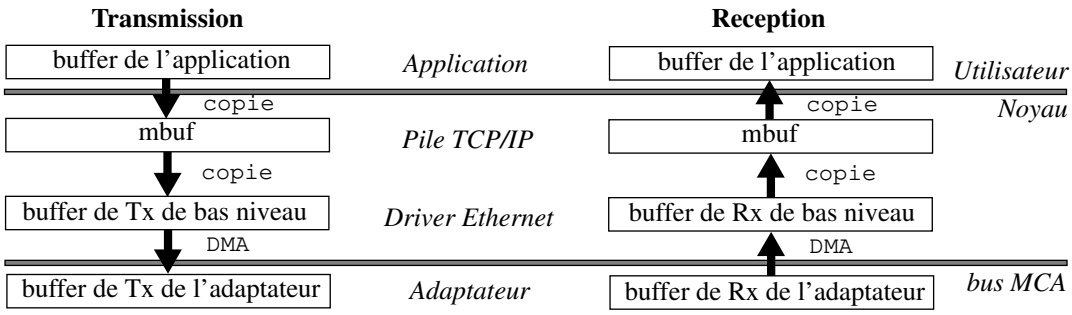

Figure 3: Mouvements de données avec l'adaptateur Ethernet hautes-performances d'IBM.

2. La translation espace virtuel/segments physiques est alors automatiquement gérée par le DMA.

2.1.3.2 L'aspect système

Le principe du 80/20

Les tâches non protocolaires ont toujours été connues comme étant coûteuses. Le principe du 80/20 [Chesson91] affirme que 80% des temps de traitement sont passés sur des tâches systèmes : gestion de buffers, copies de données entre différents domaines de protection, gestion des timers, appels système, interruptions, etc. Cela ne laisse que 20% pour les traitements protocolaires. L'arrivée de réseaux rapides va rendre l'impact négatif du système d'exploitation sur les performances globales de plus en plus évident.

La notion d'entropie appliquée au système d'exploitation

[Montz94] introduit la notion "d'entropie système" afin de souligner le fait que le système d'exploitation et la machine hôte voient leur complexité croître. Des effets imprévus et qui restent parfois inexpliqués peuvent alors résulter de leur réunion. Cela rend une optimisation globale difficile, chaque modification pouvant avoir des répercussions sur les autres composants.

[Pagels94] illustre bien cette complexité. Il explique comment le système d'exploitation, l'utilisation d'un DMA, et la présence ou non du calcul de checksum peuvent influencer le temps d'accès aux données par l'application réceptrice. En effet, un DMA transfert directement les données en mémoire principale sans faire intervenir le cache. Si UDP est utilisé sans checksum, alors la copie de données entre espaces noyau et utilisateur sera pénalisée par un fort taux d'échecs lors d'accès au cache. Sinon ces problèmes surviendront lors du calcul de checksum. Si seule la copie de données est considérée, alors d'importantes variations du temps d'exécution seront observées.

La présence d'une charge de fond, par les impacts qu'occasionnent les changements de contextes sur le comportement des caches [Mogul91], affecte sérieusement les performances. Il n'est pas toujours possible d'éviter ces traitements, de nombreux algorithmes reposant sur des traitements périodiques ou des chiens de garde.

Le support du QoS au sein du système d'exploitation

Le besoin croissant des applications pour un support intégré de la QoS concerne également le système d'exploitation qui par défaut présente nombre de lacunes. Des services fins d'allocation de buffers et de scheduling de threads, plus généralement de réservation de ressources, des APIs étendues permettant la négociation de la QoS, des services de gestion et de surveillance de la QoS sont essentiels. Tous ces aspects sont à la charge du système d'exploitation.

2.1.3.3 L'aspect protocole de communication

Etant au coeur du système de communication, les protocoles jouent un rôle essentiel. Un certain nombre de points les rendent inadaptés aux nouvelles applications et nouveaux réseaux.

Les limites des mécanismes de gestion des pertes et des déséquencements de paquets

La façon dont les protocoles actuels gèrent les pertes et les déséquencements de paquets est incompatible avec certaines classes d'applications [Clark90]. Ainsi une simple perte de paquet conduit TCP à différer la livraison aux applications des toutes les données suivantes tant que la retransmission n'a pas eu lieu. Ceci ajoute à la latence et conduit à une mauvaise utilisation du processeur (les données ne sont pas traitées progressivement lorsqu'elles arrivent mais par à-coups). Si les traitements effectués par les couches hautes sont coûteux (cas d'une couche de présentation), alors ces retards peuvent même empêcher le récepteur de suivre le rythme.

Des problèmes similaires surviennent à chaque fois qu'il y a discordance entre l'unité de transmission et l'unité de recouvrement sur erreurs [Clark90]. Un premier exemple est le mécanisme de fragmentation/réassemblage IP. Cette fonctionnalité est nécessaire lorsque la taille du datagramme que doit transmettre IP est supérieure à la taille maximale permise par le lien (ou Maximum Transmission Unit, MTU). Le datagramme est alors fragmenté, chaque fragment est transmis séparément, et est réassemblé par le destinataire. Les limitations en sont [RFC1191] :

- *des transmissions inefficaces* : dès qu'un fragment est perdu, alors le datagramme entier est jeté et est soit retransmis (TCP), soit définitivement perdu (UDP).

- *des délais supplémentaires* : le datagramme ne peut être traité par le protocole de transport qu'une fois tous ses fragments reçus et réassemblés.

- *un mauvais usage de la mémoire* : chaque fragment est mémorisé par le récepteur en attendant d'être réassemblé. Un mécanisme supplémentaire est requis afin de libérer les vieux fragments.

- *une mauvaise utilisation du processeur* : les données d'un fragment ne peuvent être traitées immédiatement.

Un deuxième exemple est le mauvais comportement du trafic TCP ou UDP au dessus d'ATM lors de congestions. Parce que l'unité de transmission est la cellule de 48 octets alors que l'unité de contrôle est le datagramme IP, une simple perte de cellule conduit à la perte du datagramme dans sa totalité [Romanow94].

Les limites des mécanismes de diffusion dans les protocoles

Le protocole IP a été étendu [RFC1112] afin de fournir un service de diffusion (multicast), c'est-à-dire la possibilité de délivrer un datagramme IP à une liste de récepteurs. Deux types d'applications peuvent bénéficier de ce service [Stevens94] :

- les applications qui doivent délivrer des données à plusieurs récepteurs, telles que les applications de vidéo-conférence, et

- les clients qui sollicitent un serveur non connu, par exemple durant la phase de démarrage

d'une station sans disque. Si RARP et BOOTP [Stevens94] utilisent actuellement un service de diffusion générale (broadcast), une solution à base de diffusion est préférable car elle ne perturbe pas les hôtes qui n'y participent pas. C'est la solution retenue dans IPv6 [Narten95].

Mais la diffusion a des limites. Parce que les protocoles de transport connectés classiques ont été définis pour un service point-à-point, la diffusion IP n'est disponible qu'avec le protocole UDP. Celui ci n'ayant aucun mécanisme de contrôle d'erreurs, les problèmes sont évités. Mais une application ayant à la fois besoin de diffusion et de garanties de livraison n'a d'autres recours que d'établir une connexion par destinataire. Ceci ajoute en complexité (gestion de plusieurs connexions) et en inefficacité (transmissions multiples des mêmes informations). Ce problème de fiabilité, pour être résolu, requiert une refonte totale des protocoles de transport.

Le manque d'informations de rétroaction

Tout d'abord les applications ne reçoivent pas d'informations concernant l'état courant du réseau (congestion, etc.). Ceci est une conséquence du modèle OSI pour lequel chaque couche est indépendante de ses voisines. Ensuite, comme TCP n'est pas adapté au temps-réel et est incompatible avec le service de diffusion, UDP lui est souvent préféré. Le problème est que UDP ne possède aucun mécanisme de contrôle de flux ou de débit.

Ces deux points font que les nouvelles applications implémentées classiquement peuvent conduire à des transmissions non contrôlées au dessus de UDP. Celles-ci peuvent alors aisément submerger le réseau, conduisant à des congestions et à un service inacceptable pour tous [Bolot94, Diot95a].

Les limites du support du QoS

La première limitation au support de la QoS provient du multiplexage des divers flux de données de la couche N en un unique flux dans la couche N-1, ce qui a pour effet de cacher les caractéristiques des flux au niveau inférieur. Cette perte d'informations entre couches de protocoles fait que les PDUs de niveau N-1 sont généralement traitées de façon égalitaire, sans tenir compte des besoins identifiés par les applications [Feldmeier90].

Table 1: Valeurs recommandées pour le champ TOS de l'en-tête IP (extrait).

Application		**Minimiser le délai**	**Maximiser le débit**	**Maximiser la fiabilité**	**Minimiser les coûts**
telnet/rlogin		oui	non	non	non
FTP	contrôle	oui	non	non	non
	données	non	oui	non	non
SMTP (mail)	commandes	oui	non	non	non
	données	non	oui	non	non
DNS	requête UDP	oui	non	non	non
	requête TCP	non	non	non	non
	transfert de zone	non	oui	non	non
NNTP (Usenet news)		non	non	non	oui

Afin de limiter cette perte d'informations, le protocole IP définit dans son en-tête un champ de type de service (TOS) composé de quatre bits : délai, débit, fiabilité, et coût financier (Table 1). Son but est de permettre aux couches supérieures de donner des indications (limitées) sur la façon dont IP doit router les paquets [RFC1349].

Plusieurs limitations font de ce mécanisme une solution partielle [RFC1349] :

- Il y a des *limitations intrinsèques* : c'est un mécanisme de recommandations qui ne fournit aucune garantie. Il ne permet pas de spécifier le niveau de service exact désiré, et il n'existe pas forcement de route disposant du TOS voulu.

- Il y a des *limitations pratiques* : ce mécanisme nécessite que chaque administrateur ait configuré ses routeurs de façon adéquate ce qui n'est pas garanti (volontairement ou accidentellement). Enfin il n'est pas supporté par tous les routeurs.

Les protocoles OSI proposent une autre approche de la QoS. Le service de transport définit des paramètres quantitatifs en terme de délai, débit, taux d'erreurs et probabilité de violation. Ces paramètres ont cependant une nature statique puisqu'ils ne peuvent pas être renégociés [Campbell94, Fdida94]. Si le réseau ne peut maintenir ses engagements initiaux, alors l'utilisateur n'a d'autre solution que de fermer la connexion pour éventuellement la rouvrir avec des paramètres moins élevés. Une deuxième limite est qu'aucun contrôle de la qualité de service disponible n'est exigé de la part du système de communication. La seule façon qu'a un utilisateur de connaître le niveau de service disponible est d'effectuer les mesures lui même. Ce service de QoS est donc très limité. Il vise essentiellement à permettre au fournisseur de services une configuration plus précise du système de communication.

Bilan sur l'aspect protocole de communication

Cette discussion sur les aspects protocolaires souligne le manque d'adaptabilité et le support limité de mécanismes évolués tels que la diffusion et la QoS au sein des protocoles actuels. Le principe d'indépendance du modèle OSI montre également ses limites pour la réalisation de systèmes de communication efficaces.

2.2 APPROCHES POUR LA CONCEPTION DE SYSTÈMES DE COMMUNICATION HAUTES PERFORMANCES

Plusieurs techniques ont été proposées pour faire face aux problèmes que nous venons d'identifier. Elles peuvent être classées en quatre catégories :

- l'adaptation mutuelle des composants du système de communication,

- l'optimisation des mécanismes de contrôle des protocoles,

- l'optimisation des manipulations de données et des accès mémoire, et
- l'accroissement des traitements par le parallélisme.

2.2.1 Adaptation mutuelle des composants du système de communication

Trois composants peuvent être identifiés : *l'application, le système de communication, et le réseau*.

Il est nécessaire que ces trois composants s'adaptent les uns aux autres. Ceci est déjà partiellement fait. Ainsi le système de communication s'adapte à la charge réseau par le mécanisme de "slow-start" de TCP, le contrôle de débit d'XTP, etc. La notion de QoS permet quant à elle au réseau et au système de communication de s'adapter aux désirs de l'application. ST-II [RFC1190], RSVP [RFC1633], et la pile de protocoles Tenet [Ferrari92, DiGenova95] sont trois approches pour le support de la QoS au sein des protocoles.

Mais d'autres formes d'adaptation plus innovatrices existent :

L'adaptation du système de communication à l'application

Le système de communication peut s'adapter à l'application. [Diot95b] montre une approche pour la conception et l'implémentation automatique de protocoles de communication adaptés à l'application. L'idée est de composer un protocole intégré à l'application qui n'inclut que les fonctionnalités requises par celle-ci.

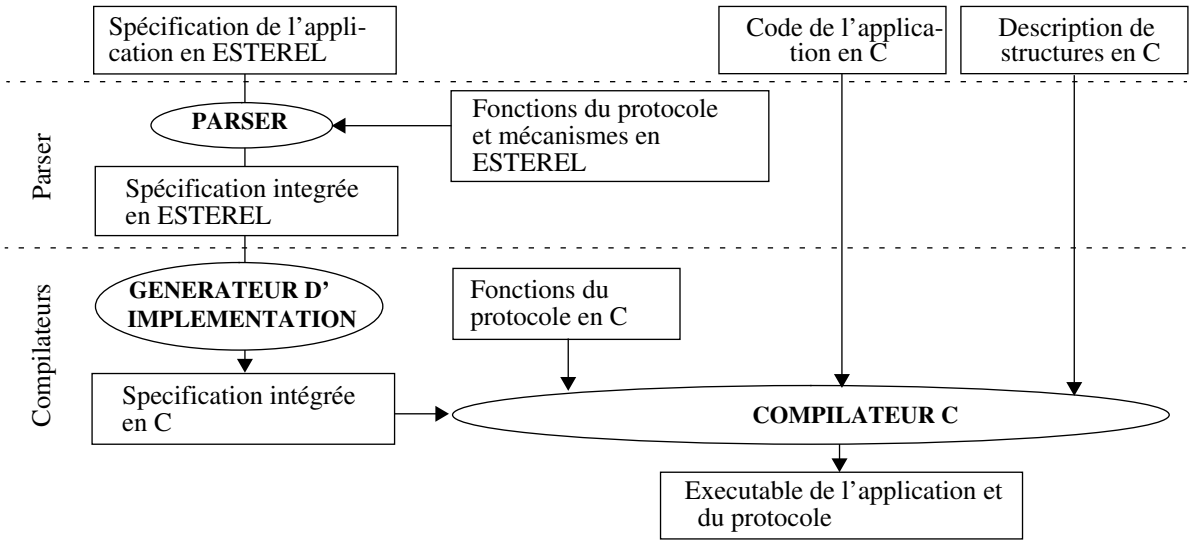

Figure 4: Architecture du compilateur de protocoles.

Une description formelle de l'application et de ses besoins est créée et envoyée à un outil, le compilateur de protocoles (Figure 4). Ce dernier assemble alors les fonctions implémentant les différents

mécanismes désirés et intègre ce protocole de synthèse directement dans l'application. Toute intervention humaine est évitée durant ces étapes. ESTEREL, un langage synchrone et réactif dédié aux descriptions de systèmes temps-réels, a été choisi comme langage de spécification [Diot94].

[Richards95] décrit un autre système d'adaptation du protocole de transport. Ce système, qui s'exécute au niveau utilisateur, est construit *dynamiquement* en prenant en compte les besoins de QoS locaux et distants. Le moteur assurant cette synthèse utilise une librairie de fonctions de protocoles. Certaines fonctions, indispensables, sont systématiquement incluses, alors que d'autres, optionnelles, sont négociées durant l'ouverture de connexion. Cette approche diffère de l'approche statique de [Diot95b] où le protocole est compilé avec l'application.

Il existe d'autres formes d'adaptation à l'application. [Edwards94, Edwards95] décrivent les cartes réseau hautes performances Jetstream/Afterburner. La méthode d'accès de bas niveau à la carte repose sur la notion de *pool*. Un pool est constitué d'un jeu de buffers résidants sur la carte et affectés à un flux de données unique, plus un jeu de primitive permettant d'allouer/libérer ces buffers, de copier les données depuis ou vers ces buffers, d'associer au pool un identificateur de canal virtuel (VCI), etc.

Le point clé de cette méthode d'accès est son extrême *flexibilité*. Une application peut ainsi décrire les opérations qu'elle désire voir appliquer à ses flux de données entrants et sortants. Ceci va bien au delà du modèle traditionnel de `send`/`receive` des méthodes d'accès Socket ou XTI (Section 3.3). Ainsi une application vidéo peut traiter des images sans avoir à lire les données : seuls quelques octets de contrôle sont examinés, les données étant pour leur part dirigées directement vers le système d'affichage.

L'adaptation de l'application au réseau

L'application doit s'adapter au réseau, c'est-à-dire à la fois à l'environnement très variable existant, depuis les liens hauts débits jusqu'aux réseaux sans fils, mais aussi à la qualité de service disponible à tout moment. C'est le concept d'application consciente du réseau, ou NCA (Network Concious Application) [Diot95]. Les applications multimedia peuvent facilement s'adapter :

- *aux variations de délai, ou gigue* : les applications qui jouent localement une séquence multimedia collectée à distance peuvent éliminer la gigue en introduisant un décalage (Section 2.1.2). Les données arrivant dans les limites fixées par ce décalage pourront ainsi être utilisées. Afin de faire face à la très grande variabilité de la gigue sur les réseaux longues distances tels que l'Internet, la valeur de ce paramètre de décalage peut elle même s'adapter à la gigue courante.

- *aux variations de débit* : si le bande passante devient rare, alors le flux de l'application doit être adapté afin que seules les données essentielles soient transmises. Cette stratégie d'adaptation au débit vise à limiter les problèmes de congestion sur le réseau (se comporter en "bon citoyen réseau") et à donner à l'application la possibilité de choisir ce qui sera transmis avec une certaine chance d'arriver à destination.

Cette adaptation peut se faire en jouant sur le taux de compression. Ainsi pour l'application de vidéo-conférence de l'Inria (IVS) [Bolot94], trois paramètres peuvent être ajustés : la fréquence de rafraîchissement, la qualité d'image, et le niveau de détection des mouvements. La technique d'encodage hiérarchique, disponible avec certains algorithmes de compression par ondelettes, est une autre technique d'adaptation. Chaque image est décomposée en plusieurs paquets qui fournissent chacun un niveau de définition accru. On attribue alors à ces paquets un niveau de priorité inversement proportionnels au niveau de définition. Ainsi en cas d'engorgement seront perdus préférentiellement les détails de l'image.

Une difficulté majeure de ces techniques consiste à recevoir l'information de rétroaction. En effet ces applications reposent souvent sur une diffusion de l'information. Solliciter des informations d'un groupe de récepteurs de taille à priori inconnue peut aisément créer un problème d'implosion. Pour l'éviter, des mécanismes spéciaux doivent être utilisées [Bolot94].

Un autre problème lié à l'utilisation de la diffusion est pour l'émetteur d'estimer le nombre de récepteurs connaissant des problèmes et de décider comment ajuster le flux de données en conséquence [Bolot94]. Doit-il privilégier les récepteurs ayant des problèmes, ou bien les autres?

2.2.2 Optimisation des mécanismes de contrôle des protocoles

Deux approches existent pour optimiser les mécanismes de contrôle des protocoles : soit on utilise mieux les protocoles existants, soit on améliore leurs mécanismes.

Un meilleur usage des protocoles existants : la technique ALF

La technique ALF (Application Level Framing) [Clark90], qui prend en compte les limites des protocoles existants, est particulièrement adaptée aux applications ayant des contraintes temps réel. La Section 2.1.3.3 a discuté des problèmes issus d'une discordance entre unités de transmission et de contrôle d'erreurs. La technique ALF soutient que les unités de données de l'application (ou ADU, Application Data Units) :

- doivent être égales aux unités de transmission, et
- doivent être à la base du contrôle d'erreurs.

Ce contrôle d'erreurs est désormais du ressort de l'application qui est la mieux placée pour savoir quelle politique appliquer. Une conséquence est que l'application émettrice doit inclure suffisamment d'informations dans une ADU pour permettre à l'application réceptrice de la traiter qu'il y ait ou non des erreurs de transmission. Ainsi une ADU contenant un fragment d'image doit inclure des informations de positionnement dans l'image afin de permettre à l'application réceptrice de la traiter même en cas de déséquencement.

Un problème cependant concerne les relations entre la taille des ADUs et la taille maximale des unités de transmission pour un chemin donné (PMTU). La taille des ADUs est liée à l'application.

C'est la taille minimale permettant à l'application de faire face à d'éventuelles erreurs. Le PMTU est au contraire lié au réseau et au(x) destinataire(s) des ADUs. Si la taille des ADUs est inférieure au PMTU, alors elles seront chacune transmises dans un seul datagramme IP et le principe ALF sera respecté. Si elle est supérieure au PMTU, alors ce n'est plus possible et un mécanisme de fragmentation doit être utilisé avec toutes les limitations que cela implique. Enfin, construire des agrégats d'ADUs peut s'avérer être très efficace dans le cas où celles-ci sont petites par rapport au PMTU. [Chrisment94] décrit un outil de transfert d'images dont les ADUs, d'une taille de 60 octets, se prêtent bien à cette technique d'agrégation.

L'optimisation des mécanismes de protocole

Les travaux sur les protocoles de communication sont essentiels. Ils incluent l'*évolution des protocoles existants*. [RFC1323] est la plus populaire des évolutions hautes performances de TCP pour les situations où le produit <bande passante*RTT> est élevé. Elle permet alors à l'émetteur de remplir le "tuyau de communication" et d'aller au delà des 64 kilo-octets de données en transit permis par les 16 bits du champ fenêtre de l'en-tête TCP. [RFC1644] est un autre exemple qui rend TCP mieux adapté aux services de transaction. Ainsi des transactions avec échange de trois segments seulement sont rendues possibles en évitant les poignées de main triples de TCP.

Parce que l'évolution des protocoles existants a des limites, la solution ultime est de *concevoir de nouveaux protocoles*. Ainsi IPv6 [Deering95] est le nouveau protocole réseau IP. Par rapport à la version 4 actuelle il possède : des capacités d'adressage étendues (128 bits) qui permettent également un routage plus performant (structures d'adresses hiérarchiques), des en-têtes simplifiés qui favorisent les traitements dans le cas général, un meilleur support des options, la possibilité d'associer un identificateur de flux à certains paquets afin qu'ils bénéficient de traitements privilégiés (va bien au delà du champ TOS d'IPv4), et finalement des possibilités d'authentification et de chiffrage. Toutes ces nouvelles possibilités rendent IPv6 mieux adapté que son prédécesseur aux nouveaux besoins.

XTP (ou eXpress Transport Protocol) [XTP95, Weaver94] représente une nouvelle génération de protocoles de transport. Le principe de base de ce protocole est de proposer de nombreux mécanismes tels que le multicast, le contrôle de débit, différentes stratégies de recouvrement sur erreur, etc., mais de n'en imposer aucune. Cet aspect, qui rejoint la discussion de la Section 2.2.1, le rend bien adapté à une large gamme de situations. Ce protocole a également été conçu pour favoriser les implémentations logicielles et matérielles par des techniques telles que des en-têtes de taille fixe (discuté dans la Section 5.3), des champs alignés sur des mots de 64 bits, des échanges de clés, la distinction entre paquets de contrôle ou de données, la distinction entre en-tête et queue (jusqu'à la version 3.6 [XTP92]), et des machines d'états finis simplifiées.

Enfin, RMTP (ou Reliable Multicast Transport Protocol) [RMTP95] est un protocole de transport orienté connexion permettant la diffusion fiable d'informations avec le protocole IP. Il pallie ainsi la limite actuelle où seul un service sans garanties (UDP) est possible avec la diffusion IP (Section 2.1.3.3).

Les intérêts d'un point de multiplexage unique

Plusieurs motivations poussent à des architectures de protocoles démultiplexées[3], en particulier pour le support de la QoS. Nous avons montré à la Section 2.1.3.3 que multiplexer conduit à perdre des informations dans les couches inférieures. La technique de classification des paquets [Wakeman95] est une solution coûteuse (car elle conduit à analyser tous les en-têtes) destinée à récupérer cette information. [Feldmeier90, Feldmeier93b] soulignent l'intérêt qu'il y a à démultiplexer le plus bas possible au sein de la pile de protocoles. L'identificateur de flux d'IPv6 et les VCI d'ATM lorsqu'ils sont affectés sur la base des connexions, sont deux façons de limiter le nombre de points de multiplexage dans la pile de communication.

2.2.3 Optimisation des manipulations de données et des accès mémoire

2.2.3.1 Moins de manipulations de données

Le déséquilibre de performances entre mémoire et processeur a incité à optimiser les manipulations de données au sein des piles de protocoles (copies de données, calcul de checksum, chiffrage, opérations de présentation, etc.). Deux voies principales ont été suivies :

- la technique ILP, et
- les systèmes de communication à copies de données minimales.

La technique ILP

La technique ILP (Integrated Layer Processing) [Clark90] vise à éviter les accès mémoires et à mieux utiliser le cache de données. Ceci passe par l'intégration de toutes les opérations de manipulation de données en une seule et unique boucle (Figure 5).

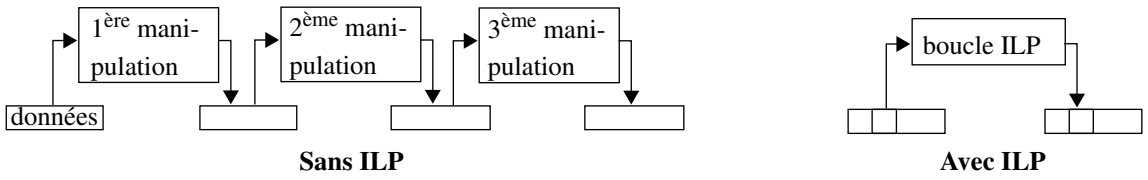

Figure 5: Le principe d'intégration des manipulations de données d'ILP.

Une implémentation ILP se doit ainsi de lire une seule fois les données, de les stocker dans les registres du processeur si possible, sinon de les conserver dans le cache du processeur, d'y appliquer toutes les manipulations requises, et enfin d'écrire les données transformées dans le

3. Nous ne traitons ici que de l'aspect multiplexage dans les protocoles, l'aspect multiplexage dans les implémentations de piles de protocoles sera étudié à la Section 3.5.1.

buffer destination. Dans ce cas idéal seuls deux accès mémoire suffisent pour chaque unité de traitement, habituellement un mot de 32 ou 64 bits. Ceci est à opposer à une implémentation classique où les manipulations sont appliquées les unes après les autres, les données intermédiaires étant à chaque fois lues puis écrites en mémoire.

En appliquant la technique ILP à des manipulations de données simples telles que les copies et le calcul de checksum, des gains de performances significatifs ont été obtenus [Clark89, Clark90]. Ces gains augmentent (théoriquement) avec le nombre de manipulations de données impliquées [Abbott93]. D'autres expériences montrent que des manipulations complexes telles que le DES (Data Encryption Standard) peuvent par contre masquer les bénéfices d'ILP [Gunningberg91]. Ces aspects seront discutés à la Section 5.1.2.

Enfin notons que la règle ALF spécifiant que ADUs et unités de transmission doivent être identiques est un prérequis lorsque ILP intègre le calcul du checksum, celui-ci portant sur le segment TCP. Nous verrons à la Section 3.5.3 comment ceci peut être garanti.

Les systèmes de communication à copies de données minimales

Une autre approche évitant les accès mémoires consiste à réduire le nombre de copies de données à son minimum, à savoir le DMA entre les buffers de l'application et la mémoire de la carte d'adaptation, puis entre la mémoire de la carte et le réseau. Plusieurs techniques existent :

[Druschel93] décrit un système de buffers appelé `fbuf` qui permet un transfert de données sans copie entre différents domaines de protection. Il repose sur une combinaison des techniques de translation de pages mémoires virtuelles et de mémoire partagée. Une abstraction telle que les `mbufs` de BSD (Section 3.3.1) peut alors facilement être superposée aux `fbufs`. Parce que différentes optimisations sont associées (localité du trafic d'E/S, utilisation de caches, gestion intégrée `mbuf/fbuf`, notion de `fbufs` volatiles, etc.), les performances des `fbufs` sont élevées, même dans le cas de petites TSDUs où il est habituellement moins coûteux de copier les données. Enfin, contrairement à d'autres techniques (cf. ci-dessous), l'architecture du système de communication n'est pas globalement remise en cause.

[Jacobson93] décrit un autre système de buffers appelé `pbuf`. Ces buffers sont alloués dans la mémoire double-accès d'une carte réseau, ce qui évite tout recours au DMA pour effectuer des transferts de données entre la carte et les buffers systèmes. C'est la technique qui a été retenue pour la carte Afterburner [Edwards94, Edwards95].

Les implémentations de niveau utilisateur de la pile de communication constituent un quatrième exemple. Historiquement elles ont été conçues pour résoudre les problèmes de performances des systèmes d'exploitation à base de micro-noyaux [Maeda93]. Du fait de leur architecture triangulaire (driver réseau dans le noyau, serveur UNIX dans l'espace utilisateur, et application), les données traversent trois espaces d'adressage au lieu de deux avec les implémentations monolithiques traditionnelles. En intégrant le système de communication directement dans l'application, le serveur UNIX est évité et le nombre de copies est ramené à deux.

Une dernière technique qui va plus loin consiste à transférer les données directement de l'adaptateur dans l'espace d'adressage de l'application [Ahlgreen93, Biersack94]. Cette architecture, devenue très populaire avec l'arrivée d'ATM, a deux implications : (1) le système de communication dans sa totalité doit résider dans l'espace utilisateur, et (2) le démultiplexage des paquets entrants doit être effectué très bas. Ce deuxième point est habituellement réalisé en exploitant un VCI dans le cas d'un réseau ATM, Jetstream [Edwards95], ou AN1 [Thekkath93]. Leur utilisation est donc limitée à certains types de réseaux et suppose de surcroît que les VCIs soient nombreux et puissent être alloués librement sur la base des connexions. [Cole95] discute cette notion de "VCI par connexion".

De façon générale, les systèmes de communication à copies minimales ont certaines limites. Nombre d'entre eux reposent sur des implémentations au niveau utilisateur d'une partie voir de la totalité de la pile de protocoles. Ceci est à la source de nombreux problèmes, particulièrement avec les protocoles orientés connexion tels que TCP (nouvelles APIs, réorganisation de l'implémentation, etc.) (Section 3.5.2). Enfin, notons que la suppression de la copie entre espaces utilisateur et noyau empêche son intégration avec le calcul de checksum. Les techniques ILP et de copies minimales sont à un certain point mutuellement exclusives.

2.2.3.2 Meilleure utilisation des caches

Une autre approche pour la minimisation du déséquilibre de performances entre mémoire et processeur consiste à améliorer le comportement du cache instructions, c'est-à-dire le taux de succès des accès au cache.

La technique usuelle est "l'inlining" qui consiste à inclure le code des fonctions brèves dans la fonction appelante plutôt que d'effectuer un appel de fonction. Cependant [Mosberger95] souligne que cela consiste en un compromis non trivial entre localité spatiale et taille du code.

Deux autres techniques existent :

- une approche compilateur, et
- une approche dynamique.

L'approche compilateur

Cette approche comporte deux étapes : l'identification du chemin de données principal tout d'abord, qui peut être faite de deux façons différentes :

- une première solution repose sur des extensions au langage afin de spécifier si un test est anticipé pour être vrai ou faux [Mosberger95]. Cela permet d'identifier aisément le code correspondant aux cas d'erreurs. Des mesures ayant montré que 50% du code système consiste en des traitements d'erreurs, cette technique peut s'avérer efficace. De plus cette approche est conservatrice dans la mesure où seuls les tests annotés sont considérés.

- Une autre solution est l'utilisation de la technique de "profiling" [Speer94]. La version non optimisée de l'exécutable est lancée avec des suites de tests significatives. Les informations de profiling sont collectées pendant ces exécutions. Contrairement à la précédente solution le code n'est pas modifié. Cependant cette technique est agressive car tout code non couvert par la suite de tests est considéré comme n'appartenant pas au chemin de données principal.

Dans un deuxième temps ces informations sont utilisées afin d'améliorer automatiquement la localité spatiale du code de l'exécutable, c'est-à-dire de garantir que le code du chemin de données principal soit consécutif en mémoire. Deux techniques existent :

- la technique "d'outlining" : les blocs (séquences de code sans instruction de branchement) rarement utilisés sont déplacés en fin de fonction et les branchements conditionnels modifiés afin de favoriser le cas courant.

- le repositionnement des procédures (technique de profiling) : on fait en sorte que les procédures couramment utilisées soient contiguës. Cela peut être vu comme l'application de "l'outlining" aux procédures plutôt qu'aux blocs.

L'approche dynamique

[Montz94] décrit un dispositif de déplacement dynamique de blocs qui permet au code du chemin critique d'être regroupé dans des régions mémoire contiguës pour un comportement du cache instructions optimal.

2.2.4 Accroissement des traitements par le parallélisme

L'utilisation d'une plate-forme multiprocesseur est une autre façon d'augmenter les performances. Deux facteurs limitants existent cependant :

- *la synchronisation* : un mécanisme de synchronisation est nécessaire afin de sérialiser les accès aux objets partagés (buffers, queues, etc.). Si ce besoin existe déjà sur un monoprocesseur (synchronisation thread-routine d'interruption et thread-thread dans le cas d'un système d'exploitation préemptible), ce besoin est exacerbé sur un multiprocesseur.

- *les changements de contextes* : l'importance des coûts de changement de contextes est plus grande sur un multiprocesseur que sur un monoprocesseur. Ceci résulte des mécanismes de synchronisation et de l'architecture du système de communication comme nous allons le voir.

Ainsi la condition pour bénéficier d'une amélioration de performances est que le taux d'accélération lié au parallélisme compense les surcoûts plus élevés dus à la synchronisation et aux changements de contextes.

Un support approprié au niveau du processeur

Les instructions atomiques `test-and-set` et `compare-and-swap` des processeurs CISC ont longtemps été à la base des mécanismes de synchronisation élémentaires de type verrous. La prise du verrou consiste en une boucle avec, selon la catégorie de verrou, soit attente active, soit blocage lorsque celui-ci n'est pas disponible. Les problèmes [Chesson94] sont que ces stratégies ont une scalabilité limitée, conduisent à des attentes importantes lorsque la thread qui détient le verrou est préemptée, et enfin imposent que la structure de données verrou ne soit pas dans les caches mémoire.

Les processeurs RISC modernes tels que le PowerPC ont remplacé ces instructions atomiques par le `load-and-reserve` et le `store-conditional` [Chesson94, IBM94, Talbot95]. Ces instructions reposent sur la notion de réservation établie lors du chargement ("load"). Si la réservation n'est pas rompue[4], alors l'écriture ("store") est validée, sinon on recommence le cycle chargement/écriture. Le fragment de code suivant en illustre l'utilisation :

```
nouv_essais: load-and-reserve reg, adr  // Charge le registre et
                                        // établit la réservation.
             <traitements utiles sur reg...>
             store-conditional reg, adr // Essaie d'écrire le résultat
                                        // en mémoire. Modifie le
                                        // registre conditionnel pour
                                        // indiquer un échec/réussite.
             branch-false nouvel_essais // Recommence si besoin.
```

Cette technique a ceci d'intéressant qu'elle permet d'insérer des traitements utiles entre les phases de chargement et d'écriture. Cette propriété permet de s'en servir pour construire à la fois les mécanismes de synchronisation classiques (verrous, etc.), mais aussi pour des manipulations élaborées (compteur, listes chaînées, etc.) [Chesson94]. La synchronisation étant intégrée aux traitements à sérialiser, son coût est moins élevé qu'avec la technique classique qui repose sur des mécanismes extérieurs. On qualifie souvent cette technique d'optimiste car l'on parie qu'il n'y aura pas de conflit, quitte à recommencer si besoin est.

Un usage approprié des mécanismes de chargement/écriture conditionnelle (compteurs de statistiques TCP/IP, gestion mémoire, etc.) permet donc de *limiter l'utilisation de verrous et d'améliorer les performances* du système de communication.

Les divers types de parallélisme

[Schmidt94] identifie deux types fondamentaux de parallélisme :

- *le parallélisme de type tâche* : chaque tâche protocolaire est prise en charge par une thread dédiée. Selon la granularité de la tâche on peut raffiner en :

4. Une réservation est perdue si un autre processeur accède le même mot mémoire entre temps.

    - parallélisme par couches où chaque protocole est attaché à une thread, et
    - parallélisme fonctionnel où les protocoles sont décomposés en diverses fonctions (traitements des en-têtes, acquittements, retransmissions, etc.) qui sont chacune attachées à une thread.

- *le parallélisme de type message* : chaque message de données ou de contrôle est pris en charge par une thread particulière. Ici aussi, selon la façon dont les messages sont attachés aux threads, on peut raffiner en :
    - parallélisme par connexion où tous les messages destinés à un contexte donné sont pris en charge par une même thread (thread par connexion), et
    - parallélisme par message où chaque message est traité par une thread quelconque (thread par message).

L'impact du type de parallélisme sur les performances

Ces quatre types de parallélisme ont un grand impact sur les performances. [Schmidt94] montre que le parallélisme par couches conduit à des coûts de synchronisation et de changement de contextes élevés car ceux-ci peuvent survenir lors de chaque passage de message à la couche suivante. Les traitements sont également limités par le protocole le plus lent. Enfin, la granularité du parallélisme est trop fine vu les temps de traitement réduits des protocoles [Bjorkman93a].

Le cas du parallélisme fonctionnel est similaire. A ces problèmes s'ajoute le fait que partitionner le protocole pour obtenir une charge équilibrée entre les threads est une tâche ardue [Rutsche92]. Ceci est d'autant plus vrai que les protocoles de communication ont rarement été conçus dans ce but. Utiliser des protocoles plus adaptés tels que XTP, disposant de machines d'états finis adéquates, peut améliorer la situation [Braun92, Braun94].

Le parallélisme de type message (par connexion ou par message) au contraire utilise mieux les processeurs [Schmidt94]. Parce que ce type de parallélisme repose sur des composants dynamiques (connections ou messages), il s'adapte facilement au nombre de processeurs disponibles. De plus les messages sont entièrement traités par une unique thread ce qui limite les changements de contextes.

[Chesson94] discute plus en détails les versions connexion ou message. Le premier a l'avantage de minimiser les conflits de synchronisation (une seule thread peut être active pour un contexte donné) et de conduire à un bon comportement des caches (meilleure affinité des traitements vis-à-vis des processeurs). Cependant le parallélisme au sein d'une connexion donnée est nul. Le parallélisme de type message au contraire offre un parallélisme maximal puisqu'une connexion peut potentiellement bénéficier de tous les processeurs. Par contre il conduit à des besoins de synchro-

nisation plus élevés puisqu'il faut synchroniser ces différentes threads entre elles.

Table 2: Taux d'accélération.

Nombre de processeurs	**UDP**	**TCP**
2	1.95	1.79
4	3.85	2.79
6	5.20	3.35
8	6.90	3.39

Enfin des essais ont montré qu'une pile TCP/IP a une scalabilité limitée, même avec un parallélisme de type message (Table 2 extraite de [Bjorkman93b]). La scalabilité de TCP est bien en deçà du taux d'accélération optimal (nombre de processeurs). La situation est meilleure avec UDP, celui ci ayant moins de ressources à partager du fait de son caractère non-connecté.

CHAPITRE 3

Notre plate-forme expérimentale

Nous avons identifié dans la section précédente les principales directions prises dans le domaine des réseaux hautes performances. A la lumière de cette étude nous décrivons maintenant les différents composants de notre plate-forme.

3.1 PRÉSENTATION DE NOS TRAVAUX

3.1.1 Les points abordés par ces travaux

Nous avons vu au chapitre précédent qu'il existe de nombreuses façons d'aborder le problème de la réalisation d'un système de communication hautes performances. Nous avons orienté nos travaux de façon à étudier un certain nombre d'aspects qui nous semblent importants, à savoir :

Les impacts du système d'exploitation et de l'environnement d'implémentation/exécution

Le but de l'environnement d'exécution est de fournir des services de base et des concepts architecturaux qui facilitent le développement des drivers de communication. Nous nous intéresserons aux deux environnements majeurs des systèmes ouverts : BSD et Streams. Cette comparaison a ceci d'intéressant que ces environnements reposent sur deux concepts radicalement opposés : une communication par appels de fonctions pour BSD, et une communication par passage de messages

pour Streams. Nous voulons apprécier l'ensemble des conséquences que peut avoir le choix de l'un ou l'autre de ces environnement.

Les impacts de l'architecture du système de communication

Des alternatives autres que la traditionnelle structure en couches propre à la plupart des systèmes de communication sont possibles. Nous nous intéresserons en particulier aux architectures démultiplexées et aux implémentations au niveau utilisateur de protocoles de communication. Notre objectif est d'obtenir une vision pratique des différents avantages et limites associés à ces alternatives. Plus particulièrement nous mettrons en évidence leurs impacts sur la complexité du chemin de données, sur le contrôle de flux, sur le support de la QoS, etc.

Les impacts du parallélisme

Les machines multiprocesseurs se généralisant, nous nous devons d'étudier le comportement des différentes alternatives proposées. Nous accorderons un intérêt tout particulier aux problèmes de scalabilité et de contentions d'accès aux verrous.

Les impacts de ILP

Si la technique ILP d'intégration de manipulations de données est du point de vue théorique simple, en revanche son incorporation dans le système de communication est complexe. Ceci est la conséquence de la remise en cause la structuration en couches que ILP impose, et de divers problèmes techniques qui s'ensuivent. Pour notre part nous nous intéresserons principalement à l'intégration des copies de données et du calcul de checksum TCP/UDP. Notre but est de montrer comment l'incorporer au système de communication, et quels sont les bénéfices et les limites que l'on peut en attendre dans des conditions réelles.

3.1.2 Présentation de notre plate-forme expérimentale

Afin d'étudier ces divers points nous avons conçu une plate-forme expérimentale (Figure 6) composée de :

- deux environnements différents : BSD et Streams,

- plusieurs piles de communication : nous avons évalué une large gamme d'architectures comprenant des implémentations TCP/IP classiques en environnement BSD et Streams, une implémentation démultiplexée sous Streams et enfin une implémentation de niveau utilisateur de TCP,

- plusieurs méthodes d'accès : chaque pile de communication est attaquée par sa méthode d'accès et/ou API privilégiées, à savoir les Sockets dans le cas de BSD, et XTI/TLI dans le cas de Streams. XTI a également été modifiée afin d'intégrer des aspects ILP dans le cas de l'implémentation Streams démultiplexée.

- un environnement intégré d'évaluation de performances que nous avons conçu autour des applications bench/benchd, et enfin

- plusieurs interfaces physiques : rebouclé mémoire (loopback), utilisé dans la plupart des tests, Ethernet et FDDI (pile BSD seulement). Il est à noter que le driver Ethernet est intégré à un environnement BSD. Son utilisation au sein d'une pile Streams rend nécessaire une adaptation BSD/Streams qui consiste en une encapsulation `mbuf`/`mblk`. Si ceci est pénalisant, ce n'est pas spécifique à Streams et les surcoûts associés doivent ainsi être ôtés.

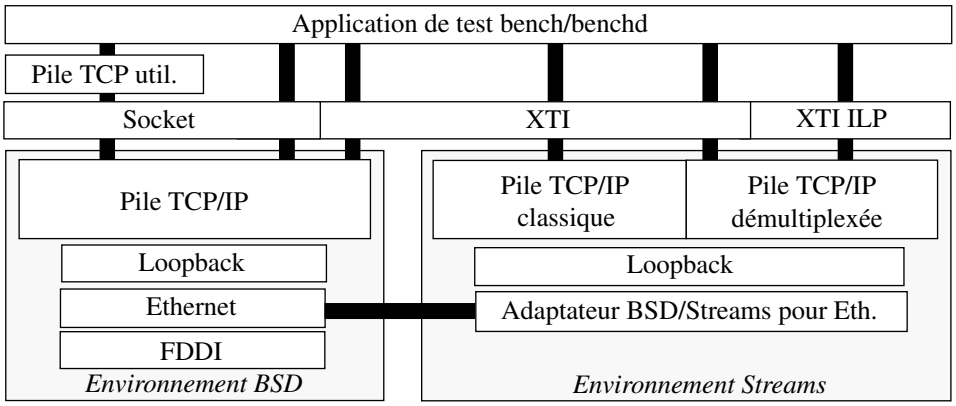

Figure 6: Notre plate-forme expérimentale.

3.2 LA MACHINE HÔTE

La plate-forme a été conçue pour fonctionner sur la gamme de machines DPX/20, ESCALA et ESTRELA de Bull S.A., fondée sur la famille de microprocesseurs POWER, et disposant d'un système d'exploitation UNIX AIX/3.2.5 ou AIX/4.1.2. Cette section décrit les points clés de ces machines. La carte réseau Ethernet a pour sa part été décrite à la Section 2.1.3.1.

3.2.1 La famille POWER de microprocesseurs

La famille POWER^{tm} de microprocesseurs RISC, née d'une collaboration entre Motorola, IBM et Apple, inclue le POWER, initiateur de la gamme, le POWER2 hautes performances, le PowerPC 601 de bas de gamme, le PowerPC 603 basse consommation, le PowerPC 604, le PowerPC 615 double personnalité, et le futur haut de gamme PowerPC 620. Le POWER, utilisé par notre machine de tests est composé de trois unités de traitement :

- l'unité de branchement (BPU, ou Branch Processing Unit) qui lit les instructions du cache d'instructions, les envoie sur les deux autres unités, exécutant pour sa part les instructions de branchement et les instructions relatives au registre conditionnel. Notons que ces deux

derniers types d'instructions peuvent être exécutés en parallèle.

- l'unité de calculs entiers (FXU, ou FiXed-point Unit) qui exécute l'arithmétique entière ainsi que les accès mémoire depuis le cache de données, et

- l'unité de calculs flottants (FPU, ou Floating-Point Unit) pour les calculs en double précision.

Ces trois unités opérant simultanément, le processeur peut exécuter un maximum de quatre instructions par cycle. Ces situations sont rares et le taux d'instructions par cycle effectif est bien inférieur (Section 4.2.3). Le compilateur C a un rôle décisif à cet égard puisqu'il a pour charge de réordonner les instructions afin que les trois unités de traitement soient utilisées du mieux possible.

3.2.2 Le DPX/20 modèle 420

Le DPX/20 modèle 420 a été utilisé dans la plupart de nos tests. Il repose sur un POWER cadencé à 41,7 MHz, inclue un cache d'instructions et un cache de données de 32 kilo-octets chacun. Ces caches sont de type associatifs de classe quatre, ce qui signifie qu'à chaque adresse physique peuvent être associées quatre lignes du cache. La mémoire principale est de 32 méga-octets ce qui s'est avéré être suffisant pour contenir le système de communication dans sa totalité, évitant ainsi tout vidage de pages sur disque. Enfin nous utilisons deux disques de 1 giga-octets chacun, le premier contenant le système AIX/3.2.5 et le second le système AIX/4.1.2.

3.2.3 Le système d'exploitation AIX

Le système d'exploitation AIX^{tm} d'IBM est un système UNIX dérivé de l'OSF/1. Il supporte toutes les nouvelles caractéristiques des systèmes d'exploitation de demain, en particulier :

- le *multi-threading* : les threads sont des sous-processus qui partagent le même espace d'adressage que le processus père. Cela rend possible des services d'IPC (Inter-Process Communications) très performants et économise de la mémoire.

- un *noyau préemptible* : chaque thread, qu'elle soit en mode utilisateur ou noyau, peut désormais être préemptée par une thread de priorité supérieure, une routine d'interruption, ou simplement parce qu'elle a fini sa tranche de temps. Le comportement temps-réel du système d'exploitation est de ce fait amélioré.

- un *noyau paginable* : le code noyau non utilisé est automatiquement vidé sur disque ce qui économise la mémoire physique.

- une *édition de liens dynamique* et des *extensions noyau* : ces techniques permettent une conception incrémentale du système d'exploitation. L'exécutable de base ne contient que les

services fondamentaux. Les services additionnels, tels que la pile TCP/IP, les drivers réseaux et l'environnement Streams, constituent des exécutables indépendants qui ne sont chargés que sur demande. La couche Socket par contre fait partie du noyau de base.

- un support pour *machines multiprocesseurs symétriques* (ou SMP, Symmetric Multi-Processor) : l'AIX/4.1.2 a été conçu avec un soucis de parallélisation efficace.

Dans la pratique deux versions majeures d'AIX doivent être distinguées :

- l'AIX/3.2.5, une version restreintes aux machines monoprocesseurs, et
- l'AIX/4.1.2 qui a été conçu pour supporter le parallélisme [Talbot95].

Ces deux versions sont radicalement opposées. De plus si l'AIX/4.1.2 existe également en version monoprocesseur, cette adaptation consiste en quelques ajustements minimaux (remplacement des verrous par un masquage d'interruptions). Les versions mono et multiprocesseurs d'AIX/4.1.2 sont donc très proches l'une de l'autre.

Une difficulté que nous avons rencontré est que l'AIX/4.1.2 était en cours de développement ce qui nous a conduit à travailler sur des versions intermédiaires. Par conséquent les expérimentations du chapitre 4 ont été réalisées soit sur AIX/3.2.5, soit sur différentes (proches) versions du premier système AIX/4.1 diffusé, depuis le "build" de janvier 1995 à celui de mai 1995. Si les courbes d'une figure donnée correspondent à la même version d'AIX, il peut y avoir des différences entre les sections.

3.3 BSD VERSUS STREAMS

Les piles de communication ne sont pas implémentées directement dans le système d'exploitation mais reposent sur un environnement qui fournit les divers services et concepts de base dont elles ont besoin. Nous nous intéressons ici aux deux environnements majeurs des systèmes ouverts : BSD, l'environnement des premières piles TCP/IP, et Streams [Streams90], introduit plus tard par AT&T. Cette section propose un aperçu critique de ces environnements.

3.3.1 L' environnement BSD

Les évolutions de l'environnement BSD liées à celles du système UNIX BSD 4.x (Berkeley Software Distribution) de l'Université de Californie à Berkeley [Stevens94]. Il est historiquement l'environnement privilégié des implémentations de TCP/IP. Cependant il est également utilisé pour d'autres familles de protocoles telles XNS (Xerox Network Systems), les couches basses de l'OSI (transport et réseau), ou les services d'IPC du domaine UNIX (Figure 7).

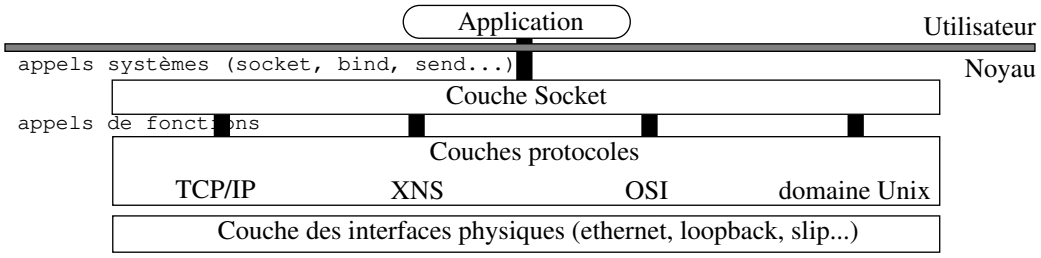

Figure 7: Le sous-système de communication BSD.

Description rapide

La couche Socket est le coeur de l'environnement BSD. Elle fournit une interface uniforme aux protocoles réseau et IPC. La sélection du protocole est faite lors de la création du point d'accès. Ensuite la couche Socket s'occupe, lors des requêtes de service des applications, d'appeler les routines adéquates. Les appels systèmes, les copies de données entre espaces utilisateur/noyau, et la gestion des buffers d'émission/réception sont sous sa responsabilité. L'API, également appelée Socket, qui est implémentée par cette couche a deux caractéristiques essentielles :

- ce n'est pas une API à caractère général mais une API orientée protocoles de communication, qui inclut par exemple des primitives de gestion des connexions,

- le code de l'API est intégralement dans le noyau. Les primitives de la librairie Socket ne sont que des empaquetages autour des appels systèmes correspondants.

L'environnement BSD fournit également des services mémoires. Les buffers, appelés `mbufs`, sont composés d'un descripteur suivi d'une petite zone de stockage. Lorsque ce buffer ne suffit pas, soit un second voir troisième et quatrième `mbufs` sont alloués et chaînés, soit un gros buffer de taille fixe appelé `cluster` est utilisé. Selon le système, un `mbuf` est long de 129 ou 256 octets et un `cluster` de 1024, 2048 octets ou davantage. La décision d'utiliser une chaîne de `mbufs` ou un unique `cluster` est un compromis entre l'efficacité des traitements et l'utilisation de la mémoire[1].

Enfin, en dehors de l'interface Socket supérieure, l'environnement BSD n'impose aucune structure à l'implémentation des piles de protocoles. Il en résulte une importante intégration des différents composants : ceux-ci dialoguent par appels de fonctions et passages de paramètres, ont accès aux structures de données socket, notamment aux buffers d'émission/réception, et réciproquement la couche Socket a accès aux structures de données des protocoles. Une autre conséquence est que la configuration des piles de communication est essentiellement définie lors du développement par le biais de tables préinitialisées. La flexibilité est donc limitée.

Support du parallélisme dans BSD

Cet environnement ne propose aucun service de synchronisation. La parallélisation des différentes

1. Ce problème est résolu avec AIX/4.1.2 où les clusters ont une taille variable.

piles de protocoles est donc laissée à la responsabilité des développeurs.

3.3.2 L'environnement Streams

L'environnement Streams a été conçu par AT&T pour la version System V Release 3 en 1986, dans le but d'apporter plus de structuration aux systèmes d'E/S, et en particulier aux TTYs.

Description rapide

La notion de stream, ou canal bidirectionnel (écriture et lecture) est centrale. Un stream permet à l'utilisateur d'un service de dialoguer avec le driver qui supporte ce service. Ce dialogue repose sur des messages qui sont envoyés dans chaque direction sur ce stream. Outres les drivers, Streams définit également des modules qui sont des unités de traitement optionnelles. Ces modules peuvent être insérés et retirés à tout moment sur un stream. Ils permettent d'effectuer des traitements supplémentaires sur les messages qui traversent ce stream sans avoir à modifier le driver sous-jacent. Streams offre également la possibilité d'empiler plusieurs drivers les uns au dessus des autres. De tels drivers sont qualifiés de multiplexeurs car ils multiplexent plusieurs streams supérieurs au dessus de plusieurs streams inférieurs. Ceci est couramment utilisé pour les piles de communication (Figure 13).

Au sein d'un driver/module les messages peuvent être soit traités immédiatement par la routine d'interface qui les reçoit (routine `put`), soit insérés dans la queue associée (queues d'écriture et de lecture du stream) et traités de façon asynchrone par la routine de `service`. L'environnement Streams est responsable du scheduling des différentes routines de service contenant des messages. Aucune hypothèse ne peut être faite sur l'algorithme de scheduling qui est spécifique à l'implémentation[2]. Le mécanisme de contrôle de flux Streams est fondé sur l'examen de la prochaine queue du stream. Si la taille cumulée des messages est supérieure à la taille maximale associée au stream (ou "high-water mark") alors la queue est dite saturée. Dans ce cas le driver/module amont en est informé et évite tout envoi de messages.

Remarque : Le fait que dans la suite les figures montrent systématiquement des queues est destiné à matérialiser les streams. Cela ne signifie pas forcément que des messages y soient insérés.

La Figure 8 représente une configuration où deux applications dialoguent avec un driver au travers de deux streams. Dans un cas deux modules additionnels ont été insérés. Le Stream-Head est responsable de l'interface UNIX/Streams et est l'équivalent de la couche Socket. Il a en charge les appels systèmes Streams, la copie de données entre les espaces utilisateur/noyau, et la création de messages. Mais si les appels systèmes Socket sont orientés protocole de transport, les appels

2. Les fonctions de service sont souvent traitées lors de la "fin d'appel système". Si cela évite tout changement de contexte, le temps de latence peut être élevé. La situation est plus complexe avec un système d'exploitation parallélisé où cette stratégie n'est pas appropriée. Un mécanisme d'interruption logicielle est utilisé avec AIX/4.1.2. Le temps de latence dépend alors du niveau d'interruption courant sur le(s) processeur(s), et ce mécanisme peut également induire un changement de contexte.

système Streams sont quant à eux génériques. Les deux principaux appels systèmes sont `putmsg` qui permet à une application d'envoyer des données, et `getmsg` qui permet à une application de récupérer les données qui ont atteint le Stream-Head. Le caractère générique est garanti par le fait que Streams n'émet aucune hypothèse sur la sémantique des données qui lui sont passées. L'avantage de cette approche est que plusieurs APIs peuvent être définies, chacune ayant la même efficacité (Section 3.4.2).

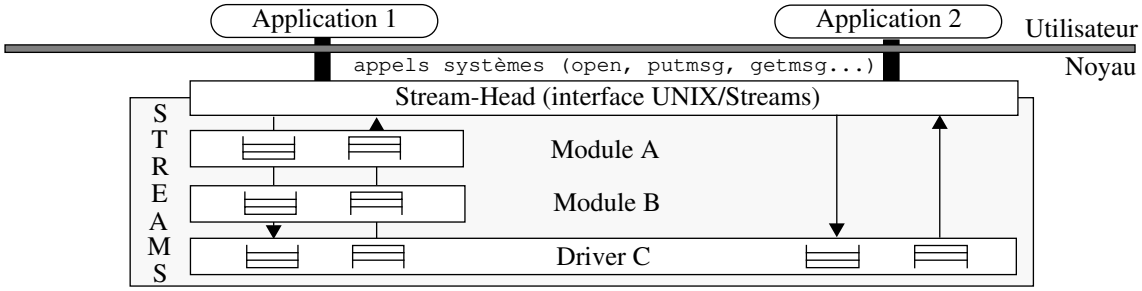

Figure 8: Une configuration Streams élémentaire.

Les services de gestion mémoire sont essentiels à Streams qui repose tout entier sur la notion de messages. Les buffers, appelés `mblk` (ou "message blocks"), sont constitués de l'association de deux structures de description (`mblk_t` et `dblk_t`) et d'un buffer de taille variable (Figure 9). L'organisation exacte de ces trois zones est spécifique à l'implémentation[3]. L'avantage des `mblks` sur les `mbufs` est leur taille variable qui garantit à la fois une bonne utilisation de la mémoire et des traitements efficaces.

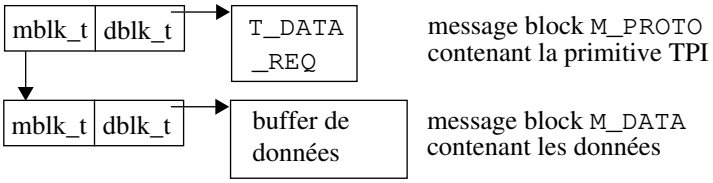

Figure 9: Message TPI contenant une requête de transmission de données.

Un message est l'association d'un ou plusieurs `mblks`. Chaque `mblk` possède un type : `M_DATA` pour les `mblks` de données, `M_PROTO` pour les `mblks` contenant une primitive d'interface, etc. Dans le cas des systèmes de communication, à chaque interface est associé un format de message standardisé. Ces interfaces normalisées sont appelées TPI (Transport Protocol Interface), NPI (Network Protocol Interface), et DLPI (Data Link Protocol Interface) [TPI92, NPI92, DLPI92]. La Figure 9 montre une TSDU composée d'un `mblk` de type `M_PROTO` contenant une primitive TPI `T_DATA_REQ`, suivi d'un `mblk` de type `M_DATA`.

3. Un environnement Streams optimisé peut allouer les deux structures de description au sein du même buffer. Il est même possible d'allouer les parties descripteur et données dans le même buffer lorsque la taille totale est appropriée.

Ces caractéristiques garantissent que les composants sont indépendants les uns des autres, et permettent des configurations dynamiques. Ainsi la configuration complète peut être décrite dans un fichier éditable qu'un outil va analyser pour ensuite générer les commandes adéquates (`ioctl`). Quelqu'un qui désirerait enregistrer tous les segments TCP échangés mais n'a pas accès à la commande `tcpdump` peut concevoir un module espion qui enregistre tous les messages contenant une primitive `N_UNITDATA_REQ/IND`. Il ne reste plus qu'à modifier le fichier de configuration afin que ce module soit inséré sous TCP. Cette souplesse permet d'éviter tout recours à des patchs ou à des modifications de sources.

Enfin il doit être noté que si l'environnement Streams est habituellement inclus dans le noyau d'UNIX, ce n'est pas une obligation. Cet environnement peut aussi être porté dans l'espace utilisateur [Schmidt94], sur d'autres systèmes, ou sur des cartes de communication intelligentes.

Support du parallélisme dans Streams

Streams a été étendu afin de faciliter le développement de composants Streams dans un environnement multiprocesseurs [Campbell91, Garg90, Kleiman92, Saxena93]. Mais ces extensions sont des solutions propriétaires; il n'y a *aucune standardisation ni consensus*. Plusieurs niveaux de synchronisation sont habituellement définis (Figure 10) :

- *Niveau module* : une unique thread est permise dans le driver. Des drivers issus d'un environnement non parallèle ne nécessiteront que des modifications minimales.

- *Niveau paire de queues* : une unique thread est permise pour chaque paire de queues (écriture et lecture). Ceci est utilisé par les composants qui ne partagent de données qu'entre les flux entrant et sortant d'un contexte.

- *Niveau queue* : une unique thread est permise pour chaque queue. Ceci est utilisé par les composants qui ne partagent aucune donnée entre flux entrant et sortant.

- *Niveau stream* : une unique thread est permise pour chaque stream. Cette solution minimise les manipulations de verrous dans le cas ou plusieurs modules sont insérés sur un stream donné, tout en préservant le parallélisme entre streams.

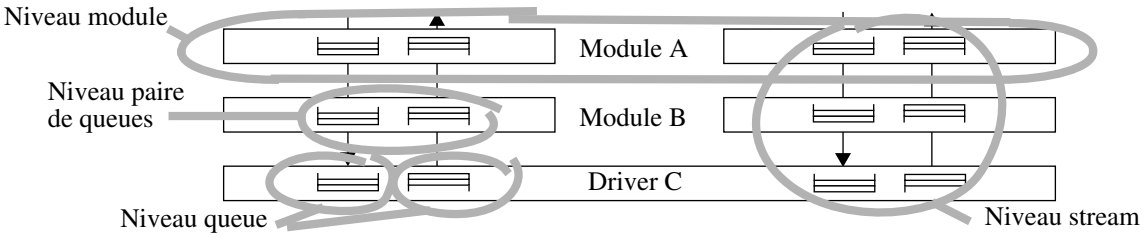

Figure 10: Les principaux niveaux de synchronisation de Streams.

Ces niveaux de synchronisation peuvent être implémentés de plusieurs façons :

- en utilisant des verrous à l'intérieur de l'environnement Streams [Kleiman92, Saxena93] : une thread qui désire effectuer du travail pour le compte d'une queue donnée doit au préalable acquérir le verrou correspondant.

- en associant des Eléments de Synchronisation (ES) aux queues : si une thread ne peut effectuer du travail pour le compte d'une queue (détecté grâce à son ES), la requête est enregistrée et la thread va ailleurs. Ce travaille sera pris en charge plus tard par la thread qui détient l'ES. Ce mécanisme offre l'avantage de minimiser le nombre de changements de contextes au prix de traitements supplémentaires en l'absence de conflits (manipulation des ES). L'environnement Streams utilisé, dérivé de celui de l'OSF/1.1, utilise ce mécanisme.

Il n'y a pas de correspondance directe entre les niveaux de synchronisation définis dans Streams et les types de parallélisme définis dans la Section 2.2.4. Streams définit pour sa part (Figure 11) :

- le *parallélisme vertical :* il dénote la possibilité pour différentes threads de traiter des messages simultanément dans les différents drivers/modules d'un chemin de données. En fait il repose sur l'utilisation systématique des queues et routines de service. C'est l'équivalent du parallélisme par couches.

- le *parallélisme horizontal :* il dénote la possibilité pour différentes threads de traiter les messages simultanément au sein des différentes queues d'un driver/module. Ceci est bien sûr incompatible avec le niveau de synchronisation module. Dans le cas d'un niveau de synchronisation par paire de queues, le parallélisme est de type connexion. Ce n'est plus le cas si le niveau queue est utilisé.

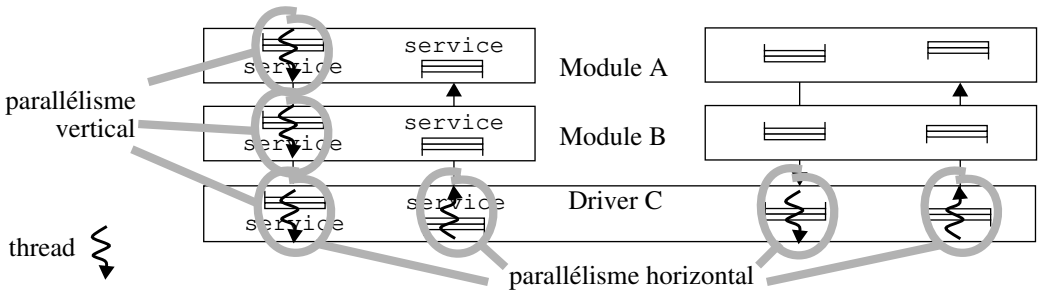

Figure 11: Le parallélisme horizontal et vertical.

3.3.3 Résumé

La Table 3 propose un synthèse de la discussion BSD versus Streams. Elle souligne les différents objectifs suivis : offrir un environnement limité permettant une intégration maximale pour BSD, ou bien un environnement complet destiné à faciliter et uniformiser le développement de systèmes d'E/S pour Streams. Les solutions adoptées sont radicalement différentes dans chaque cas. Nous allons voir dans les sections suivantes les implications de ces choix dans le détail.

Table 3: BSD versus Streams, résumé des différences.

	Environnement BSD	**Environnement Streams**
Concepts clés et buts	favoriser l'intégration et les performances	favoriser l'indépendance, la portabilité et la flexibilité
	environnement minimum (interface haute, services de base, pas de facilités de synchronisation)	environnement complet (interface haute, services de base, scheduling, contrôle de flux et mécanismes de synchronisation)
utilisation	restreint aux systèmes de communication	utilisé pour divers systèmes d'E/S
moyens	communication par appels de fonctions	communication par messages
	interfaces internes propriétaires	interfaces internes normalisées par l'X/Open
gestion mémoire	mauvaise (taille fixe, non homogène)	efficace (taille variable, un seul type)

3.4 IMPLÉMENTATION CLASSIQUE D'UNE PILE TCP/IP SOUS BSD ET STREAMS

Cette section décrit les implémentations classiques d'une pile TCP/IP qui sont trouvées dans les environnements BSD et Streams. Ces implémentations seront prises comme point de référence pour les évaluations de performances du Chapitre 4. Elles sont aussi le point de départ des alternatives que nous décrivons dans les Section 3.5.1 et Section 3.5.2.

3.4.1 Une pile TCP/IP classique sous BSD

L'implémentation BSD est le standard de fait pour TCP/IP. Elle provient du Computer System Research Group de l'Université de Californie, Berkeley. Cette implémentation, qui a été grandement optimisée au cours des années, est distribuée librement avec les différentes versions du système BSD 4.x. Elle a été intégrée à de nombreux systèmes d'exploitation commerciaux tels que l'AIX/3.2.5 et l'AIX/4.1.2, HP-UX, ou SunOS 4.x (avant que Sun ne migre sous Streams, voir plus loin).

La Table 4 montre les évolutions majeures de TCP/IP et de cette implémentation suivant les différentes versions de BSD [Stevens94], et la Figure 12 représente un extrait de la pile TCP/IP BSD (manquent IGMP, rawIP, BPF, etc.).

Table 4: Evolutions de TCP/IP et de son implémentation suivant les versions de BSD.

4.2 BSD (1983)	première version de TCP/IP largement diffusée
4.3 BSD (1986)	améliorations de performances de TCP
4.3 BSD Tahoe (1988)	ajout des mécanismes de "slow start", "congestion avoidance", et de retransmission rapide
4.3 BSD Reno (1990)	ajout du mécanisme de prédiction des en-têtes TCP, recouvrement sur erreur rapide, changements dans la gestion de la table de routage
4.4 BSD (1993)	support de la diffusion, RFC1323

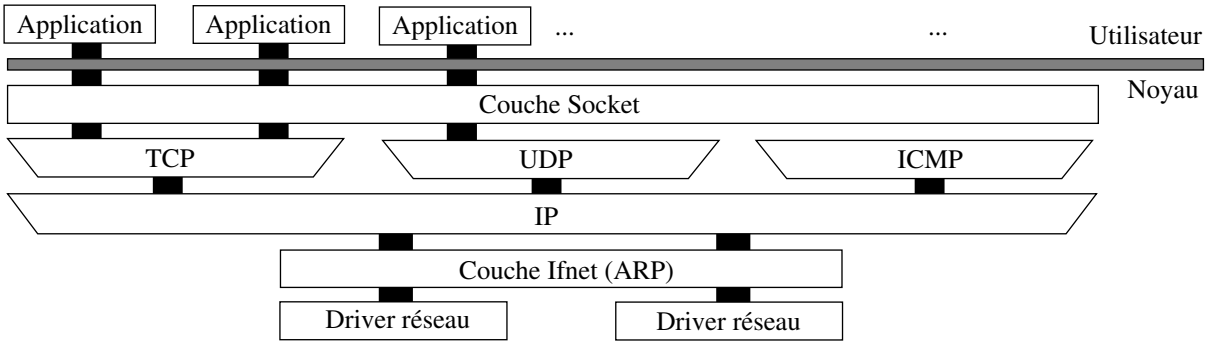

Figure 12: Extrait de la pile TCP/IP sous BSD.

Un certain nombre de composants de la Figure 12 ont été décrits à la Section 3.3.1. Pour sa part, la couche Ifnet fait le lien entre le driver IP et les différents drivers réseau en créant une interface unifiée. C'est également l'endroit où se trouve le protocole ARP.

Disposer d'une API XTI/TLI est possible avec BSD. Dans l'OSF/1 et AIX ceci est réalisé au moyen d'un driver Streams spécial, XTISO. Ce driver réalise les conversions requises entre les abstractions Socket et Streams. En raison des très nombreuses redondances que cela induit (encapsulation `mblk`/`mbuf`, double méthode d'accès) les performances sont médiocres (divisées par deux).

Parallélisation de l'implémentation BSD

La parallélisation de l'implémentation BSD est traditionnelle. Elle s'efforce de minimiser le travail réalisé sous interruption et de transformer les problèmes de synchronisation au niveau interruption en problèmes de synchronisation au niveau thread plus faciles à résoudre. Dans ce but [Boykin90] définit plusieurs threads noyau qui ont pour tâche de traiter les interruptions Ethernet et les différents événements timer : timers lent et rapide de TCP, timer de réassemblage des fragments IP, timer ARP de vieillissement des entrées, etc. Enfin, plusieurs verrous de type attente active ou lecteurs/écrivain sont ajoutés au code.

Le parallélisme est traditionnellement de type Message (Section 2.2.4), c'est-à-dire que chaque paquet entrant est pris en charge par la première thread disponible. D'autres stratégies sont cependant possibles [Chesson94].

3.4.2 Une pile TCP/IP classique sous Streams

Les implémentations Streams ont été vues comme un moyen d'apporter des aspects de flexibilité et de portabilité aux implémentations BSD monolithiques. Cependant le code de base reste la souche BSD. Streams est actuellement utilisé comme environnement réseau de plusieurs systèmes d'exploitation majeurs : UNIX System V Release 4, Solaris [Kleiman92] et WindowsNT

[WindowsNT93]. Sous AIX/4.1 par contre, seuls les TTYs et la pile OSI sont Streams.

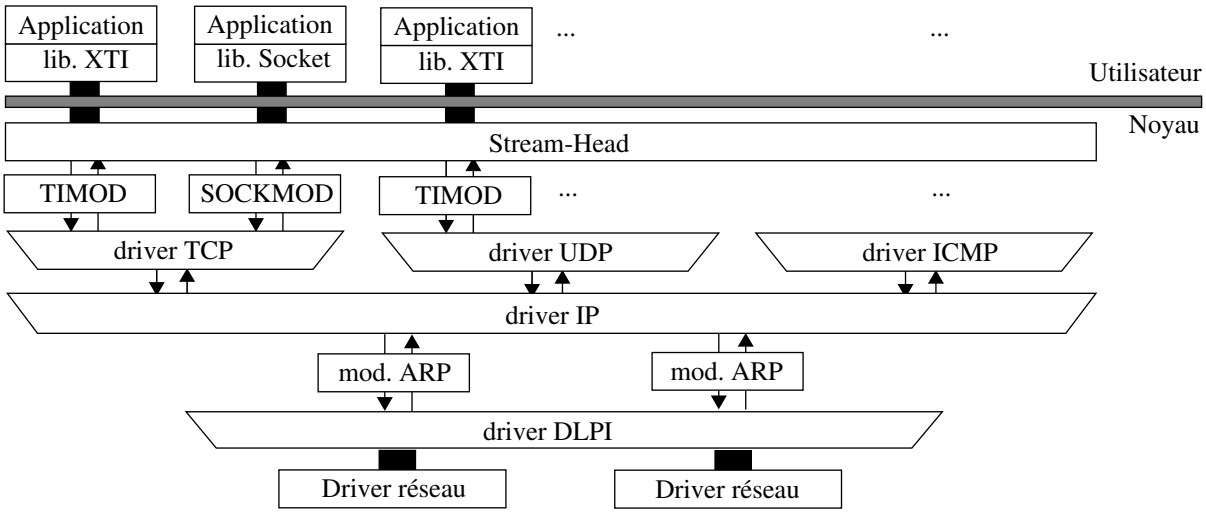

Figure 13: Extrait de la pile TCP/IP sous Streams.

La Figure 13 montre une pile TCP/IP sous Streams. On identifie les composants suivants :

- la librairie XTI (X/Open Transport Library) [XTI93], un sur-ensemble de la librairie TLI d'AT&T (Transport Layer Interface). Le fait que la plus grande partie des traitements s'effectue au niveau Utilisateur est la raison pour laquelle TIMOD, un module coopératif, est utilisé. Ce module contient suffisamment d'informations pour permettre une resynchronisation après un `fork`[4],

- la librairie Socket et le module SOCKMOD opèrent de façon similaire. Ils implémentent l'API Socket au dessus des appels systèmes Streams génériques. Les performances sont de ce fait identiques à celles de XTI,

- les drivers multiplexés TCP, UDP, IP, ICMP, IGMP, RawIP (pas tous représentés),

- le module ARP, et

- le driver DLPI qui fournit une interface commune entre les différents drivers réseaux propriétaires et les piles de communication Streams. Il y a un stream entre les drivers IP et DLPI par interface réseau logique.

4. La difficulté vient du fait que le driver de transport traite tous les utilisateurs d'un point d'accès (à la suite d'un `fork` par exemple) comme un unique utilisateur. Ces utilisateurs multiples doivent alors coordonner leur activité afin de respecter l'état du point d'accès transport. Le module TIMOD est le lieu où cet état est maintenu. Se situant dans le noyau cette information est unique et non volatile.

3.4.3 Limites à priori de l'implémentation Streams à la lumière de BSD

Nous nous intéressons ici essentiellement aux limites associées à l'architecture de l'implémentation Streams classique. D'autres limites, propres à Streams, seront développées plus tard.

L'utilisation des queues et des routines de service

Les protocoles de transport insèrent habituellement les requêtes de transmission de données dans la queue d'écriture du stream et les traitent dans la routine de service associée. Cette technique permet un contrôle de flux aisé de l'application : lorsque la fenêtre d'émission de TCP est saturée, la queue d'écriture le devient aussi (la taille de cette queue a été initialisée en conséquence). Ceci est détecté par le Stream-Head et l'application est bloquée lors du prochain `putmsg`. Cette technique a cependant deux inconvénients :

- l'insertion et le retrait d'un message sont des opérations coûteuses qui interviennent de surcroît sur le chemin de données principale. Un service dédié serait plus performant ici.

- Le scheduling des fonctions de service ajoute un temps de latence (fonction de l'implémentation de l'environnement). Ceci est pénalisant car il retarde la transmission possible des paquets.

Le contrôle de flux Streams

Parce qu'un driver multiplexé par définition multiplexe plusieurs streams supérieurs au dessus de plusieurs streams inférieurs, ces streams ne sont pas directement liés. Par conséquent le contrôle de flux Streams, local au stream, ne peut voir à travers un driver ce qui est source de problèmes. Par exemple si le driver DLPI est saturé et si tous les drivers intermédiaires utilisent des queues, alors l'application ne pourra être bloquée qu'une fois ces queues pleines. Si certains drivers n'utilisent pas de queue (c'est généralement le cas d'IP), alors l'information de saturation ne pourra être propagée vers l'application. La situation est similaire pour le flux entrant.

Ces problèmes de contrôle de flux signifient que des ressources mémoire et processeur peuvent être monopolisées par des tâches non productives (les paquets sont soit perdus soit mis en queue) au détriment des composants qui auraient besoin de ces ressources.

La gestion des buffers coté entrant : une limitation fonctionnelle de Streams

Les expérimentations ont montré que les médiocres performances de l'implémentation Streams classique proviennent en partie de la stratégie de gestion des buffers du Stream-Head coté réception. Contrairement à la couche Socket qui par défaut essaie d'optimiser le remplissage des buffers de l'application, le Stream-Head retourne les données à l'application dès qu'elles arrivent. [Streams90] ne propose aucune autre stratégie. Une conséquence directe est qu'un plus grand nombre d'appels systèmes de réception (`getmsg`/`read`) est requis. De nombreuses implémentations Streams ne se soucient pas de ce problème.

La parallélisation

La parallélisation de cette implémentation Streams classique révèle plusieurs limites :

- Le concept d'implémentation en couches promu par Streams et l'idée qu'effectuer le travail dans les routines de service est une bonne façon de paralléliser le système de communication ont conduit à la notion de parallélisme vertical (Section 3.3.2). Malheureusement ce type de parallélisme s'avère médiocre (Section 2.2.4).

- La synchronisation proposée par notre environnement Streams ne peut être désengagée, c'est-à-dire que le niveau "queue" est le niveau de parallélisme maximal permis. Il en résulte que la parallélisation des drivers ayant peu de points d'accès tels que DLPI et IP conduit à des problèmes de sérialisation (on ne peut traiter qu'un message à la fois). Une solution pour préserver le parallélisme est de multiplier artificiellement le nombre de points d'accès à ces drivers, c'est-à-dire d'ajouter artificiellement des streams. Le problème est résolu au prix d'une complexité accrue.

- Enfin un driver multiplexé ne peut pas exploiter efficacement les différents niveaux de synchronisation Streams. La Figure 16-gauche montre que les traitements TCP sont effectués à la fois dans la paire de queue inférieure (démultiplexage), dans la paire de queue du contexte (fin de traitement du paquet), et dans un contexte d'interruption pour les traitements liés aux timers. La parallélisation conduit de ce fait à un usage important de verrous privés [Heavens92]. Ceci est un problème lorsque la synchronisation Streams ne peut être évitée, car deux niveaux de synchronisation existent alors, l'un d'eux n'étant pas utilisé.

3.5 LES ALTERNATIVES

Les limitations que nous avons analysées dans les précédentes sections nous ont conduit à modifier les implémentations BSD et Streams de trois façons : une implémentation démultiplexée, une implémentation au niveau utilisateur, et une méthode d'accès ILP.

3.5.1 Une implémentation démultiplexée sous Streams

3.5.1.1 Architecture de l'implémentation démultiplexée

L'analyse détaillée des problèmes rencontrés avec l'implémention classique sous Streams nous a conduit à nous intéresser à une architecture démultiplexée regroupant toutes les tâches de multiplexage/démultiplexage en un seul lieu [Roca94]. Dans cette approche tous les protocoles sont implémentés sous forme de modules plutôt que de drivers multiplexés. Lorsqu'une application ouvre un point d'accès transport, un stream est crée et les modules protocolaires appropriés sont automatiquement insérés sur ce stream. Nous appelons *Canal de Communication* (ou CC)

l'association d'un stream et de ses modules.

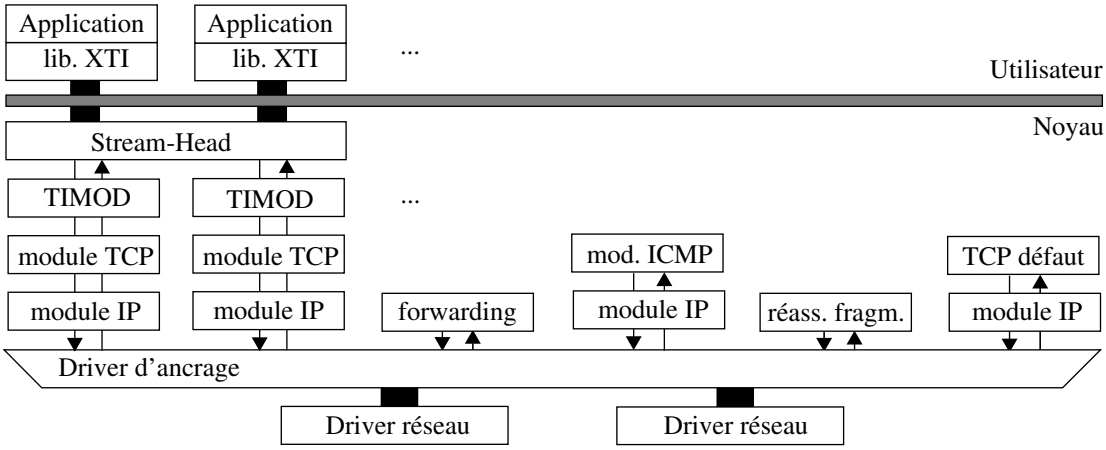

Figure 14: La pile TCP/IP démultiplexée sous Streams.

Le driver d'ancrage

Le driver d'ancrage a trois rôles :

- il sert de point d'ancrage pour les différents streams,
- il fournit une interface de type DLPI aux drivers réseaux[5], et enfin
- il démultiplexe les paquets entrants. Cette opération est fondée sur l'examen des différents en-têtes de protocoles du paquet. Dans le cas d'Ethernet cela inclue le champ Ethertype, le champ protocole d'IP, les adresses IP source/destination, et les numéros de port source/destination.

Les canaux de communication applicatifs

Les différents canaux TCP/IP et UDP/IP sont les endroits où les traitements protocolaires ont lieu. Ils sont créés à chaque fois que l'application ouvre un point d'accès transport.

Coté réception, le checksum est calculé par les modules. En conséquence, le démultiplexage effectué dans le driver d'ancrage intervient avant la vérification du checksum. Ceci repose sur l'hypothèse qu'aucune erreur n'est présente, mais un paquet erroné orienté sur un mauvais CC sera de toutes façons rejeté plus tard.

5. La raison pour laquelle cette interface n'est pas complètement conforme à l'interface DLPI est que DLPI est prévu pour le modèle en couches traditionnel où chaque stream entre IP et DLPI correspond à une interface réseau unique. Ce n'est plus le cas dans l'architecture démultiplexée et un champ a été ajouté à la primitive DL_DATA_REQ afin d'indiquer l'interface choisie pour ce paquet

Une autre remarque est que le module IP joue désormais un faible rôle coté réception. La plupart des traitements IP destinés à déterminer si un paquet a atteint sa destination et si il est fragmenté ont en effet été déplacés dans le driver d'ancrage.

Les canaux de communication supplémentaires

Des tâches telles que le "forwarding" des paquets (action de router les paquets qui ne sont pas à destination vers le prochain noeud), les traitements ICMP, et le réassemblage des fragments IP ne sont pas liées à un point d'accès transport particulier. Ne pouvant être traitées par un CC standard, ces tâches sont prises en charge par des CC dédiés créés lors de la mise en place générale.

Cette architecture démultiplexée a mis en évidence une difficulté supplémentaire : en raison de la notion de fermeture de connexion douce, une connexion TCP peut vivre plus longtemps que l'application. C'est le cas lorsque l'application ferme son point d'accès alors qu'il reste des paquets à échanger (données ou acquittements). Avec une implémentation TCP/IP standard, l'accès est fermé mais le contexte TCP reste actif et peut continuer d'envoyer et recevoir des paquets. Avec l'implémentation démultiplexée, fermer l'accès démantèle le CC dans sa totalité. La solution consiste à déplacer le contexte TCP vers un CC TCP permanent. Tous les paquets seront alors pris en charge par ce CC.

Les traitements ARP

ARP n'est pas représenté dans la Figure 14 car il est inclus dans le driver réseau. Ceci est une conséquence de l'utilisation du driver Ethernet BSD. Une solution plus propre consiste à ajouter un CC pour le protocole ARP et à faire la translation d'adresse dans le driver d'ancrage.

3.5.1.2 Bénéfices de cette approche

Les intérêts de cette approche résident dans les points suivants :

Les impacts sur le chemin de données principal

L'architecture des composants d'un CC est simplifiée par la connaissance précoce des protocoles utilisés (défini lors de l'ouverture du CC) et par la suppression de tout multiplexage. Un module ne définissant qu'un seul chemin de données dans chaque sens a une structure plus simple qu'un driver multiplexeur qui doit faire le lien entre les streams supérieurs et inférieurs. Le chemin de données principal est donc (légèrement) simplifié.

D'un autre coté le fait que le démultiplexage intervienne avant le calcul de checksum ajoute une conversion réseau-vers-hôte (`ntoh`, ou "network to host") additionnelle sur les machines qui ne sont pas big-endian[6]. En effet, dans une pile classique un paquet entrant est reçu dans le format

6. L'ordre des octets dans un mot de 32 bits est dépendant du processeur. Deux possibilités principales sont le format big-endian utilisé par la famille POWER et par convention par les en-têtes de paquets sur le réseau, et le format little-endian utilisé par la famille de processeurs Intel.

réseau (big-endian), le checksum est calculé, les en-têtes sont convertis au format hôte et les traitements protocolaires ont lieu. Avec l'approche démultiplexée les champs protocoles du paquet entrant sont accédés alors qu'ils sont encore au format hôte.

Enfin l'approche démultiplexée conduit à un accès supplémentaire aux en-têtes (driver d'ancrage) par rapport aux solutions traditionnelles où les en-têtes ne sont accédés que dans IP et TCP/UDP.

Dans l'ensemble nous pensons que l'approche architecture démultiplexée ni ne simplifie de façon significative le chemin de données principale, ni ne le complique. *Les fonctionnalités sont déplacées plutôt que simplifiées*.

Les impacts sur le contrôle de flux local

Le contrôle de flux local est grandement amélioré. Comme ce contrôle se fait au niveau du stream, avoir un unique stream depuis l'application jusque dans le driver d'ancrage signifie qu'une application gourmande peut immédiatement être bloquée si le driver réseau est saturé (flux sortant). Cette technique *revient à créer un mécanisme de contrôle d'accès au médium*.

L'algorithme utilisé pour implémenter ce contrôle de flux coté émission est le suivant : si, lors d'une tentative de transmission, le driver réseau indique qu'il est saturé, alors le driver d'ancrage insère le paquet dans la queue d'écriture du CC. Celui-ci devient bloqué[7] et il est inséré dans une liste de CC dans la même situation. Le module de transport qui détecte cette saturation évitera par la suite de créer de nouveaux segments. Enfin un timer de type "chien de garde" est armé au sein du driver d'ancrage afin de relancer plus tard tous les CC bloqués.

Réciproquement en réception, après avoir démultiplexé un paquet, le driver d'ancrage peut détecter, par simple vérification de l'état de saturation du CC, si l'application parvient à suivre le rythme. Il peut alors décider de l'attitude à adopter : faire remonter le paquet, jeter un paquet UDP en cas de saturation, etc. Dans tous les cas, ce choix est effectué le plus tôt possible, avant tout traitement important.

Les impacts sur le support de la QoS

L'algorithme utilisé pour décider quand bloquer ou débloquer un CC peut être fondé sur des informations de saturation comme on vient de voir, mais aussi sur un mécanisme de déblocage périodique dans le cas où l'on désire effectuer un contrôle de débit sur un flux UDP par exemple, ou enfin sur des notions de priorité.

Montrons comment on peut construire un mécanisme de support de la QoS. Le driver d'ancrage est à cet égard privilégié : il a accès à tous les CC et peut être étendu afin d'avoir connaissance des besoins en QoS de chacun d'eux. C'est là que peut être implémenté un mécanisme de contrôle d'accès de type CBQ (Class Based Queueing) [Wakeman95]. Ce mécanisme peut aussi être étendu

7. Le seuil de remplissage de la queue a été initialisé à un octet. Ainsi la queue est déclarée pleine chaque fois qu'elle contient au moins un message, et vide sinon. On empêche la procédure de service d'être schedulée grâce à la routine `noenable` de Streams.

aux différents flux de données qui traversent la machine lorsque celle-ci est utilisée en tant que routeur. Dans ce cas là, le module de “forwarding” de la Figure 14 est étendu pour inclure l’aspect gestion des queues de priorité.

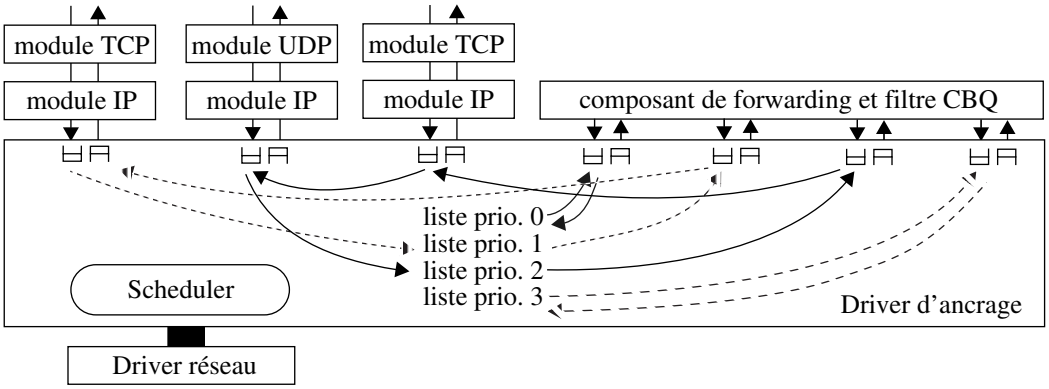

Figure 15: Extension QoS de la pile TCP/IP démultiplexée sous Streams.

La Figure 15 montre l’architecture obtenue. Les différentes queues sont chaînées en listes de priorité. Ce chaînage est à la fois dynamique pour les CC (modifié à chaque ouverture/fermeture de CC), et statique pour le composant de “forwarding” (mis en place à la configuration). Le “scheduler” est un composant asynchrone qui, à la fin de chaque transmission sur le réseau, va choisir le prochain paquet à transmettre en commençant par les listes les plus prioritaires. Un avantage de cette architecture est de traiter de façon uniforme le cas où la machine possède une fonctionnalité routage (composant de “forwarding”/filtre CBQ présent), et le cas où elle constitue une station terminale sur le réseau (seuls les CC sont présents).

L’approche démultiplexée va en fait plus loin que le simple contrôle d’accès au médium, en bloquant automatiquement toutes les applications qui ne sont pas les plus prioritaires à un moment donné. *Agir sur le blocage des CC est ainsi un moyen aisé et efficace de contrôler l’allocation des ressources processeur, mémoire et médium, en privilégiant certains flux par rapport à d’autres.*

Nous n’avons décrit que le moteur du support système de la QoS. Des mécanismes connexes tels que les politiques de scheduling, le contrôle d’admission, le contrôle du respect des engagements de QoS, etc. sont nécessaires. Ils ne seront pas abordés car trop éloignés de notre propos.

Les impacts sur le parallélisme

Extension du parallélisme à l’ensemble de la pile de communication

L’approche démultiplexée étend naturellement les points d’accès aux composants. Le parallélisme horizontal (Section 3.3.2) disponible dans TCP et UDP est maintenant étendu à l’ensemble du CC. Ceci est vrai à la fois pour le flux sortant et pour le flux entrant. En outres le parallélisme est désormais de type connexion (Section 2.2.4) ce qui peut induire un léger avantage par rapport au parallélisme de type message de BSD.

Utilisation réduite des verrous

Avec une implémentation Streams classique le niveau “paire de queues” ne peut être utilisé pour synchroniser les traitements TCP (Section 3.4.3). Ceci a été rendu possible avec l’implémentation démultiplexée par :

- le démultiplexage précoce des paquets : ils sont maintenant reçus directement sur la bonne queue, et
- l’intégration des traitements de timers dans TCP. Lorsqu’une routine de timer identifie que des traitements conséquents doivent être effectués (acquittement différé, échéance de timer) un message est créé et envoyé à la queue appropriée. De ce fait les traitements sont effectués sous le contrôle de Streams.

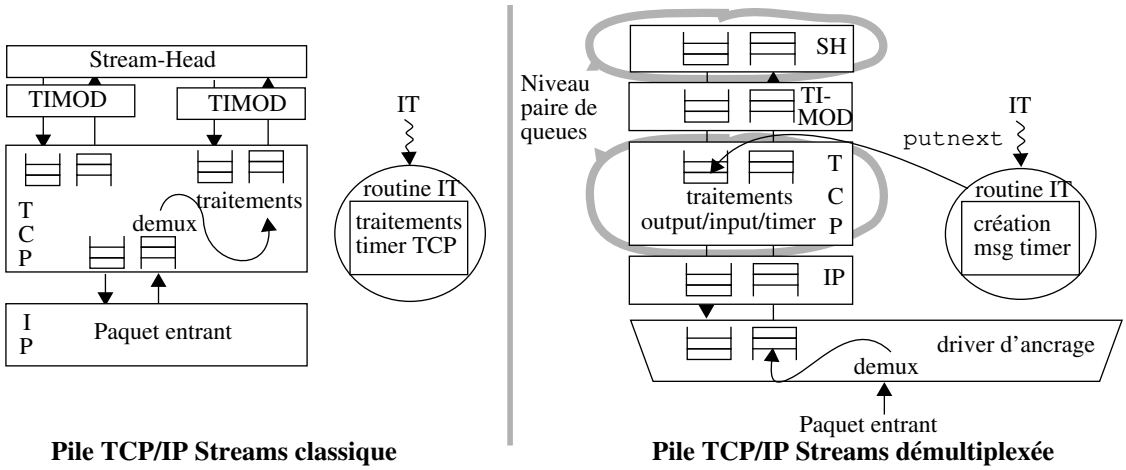

Figure 16: Traitement des flux entrant/sortant et des timers pour les implémentations classique et démultiplexée.

Grâce à la synchronisation Streams, l’utilisation de verrous privés est réduite à son strict minimum. Au sein de TCP, seule la liste partagée des contextes est protégée. Ce verrou n’est maintenu que lorsque la liste est modifiée (création et destruction de contexte) ou traversée (démultiplexage dans le driver d’ancrage, timers).

Afin de réduire plus encore l’utilisation de ce verrou nous avons fusionné les deux routines de timers de TCP : `tcp_fasttimo` (acquittements différés) et `tcp_slowtimo` (timers des contextes TCP). La nouvelle routine est exécutée avec une fréquence de 4 Hz proche des 5 Hz de `tcp_fasttimo`[8]. Le traitement des acquittements différés est systématiquement effectué, alors que le traitement des timers des contextes est fait une fois sur deux. Il y a ainsi moins d’interrup-

8. Cela est rendu possible par la nature de `tcp_fasttimo` : c’est une astuce destinée à diminuer le nombre de paquets d’acquittements et à les inclure dans d’éventuels paquets de données circulant en sens inverse. La routine `tcp_slowtimo` qui gère tous les timers des contextes TCP continue à être appelée avec la même fréquence de 2 Hz.

tions (4 IT/s au lieu de 5+2) et moins de manipulations de verrous (la liste de contextes est traversée 4 fois par seconde au lieu de 5+2).

Définition d'un "niveau de synchronisation nul"

La synchronisation Streams s'est avérée être coûteuse (chaque `putnext` effectue au minimum deux séquences de verrouillage/déverrouillage) (Section 4.2.1.3) et pas toujours nécessaire. Ainsi nous définissons un niveau supplémentaire qui consiste à ne rien imposer. Il est laissé au développeur le soin d'ajouter si besoin la synchronisation.

Ainsi seuls TCP et le Stream-Head continuent d'utiliser une synchronisation de type paire de queues. Les autres composants, ayant des besoins moindres (TIMOD) ou peu adaptés à la synchronisation Streams (accès à des tables globales), utilisent un niveau nul associés si besoin à des verrous privés (Figure 16-droite).

Facilité de développement

Comme toute la majorité de la synchronisation (hormis le cas du niveau de synchronisation nul) est prise en charge par l'environnement Streams, le support du parallélisme est aisé. La situation est différente avec une implémentation BSD.

3.5.1.3 Techniques supplémentaires

Des techniques supplémentaires ont été utilisées afin de remédier aux problèmes qui ne sont pas liés à l'architecture de la pile de communication :

A propos de l'utilisation des queues et routines de service

La mise en queue des messages et les traitements asynchrones dans les routines de services ont été bannies du chemin de données principal. Au lieu de cela :

- un service de queues spécialisé, plus efficace que la version Streams car plus simple, est utilisé par TCP afin de conserver les données entrantes et sortantes,

- les TSDUs sont systématiquement traitées au sein des routines `put` afin d'éviter tout délai supplémentaire,

- le contrôle de flux local (blocage de l'application) est réalisé en insérant un message spécial appelé "gros-message" dans la queue d'écriture (il s'agit en fait d'un message de taille supérieure à la taille maximale de la queue) et à le retirer lorsque la condition de blocage disparaît. Parallèlement on empêche la routine de `service` d'être schedulée lors de l'insertion du "gros-message" au moyen de `noenable`[9].

9. Un service Streams à définir de blocage/déblocage permettrait d'éviter l'utilisation artificielle de ce "gros-message" tout en étant plus performant (on se contente d'agir sur l'indicateur de saturation de la queue).

La gestion des buffers coté entrant

Ainsi qu'il a été précisé à la Section 3.4.3, le Stream-Head retourne systématiquement les données à l'application dès qu'elles lui arrivent, augmentant ainsi le nombre d'appels système de réception. Plusieurs stratégies sont envisageables pour remédier à cela :

- *TCP retient les données tant qu'elles ne sont pas acquittées* : c'est une solution partielle car les buffers applicatifs de taille supérieure à deux MSS (Maximum Segment Size) ne sont pas remplis[10]. Cette stratégie a cependant l'avantage de la portabilité puisqu'elle ne requiert aucune modification des applications ou de Streams.

- *TCP retient les données tant qu'il ne peut pas remplir le buffer applicatif* : le Stream-Head informe TCP de la taille du buffer de l'application au moyen d'un message spécial (`M_READ`). TCP conserve alors les données reçues tant qu'elles ne sont pas en quantité suffisante pour remplir ce buffer. Cette solution est partielle (les buffers de taille supérieure à la taille de la fenêtre de réception ne sont pas remplis), à moitié portable ([Streams90] ne définit les messages `M_READ` que pour les appels systèmes `read`, pas pour `getmsg`), et diffère les acquittements ce qui peut conduire à des situations d'interblocage.

- *TCP retient les données mais permet les acquittements* : c'est la même stratégie que précédemment mais TCP est ici autorisé à acquitter les données reçues. Les acquittements ne sont plus différés et les buffers applicatifs de taille supérieure à la fenêtre de réception peuvent désormais être remplis. Cependant cela conduit à une grosse utilisation mémoire puisque les données dans TCP monopolisent presque autant de place que le buffer de l'application.

  Une variante qui résout ce problème de mémoire consiste à ce que TCP copie les données directement dans le buffer applicatif (c'est de cette façon que BSD fonctionne). Cette technique autorise également une intégration copie/checksum coté entrant. Mais les problèmes de portabilité sont très importants (Section 3.5.3.2).

- *Définition d'une option qui force le Stream-Head à remplir le buffer applicatif* : cette technique résout le problème posé sans créer de difficultés particulières. Mais ce n'est pas portable (non défini dans [Streams90]) et dans le cas d'un trafic interactif où il n'est pas possible d'attendre la fin du remplissage du buffer, un message additionnel doit être envoyé au Stream-Head pour l'en informer. L'environnement Streams utilisé définit cette option mais uniquement dans le cas des appels systèmes `read`. En outres il est laissé à la charge de l'application de décider ou non de son utilisation (c'est elle qui a la connaissance du type de trafic attendu).

Notre implémentation démultiplexée utilise la seconde stratégie consistant à retenir les données dans TCP. L'environnement Streams a été étendu afin de générer un message `M_READ` dans le cas du `getmsg` également. Générer un message supplémentaire lors de chaque appel système `getmsg`/

10. La valeur 2*MSS provient de la stratégie d'acquittement de TCP. Lorsque les segments ont une taille maximale, TCP ne retourne un acquittement qu'une fois sur deux.

`read` peut sembler prohibitif, mais en fait son coût est faible face à celui d'un appel système. A la lumière de notre expérience nous pensons que la dernière solution est préférable.

3.5.2 Une implémentation au niveau utilisateur de TCP

Plusieurs techniques hautes performances reposent sur des implémentations de la pile de communication au niveau utilisateur : les protocoles composés pour les besoins d'une application spécifique (Section 2.2.1), les systèmes de communication à copies de données minimales, et certaines solutions ILP (Section 2.2.3). Afin d'avoir un environnement d'expérimentation pour ILP [Braun95a] et pour les techniques de composition de protocoles, nous avons conçu une implémentation au niveau utilisateur de TCP dérivée de BSD [Braun95b]. Elle est divisée en :

- un driver dans l'espace noyau : son rôle est de démultiplexer les paquets IP entrants vers la connexion TCP appropriée, c'est-à-dire vers l'application correspondante. Dans le cas d'expérimentations où la compatibilité avec les piles TCP/IP standards n'est pas essentielle, ce driver peut être remplacé par UDP pour lequel le calcul de checksum aurait été supprimé. Ces deux solutions ont des performances similaires.

- une librairie contenant l'exécutable de TCP qui est liée à l'application.

Nous avons du faire face à deux types de difficultés :

Les protocoles s'exécutent dans des espaces d'adressage différents

- Un lieu partagé de dépôt d'informations est requis, sous forme d'un démon dans l'espace utilisateur ou d'un driver noyau. Dans notre cas son seul but est de garantir l'unicité des numéros de port. Dans les expérimentations où la pile TCP/IP complète est déplacée au niveau utilisateur, il devra également contenir la table de routage IP, la table ARP, etc. La communication avec ce démon/driver est coûteuse ce qui conduit à l'utilisation de techniques de cache avec des mises-à-jour régulières [Maeda93, Jain94].

- Un démultiplexage précoce des paquets entrants est nécessaire. C'est le rôle de notre driver de démultiplexage. D'autres systèmes utilisent des composants de filtrage de paquets [Maeda92] ou bien de démultiplexage au niveau de la carte d'adaptation/driver réseau. La technique de filtrage a souvent des problèmes de performance lorsque le nombre de connexions augmente (mauvaise scalabilité) d'où de nombreux travaux pour résoudre cet aspect [McCanne93, Yuhara94, Bailey94, Schmidt94]. Le problème du démultiplexage au niveau carte d'adaptation est qu'il est presque entièrement limité à ATM (ou réseau similaire) et suppose que les VCIs puissent être alloués librement (Section 2.2.3).

- Il y a enfin des difficultés liées à la gestion des timers. L'implémentation noyau de TCP utilise deux routines (`tcp_fasttimo` et `tcp_slowtimo`) appelées périodiquement. Un unique changement de contexte permet le contrôle de tous les timers de toutes les connexions TCP. Ceci n'est plus possible et utiliser une stratégie similaire (routines appelées périodi-

quement) conduit à un grand nombre d'interruptions et changements de contextes. Notre implémentation repose sur l'emploi de l'appel système `select` au lieu des `recv/read`. La routine `tcp_fasttimo` n'est plus utilisée. L'application reçoit les données jusqu'à ce qu'il n'y en ait plus. TCP regarde alors si un acquittement doit être retourné. La réception d'un paquet déséquencé conduira également à la transmission d'un acquittement.

La position dans la pile de communication de la frontière utilisateur/noyau est déplacée

- Les APIs Socket et XTI imposent que l'application puisse librement réutiliser ses buffers au retour de l'appel système d'émission. Ceci est rendu possible par la copie de données entre les espaces utilisateur et noyau. Ce n'est plus le cas ici. Les donnés devant être conservées par TCP en cas de retransmission, une copie supplémentaire est faite dans un buffer interne à TCP. Une solution à ce problème nécessite de définir une nouvelle API qui éviterait cette copie par l'emploi d'un buffer circulaire [Ahlgren93, Maeda93]. Mais la compatibilité au niveau applicatif est alors compromise.

- Recevoir les données du réseau crée des problèmes de buffers. Afin d'éviter une copie de données intermédiaire, l'appel système `readv` est utilisé : l'en-tête TCP est copié dans un buffer temporaire et la partie données directement dans le buffer de l'application. La condition préalable est que le paquet ne contienne aucune option et ne soit pas déséquencé. Avec cette optimisation le nombre de copies de données durant une réception est le même que pour une implémentation noyau.

- Les traversées de frontières de domaines sont plus nombreuses. Les acquittements sont traités au niveau utilisateur et non plus dans le noyau. La taille des données traversant la frontière utilisateur/noyau est également plus faible : elle est désormais limitée par le PMTU (1500 octets avec Ethernet), alors qu'elle était fixée par l'application dans le cas d'une implémentation noyau, généralement avec une valeur élevée et multiple de un kilo-octet pour des considérations d'efficacité. Enfin l'utilisation du `select` conduit à doubler le nombre d'appels systèmes durant les réceptions (il doit être suivi d'un `recv`).

- Le support des appels systèmes `fork` et `exec` est problématique. Avec une implémentation noyau, seule l'application et peut être l'API sont concernées (Section 3.4.2). Avec une implémentation utilisateur, le contexte TCP est lui aussi dupliqué (`fork`) ou réinitialisé (`exec`).

3.5.3 Une méthode d'accès ILP

Dans cette section nous montrons comment la gestion des buffers peut être simplifiée et comment une technique ILP peut être introduite dans une implémentation TCP/IP traditionnelle. L'intégration ILP est limitée dans la mesure où elle ne concerne que la copie de données entre espaces utilisateur et noyau, et le calcul de checksum de TCP ou UDP. Parce que les problèmes sont totalement différents, nous distinguons le cas du flux sortant de celui du flux entrant.

3.5.3.1 Le cas du flux sortant

Les limitations des méthodes d'accès actuelles

Considérons le cas d'une méthode d'accès traditionnelle utilisée avec un protocole de type "stream" tel que TCP, c'est-à-dire voyant les TSDUs comme une succession d'octets susceptibles d'être réorganisés en segments comme bon lui semble. La gestion des buffers durant la copies de données entre les espaces utilisateur et noyau est alors faite sans considérations pour les frontières des futurs segments (elles ne sont pas connues) (Figure 17-gauche).

Ceci a plusieurs conséquences :

- un segment est généralement soit à cheval sur plusieurs buffers soit ne constitue qu'une partie d'un buffer. Cela complique les opérations de recherche, duplication, et libération de segments dans la liste de transmission de TCP. D'une part ces opérations reposent sur des calculs d'offset, d'autre part la duplication d'un segment chevauchant plusieurs buffers nécessite plusieurs descripteurs[11], un par buffer.

- un segment pouvant commencer n'importe où dans le buffer, aucune place ne peut être réservée pour les futurs en-têtes de protocoles. Un buffer supplémentaire doit être alloué dans ce but.

- parce les frontières de segments ne sont pas encore connues durant la copie entre espaces utilisateur et noyau, il n'est pas possible d'intégrer celle-ci avec le calcul de checksum qui lui se fait sur la base d'un segment.

Ces limitations n'existent pas avec un protocole de type datagrammes tel que UDP : à chaque TSDU correspond exactement un datagramme UDP. Un usage intelligent de cette caractéristique et le fait que UDP n'a pas à conserver de copies des données peut aisément résoudre les limites précédentes. Ainsi l'intégration copie/checksum a été ajoutée au driver UDP d'AIX/4.1.2.

Une méthode d'accès ILP

De cette discussion il ressort que les limitations de la méthode d'accès dans le cas de TCP proviennent de la connaissance tardive des frontières de segments. Nous proposons donc de *déplacer la fonctionnalité de segmentation des TSDUs depuis TCP vers la librairie de l'API* (XTI dans notre cas), au niveau utilisateur. Cette technique consiste à identifier dans le flux de données de l'application les frontières de segments, et à contrôler durant l'appel système la façon dont les données sont copiées dans les buffers noyaux (Figure 17-droite).

11. Les données ne sont pas copiées durant une duplication. A la place, un nouveau buffer est alloué qui pointe vers le buffer de données initial.

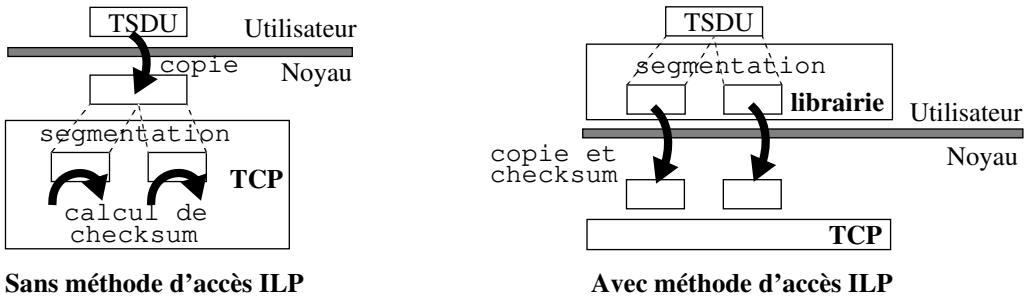

Figure 17: Comparaison des méthodes d'accès traditionnelle et ILP.

Les avantages sont multiples :

- un segment correspond à un buffer noyau, simplifiant ainsi les opérations de recherche, duplication et libération dans la liste de transmission de TCP,
- de la place peut être réservée dans le buffer noyau pour les divers en-têtes de protocoles avec de la place pour d'éventuelles options,
- la copie de données peut être intégrée au calcul de checksum, et
- le nombre d'appels systèmes peut être considérablement réduit lorsque l'application utilise de petites TSDUs et n'est pas préoccupée par la latence (FTP en mode ASCII par exemple). Dans ce cas l'API attend d'avoir suffisamment de données pour remplir un segment de taille maximale. En cas de TSDU isolée ou si la latence importe, l'application positionne un paramètre `PUSH` que la librairie reconnaît, et les données sont alors immédiatement passées à TCP.

D'un autre coté cette méthode d'accès crée un certain nombre de difficultés, en particulier :

- TCP peut être amené à resegmenter les "segments" anticipés par la librairie. C'est le cas lorsque TCP désire sonder la fenêtre de l'hôte distant [Stevens94] (segment de un octet). Dans ce cas, un nouveau buffer est alloué, les données y sont copiées et le checksum recalculé. La règle "un segment, un buffer" est ainsi préservée.
- Des copies de données supplémentaires peuvent être nécessaires dans la librairie. Cela survient lorsque le contrôle doit être retourné à l'application alors que les données n'ont pas encore été envoyées à TCP (même problème qu'à la Section 3.5.2).
- Enfin un nouvel appel système, `putextmsg`, a été conçu. Il reconnaît les frontières de blocs de données (segments identifiés par la librairie), alloue les buffers noyau en conséquence, et effectue l'opération de copie/checksum. Il est important que plusieurs segments puissent être envoyés en un unique appel système en raison de son coût élevé.

Une caractéristique importante de cette technique est d'être quasiment transparente aux applica-

tions : tout est caché dans la méthode d'accès (librairie, Stream-Head, TCP/UDP). La seule modification au sein de l'application consiste en la gestion du paramètre `PUSH`.

Cette technique a aussi quelques points communs avec le concept ALF dans le sens où la segmentation dans l'espace utilisateur permet d'avoir des ADUs égales aux unités de transmission de données (Section 2.2.2). Cependant cette méthode d'accès n'impose rien quant à l'aspect contrôle d'erreurs.

Enfin, si ce type d'intégration n'est pas récent ([RFC817, Clark89, Clark90] en discutent déjà, voir donnent des résultats de performance), nous montrons ici comment cela peut être intégré à TCP/IP et quels bénéfices on peut en attendre dans la pratique.

3.5.3.2 Le cas du flux entrant

La situation est complètement différente du coté du flux entrant. Ici les principaux problèmes qui se posent sont des contraintes d'ordonnancement entre les différentes tâches des protocoles : le checksum doit être vérifié tout d'abord, ce qui permet le traitement des en-têtes, et enfin les données sont copiées dans le buffer de l'application. Introduire ILP dans cette séquence impose :

- une hypothèse optimiste : le paquet doit tout d'abord être supposé sans erreur. L'en-tête est alors utilisé afin de déterminer si le paquet est dans l'ordre, et si c'est le cas, les données sont copiées dans le buffer et le checksum calculé dans la foulée. Enfin l'hypothèse d'intégrité du paquet est vérifiée et la réception validée.

- la connaissance des caractéristiques des buffers applicatifs (taille, adresse) : le driver TCP ou UDP a besoin de ces informations lors de la copie alors qu'elles sont traditionnellement détenues par le Stream-Head ou la couche Socket.

  Avec Streams une réorganisation complète de la méthode d'accès est nécessaire : le message `M_READ` doit être étendu pour contenir également l'adresse du buffer, les fonctions `getmsg`/`read` du Stream-Head doivent être modifiées puisque les données seront déjà copiées dans le buffer, enfin un nouveau type de message doit être créé afin d'avertir le Stream-Head que l'appel système de réception est fini (il n'y a plus de message `T_DATA_IND` remonté au Stream-Head).

  Avec BSD la solution consiste à intégrer le traitement de la couche Socket dans TCP (c'est permis par BSD). La conséquence est que le traitement de TCP est désormais effectué dans la thread de l'application [Jacobson93].

Cette comparaison montre qu'une méthode d'accès ILP coté réception est plus aisément et efficacement implémentée dans un environnement BSD que Streams. Le principe d'indépendance des composants promu par Streams soulève ici de nombreux problèmes, et c'est la raison pour laquelle cela n'a pas été inclus à notre plate-forme expérimentale. Sinon avoir une méthode d'accès ILP coté réception n'a aucun impact sur les applications. Plus encore que pour le flux sortant, tout est caché dans le noyau.

3.6 LES OUTILS D'ÉVALUATION DE PERFORMANCES

Nous décrivons maintenant les différents outils à notre disposition pour la phase d'analyse de performances. Nous distinguons tout d'abord un environnement intégré d'évaluation de performances que nous avons conçu, puis trois outils d'AIX qui permettent plusieurs types d'études complémentaires. Nous essayons à chaque fois de mettre en évidence les possibilités et limites de l'outil.

3.6.1 Un environnement intégré d'évaluation de performances

Afin de faciliter l'évaluation de performances de notre plate-forme expérimentale, nous avons conçu, amélioré, et intégré plusieurs outils. Cet environnement permet la collecte automatique de mesures de performances, leur analyse, et la mise en forme graphique des résultats [Roca95a]. Il est composé de :

- *bench/benchd* : cette application de type client/serveur effectue des transferts de données intensifs. Elle collecte à chaque extrémité des statistiques au niveau applicatif du transfert de données (débit, temps d'aller-retour, remplissage des buffers de réception, etc.) ainsi que des événements systèmes (charge par processeur, utilisation des verrous, etc.).

  Divers scénarios sont possibles, depuis les transferts de données de masse (type FTP) pour l'étude du débit, jusqu'aux transferts de données en mode écho (type telnet) pour les études de RTT, en utilisant soit TCP soit UDP, une ou plusieurs connexions, etc. Plusieurs paramètres peuvent être positionnés : taille des sockets, taille des buffers de l'application, mode `NO_DELAY` de TCP, etc.

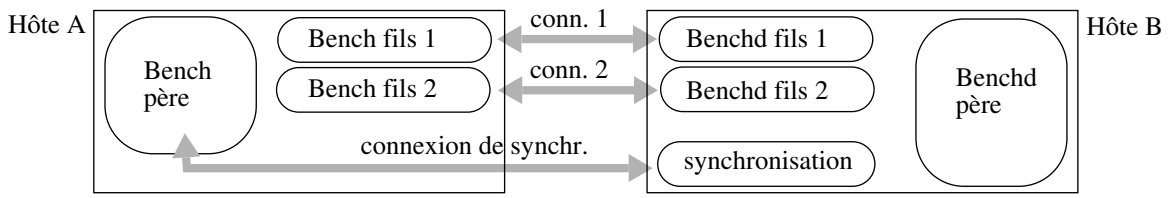

Figure 18: Organisation des applications client/serveur bench/benchd.

La Figure 18 montre la configuration de bench/benchd durant des tests mettant en oeuvre deux connexions. A chaque extrémité un processus est créé par le processus père afin de prendre en charge cette connexion. Une connexion supplémentaire permet de synchroniser les deux extrémités et d'échanger des informations (taille des TSDUs, nombre de connexions) qui sont ensuite enregistrées dans les fichiers de trace. N'étant active qu'en début et fin de session de test, l'influence de cette connexion supplémentaire n'est pas significative.

- *bencht/benchdt* : ce sont des fichiers de commandes shells qui permettent de lancer bench/benchd répétitivement en faisant varier le paramètre “taille de TSDU” du client.

- *benchsort* : ce fichier `awk` analyse les diverses traces relatives aux activités réseau et système générées par bench/benchd. benchsort crée en sortie des fichiers de données et de commandes compatibles avec gnuplot.

- *gnuplot* : un outil GNU spécialisé dans la création de courbes [gnuplot94].

3.6.2 Les autres outils

3.6.2.1 L'outil de trace

L'outil AIX de trace (`trchk` et `trcrpt`) permet la collecte d'informations relatives aux événements applicatifs et systèmes. Un estampillage temporel est associé à chaque trace. Son utilisation se fait en trois étapes :

- ajout des sondes à des points clés du composant testé,

- exécution du programme avec collecte de traces, et

- utilisation d'un outil d'analyse et de mise en forme du fichier binaire contenant les traces.

La précision des mesures obtenues dépend de deux facteurs :

- la précision du timer haute-résolution de la machine : elle est supérieure à 10 cycles horloges ce qui sur notre DPX/20 420 correspond à 0.2 µs.

- l'overhead du système d'enregistrement de traces lui même. Il est constant, autour de 1.5 µs, soit 50 instructions.

Cependant des expérimentations en grandeur nature ont montré que les mesures obtenues ont une large dispersion. Dans les cas extrêmes, une variation de 100% du temps de transmission d'une TSDU a été observée. L'influence des événements systèmes principaux tels que les changements de contextes, les interruptions, la gestion mémoire virtuelle etc., a déjà été éliminée. Ce phénomène d'instabilité a deux origines :

- algorithmique : suivant la présence ou non d'un buffer libre dans un cache du système de gestion mémoire, le temps d'allocation d'un buffer peut varier de façon conséquente. Ce niveau de précision n'est pas disponible dans les informations collectées et ajouter davantage de sondes n'est guère possible.

- matérielle : par exemple le comportement des caches du processeur n'est pas visible.

En conclusion nous pouvons dire que ce type d'outil de trace :

- est utile pour étudier le comportement système de bas niveau : gestion des threads, interruptions, préemptions, etc.,
- fournit une vision de granularité intermédiaire du comportement et des performances d'un composant. Plus il y a de traces, meilleure est la visibilité, mais aussi plus élevé est l'overhead. Cela va dans le sens du principe d'incertitude d'Heisenberg, à savoir qu'observer un système perturbe ce système, et
- rend délicates les évaluations de performances fines en raison des problèmes d'instabilité observés et des limites de l'outil. Les résultats donnés à la Section 4.2.1 sont ainsi des moyennes calculées sur plusieurs dizaines d'échantillons.

3.6.2.2 L'outil de comptage d'instructions

Cet outil AIX permet de dénombrer dynamiquement les instructions exécutées. Il a plusieurs intérêts par rapport à l'outil de trace :

- Il fonctionne avec tous les exécutables : il n'y a ni sonde à insérer, ni paramètre de compilation à ajouter. Toutes les routines traversées durant l'exécution sont donc prises en compte, y compris celles résultant d'un éventuel aléa algorithmique (Section 3.6.2.1).
- L'outil d'analyse et de mise en forme des résultats est très puissant et permet d'obtenir entre autres des rapports de type profiling. Des analyses fines sont donc aisément permises.
- Si le temps d'exécution du système étudié est très sérieusement affecté (l'outil d'analyse est actif), en revanche le nombre d'instructions ne l'est pas (la présence de l'outil n'ajoute pas d'instruction parasite).
- Les résultats sont en outres extrêmement stables entre les essais. Ceci est le résultat d'un mécanisme de convergence intégré à l'outil : celui-ci fait plusieurs passes et ne s'arrête que lorsqu'il y a convergence.
- D'un autre coté cet outil ne tient pas compte d'un aspect essentiel : le taux d'instructions exécutées par cycle qui varie très largement ainsi que nous le montrons à la Section 4.2.3.

3.6.2.3 L'outil d'analyse des contentions d'accès aux verrous

L'outil `lockstat` fournit des informations sur l'utilisation et les contentions d'accès aux verrous sur une période donnée grâce à des informations collectées automatiquement par le système d'exploitation. Cet outil a été intégré à notre environnement d'évaluation de performances ce qui permet de visualiser l'évolution de l'utilisation et des contentions sur les verrous en fonction de l'évolution des paramètres de transfert de données.

Si cet outil est commode, il a une limitation majeure : il ne tient pas compte du temps d'attente active lorsqu'il y a contention[12], cette information n'étant pas collectée par le système d'exploitation. Seul le nombre de contentions est pris en considération. Cette caractéristique peut sous estimer certains résultats, en particulier les contentions d'accès aux verrous qui sont conservés longtemps.

12. En raison de leur faible surcoût et du fait qu'ils puissent être indifféremment utilisés dans un contexte thread ou routine d'interruption, le système d'exploitation emploie essentiellement des verrous à attente active ("spin locks") plutôt que des verrous de type écrivain/lecteurs susceptibles de bloquer la thread en cas de conflit.

CHAPITRE 4

Etude de Performances

Ce chapitre rend compte de l'étude de performances que nous avons menée sur la plate-forme expérimentale afin d'évaluer l'opportunité des solutions que nous avons décrites au chapitre précédent. Une caractéristique importante de ce travail est qu'il est fondé sur des essais en grandeur nature et non sur des simulations, des micro-benchmarks, ou encore des prototypes implémentés dans un environnement de simulation au niveau utilisateur.

Cette étude comporte deux approches : (1) *des expériences de haut niveau (mesures de débit et latence au niveau utilisateur) et* (2) *des expériences de bas niveau (traces systèmes et comptages d'instructions). Ces deux approches sont complémentaires : les expériences de haut niveau permettent l'analyse du système de communication de façon globale, en prenant en compte son comportement dynamique. A l'opposé, les expériences de bas niveau permettent une analyse détaillée, montrant des phénomènes cachés à l'approche de haut niveau. Une discussion générale synthétisant tous les résultats obtenus est proposée au Chapitre 5.*

4.1 EXPÉRIENCES DE HAUT NIVEAU

Les expériences de haut niveau consistent en des mesures de débit et de latence au niveau utilisateur. Elles ont été effectuées au moyen de l'environnement d'évaluation de performances décrit à la Section 3.6.1. L'outil bench/benchd utilisé est suffisamment flexible pour permettre une large gamme d'expérimentations.

Les tests mettent en jeu deux types de transferts : les transferts de masse (type FTP) pour lesquels l'émetteur s'efforce de saturer le récepteur, et les transferts en mode écho (type telnet) où toutes les données envoyées sont retournées en écho. Ces deux types de mesures permettent d'analyser les aspects débit et latence du système de communication.

Dans ces tests le buffer de réception de l'application est systématiquement de 8 kilo-octets, et les fenêtres d'émission/réception de 16 kilo-octets. Chaque séquence de test dure entre 10 et 30 secondes et le point de mesure retenu est la moyenne sur cet intervalle. Cela correspond à un échange de quelques milliers à quelques centaines de milliers de TSDUs suivant leur taille et le type de transfert choisi.

La plupart des tests de haut niveau ont été effectués en rebouclé mémoire. La raison en est qu'un réseau Ethernet est très aisément saturé avec les stations de travail actuelles. Ainsi en utilisant des TSDUs d'un kilo-octet, l'adaptateur et le médium Ethernet sont saturés alors que le processeur de la station émettrice est chargé à 35% seulement. De la même façon un quadri-processeur ESCALA sature aisément un anneau FDDI tout en ayant une charge moyenne des processeurs de 20 à 45%.

Les machines utilisées durant ces test sont le DPX/20 modèle 420 décrit à la Section 3.2.2 et un ESCALA octo-processeur utilisant des PowerPC 601 cadencés à 66 MHz. La nature du système d'exploitation revêt une importance capitale. Il s'agit soit de l'AIX/3.2.5 (DPX/20), soit de l'AIX/4.1.2 dans une version monoprocesseur (DPX/20) ou multiprocesseur (ESCALA).

4.1.1 Comparaison des différentes architectures

4.1.1.1 Les piles TCP/IP BSD et Streams classiques

La Figure 19 montre les performances des piles BSD et Streams classiques. Les transferts se font au travers d'une connexion TCP unique au dessus d'une interface rebouclée mémoire (MSS de 1460 octets). La machine utilisée est un DPX/20 sous AIX/3.2.5.

⇨ Ces expériences montrent les *performances limitées de la pile Streams classique à la fois en terme de débit et de latence*.

La stratégie de gestion des buffers de réception en est largement responsable : le Stream-Head remonte les données à l'application dès qu'elles arrivent au lieu d'optimiser le remplissage du buffer de réception (Section 3.4.3). Le nombre accru d'appels systèmes qui en résulte justifie les médiocres performances en débit (Figure 19-gauche). Du point de vue latence (Figure 19-droite.), cette stratégie impose également un cycle `recv`/`send` supplémentaire à chaque frontière de MSS, puisqu'un paquet supplémentaire est alors reçu. Au contraire la pile BSD rassemble les données dans la socket de réception avant de les retourner à l'application. La différence de comportement est également visible à la frontière de 8 kilo-octets (taille du buffer de réception de l'application). Avec la pile BSD un nouveau cycle recv/send est nécessaire, d'où la marche que l'on observe, alors que ce buffer n'est de toutes façons jamais rempli dans le cas de Streams.

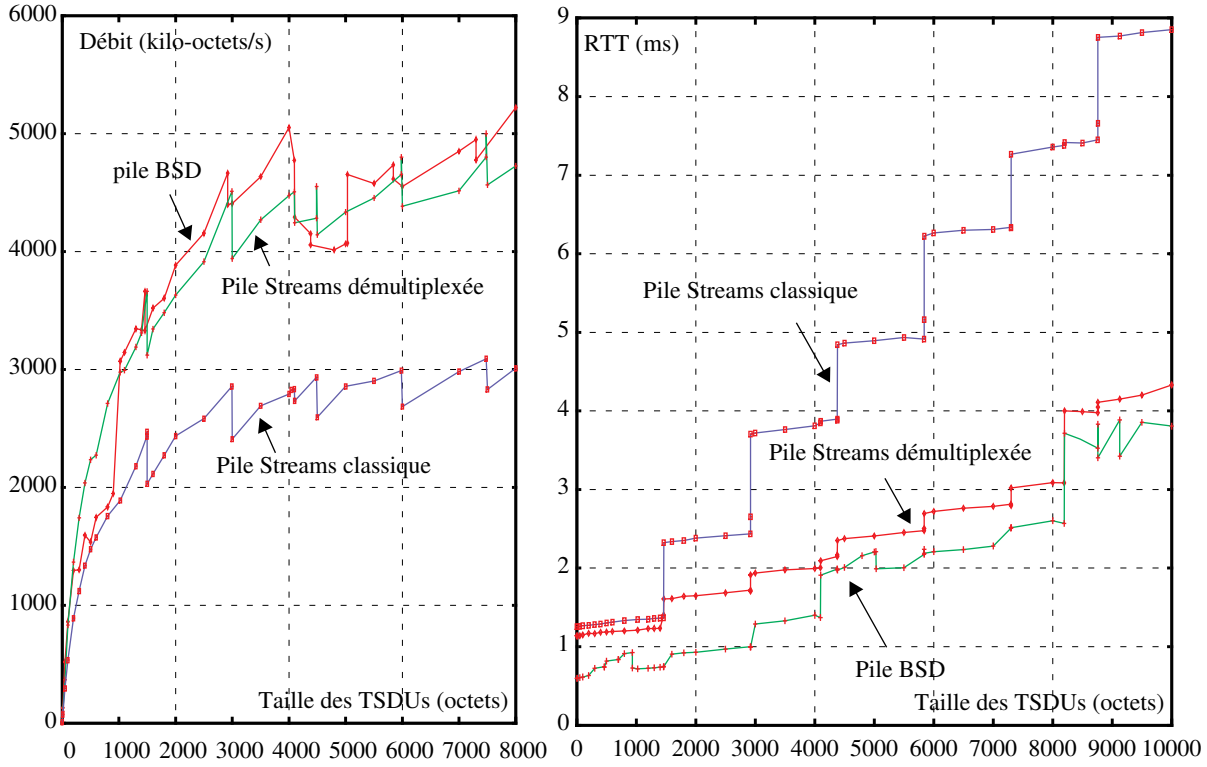

Figure 19: (gauche) Comparaison des débits durant des transferts de données de masse; (droite) Comparaison des RTTs durant des transferts en mode écho; DPX/20 sous AIX/3.2.5.

4.1.1.2 La pile TCP/IP Streams démultiplexée

Les expériences de base

Nous commençons avec des expériences qui ne tirent aucun profit direct de l'architecture démultiplexée de cette pile. Les conditions sont les mêmes que précédemment.

⇨ La Figure 19-gauche montre qu'*une implémentation Streams soigneusement conçue peut quasiment atteindre le même niveau de débit qu'une implémentation BSD* sous AIX/3.2.5.

La pile BSD est en effet pénalisée par sa gestion mémoire qui alloue jusqu'à quatre `mbufs` (228 ou 236 octets) pour les requêtes entre 1 et 936 octets, et un unique `cluster` de 4 kilo-octets au dessus. Le même phénomène se produit aux multiples de 4 kilo-octets. Cela se traduit par des chutes de performances visibles sur les courbes. La pile démultiplexée ayant une meilleure gestion mémoire parvient à dépasser BSD dans ces zones.

⇨ En revanche la pile démultiplexée a un *moins bon comportement avec les tests de latence* (Figure 19-droite).

Ceci est lié à la stratégie de gestion des buffers de réception utilisée (Section 3.4.3). Le coût de la création d'un message M_READ pour chaque appel système de réception n'est plus compensé par

le remplissage de ce buffer (n'arrive qu'avec des TSDUs de 8 kilo-octets). Ceci est particulièrement vrai avec les petites TSDUs où ce coût est proportionnellement plus élevé.

Analyse du contrôle de flux

Saturer un réseau Ethernet ou FDDI est aisé avec les stations de travail actuelles. Nous désirons maintenant apprécier l'opportunité d'avoir un mécanisme de contrôle de flux local (cas de la pile démultiplexée, Section 3.5.1.2) qui bloque l'application dans le cas où la carte d'adaptation et/ou le médium sont saturés. Ce contrôle de flux Streams est donc utilisé en tant que contrôle d'accès au médium. Pour cela nous juxtaposons un flux UDP à une connexion TCP entre deux DPX/20 reliés par un réseau Ethernet privé, sous AIX/4.1.2. Si TCP peut s'adapter à la bande passante disponible grâce à ses mécanismes de contrôle de flux/débit, UDP ne le peut pas.

Table 5: Débits coté émission et réception dans le cas des piles BSD et démultiplexée au dessus d'Ethernet; DPX/20 sous AIX/4.1.2.

taille des TSDUs et protocoles		pile TCP/IP classique BSD		pile TCP/IP démultiplexée Streams	
		émetteur (kilo-octets/s)	récepteur (kilo-octets/s)	émetteur (kilo-octets/s)	récepteur (kilo-octets/s)
512	TCP	0	0	244	244
	UDP	2244	511	597	470
1024	TCP	0	0	165	165
	UDP	4703	964	780	780
4096	TCP	0	0	145	145
	UDP [a]	9782	0	851	851

a. Les datagrammes UDP sont ici fragmentés par IP. Ainsi perdre un seul fragment IP conduit à perdre le datagramme dans sa totalité.

La Table 5 montre que l'absence de contrôle de flux local (cas de BSD) conduit UDP à monopoliser toute la bande passante au détriment de TCP. Le fait que le débit UDP en réception ne soit qu'une fraction du débit UDP en émission montre également qu'une portion significative des paquets est perdue. Ces deux problèmes sont évités dès lors que l'on introduit un contrôle de flux local.

Ces essais peuvent paraître artificiels : UDP n'est pas adapté à un transfert de données de masse à moins qu'il existe un mécanisme de rétroaction (Section 2.2.1). Dans ce cas une cohabitation UDP/TCP est possible.

⇨ Cependant le mécanisme de contrôle de flux local au niveau système, et les mécanismes de contrôle de flux/débit au niveau protocoles ou applications sont *complémentaires*. *L'avantage du contrôle de flux local est de permettre une bonne réponse aux phénomènes de congestion transitoires.*

Ces congestions sont dues par exemple à des transferts UDP brefs mais importants sur une machine déjà chargée. En raison de la faible durée du transfert de données, celui-ci n'est contrôlé par aucun mécanisme de rétroaction[1]. Dans ce cas le contrôle de flux local :

- évite aux paquets d'être perdus. Ainsi TCP n'a pas à réduire sa fenêtre de congestion si l'engorgement n'est que temporaire,
- réagit immédiatement aux congestions. Les autres solutions reposent sur des messages d'indication d'erreur ou sur des arrivées à échéances de timers, ce qui ajoute un RTT voir davantage avant que l'émetteur ne réagisse, et enfin
- est indépendant de la nature des protocoles utilisés et du nombre de connexions actives.

⇨ Au delà de ces aspects, *l'intérêt majeur de ces tests est de valider le mécanisme de blocage/déblocage des CC*. L'algorithme utilisé pour décider quand bloquer ou débloquer dépend ensuite du contexte (Section 3.5.1.2).

4.1.1.3 Implémentation au niveau utilisateur de TCP

Cette section compare les performances de l'implémentation au niveau utilisateur de TCP avec la pile BSD classique. L'implémentation utilisateur est configurée pour travailler au dessus de UDP/IP BSD pour lequel le calcul de checksum a été invalidé. Les tests consistent en un transfert de données de masse en rebouclé mémoire sous AIX/3.2.5.

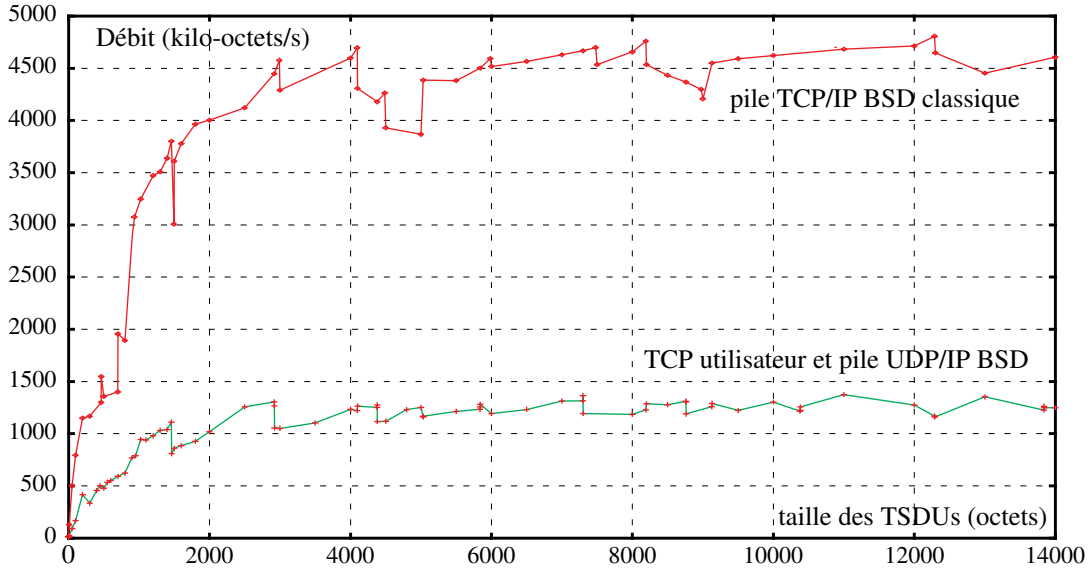

Figure 20: Comparaison des débits des implémentations BSD et TCP au niveau utilisateur; DPX/20 sous AIX/3.2.5.

1. Ce problème est similaire à celui des hordes de souris par opposition aux éléphants sur certaines liaisons transatlantiques [Crowcroft95].

⇨ La Figure 20 montre clairement les *performances très limitées de l'implémentation au niveau utilisateur de TCP.*

Ceci est le résultat de l'éclatement en plusieurs espaces d'adressages et du déplacement de la frontière utilisateur/noyau. Il y a plus de copies de données et plus de traversées de domaines (Section 3.5.2). Une pile de communication de niveau utilisateur implémentée au dessus d'un système d'exploitation standard ne peut pas rivaliser avec une implémentation noyau. [Edwards95] montre de la même façon qu'une implémentation de TCP/IP au niveau utilisateur très optimisée ne peut rivaliser avec une implémentation noyau possédant le même niveau d'optimisation. Les avantages des implémentations au niveau utilisateur sont ailleurs, ainsi que nous le montrons à la Section 5.2.3.2.

4.1.2 Evaluation des architectures sur des multiprocesseurs

Nous abordons maintenant des essais destinés à évaluer le comportement de notre pile démultiplexée en environnement multiprocesseurs. Nous présentons d'abord les mesures de performances, pour ensuite nous intéresser aux problèmes de contention sur les verrous.

4.1.2.1 Performances et taux d'accélération

La Figure 21 montre les performances des piles BSD et démultiplexée Streams durant des transferts de masse avec TCP au dessus d'une interface rebouclée mémoire. Les machines utilisées sont des quadri et octo-processeurs sous AIX/4.1.2. Les débits obtenus individuellement sur chaque connexion sont cumulés afin de mettre en évidence l'impact de l'augmentation du nombre de connexions sur l'utilisation des processeurs.

Tout d'abord, saturer les processeurs requiert qu'il y ait un nombre de connexions au moins égal au nombre de processeurs. C'est la raison pour laquelle les courbes correspondant à une seule connexion TCP ont des performances limitées.

⇨ La Figure 21 montre que la pile BSD a des performances plus élevées que la pile démultiplexée avec quatre processeurs. Ceci est particulièrement vrai avec les petites TSDUs.

Ceci est fortement lié aux surcoûts très élevés imposés par l'environnement Streams pour l'AIX/4.1.2 (Section 4.2.1.3).

⇨ La situation est moins nette avec huit processeurs. La pile BSD connaît même une chute de performances lorsque le nombre de connexions augmente (4, 8 puis 16 connexions) probablement due à un accroissement des contentions sur les verrous. Ce comportement n'est pas observé avec la pile démultiplexée.

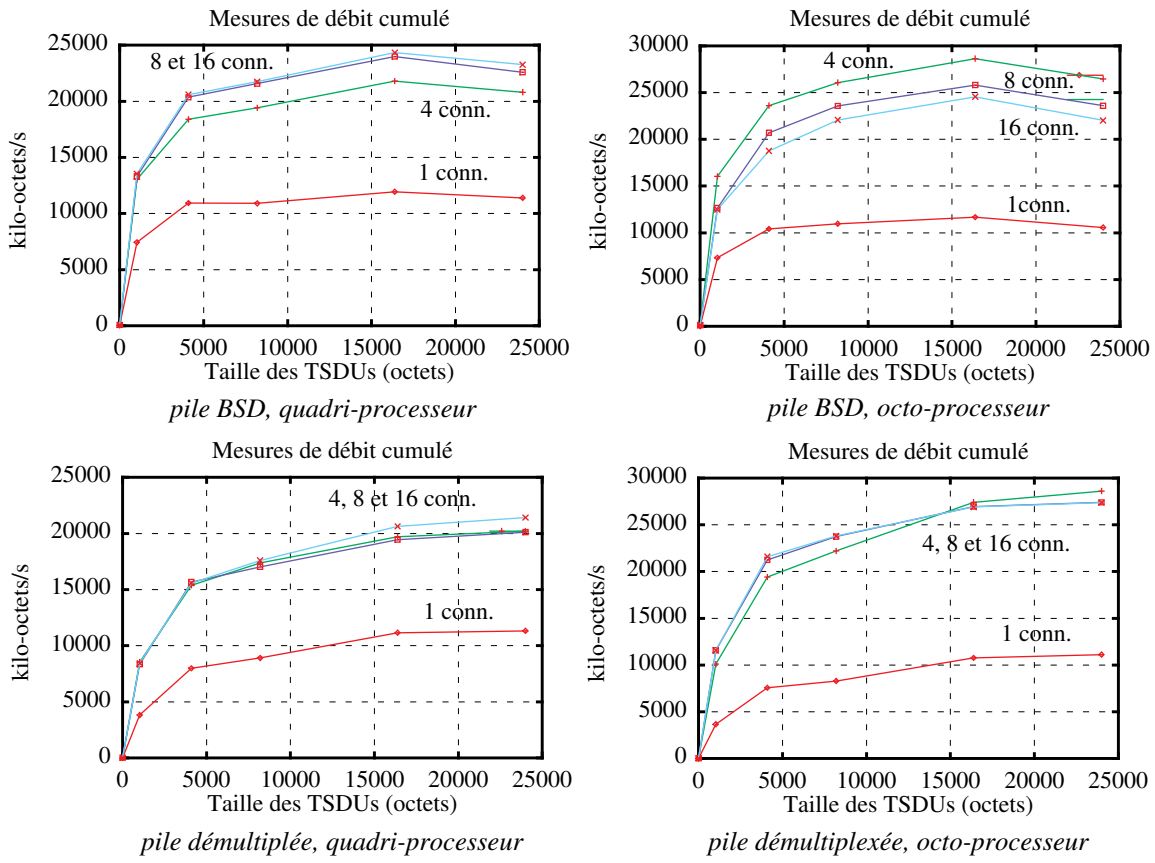

Figure 21: Comparaison des débits cumulés des piles BSD et démultiplexée; ESCALA quadri et octo-processeur sous AIX/4.1.2.

Les taux d'accélération (rapport entre les performances sur un monoprocesseur et un quadri ou octo-processeur) qui s'y rapportent pour des TSDUs de 16 kilo-octets sont donnés dans la Table 6. Les situations pour lesquelles le nombre de connexions est inférieur au nombre de processeurs ne sont pas prises en compte car le taux de charge des processeurs n'est pas maximal ce qui fausse les calculs.

Table 6: taux d'accélération des piles BSD et démultiplexée; ESCALA quadri et octo-processeur sous AIX/4.1.2.

	pile TCP/IP BSD		**pile TCP/IP démultiplexée**	
	4 connexions	**8 connexions**	**4 connexions**	**8 connexions**
mono vers quadri-processeurs	2,0	2,2	2,1	2,6
mono vers octo-processeurs	NS	2,4	NS	3,6

Cette table montre deux choses :

⇨ En dépit de performances plus faibles, *la pile démultiplexée Streams a un meilleur taux*

d'accélération que la pile BSD.

⇨ Cependant *ces taux d'accélérations sont faibles*, bien en deçà des valeurs optimales (égales au nombre de processeurs).

4.1.2.2 Contentions d'accès aux verrous

La section précédente a montré que la parallélisation du système de communication est peu performante. Des mesures sur l'évolution du nombre de changements de contextes ont montré que ces derniers évoluaient de façon similaire aux taux d'accélération. Les changements de contextes ne constituent donc pas, dans le cas présent, le facteur limitant. Nous nous sommes donc intéressés aux contentions d'accès aux verrous. Ceux-ci ont été classés en différentes catégories (Table 7).

Table 7: Classification des verrous.

Nom	**Applicabilité**	**Description**
verrous mémoire	BSD et Streams	services de gestion mémoire
verrous Socket	BSD	protection des contextes TCP et des sockets d'émission/ réception
autres verrous INET	BSD et Streams	liste de contextes TCP, table de routage, queue d'interruption IP, etc.
verrous Streams	Streams	environnement Streams, ES de protection des contextes TCP
verrous de gestion des threads	BSD et Streams	appels systèmes, gestion des threads et des timers
verrous OS divers	BSD et Streams	le restant

Les Figure 22, Figure 23 et Figure 24 montrent les statistiques de contentions d'accès aux verrous dans le cas de transferts de données de masse avec TCP sur un quadri-processeur. Parce que l'outil `lockstat` compte le nombre de conflits plutôt que le temps cumulé des conflits (Section 3.6.2.3), les contentions relatives aux verrous qui sont conservés longtemps (Socket par exemple) peuvent être sous-estimées. Cela peut biaiser l'interprétation des courbes obtenues.

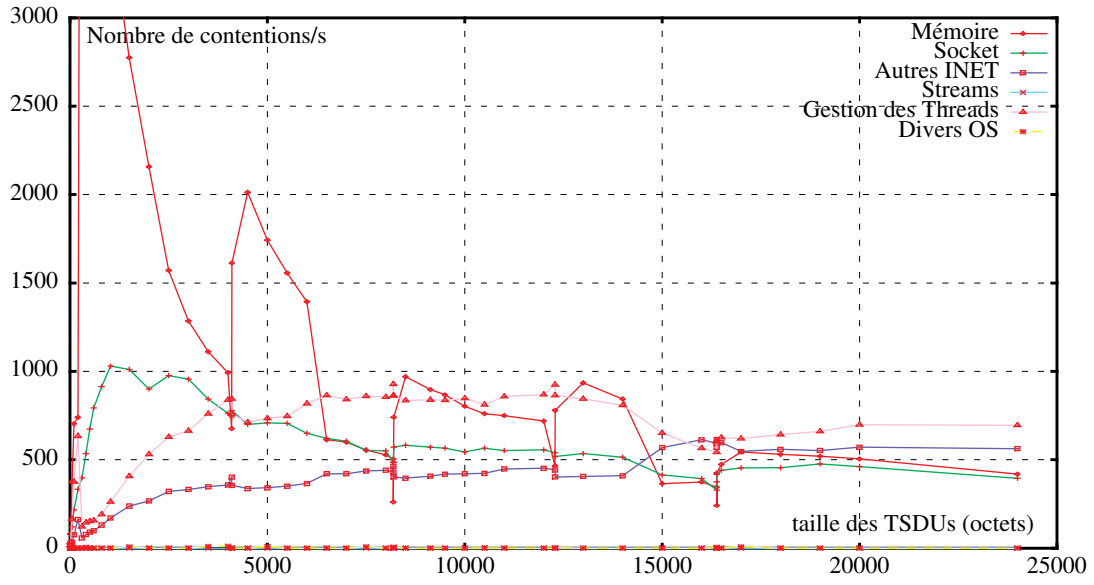

Figure 22: Contentions pour la pile BSD, 4 connexions, quadri-processeur.

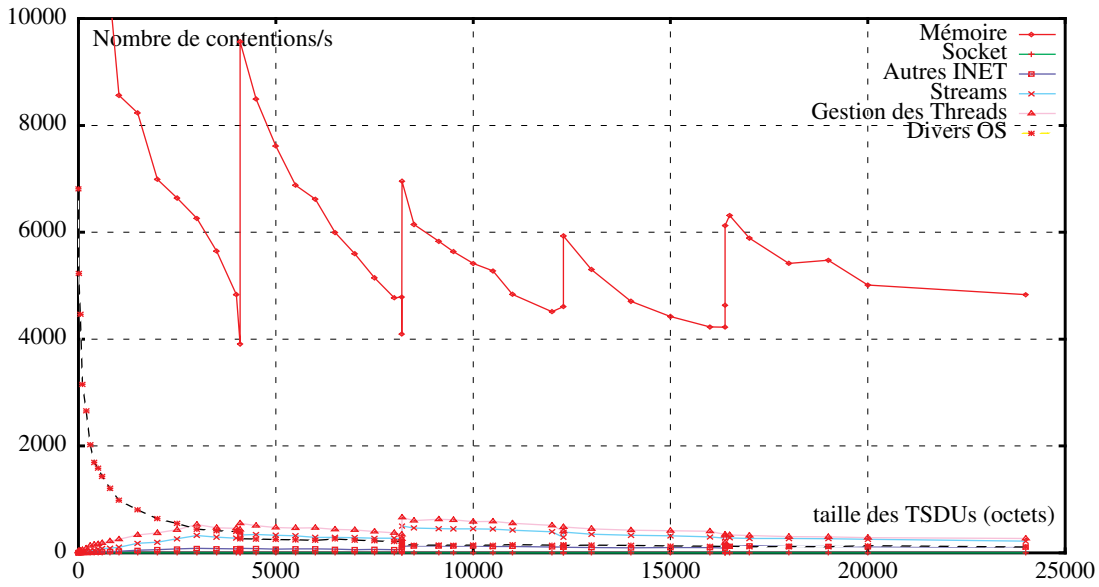

Figure 23: Contentions pour la pile démultiplexée, 4 connexions, quadri-processeur.

Un agrandissement des courbes de contentions pour la pile démultiplexée est donnée ci-dessous.

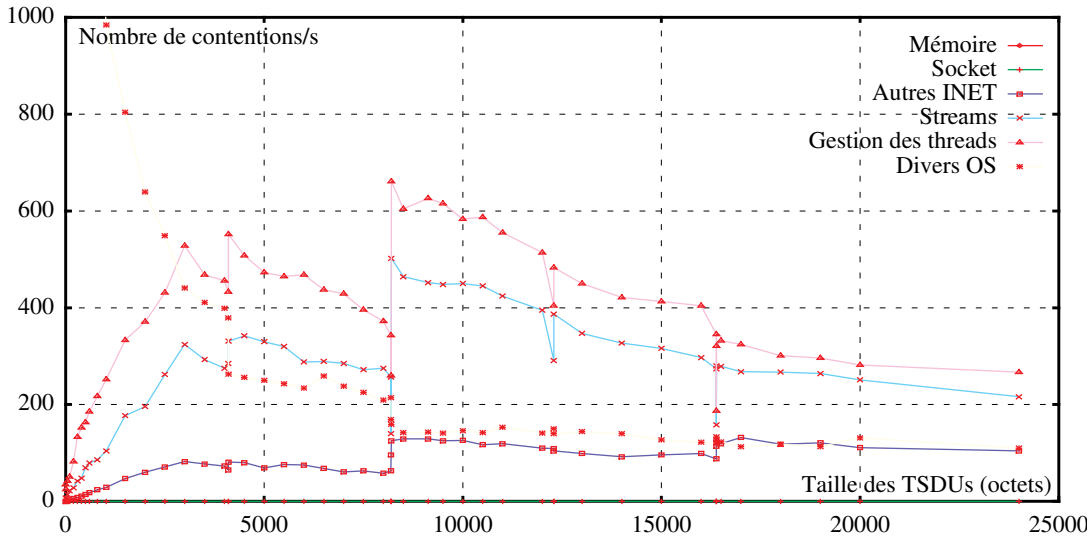

Figure 24: Agrandissement des courbes de contentions pour la pile démultiplexée (ne montre pas les contentions mémoires), 4 connexions, quadri-processeur.

⇨ Ces expériences montrent que *les contentions mémoire dominent*.

Ceci est tout particulièrement vrai dans le cas de la pile démultiplexée. La raison en est que les implémentations Streams font un grand usage de petits buffers (Section 4.2.2.2) : ils sont utilisés pour contenir les primitives TPI, NPI, et DLPI, les messages `M_READ` et les messages d'arrivée à échéance de timers.

On remarque aussi que les petites TSDUs créent systématiquement une grosse activité mémoire. Ceci est un effet secondaire : **(1)** du faible coût des manipulations de données (copie et calcul de checksum), et **(2)** de l'algorithme de Nagle [RFC896] qui regroupe les petites TSDUs afin d'amortir les coûts de transmission. Chaque appel système induisant peu de traitements, il sont plus nombreux et de ce fait créent une forte activité d'allocation/libération.

Le comportement des contentions mémoire au voisinage immédiat de zéro est intéressant : il n'y a aucune contention dans le cas de BSD alors que la contention est d'emblée maximale avec la pile démultiplexée. Ceci est une autre conséquence de l'opposition intégration/indépendance des composants : avec BSD la couche Socket essaie systématiquement de copier les données de l'application à la fin du précédent buffer de transmission de la socket. La plupart du temps cela permet d'éviter de recourir à une allocation de `mbuf`. Au contraire le Stream-Head n'a pas accès à la liste d'émission de TCP et ainsi alloue systématiquement un nouveau buffer pour y copier les données de la TSDU. Ceci conduit à de nombreuses allocations/libérations de petits buffers d'où d'importants risques de conflits.

⇨ La seconde cause de contentions, spécifique à BSD, est le verrou sur les Sockets en dessous de 4 kilo-octets, pour devenir ensuite le verrou de gestion des threads. Les autres verrous TCP/IP n'arrivent qu'en quatrième position avec une importance qui décroît peu à peu avec la taille des TSDUs (ces verrous sont principalement utilisés lorsque des paquets sont transmis).

⇨ Dans le cas de la pile démultiplexée les contentions autres que mémoires sont proportionnellement très faibles. Les résultats montrent que si la synchronisation Streams est la principale source d'*utilisation* de verrous (avant la gestion mémoire, résultat non montré sur les courbes), cela ne crée en revanche que peu de conflits. Ceci est lié au mécanisme de synchronisation Streams (Section 3.3.2) et au parallélisme de type connexion utilisé.

4.1.2.3 Bilan

Les expérimentations sur machines multiprocesseurs ont montré que les performances de TCP sont faibles. Ceci est en particulier dû à la synchronisation d'origine système (gestion mémoire et thread) qui est la principale source de contention, bien avant la synchronisation d'origine protocole. L'utilisation de systèmes d'exploitation multiprocesseurs sur une machine monoprocesseur a également un coût. Cet aspect est étudié à la Section 4.2.1.3.

4.1.3 Evaluation de ILP

Deux types d'expériences ont été réalisées :

- des micro-tests qui donnent le gain théorique maximal qui peut être tiré de l'intégration copie de données/calcul de checksum, et
- des expériences en grandeur nature avec la méthode d'accès ILP qui en prennent en compte tous les autres avantages et limites.

4.1.3.1 Bénéfices liés à l'intégration copie/checksum

Les performances des routines de copie et de calcul de checksum sont exprimées par les équations 1 et 2. Elles donnent le temps de traitement (µs) en fonction de la taille du bloc de données (octets) :

$$t_{\text{copie Utilisateur->Noyau}}(\text{taille}) = 3.5 + 0.0213 * \text{taille} \quad (1)$$

$$t_{\text{checksum TCP}}(\text{taille}) = 7.5 + 0.0286 * (40 + \text{taille}) \quad (2)$$

En les additionnant on obtient l'équation de coût total des manipulations de données :

$$t_{\text{total_sans_intégration}}(\text{taille}) = 12.1 + 0.0499 * \text{taille} \quad (3)$$

Au contraire, avec une intégration copie/checksum :

$$t_{\text{copie/checksum}}(\text{taille}) = 11.5 + 0.0261 * \text{taille} \quad (4)$$

$$t_{\text{TCP pseudo-en-tête}} = 10 \quad (5)$$

soit en les ajoutant :

$$t_{total_avec_intégration}(taille) = 21.5 + 0.0261 * taille \quad (6)$$

Remarques : Le coefficient multiplicateur de l'équation 4 est inférieur à celui de l'équation 2 car la routine de copie/checksum est en assembleur alors que la routine de checksum est en C. L'addition de 40 octets dans l'équation 2 est liée à la présence du pseudo-en-tête.

L'équation 6 confirme qu'intégrer les routines de copie et de calcul de checksum seulement est déjà intéressant (coefficient multiplicateur plus faible). Cette intégration compense les coûts fixes (constantes des équations) à partir de 400 octets. La Table 8 donne le gain de performances pour deux tailles de blocs particulières.

Table 8: Gains liés à l'intégration copie/checksum.

Taille des blocs de données	**temps de traitement sans intégration**	**temps de traitement avec intégration**	**gains**
1460 octets	85.0 µs	59.6 µs	30%
3460 octets	184.8 µs	111.8 µs	39%

⇨ Ainsi plus le bloc de données est important, meilleur est le gain. Cependant deux aspects limitent la taille de ce bloc :

- la *taille maximale des buffers systèmes* (taille du `cluster` de BSD, ou taille maximale du buffer de données Streams). Si le bloc de données doit être éclaté, alors un surcoût de 11,5 µs (routine de copie/checksum) doit être ajouté pour chaque buffer supplémentaire, et

- le *MSS* puisque le calcul de checksum ne peut se faire que sur un segment TCP/UDP.

4.1.3.2 Impact de la méthode d'accès ILP

Nous nous intéressons maintenant à la méthode d'accès ILP complète, avec ses avantages et limites (Section 3.5.3.1). Les tests sont soit des transferts de masse, soit des transferts en mode écho. Dans les deux cas on utilise une connexion TCP au dessus de l'interface rebouclée mémoire. La machine utilisée est un DPX/20 sous AIX/4.1.2. En raison de l'importance essentielle du MSS, ces tests utilisent deux tailles de MTU différentes : 1500 octets (Ethernet) et 3500 octets.

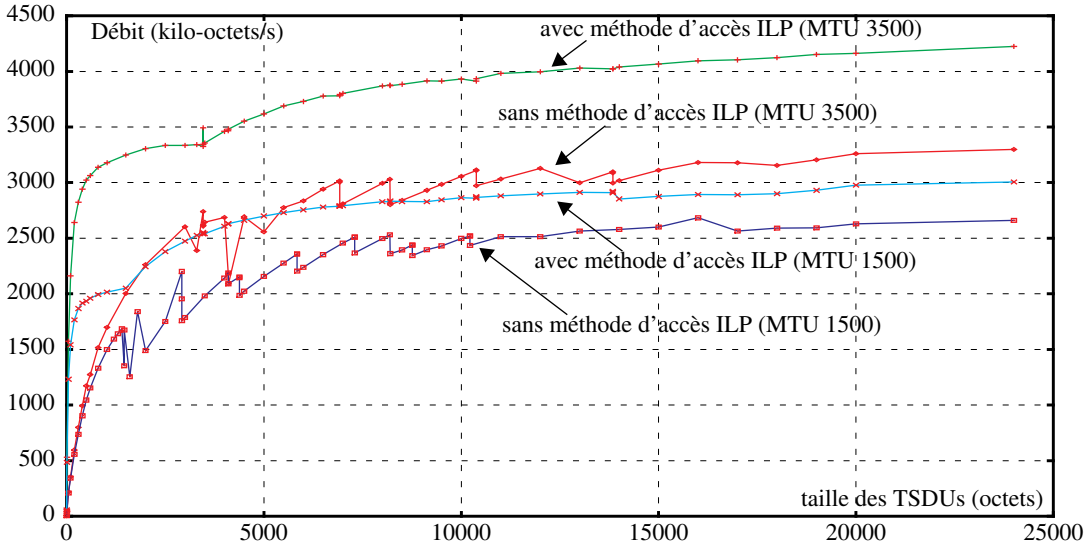

Figure 25: Comparaison des débits obtenus avec ou sans méthode d'accès ILP; DPX/20 sous AIX/4.1.2.

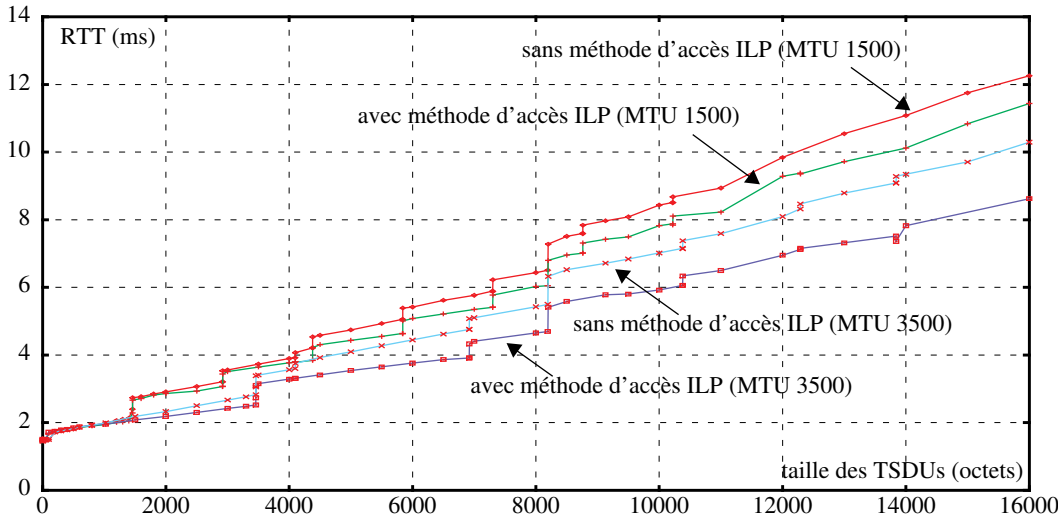

Figure 26: Comparaison des RTTs avec ou sans méthode d'accès ILP; DPX/20 sous AIX/4.1.2.

⇨ La Figure 25 montre que la méthode d'accès ILP permet des *gains extrêmement élevés*, autour de 400%, *pour de petites TSDUs*.

Ceci est le résultat de la possibilité d'accumulation de données dans la librairie XTI qui réduit considérablement le nombre d'appels systèmes (Section 3.5.3.1). Cette accumulation n'a plus d'impact significatif pour des TSDUs de taille supérieure au MSS. Ainsi :

⇨ Les *gains asymptotiques* pour de grosses TSDUs sont *plus modestes*, de 13% pour une MTU de 1500 octets à 28% pour une MTU de 3500 octets.

⇨ Les tests *en mode écho* de la Figure 26 révèlent des *gains plus limités*.

L'utilisation de l'argument `PUSH` dans les requêtes de transmission (envoi immédiat des données) en est la cause et les gains ne sont dûs qu'aux aspects intégration copie/checksum et meilleure gestion mémoire. La Figure 26 montre qu'utiliser la méthode d'accès ILP permet des réductions de 6%-8% du RTT au dessus de 5 kilo-octets avec une MTU de 1500 octets, et de 13%-16% avec une MTU de 3500 octets. Elle montre aussi que cette méthode d'accès devient intéressante à partir de 800 octets.

4.1.3.3 Discussion

Les tests précédents ont montré que si l'intégration des routines de copie de données avec le calcul de checksum est intéressante, l'application de cette technique aux systèmes de communication conduit à des *résultats inégaux* :

- La méthode d'accès ILP est principalement bénéfique aux applications effectuant des transferts de masse. Elle améliore non seulement leurs performances mais les rend aussi moins dépendantes de la taille des TSDUs. Les gains sont cependant largement liés à l'aspect accumulation de données dans la librairie.

- Les tests de RTT pour lesquels ce facteur est supprimé conduisent à des gains de performances plus faibles et seulement dans le cas de grosses TSDUs. Les applications interactives utilisant de petits messages (telnet, rlogin etc.) ne tireront que peu de bénéfices de cette technique.

Un autre résultat est que l'*effet de la MTU est essentiel*. La raison en est que la méthode d'accès ILP a des coûts fixes plus élevés. L'arrivée de nouvelles technologies réseau utilisant une MTU importante[2] contribuera à rendre cette méthode d'accès plus intéressante.

Enfin l'impact de la méthode d'accès ILP sur les traitements du flux sortant est sous-estimé. En effet, les tests étant effectués en rebouclé mémoire, ils incluent également le traitement des paquets entrants. En conséquence les nombres donnés dans la Table 8 ne peuvent pas être comparés directement aux résultats de la Section 4.1.3.2.

Une autre raison rendant la comparaison difficile est que *l'application d'ILP n'est qu'un des aspects de cette méthode d'accès*. Des résultats complémentaires fondés sur une étude de bas niveau sont donnés dans la Section 4.2.2.2. Ils soulignent en particulier l'importance de l'amélioration de la gestion mémoire.

2. La MTU par défaut de la technique IP sur ATM classique est de 9128 octets [RFC1577] et la MTU de FDDI est de 4500 octets [IBM93].

4.2 EXPÉRIENCES DE BAS NIVEAU

Nous avons jusqu'à présent étudié les performances des différentes piles de communication par le biais de mesures de débit et de temps d'aller-retour au niveau utilisateur. Nous analysons maintenant leur comportement de bas niveau sous trois points de vue différents : les temps de traitement, les instructions déroulées, et les ratios instructions/cycle. Dans chacune de ces sections, nous commençons par présenter les résultats d'expérience, ensuite nous les commentons.

4.2.1 Temps de traitement

Cette section analyse la distribution des temps de traitement parmi les différentes couches du système de communication. Les tests ont été effectués sur un DPX/20 qui utilise soit l'AIX/3.2.5, soit l'AIX/4.1.2. Ils consistent à transmettre des TSDUs de 1024 octets sur une connexion TCP au dessus d'un interface Ethernet (AIX/4.1.2) ou rebouclée mémoire (AIX/3.2.5). Afin d'être sûr que les TSDUs ne sont pas regroupées, l'option `TCP_NODELAY` [Stevens95] est systématiquement utilisée.

4.2.1.1 Distribution des temps de traitement sous AIX/3.2.5

La Figure 27 donne la distribution des temps de traitement dans le cas des piles TCP/IP BSD et démultiplexée (sans méthode d'accès ILP) sous AIX/3.2.5. Seul le coté émission est représenté. Notons que la pile BSD utilise une interface Ethernet alors que la pile démultiplexée utilise une interface rebouclée.

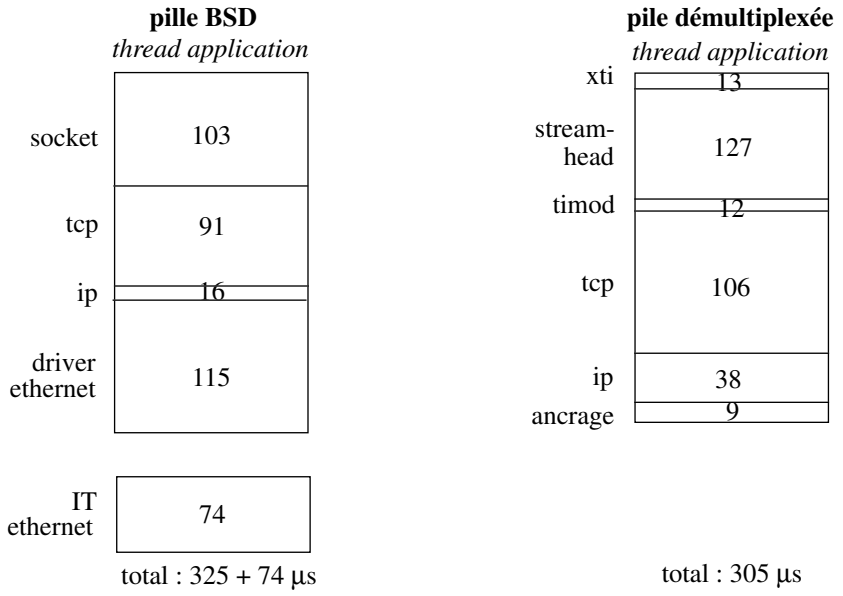

Figure 27: Temps de traitement d'un segment TCP de 1024 octets avec la pile BSD sur Ethernet (gauche) et la pile démultiplexée sur l'interface rebouclée (droite).

4.2.1.2 Distribution des temps de traitement sous AIX/4.1.2

Les Figure 28 et Figure 29 montrent la distribution des temps de traitement dans le cas d'AIX/4.1.2. Tous les événements relatifs à l'émission et à la réception des paquets au dessus d'Ethernet sont représentés.

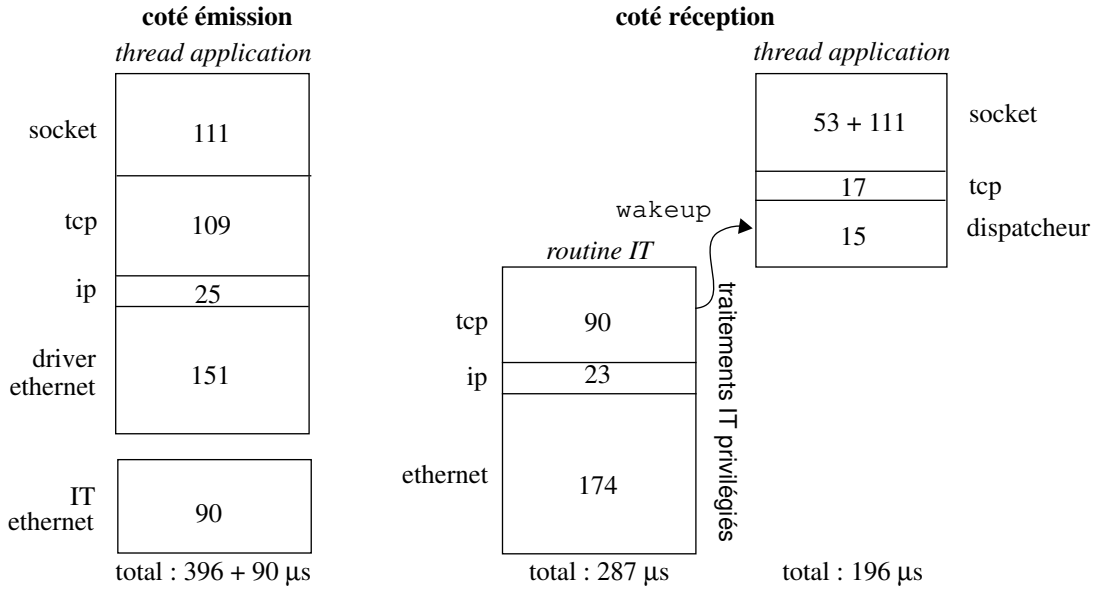

Figure 28: Temps de traitement d'un segment TCP de 1024 octets avec la pile BSD.

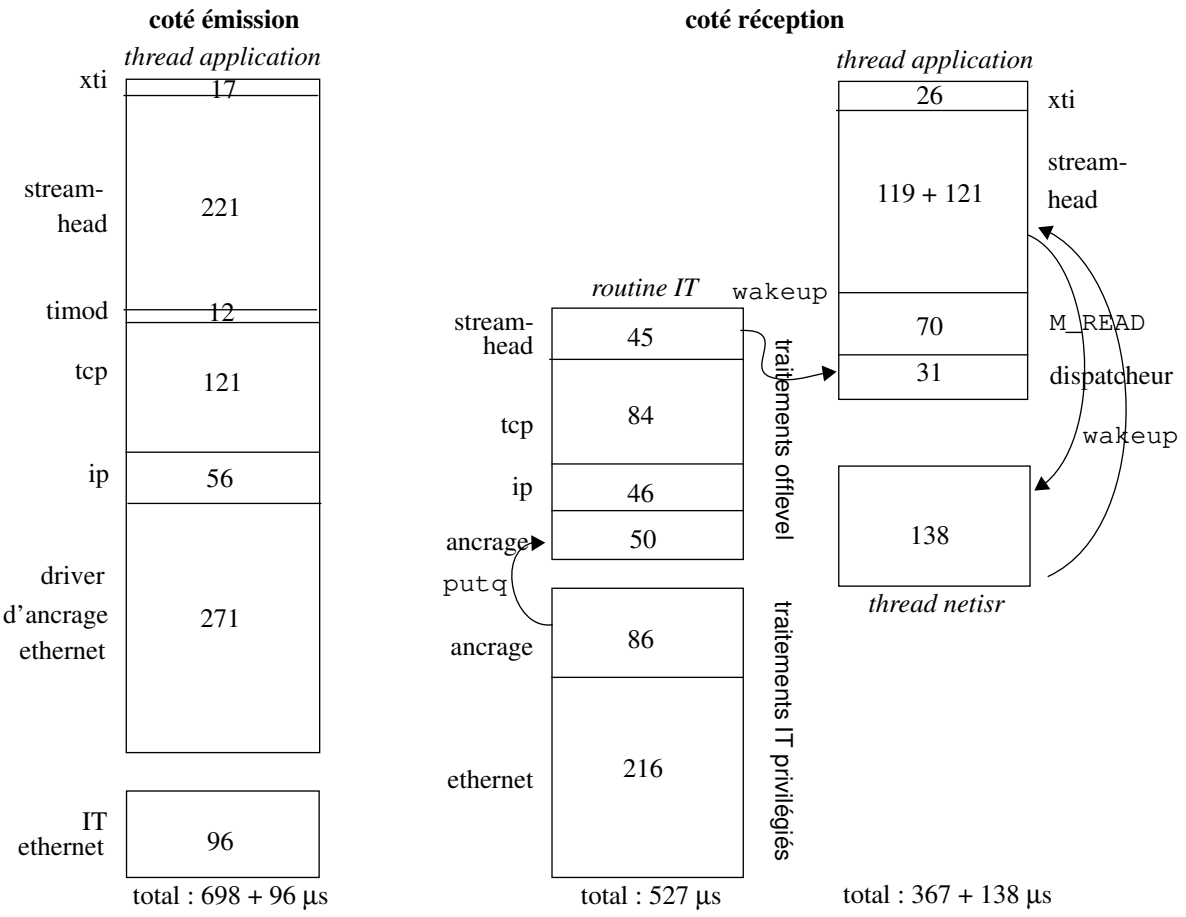

Figure 29: Temps de traitement d'un segment TCP de 1024 octets avec la pile démultiplexée.

Remarques :

1. Durant ces tests il est supposé que l'application réceptrice a déjà effectué l'appel système `recv/getmsg` et attend que des données arrivent. Le premier nombre (avant le signe +) dans la couche Socket/Stream-Head est le temps passé durant la première partie de l'appel système (avant blocage), et le second nombre est le temps passé pour achever cet appel système.

2. Chaque sonde de performances ajoute un surcoût constant de 1,5 µs (ou 50 instructions) qui n'a pas été retranché. Pour le flux sortant, il y a 59 traces dans le cas de la pile BSD et 87 traces dans le cas de la pile démultiplexée[3]. Cela représente un total de 88 µs (ou 2964 instructions) pour BSD et 130 µs (ou 4800 instructions) pour la pile démultiplexée. Ces surcoûts n'ont pas été estimés pour le flux entrant.

3. Toutes les traces de toutes les sondes sont collectées, et une fois le test achevé, le filtrage permet de ne retenir que celles qui nous intéressent.

4.2.1.3 Discussion

Les stratégies de gestion des threads dans le cas des piles BSD et démultiplexée

Les essais précédents soulignent d'importantes différences dans la stratégie de gestion des threads coté réception (le cas du flux sortant est trivial puisque la thread de l'application effectue tous les traitements) :

⇨ *La stratégie relative à la routine d'interruption de la pile BSD d'AIX/4.1.2 privilégie le temps de traitement aux caractéristiques temps réel du système d'exploitation.*

Avec une pile BSD les paquets entrants sont traditionnellement insérés dans une queue de IP et pris en charge par une thread dédiée [Stevens95]. Les traitements sous IT sont donc minimisés. Cette caractéristique a été supprimée de la pile BSD d'AIX/4.1.2 afin d'économiser un changement de contexte, et tous les traitements jusqu'à TCP se font dans un contexte d'interruption. Pour sa part la pile démultiplexée Streams utilise `putq`[4] afin d'effectuer les traitements coûteux (IP et TCP) à un faible niveau de priorité (qualifié "d'offlevel" sur la Figure 29).

⇨ *La stratégie utilisée par la pile démultiplexée quant au réveil de la thread applicative diminue le nombre de changements de contextes par rapport à BSD.*

En effet, avec BSD, pour chaque segment TCP de données reçu, la thread applicative est réveillée et copie les données dans le buffer de l'application. Ensuite soit l'appel système se finit, soit la thread se bloque en attendant des données supplémentaires. Dans ce deuxième cas la simple nécessité de copier les données aura conduit à deux changements de contextes. En revanche, la stratégie de réception des données de la pile démultiplexée fait que la thread applicative n'est réveillée que si TCP sait que l'appel système est terminé.

Enfin la présence de la thread `netisr` de la Figure 29 est due à une caractéristique coûteuse de notre environnement Streams : lorsque Streams réalise qu'un `mblk` utilise un buffer de données privé, sa libération est effectuée par une thread dédiée ce qui ajoute deux changements de contextes. Dans notre cas c'est le résultat de l'encapsulation `mbuf`/`mblk` (Section 3.1) due à la présence d'un driver Ethernet BSD. Cet overhead ne doit donc pas être pris en compte.

⇨ Cette analyse montre que *la pile démultiplexée Streams a une meilleure stratégie de gestion des threads* que la pile BSD.

L'évolution des environnements Streams et BSD de l'AIX/3.2.5 à l'AIX/4.1.2

La comparaison des Figure 27 et Figure 29 montre l'évolution de l'environnement Streams entre les versions 3.2.5 et 4.1.2 d'AIX.

⇨ *La parallélisation de l'environnement Streams a conduit à des choix de conception coûteux.*

4. `putq` insère un message dans une queue et schedule la routine de service associée.

Ainsi :

- La routine `putnext`[5] gère maintenant les niveaux de synchronisation Streams (Section 3.3.2) ce qui nécessite 12 µs (ou 230 instructions) en l'absence de conflit. Au contraire `putnext` effectue un simple appel de fonction dans le cas d'AIX/3.2.5. Même si l'ajout du niveau de synchronisation nul permet d'éviter cet overhead dans certains cas, la synchronisation de TCP et du Stream-Head reposent toujours sur un niveau de type paire de queues.

- La routine `putq` est un autre exemple. Avec AIX/4.1.2, lorsque `putq` est appelé d'un contexte thread, une interruption de faible priorité est déclenchée pour prendre en charge le traitement de la routine de service. A l'inverse, avec AIX/3.2.5 cette routine de service est traitée par le même thread[6]. La stratégie d'AIX/4.1.2 conduisant à des changements de contextes additionnels, elle rend l'utilisation des queues Streams prohibitive. La seule exception concerne l'utilisation de `putq` dans une routine d'interruption afin de traiter les tâches coûteuses à un plus faible niveau de priorité (cf. ci-dessus).

En conséquence, les temps de traitement dans le Stream-Head ont augmenté de 74% entre les deux version d'AIX et les temps de traitement du flux sortant (depuis la librairie XTI jusqu'au module IP) de 44%.

⇨ En revanche, *la parallélisation de la pile BSD a été moins coûteuse*.

En effet, les temps de traitement (depuis la couche Socket jusqu'à IP) n'ont connu que 17% d'augmentation. Ceci est lié à la parallélisation conservatrice de la pile BSD qui s'est limitée à l'ajout de verrous. La structure profonde de la pile n'a pas été modifiée et aucun service complexe n'a été ajouté.

Cette analyse montre que la parallélisation d'un système d'exploitation a un coût qui n'est justifié que dans le cas d'une machine multiprocesseur. Faire partager le même système aux machines mono et multiprocesseurs d'une gamme s'avère pénalisant. Beaucoup d'attention doit également être apportée à l'aspect simplicité. Parce que Streams n'a pas respecté ce principe, il est devenu prohibitif.

L'importance des tâches non protocolaires

Les traitements TCP/IP/ARP représentent une faible fraction du temps de traitement total sous AIX/4.1.2 : respectivement 30% et 21% pour les piles BSD et démultiplexée (Table 9).

5. `putnext` envoie un message à la routine de type `put` du prochain driver/module du stream.
6. Avant de rendre le contrôle à l'application à la fin d'un appel système, l'environnement Streams d'AIX/3.2.5 vérifie si il y a une routine de service en attente.

Table 9: Importance relative des traitements TCP/IP/ARP dans AIX/4.1.2.

	temps de traitement total (émission & réception)	temps de traitement TCP/IP/ARP	ratio TCP/IP sur temps total
pile BSD	969 µs	287 µs (23 µs pour ARP)	30%
pile démultiplexée	1688 µs (thread `netisr` supprimée)	348 µs (23 µs pour ARP et 18 µs pour le démultiplexage du paquet entrant)	21%

⇨ *Les traitements Ethernet* (au sein de la thread applicative et de la routine d'interruption) *sont la principale cause d'overhead.*

Une fois les traitements ARP déduits, ils représentent 218 µs (45%) pour BSD et 344 µs (43%) pour la pile démultiplexée coté émission.

⇨*La méthode d'accès est une autre source importante d'overhead, tout particulièrement pour Streams.*

Elle représente 111 µs (23%) pour la pile BSD et 250 µs (31%) pour la pile démultiplexée coté émission. La Figure 30 montre une comparaison de ces deux méthodes d'accès.

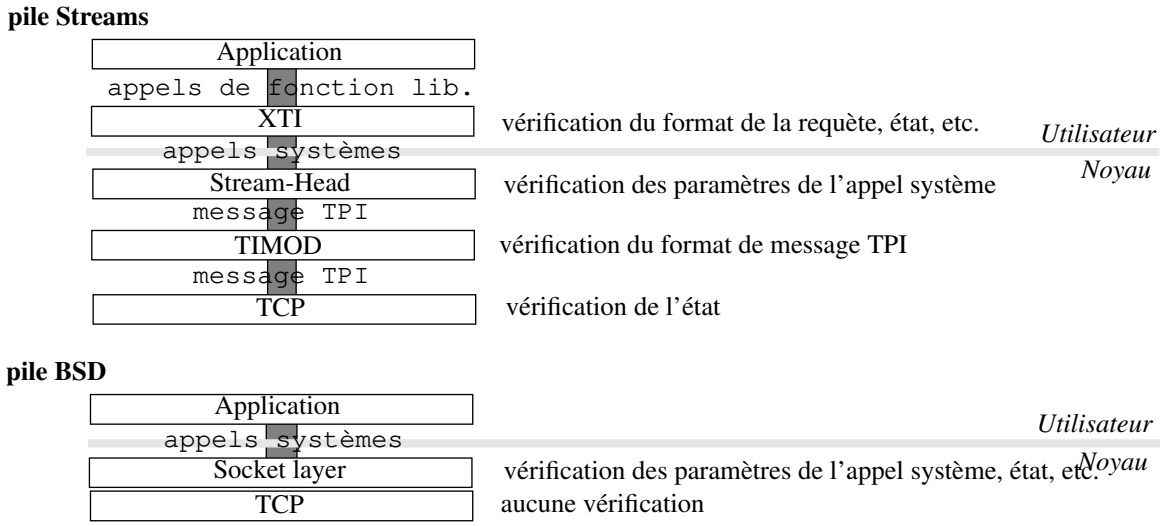

Figure 30: Comparaison des méthodes d'accès BSD and Streams.

Dans le cas de Streams, des considérations structurelles qui sont les conséquences du principe de communication par messages, associées à une parallelisation coûteuse justifient aisément le coût très élevé de sa méthode d'accès. Celle-ci repose en effet sur plusieurs composants indépendants : librairie de niveau utilisateur, Stream-Head et module TIMOD, ce qui a plusieurs conséquences :

- Ces composants sont répartis autour de la frontière entre espaces utilisateur et noyau. Par

conséquent les composants noyau ne peuvent pas faire confiance aux composants utilisateurs ce qui conduit à des vérifications multiples. Egalement, le fait que la plupart des traitements XTI soient faits au niveau utilisateur impose la présence du module TIMOD (Section 3.4.2).

- Les principes d'indépendance et de "méfiance vis-à-vis du voisin" conduisent tous les composants (la librairie, TIMOD et TCP/UDP) à maintenir leur propre machine d'états finis, d'où des redondances.

- Enfin les communications entre ces composants utilisent différents formalismes : appels de fonctions de la librairie, appels systèmes et messages Streams. Chaque requête doit être convertie entre ces différents formats et plusieurs vérifications sont faites à chaque fois.

Dans la cas de BSD l'intégration des couches Socket et protocoles de transport conduit à plus d'efficacité. En effet, les composants étant dans le noyau, ils sont considérés comme étant fiables, sont moins nombreux, et enfin emploient un formalisme unique. Cependant cette intégration est faite au détriment de la souplesse, et des APIs autres que les Sockets sont difficilement envisageables (Section 3.3.1).

4.2.2 Instructions déroulées

Les expériences suivantes ont été réalisées dans les mêmes conditions qu'à la Section 4.2.1 à la différence près que seul l'AIX/4.1.2 a été utilisé.

4.2.2.1 Distribution des instructions déroulées sous AIX/4.1.2

La Figure 31 montre la distribution des instructions déroulées pour les piles BSD et démultiplexée avec ou sans la méthode d'accès ILP, au dessus d'une interface Ethernet. Contrairement aux sondes de performances, l'outil de comptage d'instructions n'introduit ici aucun overhead.

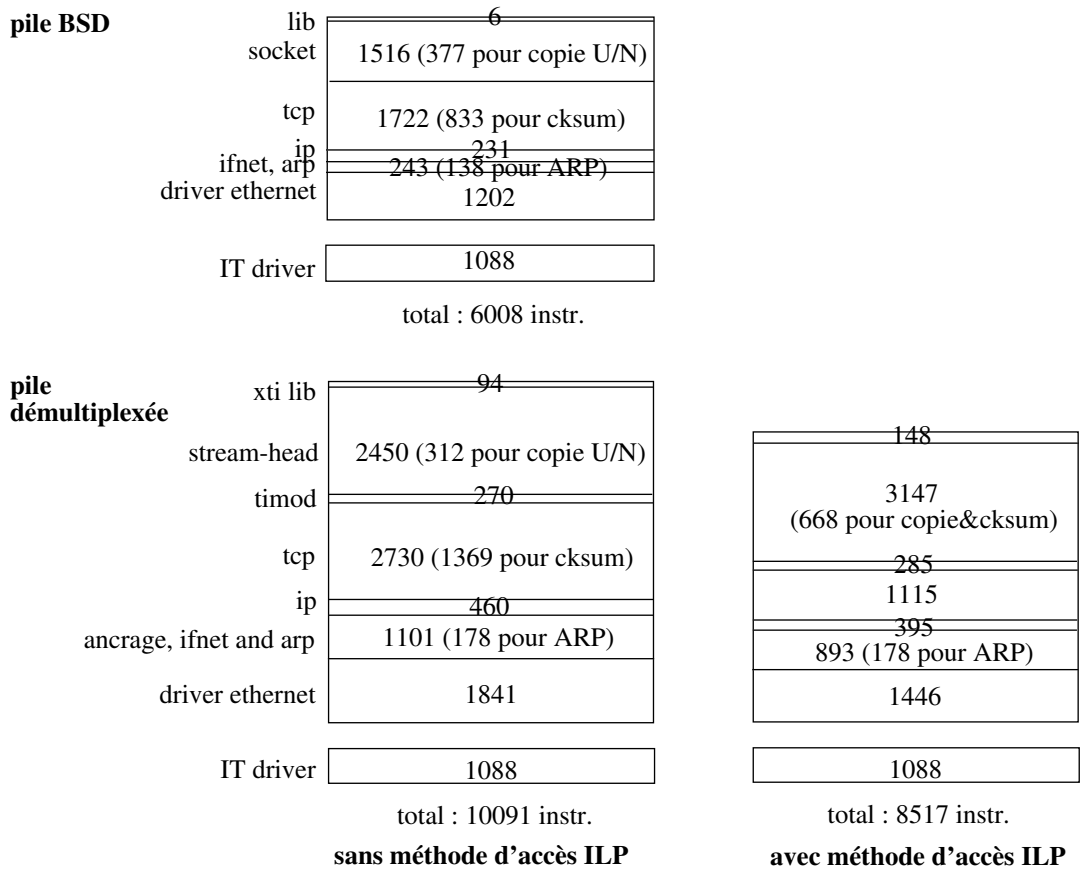

Figure 31: Instructions déroulées lors de l'émission d'un segment TCP de 1024 octets avec les différentes piles.

La distribution des instructions, au lieu de se faire selon les couches du système de communication, peut se faire selon différents aspects clés (Table 10). Ce type de représentation offre l'avantage de permettre une comparaison plus simple des différentes solutions. Le résultat de cette distribution est présenté dans la Table 11.

Table 10: Aspects clés retenus.

Niveau utilisateur	Tout les traitements effectués au niveau utilisateur, essentiellement la librairie XTI dans le cas de Streams.
Méthode d'accès	Avec la pile BSD il s'agit de la couche Socket. Avec Streams il s'agit du Stream-Head, de TIMOD, et des routines Streams telles que putnext. Les copies de données Utilisateur/Noyau ne sont pas incluses.
Protocoles	TCP, IP et ARP. Dans le cas de Streams, la gestion des messages TPI/NPI/DLPI est incluse. Les calculs de checksum ne sont pas inclus.
Mémoire	Allocation/libération des buffers (`mbuf` pour BSD et `mblk` pour Streams).
Manipulations de données	Copie Utilisateur/Noyau, calcul de checksum, copie de données dans le driver Ethernet (Section 2.1.3.1).

Table 10: Aspects clés retenus.

Ethernet	Driver Ethernet. Dans le cas de Streams cela inclue le driver d'ancrage et l'adaptation Streams/BSD.
Gestion des interruptions	Gestion bas niveau des interruptions par le système d'exploitation.
Synchronisation	Masquage/démasquage des interruptions et gestion des verrous.
OS de bas niveau	Autres activités du système d'exploitation.

Table 11: Instructions déroulées lors de l'émission d'un segment TCP de 1024 octets avec les différentes piles.

	pile TCP/IP BSD		pile démultiplexée		pile démultiplexée avec méthode d'accès ILP	
	nombre instructions	% relatif	nombre instructions	% relatif	nombre instructions	% relatif
Niveau utilisateur	6	0.1	94	0.9	148	1.7
Méthode d'accès	525	8.8	1234	12.2	1373	16.1
Protocoles	740	12.3	871	8.6	943	11.1
Mémoire	618	10.3	1776	17.6	1407	16.5
Manipulations données	1607	26.7	2037	20.2	950	11.2
Ethernet	1102	18.3	1266	12.5	1241	14.6
Gestion interruptions	185	3.1	185	1.8	185	2.2
Synchronisation	464	7.7	1601	15.9	1277	15.0
OS de bas niveau	761	12.7	1027	10.2	993	11.7
Total	6008	100.0	10091	100.0	8517	100.0

4.2.2.2 Discussion

Suite de la comparaison BSD/Streams

Cette discussion est la suite de la Section 4.2.1.3 et nous nous intéressons ici aux informations supplémentaires apportées par les expériences de comptage d'instructions.

⇨ Une pile Streams utilise deux fois plus de buffers qu'une pile BSD.

Ces expériences montrent que, une fois déduits les deux buffers liés à l'encapsulation `mblk`-vers-`mbuf` (ou 543 instructions), six buffers sont requis pour la pile démultiplexée au lieu de trois pour la pile BSD.

⇨ Le surcoût imposé par la synchronisation Streams est trois fois celui de BSD.

Lorsque l'impact de l'encapsulation `mblk`-vers-`mbuf` a été déduit (162 instructions), le coût de la synchronisation Streams s'élève à : 1601 - 162 = 1439 instructions comparé aux 464 instructions de BSD. Ceci est un overhead majeur sur un monoprocesseur pour lequel peu de contentions sont attendues. C'est une autre preuve du coût lié à l'emploi d'un système d'exploitation multiprocesseur sur un monoprocesseur.

⇨ Les coûts des traitements purement protocolaires des piles BSD et démultiplexée sont proches l'un de l'autre.

Ceci provient de l'origine commune des deux implémentations : la souche BSD. Les 131 instructions supplémentaires pour la pile démultiplexée sont dues à l'initialisation et au décodage des messages TPI/NPI/DLPI.

L'impact de la méthode d'accès ILP

⇨ La Table 11 montre que la *principale contribution de la méthode d'accès ILP provient de l'intégration des manipulations de données* (1087 instructions).

⇨ Elle montre également qu'*avoir une meilleure gestion mémoire a un impact très important*.

La possibilité de réserver de la place pour les en-têtes des protocoles et d'avoir exactement un segment par buffer (Section 3.5.3.1) sont bénéfiques à la fois pour la gestion mémoire (moins d'allocations/libérations) et pour la synchronisation (la gestion mémoire repose de façon importante sur la synchronisation, Section 4.1.2.3). Ces deux aspects permettent une économie de 693 instructions, soit plus de la moitié de celle de l'intégration ILP. Ce résultat souligne une fois de plus l'importance des considérations de gestion mémoire.

⇨ En comparaison l'augmentation des traitements requis au sein de la méthode d'accès est limitée et ne représente que 265 instructions.

4.2.2.3 Comparaison avec d'autres expériences de comptage d'instructions

Plusieurs expériences de comptage d'instructions peuvent être trouvées dans la littérature. Elles fournissent cependant des chiffres bien plus faibles que les nôtres :

[Clark89] analyse une pile TCP/IP tournant sur une machine CISC d'ancienne génération. Il montre que les traitements TCP et IP représentent approximativement 300 instructions. Cependant cette valeur :

- ne tient pas compte des manipulations de données (copie utilisateur/noyau, checksum),
- suppose l'utilisation de services de gestion mémoire privés, et non les traditionnels `mbufs`,

- ne tient pas compte des traitements dus au driver réseau sous-jacent, et
- plus généralement il est supposé que l'implémentation a été retirée d'UNIX de peur que les résultats ne soient perturbés par des traitements non attendus au sein du système d'exploitation.

Ce n'est pas l'approche que nous avons choisie. [Clark89] décrit ensuite des expériences en grandeur nature sur un Sun-3/60 (MC68020 cadencé à 20 MHz) au dessus d'Ethernet. Les auteurs concluent que les opérations de manipulation de données dominent largement les temps de traitements (771 µs par opposition à 100 µs) et que l'overhead lié au système est large comparé à celui des traitements des protocoles (240 µs vs. 100 µs). Ceci va dans la même direction que nos résultats.

[Jacobson93] affirme pour sa part que si l'on ne considère que les aspects de contrôle de TCP, en particulier l'analyse de l'en-tête, alors un segment entrant peut être traité en 30 instructions. Plusieurs tâches essentielles sont de ce fait omises :

- le démultiplexage des segments TCP (10 instructions si les informations se trouvent dans le cache de démultiplexage),
- le réveil de la thread de l'application (dans cette implémentation les traitements TCP sont faits par la thread de l'application) (dans notre cas un changement de contexte requiert 285 instructions),
- la copie/checksum (nous avons trouvé que 668 instructions étaient requises dans le cas d'un segment de 1024 octets), et enfin
- la gestion mémoire. Notons que la présence de nouveaux buffers, les `pbufs` (Section 2.2.3.1), et les hypothèses associées (exactement un paquet par `pbuf`) améliorent grandement cet aspect.

Là encore l'affirmation "TCP en 30 instructions" ne peut pas être comparée directement avec nos résultats où les traitements des protocoles sont mélangés avec les autres tâches.

La conclusion, conforme avec nos résultats, qui peut être tirée de ces expériences est que les traitements de contrôle des protocoles sont habituellement peu coûteux. Mais dès que ces protocoles sont intégrés à un environnement de travail, alors des contraintes techniques multiples submergent les traitements purement protocolaires.

4.2.3 Ratios instructions/cycle

Nous avons vu à la Section 3.2.1 qu'un processeur POWER est composé de trois unités de traitement opérant simultanément. Cela autorise le processeur à exécuter un maximum de quatre instructions par cycle. Le but de cette section est d'étudier les ratios instructions/cycle effectifs

durant des transferts réseaux. Ces ratios sont donnés par l'équation :

$$\frac{\text{durée du cycle}}{\text{durée d une instruction}} = \frac{1/\text{fréquence horloge}}{\text{durée de test}/\text{nombre d instructions exécutées}}$$

La fréquence de l'horloge est de 41,7 MHz. Les durées des tests sont celles de la Section 4.2.1. Comme ces chiffres incluent l'overhead des sondes de performances, des corrections sont apportées[7].

4.2.3.1 Ratios instructions/cycle moyens

Nous évaluons tout d'abord les ratios moyens lors de l'envoi d'une TSDU de 1024 octets.

- *pile BSD :* le temps de traitement est de 486 µs et les sondes sont appelées 59 fois, soit un surcoût de 2964 instructions. Donc :

$$\frac{1/41.7\times10^{6}}{486\times10^{-6}/(6008+2964)} = 0.443 \text{ instructions/cycle}$$

- *pile démultiplexée :* le temps de traitement total est de 10091 µs et les sondes sont appelées 87 fois soit un surcoût de 4800 instructions. Donc :

$$\frac{1/41.7\times10^{6}}{794\times10^{-6}/(10091+4800)} = 0.450 \text{ instructions/cycle}$$

- *pile démultiplexée avec méthode d'accès ILP :* le temps de traitement est de 755 µs, les sondes étant la encore appelées 87 fois. Donc :

$$\frac{1/41.7\times10^{6}}{755\times10^{-6}/(8517+4800)} = 0.423 \text{ instructions/cycle}$$

Plusieurs conclusions peuvent être tirées :

⇨ *Le ratio instructions/cycle moyen est nettement plus faible que le ratio maximum théorique.*

Ceci est en accord avec [Mosberger95] qui rapporte qu'un processeur DEC Alpha n'est jamais actif plus de 37% du temps.

⇨ *Les deux piles non ILP ont des ratios similaires. La pile ILP a un ratio légèrement plus faible.*

Ce phénomène est expliqué plus loin.

7. L'évaluation des corrections se fait en utilisant l'outil de comptage d'instructions alors que les sondes de performances sont activées.

4.2.3.2 Ratios instructions/cycle durant les manipulations de données

Copies de données entre espaces utilisateur et noyau

Lorsque `copyin` est utilisé pour réaliser la copie Utilisateur/Noyau (cas de Streams), nous avons :

$$t_{\text{copie Utilisateur/Noyau}}(1024) = 23\ \mu s \text{ avec la pile démultiplexée}$$

Les sondes de performance ajoutant 50 instructions, on a :

$$\frac{1/41.7\times10^{6}}{23\times10^{-6}/(312+50)} = 0.374 \text{ instructions/cycle}$$

⇨ Ce *ratio est très faible*, inférieur même au ratio moyen. Ceci s'explique :

- par le fait que *les accès mémoire sont un facteur limitant*, et
- par la structure de la routine assembleur de copie. Celle ci est centrée autour d'une boucle de quatre instructions qui effectuent le transfert par blocs de 16 octets simultanément. Ceci est économique dans le sens où peu d'instructions suffisent à transférer un bloc, cependant les instructions de chargement et d'écriture de plusieurs mots nécessitent chacune plusieurs cycles [Moore93].

Calcul du checksum

Le calcul du checksum sur le pseudo-en-tête TCP/IP et le bloc de données de 1024 octets nécessite :

$$t_{\text{checksum}}(1064) = 17\ \mu s \text{ avec la pile BSD}$$

$$t_{\text{checksum}}(1064) = 37\ \mu s \text{ avec la pile démultiplexée}$$

Les différences sont dues au fait que la pile démultiplexée utilise une routine C alors que la pile BSD utilise une routine assembleur optimisée. On a donc :

$$\frac{1/41.7\times10^{6}}{17\times10^{-6}/(833+50)} = 1.246 \text{ instructions/cycle pour BSD}$$

$$\frac{1/41.7\times10^{6}}{37\times10^{-6}/(1369+50)} = 0.920 \text{ instructions/cycle pour la pile démultiplexée}$$

⇨ Ces ratios sont nettement plus élevés que la moyenne. Cela montre que le processeur n'est plus limité par les accès mémoire et que ses unités de traitement sont mieux utilisées, notamment dans le cas de la routine assembleur.

Intégration copie de données et calcul de checksum

La routine copie/checksum utilisée par le Stream-Head nécessite :

$$t_{copie_et_cksum}(1024) = 38\ \mu s \text{ avec la pile démultiplexée}$$

d'où :

$$\frac{1/41.7\times10^{6}}{38\times10^{-6}/(668+50)} = 0.453 \text{ instructions/cycle}$$

⇨ Ce ratio est davantage *typique de celui de la routine de copie* que de celui de la routine de calcul de checksum. Ceci justifie le plus faible ratio instructions/cycle moyen observé pour la pile démultiplexée utilisant la méthode d'accès ILP.

4.2.3.3 Discussion

Ces expériences d'évaluation des ratios instructions/cycle montrent que :

- *Le ratio instructions/cycle moyen est très faible*. Les opportunités de traitements concurrents des différentes unités du processeur sont rares. Le processeur est soit limité par les accès instructions (cas général) soit par les accès données (cas des manipulations de données).

- *Les ratios instructions/cycle varient largement durant les traitements des protocoles*. Les meilleurs ratios sont obtenus avec les boucles denses (peu d'accès aux instructions) qui effectuent des traitements intensifs sur un faible jeu de données (peu d'accès aux données). Une bonne alternance entre **(1)** les accès mémoire et calculs entiers, **(2)** les calculs sur les registres conditionnels, et **(3)** les instructions de branchement est nécessaire pour favoriser le traitement parallèle des différentes unités du processeur. La routine de checksum qui se contente d'un accès en lecture aux données, répond (partiellement) à ces critères.

- *Avoir des ratios instructions/cycles élevés n'est pas nécessairement le but*. Ce qui compte est le temps (exprimé en secondes ou en cycles) requis pour effectuer une tâche donnée. Le ratio instructions/cycle donne seulement une idée du rendement du processeur. Le même niveau de performances peut être obtenu soit avec un rendement du processeur élevé et un grand nombre d'instructions, soit au contraire avec un faible rendement du processeur associé à un nombre d'instructions faible, dès lors que le rapport entre les deux reste constant.

Ces résultats relativisent les expériences de comptage d'instructions qui ne prennent pas en compte l'efficacité d'utilisation du processeur. Les expériences d'évaluation fine des temps de traitement ont aussi leur limites dans la mesure où elles reposent sur l'ajout explicite de sondes dans le code, introduisent des surcoûts non négligeables, et cachent certains phénomènes de bas niveau.

La complexité croissante des processeurs et des systèmes d'exploitation souligne le besoin urgent de moniteurs de performances intégrés au processeur [Mosberger95] afin d'évaluer de façon fiable

le nombre de cycles écoulés, les comportements des caches instructions et données, du TLB, etc.

CHAPITRE 5

Discussion

Dans cette section nous tirons les conclusions de l'expérience pratique et théorique obtenue et nous proposons des axes pour améliorer encore le système de communication. Ces solutions sont complémentaires et non opposées à celles de la Section 2.2. La discussion est classée selon les trois aspects identifiés à la Section 2.1.3 : architecture de la machine hôte, système d'exploitation et protocoles de communication.

5.1 ASPECT ARCHITECTURE DE LA MACHINE HÔTE

La machine hôte est essentielle dans la mesure où elle constitue la fondation au dessus de laquelle est construit le système de communication. Les diverses expériences que nous avons effectuées nous ont conduit à plusieurs conclusions.

5.1.1 Limites du parallélisme

L'arrivée des machines multiprocesseurs a mis l'accent sur la parallélisation du système de communication. Les expériences ont montré que trois conditions doivent être réunies :

Un support approprié au niveau du processeur

Un emploi judicieux des instructions `load-and-reserve` et `store-conditional` (Section 2.2.4) que l'on trouve sur les processeurs RISC modernes permet une réduction significative de l'emploi des verrous. La synchronisation devient ainsi mois coûteuse.

Un support approprié du système d'exploitation

Les expériences ont montré que les contentions d'accès aux verrous ont essentiellement pour origine des tâches systèmes et non des tâches protocolaires. Ceci est particulièrement vrai pour la gestion mémoire qui joue un rôle capital avec les implémentations Streams. Un grand soin doit donc être apporté à la parallélisation des fonctionnalités systèmes de base.

Les services de synchronisation proposés par l'environnement Streams utilisé (Section 3.3.2) se sont révélés inadéquats : si leur utilisation est commode (tout est pris en charge par l'environnement), en revanche leur implémentation au moyen d'éléments de synchronisation est trop coûteuse.

Une architecture du système de communication appropriée

L'architecture du système de communication doit naturellement minimiser les conflits de synchronisation et les changements de contextes. Deux types de parallélisme ont été distingués (Section 2.2.4) :

- le parallélisme de type tâche : il inclue les parallélismes fonctionnel et pipeline, encore appelé vertical dans le jargon Streams. Ce type de parallélisme, à la source de nombreux problèmes (nombre élevé de changements de contextes, performances limitées par le composant le plus lent, etc.), doit être évité.

- le parallélisme de type message : le parallélisme par connexion de la pile démultiplexée et par message de la pile BSD font tous deux partie de cette catégorie. Nous pensons que le modèle connexion, évitant les contentions par une bonne gestion des threads, a un léger avantage sur le modèle message. Nos tests ont montré que la pile démultiplexée avait en effet une meilleure scalabilité que la pile BSD. Cependant il est difficile de départager l'impact de l'architecture (connexion/message) de celui du système d'exploitation (Streams/BSD).

En dépit de ces techniques, les expériences ont montré une *faible scalabilité* avec TCP. Le système de communication d'un octo-processeur est au mieux 3,5 fois plus rapide que celui d'un monoprocesseur.

De plus *la généralisation des machines multiprocesseurs symétriques s'est faite au détriment des performances des machines monoprocesseurs*. En effet, des considérations de coût imposent que les machines d'un constructeur donné partagent le même système d'exploitation. Parce qu'un monoprocesseur est vu comme un cas particulier de multiprocesseur, le système d'exploitation est

désormais conçu pour supporter le parallélisme et seuls quelques ajustements minimaux sont faits pour prendre en compte les hypothèses simplificatrices des monoprocesseurs (Section 3.2.3). Malgré ces ajustements, les performances sur un monoprocesseur sont plus faibles que celles qui auraient été obtenues avec un système d'exploitation restreint à ces machines[1].

Etant donné les piètres résultats obtenus, une alternative pourrait être de se ramener à une implémentation non parallélisée en dédiant un processeur au système de communication. Cependant cette solution conduit à des problèmes d'équilibre de charge. Si la machine est utilisée pour des traitements intensifs demandant peu de communications, alors le processeur de communication peut être sous utilisé. A l'opposé, dans le cas de tâches effectuant de nombreuses communications, ce processeur peut devenir le facteur limitant. Une variante serait de partager le processeur de communication lorsque ce dernier est sous-utilisé. Mais là encore cela peut conduire à des problèmes si cette situation n'était que transitoire et qu'une période de forte activité suit. En conclusion, déplacer le système de communication sur un processeur dédié n'est pas la solution.

De cette étude il ressort que :

- les machines multiprocesseurs sont certainement valables dans le cas d'applications nécessitant des traitements intensifs, telles que les applications de calcul scientifique. C'est alors une façon aisée d'augmenter la puissance de la machine. Mais leur utilisation dans l'unique but d'accélérer les communications est contestable.

- même si sa scalabilité est limitée, un système de communication parallélisé reste, à défaut de mieux, la "moins mauvaise" solution dans le cas d'une machine multiprocesseur.

5.1.2 Limites de ILP

La mémoire est désormais bien identifiée comme étant le principal facteur limitant des stations de travail actuelles. ILP est l'une des réponses à cette situation (Section 2.2.3). Plusieurs contraintes tendent cependant à en limiter les bénéfices :

- Les fonctions de manipulation ne doivent pas imposer que les données soient traitées séquentiellement. Ainsi l'algorithme de calcul du CRC n'est pas adapté.

- *La complexité des manipulations de données doit être faible*. En effet, si ILP améliore la localité des accès aux données manipulées (les manipulations sont faites lorsque les données sont dans le cache ou les registres), c'est au détriment de la localité des accès aux instructions et données des fonctions de manipulation (chaque fonction opère successivement sur la petite unité de données courante, puis le processus recommence pour l'unité suivante). Si les manipulations sont trop complexes, alors leur code et données[2] peuvent être trop larges pour

1. Ce phénomène est parfois caché par le fait que des optimisations, n'existant pas sur l'ancienne gamme de systèmes d'exploitation monoprocesseurs, ont été réalisées pour compenser les problèmes de performances.
2. Certaines fonctions emploient de grosses tables pré-calculées.

tenir dans les caches. [Braun95a] montre que ILP peut ainsi conduire à des taux d'échecs d'accès aux caches supérieurs à ceux d'une pile de communication classique.

Ces problèmes justifient le fait que les gains obtenus dans [Braun95a] par l'intégration de quatre fonctions de manipulation, en l'occurrence le marshalling, le chiffrage, le calcul de checksum et la copie de données ne soient que de 10% à 20%. C'est peu vu le nombre élevé de manipulations.

Dans ce travail nous avons montré comment une intégration moins ambitieuse, à savoir la copie de données entre espaces utilisateur et noyau et le calcul de checksum, peut être réalisée et quels bénéfices on peut en retirer. Parce que ces deux manipulations sont élémentaires, les limitations précédentes ne s'appliquent pas. Les expériences ont montré que (Section 4.1.3 et Section 4.2.2.2) :

- les résultats sont inégaux et dépendent largement de la nature de l'application. Les applications interactives manipulant des petites TSDUs ne sont pas adaptées.

- si l'utilisation de grosses TSDUs est bénéfique, l'intégration ILP est, du fait que le checksum soit calculé sur la base du segment TCP, limitée par le PMTU. Si les gains sont réduits dans le cas d'Ethernet, ils vont devenir plus attractifs avec les nouvelles technologies réseau (FDDI, ATM, etc.) utilisant de grosses MTUs.

- les gains liés à ILP ne sont en fait qu'un des aspects. La méthode d'accès ILP a également un impact très bénéfique sur la gestion des buffers, et par conséquent sur la synchronisation. La réduction du nombre d'appels systèmes peut aussi permettre sous certaines conditions (petites TSDUs et pas de préoccupations de latence) des gains de performances de 400%.

Ainsi la méthode d'accès ILP peut s'avérer être bénéfique, plus ou moins suivant ses conditions d'utilisation, mais l'intégration des manipulations de données n'est que l'un des facteurs.

5.2 ASPECT SYSTÈME D'EXPLOITATION

Par système d'exploitation nous entendons non seulement l'OS et l'environnement d'exécution du système de communication, mais également des considérations d'architecture. Ces points sont essentiels et les solutions actuelles conduisent parfois à de sérieuses limites.

5.2.1 Limites des environnements actuels

5.2.1.1 Limites de Streams

Les expériences ont montré que l'environnement Streams actuel ne peut pas être considéré comme étant adapté aux systèmes de communication hautes performances. Il y a plusieurs raisons à cela :

Limitations intrinsèques

Plusieurs facteurs intrinsèques font de Streams un environnement coûteux :

- sa communication par messages : elle nécessite l'allocation de buffers supplémentaires, l'initialisation puis le décodage de messages conformes aux standards TPI, NPI et DLPI, et enfin leur libération.

- sa méthode d'accès par couches (Figure 30 page 92) : elle est composée de multiples composants indépendants répartis autour de la frontière utilisateur/noyau. Plusieurs abstractions (appels de fonction de la librairie, appels système, et messages TPI) et plusieurs niveaux de vérification (les composants étant indépendants, ils ne peuvent pas se faire confiance mutuellement) sont donc requis. Cette structure en couches empêche ou rend difficiles certaines optimisations (méthode d'accès ILP coté entrant par exemple).

Limitations fonctionnelles

Le fait qu'il n'y ait pas de mécanisme standard pour contrôler et optimiser le remplissage des buffers de réception des applications est une très grosse limitation. Ceci est probablement un héritage de son utilisation pour les drivers TTY pour lesquels peu d'octets sont transférés simultanément.

Un autre problème, moins sérieux, provient de l'absence de possibilité de "scattering/gathering". Les appels systèmes `readv`/`writev` disponibles avec la méthode d'accès Socket ne sont pas supportés par Streams. L'application a de ce fait moins de liberté pour choisir la stratégie de gestion mémoire qui convient le mieux à ses besoins, ce qui peut conduire dans certains cas à des copies de données supplémentaires.

Problèmes propres à l'environnement Streams utilisé

Certains problèmes sont liés à l'implémentation de l'environnement Streams utilisé. Ceci est particulièrement vrai de la version parallélisée de Streams. En effet, les aspects scheduling et synchronisation sont complexes et plusieurs stratégies sont possibles. Ainsi nous avons vu que les mécanismes de synchronisation utilisés ont un coût très élevé. La Section 4.2.1.3 donne d'autres exemples. Les problèmes sont moins importants sur les versions strictement monoprocesseurs de Streams, sous AIX/3.2.5 par exemple.

Mauvaise utilisation

Le fait que Streams promeuve le concept d'indépendance des composants avec des interfaces bien définies a conduit de nombreux implémenteurs à concevoir des implémentations en couches et à faire un grand usage des fonctions de service. Sur les multiprocesseurs un nom spécifique, "parallélisme vertical", a même été donné au type de parallélisme qui en résulte. Malheureusement les notions de couches et de parallélisme vertical sont totalement inefficaces. Cette situation caricaturale provient d'une mauvaise connaissance des techniques

d'implémentation hautes performances. Nos travaux ont montré que des solutions sont heureusement possibles. Cela passe par plus de synchronisme et plus de simplicité. Une discussion générale est faite à la Section 5.2.2 et des compléments à l'Annexe A.

5.2.1.2 Limites de BSD

La nature intégrée de l'environnement BSD limite naturellement les problèmes de performances. Mais BSD n'est pas parfait et souffre en particulier :

- de mauvais services mémoire : la gestion mémoire `mbuf` qui définit deux types de buffers de taille fixe n'est pas adaptée et conduit soit à des traitements coûteux, soit à une mauvaise utilisation de la mémoire.

- de nombreux changements de contextes coté entrant : à chaque réception de données, TCP réveille la thread applicative même si le contrôle ne peut être retourné de suite à l'application (Section 4.2.1.3).

- d'un manque de flexibilité : étant difficile de composer une pile de protocoles qui convienne exactement aux besoins de l'application, les protocoles "optionnels" tels que RPC sont déplacés hors du noyau. Leur efficacité est de ce fait moindre (Section 5.2.2).

- d'une intégration de la méthode d'accès et de l'API : les deux notions sont mélangées, rendant le support d'APIs non Socket (XTI par exemple) difficile et peu efficace (Section 3.4.1). C'est un autre exemple du manque de flexibilité de BSD.

5.2.2 Caractéristiques d'un environnement de système de communication hautes performances

Nous pouvons maintenant définir les principes généraux qu'un environnement hautes performances se doit de respecter :

- favoriser la simplicité,

- favoriser le synchronisme,

- conserver une certaine flexibilité,

- éviter les implémentations en couches inutiles, et

- intégrer les drivers réseaux.

Favoriser la simplicité

Les solutions simples (et non simplistes) sont souvent les meilleures. Le principe K.I.S.S., "Keep

It Simple and Stupid", affirme que c'est le meilleur moyen pour obtenir un système opérationnel. Parce que l'environnement Streams parallélisé n'a pas respecté ce principe, son usage est devenu problématique.

Favoriser le synchronisme

Plus de synchronisme signifie que les requêtes de transmission de données et les paquets entrants doivent être traités par une unique thread. Ce paradigme réduit naturellement le nombre de changements de contextes, favorise le comportement des caches [Mogul91], et est bien adapté au parallélisme (Section 5.1.1).

X-kernel va plus loin dans cette direction. [Peterson90] décrit le mécanisme de "upcall" qui permet à une thread noyau d'invoquer une procédure de niveau utilisateur. Ainsi un paquet entrant est traité intégralement par la même thread, depuis les couches de protocoles jusqu'à l'application. Ceci économise un changement de contexte par rapport aux environnements traditionnels où la thread applicative est réveillée par la thread noyau.

Conserver une certaine flexibilité

Nous pouvons distinguer deux niveaux de flexibilité : au niveau de la configuration et au niveau de la méthode d'accès.

1- *Flexibilité au niveau de la configuration*

Avoir la possibilité de composer dynamiquement la pile de communication exacte qui convient le mieux à un besoin particulier est un avantage. [Hutchinson91] montre que, grâce à la flexibilité apportée par x-kernel, les protocoles "optionnels" tels que RPC peuvent être implémentés dans le noyau. La gestion mémoire est de ce fait simplifiée et l'efficacité améliorée. A l'opposé, les systèmes monolithiques déplacent habituellement ces protocoles hors du noyau.

2- *Flexibilité au niveau de la méthode d'accès*

Deux types de méthodes d'accès ont été rencontrés dans ces travaux (Figure 30 page 92) :

- *un jeu d'appels systèmes dédiés* qui forment l'API Socket. Ici tout le code se trouve dans le noyau ce qui le rend efficace et peu sensible aux événements de niveau utilisateur, et

- *un jeu d'appels systèmes génériques*, ceux de Streams, qui sont utilisés pour créer différentes APIs : XTI, Socket, TTY, etc. L'API repose essentiellement sur une librairie de niveau utilisateur dont le rôle est de changer d'abstraction : primitives XTI/messages TPI par exemple.

Ces deux méthodes d'accès ont des limites :

- Une méthode d'accès qui fusionne les appels systèmes avec l'API empêche l'utilisation efficace de nouvelles APIs (Section 3.4.1).

- Elles sont *limitées*. Les fonctionnalités qui n'ont pas été envisagées au départ ne peuvent pas être supportées. Ainsi le service de "scattering/gathering" disponible avec les Sockets est absent des appels systèmes Streams et doit être émulé, perdant alors tout intérêt du point de vue performances.

- Parce qu'elles ont été conçues avec un schéma d'API standard, elles sont très *traditionnelles*. Les fonctionnalités nouvelles requièrent le développement d'appels systèmes supplémentaires.

  En particulier l'application *manque de contrôle sur le flux de données et la gestion des buffers*. Parce que ce n'était pas disponible, la méthode d'accès ILP a nécessité la conception d'un nouvel appel système afin de permettre à l'application de contrôler la façon dont ses données doivent être affectées aux buffers systèmes (Section 3.5.3.1). Un autre exemple est une application de vidéo-sur-demande qui a besoin de router les données depuis le disque vers le réseau. Avec une API Socket ou XTI, les données doivent nécessairement transiter par l'espace utilisateur. Afin d'éviter les problèmes de performance qui en découlent, une solution courante consiste à implémenter un agent dans le noyau pour contrôler ce transfert sans avoir à passer par l'espace utilisateur. Cette solution est malheureusement complexe et non portable.

Une solution plus flexible est requise. Nous proposons pour cela un modèle de méthode d'accès à deux niveaux qui peut être vue comme une généralisation de la notion de "pool" (Section 2.2.1) :

- un jeu de briques élémentaires génériques qui regroupent des services d'allocation et de libération de buffers noyau, de copie entre les buffers noyau et utilisateur, de copie avec calcul de checksum intégré, de mise en place d'un flux entre deux point d'accès, l'un d'eux pouvant être une point d'accès TCP et l'autre un descripteur de fichier, etc.

- un mécanisme permettant à l'application ou à une librairie de composer des appels systèmes spécifiques à partir de ces briques élémentaires.

Si les briques de base et le mécanisme de composition sont standardisés, il est alors possible d'implémenter toutes les APIs de façon portable. Dans le même temps, les applications ayant des besoins spécifiques peuvent aisément construire leur propre jeu d'appels systèmes et leur propre API. Ainsi une application de vidéo-conférence peut contrôler totalement le flux d'informations depuis l'espace utilisateur, sans être pénalisé par des transferts de données entre domaines.

Eviter les implémentations en couches inutiles

Nous pouvons là aussi distinguer deux niveaux distincts :

1- *La notion de couches dans le système de communication*

Les limitations intrinsèques de Streams constituent le prix à payer pour les avantages de modularité, flexibilité et de portabilité. Mais ces notions sont elles essentielles ? On peut ainsi se

demander si l'interface NPI entre les protocoles de transport et de réseau est bénéfique. Parce que ces deux couches ont habituellement la même origine, utiliser un format de message privé ne compromet ni la flexibilité ni la portabilité, et permet aux protocoles de n'échanger que les informations pertinentes. On peut éventuellement aller plus loin et remplacer cette interface à base de messages par une interface à base d'appels de fonctions. *Les interfaces normalisées n'ont d'intérêt que lorsqu'elles sont utilisées à bon escient.*

2- *La notion de couches dans le système d'exploitation*

La plupart des systèmes d'exploitation sont caractérisés par la distinction de plusieurs domaines de protection. C'est un moyen efficace de garantir confidentialité et indépendance entre applications, et d'implémenter des services généraux accessibles par chacun. Mais ici aussi l'architecture en couches a des impacts négatifs. La séparation utilisateur/noyau d'UNIX conduit à des copies de données et des changements d'abstractions qui affectent sérieusement les performances. On peut dès lors se poser la question de savoir si la structure en couches du système d'exploitation doit être maintenue ou non.

Les difficultés rencontrées lors de l'implémentation au niveau utilisateur de TCP montrent combien il peut être délicat de vouloir sortir un protocole du noyau (Section 5.2.3.2). C'est pourquoi nous pensons que dans l'ensemble la structure en couches est bénéfique et susceptible de satisfaire bon nombre d'applications, notamment celles qui communiquent peu et qui sont de ce fait peu affectées par les traversées de domaines.

Il est cependant nécessaire de proposer de nouveaux services systèmes afin d'apporter des solutions appropriées aux applications ayant des besoins spécifiques, et de limiter l'impact des traversées de domaines. Un certain nombre de ces services ont été recensés dans la Section 2.2.3 (`fbuf` en particulier). La méthode d'accès flexible définie ci-dessus est un autre exemple de technique permettant de s'affranchir dans une certaine mesure de la structure en couches du système d'exploitation.

Intégrer les drivers réseaux au système de communication

La tendance actuelle est de cacher la couche physique des couches supérieures afin de favoriser la portabilité du système de communication. C'est la raison d'être des interfaces CDLI, DLPI et NDIS [WindowsNT93]. Cependant nous avons montré que cette approche est inappropriée. L'intégration des drivers réseaux répond à plusieurs objectifs :

- permettre un contrôle d'accès au médium. La Section 3.5.1.2 a montré que cela pouvait être fait efficacement tout en restant indépendant des protocoles,

- offrir des services de gestion mémoire qui limitent le nombre de copies de données. Les `pbufs` et la carte Afterburner en sont deux exemples (Section 2.2.3). L'inconvénient est que ces solutions nécessitent un support matériel adéquat sous forme d'une mémoire double accès de taille suffisante.

- fournir des services de démultiplexage de bas niveau. Si cela est habituellement réalisé dans le cas d'ATM en tirant parti d'une association VCI-par-connexion (Section 2.2.3), cela peut être fait également avec des médias standards au sein du driver réseau.

5.2.3 Architecture de la pile de communication

5.2.3.1 Implémentations démultiplexées

La notion centrale de notre pile TCP/IP démultiplexée est celle de canal de communication (CC). Un CC est un chemin de données direct (c'est-à-dire sans multiplexage) entre l'application et le driver réseau. La notion de CC est conforme au principe de synchronisme défini précédemment puisque ces canaux sont désormais à la base de la gestion des threads.

Mais avoir une architecture démultiplexée n'est pas un but en soi. Ainsi elle ne simplifie pas de façon significative le chemin de données principal. En revanche, lorsqu'elle est couplée avec le contrôle de flux Streams, la nature démultiplexée de la pile est un atout majeur. Les CC deviennent alors la base (Section 3.5.1.2) :

- du contrôle de flux local, et
- du support de la QoS.

Il en résulte une gestion aisée des trois ressources de base de la machine : mémoire, processeur et médium de communication. Enfin, ce support de la QoS peut être utilisée dans le cas où la machine est configurée en système terminal ou en routeur.

Ainsi nous voyons que cette architecture repose grandement sur l'environnement Streams, et en exploite au maximum ses points forts : la communication par messages, la flexibilité qui en résulte, et le contrôle de flux. C'est la raison pour laquelle cette architecture n'est pas reproductible avec un environnement BSD.

Cependant les surcoûts très élevés introduits par l'environnement Streams utilisé cachent certains bénéfices de notre architecture. De plus cette approche nécessite un certain nombre de bouleversements : de l'intelligence est apportée à des composants, l'interface DLPI est modifiée, la gestion des connexions en fin de vie passe parfois par un CC dédié, etc. Ces inconvénients restent tout de même limités face aux bénéfices obtenus.

Positionnement du mécanisme de support de la QoS proposé par rapport à d'autres travaux

Différents degrés de support de la QoS au niveau système existent[3]. A un premier extrême, les environnements de communication classiques ne proposent aucun mécanisme particulier, et tous

3. Nous discutons ici du support système (donc local) de la QoS. Une solution globale de bout en bout nécessiterait bien sûr des mécanismes applicatifs et protocolaires (Section 2.2.1) supplémentaires.

les flux sont traités de façon égalitaire.

A l'opposé, le système QoS-A [Campbell94, Robin94] fournit des *garanties strictes* de respect des engagements de QoS. Ceci passe par l'utilisation de mécanismes de réservation de ressources lors de l'établissement du flux, de verrouillage en mémoire physique des buffers, d'allocation de ressources, de mise en forme des flux de données, de surveillance des paramètres de performances, etc. Une remarque que l'on peut faire à ce système est de reposer entièrement sur une infrastructure ATM et des protocoles particuliers, en l'occurrence IP++, proche de IPv6, dont les identificateurs de flux et de QoS sont largement utilisés, et sur un protocole de transport de type flots continus, METS. L'utilisation du système QoS-A dans un cadre plus classique n'est pas abordé.

L'approche "canal de communication" que nous proposons est beaucoup plus limitée. Elle permet une gestion intelligente des ressources essentielles de la machine hôte, en privilégiant en permanence les flux prioritaires. Elle peut être vue comme une gestion de type "meilleur effort" de la QoS. Il n'est donc pas possible de fournir des garanties strictes de respect de QoS uniquement sur la base de ces mécanismes. La fourniture de garanties statistiques passe obligatoirement par l'ajout d'un mécanisme de contrôle d'admission.

En résumé, l'inconvénient de l'approche QoS-A est de nécessiter un contrôle très fin et très complexe du système de communication. A l'opposé l'approche "canal de communication" permet un contrôle des allocations de ressources très simple qui peut fournir des garanties de QoS (locales) suffisantes pour de nombreuses applications.

Les autres architectures démultiplexées

Implémenter une pile de communication au niveau utilisateur conduit également à une architecture démultiplexée. Cependant avoir une architecture démultiplexée est généralement davantage un sous-produit que le but premier.

5.2.3.2 Implémentations de niveau utilisateur

Implémenter des protocoles au niveau utilisateur n'est pas un but en soi mais un moyen. Nous avons montré que ces implémentations souffrent de nombreuses limitations qui proviennent soit du fait que les protocoles sont dans des espaces d'adressage différents, soit du fait que la position dans la pile de communication de la frontière utilisateur/noyau a été déplacée (Section 3.5.2). Trois aspects doivent être distingués et vont déterminer l'ampleur des problèmes et la façon d'y faire face :

- Un premier point est de savoir *quelle partie* de la pile de communication doit être déplacée dans l'espace utilisateur : totalité ou seulement un protocole (notre cas). Les problèmes n'auront bien sûr pas la même ampleur dans chacun de ces cas.

- Ensuite vient la *nature des protocoles*. Les problèmes majeurs sont rencontrés avec les protocoles connectés pour lesquels la fiabilisassent des transmissions nécessite l'utilisation de timers, la conservation d'une copie des données dans le cas où il faille retransmettre, une

gestion adéquate des fermetures de connexion, etc. Tous ces problèmes sont absents dans le cas de protocoles non connectés, et une implémentation de niveau utilisateur est alors aisée.

- Enfin des *libertés permises* vont dépendre l'efficacité des méthodes mises en oeuvre pour résoudre les problèmes. Utiliser un système UNIX avec des drivers réseaux classiques et conserver une API Socket comme nous l'avons fait est trop rigide. Au contraire l'efficacité passe par l'emploi de méthodes d'accès spécialisées évitant les copies supplémentaires, par un démultiplexage très bas au sein du driver réseau voir de la carte d'adaptation, etc.

L'utilisation des implémentations de niveau utilisateur sera ainsi plus ou moins appropriée suivant le but recherché :

- Elles sont des alternatives intéressantes dans le cas des systèmes à base de micro-noyau. Elles permettent d'améliorer la médiocre architecture triangulaire (driver réseau, serveur UNIX de niveau utilisateur, et application) de ces systèmes. Cependant elles ne peuvent rivaliser avec les implémentations noyau traditionnelles.

- Elles sont aussi intéressantes lorsque l'on s'efforce de réduire le nombre de copies de données puisqu'elles permettent de court-circuiter complètement le noyau.

- Leur utilisation dans le cas d'implémentations ILP est par contre contestable. Des expériences [Braun95a, Roca95b] ont montré qu'appliquer ILP aux fonctions de présentation seules en conservant le protocole TCP dans le noyau était préférable. Les gains obtenus par une intégration de la routine de calcul de checksum TCP ne compensent en effet pas le surcoût lié à une implémentation au niveau utilisateur.

- Enfin elles sont intéressantes lors de l'adaptation des protocoles à l'application (Section 2.2.1). Parce que seules les fonctionnalités nécessaires sont retenues, cette technique est justifiée à chaque fois que le facteur limitant est lié aux mécanismes protocolaires et non aux traitements sur la station de travail. C'est le cas des applications multimédias opérant sur des connexions WAN.

5.3 ASPECT PROTOCOLE DE COMMUNICATION

Etant au coeur du système de communication, les protocoles jouent un rôle décisif. L'adéquation de leurs mécanismes au réseau sous-jacent et aux besoins des applications est essentielle. Plusieurs questions peuvent être posées : de nouveaux protocoles sont ils requis ? Dans quels buts ? Afin de supporter de nouveaux services et mécanismes ou bien afin de favoriser des implémentations efficaces ? Quelles sont les caractéristiques susceptibles d'améliorer ce dernier point ?

Il est évident que des considérations d'implémentations ne peuvent à elles seules justifier de la migration vers une nouvelle suite de protocoles. Mais lorsqu'il y a nécessité de changement, alors le fait que, sauf cas particuliers (liaisons asynchrones, réseaux mobiles), le système de communi-

cation soit devenu le facteur limitant doit conduire à apporter un grand soin à l'aspect efficacité de l'implémentation.

C'est ce qui nous intéresse ici. De nombreuses techniques sont d'ores et déjà connues : champs alignés sur des frontières de mots de 32 ou 64 bits, etc. Nous ne les présentons pas ici et nous n'insistons que sur deux aspects qui ont été moins couverts dans la littérature : en-têtes de taille fixe ou bien prédictible, et mécanisme de démultiplexage performant.

En-têtes de taille fixe ou bien prédictible

L'utilisation d'options par opposition aux en-têtes de taille fixe doit être vue comme une opposition entre flexibilité et efficacité. Il est couramment reconnu qu'avoir des options, et de ce fait une taille variable d'en-têtes, doit être évité afin de favoriser les traitements. C'est la solution retenue pour XTP. Cependant la plupart des extensions à TCP ont été rendues possibles par la présence d'options. Les surcoûts liés au traitement de ces options dans le cas des RFC1323 et RFC1644 sont largement compensés par les bénéfices retirés, notamment du point de vue performances. Toutes choses considérées, nous pensons que *les options, par la souplesse qu'elles apportent, sont bénéfiques, même d'un point de vue performances*.

La présence d'options n'est pas la seule cause d'en-têtes de taille variable. Dans le monde OSI les adresses ont également une taille variable pouvant aller jusqu'à 20 octets. C'est visiblement trop de souplesse. Une taille importante mais fixe (de même que dans IPv6) est préférable.

Si les en-têtes de taille fixe ne sont pas recommandés, en revanche *la taille des en-têtes se doit d'être prédictible*. C'est le cas du RFC1323 : une fois la connexion établie les deux hôtes savent que les paquets successifs contiendront une option "timestamp". Cette connaissance peut être ensuite utilisée lors des traitements des protocoles. Par exemple avoir des en-têtes de taille prévisible est suffisant pour permettre d'utiliser une technique de "scattering" avec notre TCP de niveau utilisateur. La taille de l'en-tête étant connue, les données peuvent être directement copiées dans le buffer de l'application.

Mécanisme de démultiplexage performant

Le démultiplexage précoce des paquets entrants est devenu très populaire. C'est nécessaire dans le cas de notre pile démultiplexée et des implémentations de niveau utilisateur. Les protocoles de communication actuels ne favorisent pas cette opération de démultiplexage : analyse des divers en-têtes de protocoles, recherche linéaire ou de type hachage au sein de tables. Ceci a conduit à utiliser ATM dont les VCIs peuvent théoriquement être associés aux connexions de transport (Section 2.2.3). Cette solution est restrictive (on est lié à une technologie réseau) et non garantie (ce type d'allocation de VCIs sera t'il possible du point de vue technique et tarification).

Nous pensons qu'une solution protocole est préférable. XTP [XTP95] propose un mécanisme d'échange de clés destiné à faciliter les opérations de démultiplexage. IPv6 [Deering95] définit un identificateur de flux qui peut potentiellement être alloué par connexion. Son utilisation précise reste cependant à définir [RFC1809]. Notons que cette approche est différente de celle d'XTP

puisqu'elle induit une redondance (le démultiplexage peut se faire soit traditionnellement, soit par l'identificateur de flux) ce qui permet son utilisation avec des protocoles de transport traditionnels.

CHAPITRE 6

Conclusion

Durant ces travaux nous avons étudié un large panorama de techniques d'implémentation hautes performances. Notre plate-forme expérimentale nous a conduit à analyser les relations entre l'efficacité de ces techniques et leur environnement d'implantation, à savoir les aspects architecture de la machine hôte, système d'exploitation, et protocole de communication.

Ainsi, pour l'aspect architecture de la machine hôte, nous montrons que la parallélisation du système de communication est décevante. Elle est en particulier grandement limitée par les contentions sur les verrous d'origine système. De plus la généralisation des machines multiprocesseurs s'est faite au détriment des machines monoprocesseurs puisque l'une et l'autre partagent désormais le même système d'exploitation parallélisé.

Nous avons également montré comment la technique ILP, destinée à limiter les accès mémoire, peut être intégrée à la méthode d'accès d'une pile TCP/IP. Cependant, ses bénéfices dépendent largement des conditions d'utilisation (MTU, taille des TSDUs), de la nature de l'application (interactive ou effectuant des transferts de masses), et de celle des fonctions de manipulation de données (complexité des algorithmes). Notre méthode d'accès ILP a aussi révélé que l'amélioration de la gestion mémoire et la réduction du nombre d'appels systèmes ont des impacts de performance tout aussi importants que la technique ILP.

Du point de vue système, l'étude de deux environnements d'exécution de protocoles radicalement opposés, BSD et Streams, a permis d'identifier leurs faiblesses. En particulier nous montrons que l'environnement Streams actuel n'est pas adapté aux hautes performances, car il souffre de limita-

tions fonctionnelles, de choix d'implémentation coûteux, et surtout d'une mauvaise utilisation. Ceci nous a conduit à définir un ensemble de principes qu'un environnement hautes performances se doit de respecter : simplicité, synchronisme, flexibilité, suppression des implémentations en couches, et intégration des drivers réseaux.

L'architecture de la pile de communication s'avère elle aussi essentielle. Nous montrons qu'une architecture démultiplexée, avec des chemins de données directs entre applications et drivers réseaux, est très largement bénéfique. Elle permet un excellent support du contrôle de flux local et des mécanismes systèmes de gestion de la QoS. Le contrôle des ressources mémoire, processeur, et médium est alors à la fois simple et efficace.

En revanche les implémentations de niveau utilisateur souffrent de nombreuses limites. Mais ces implémentations restent indispensables, en particulier pour apporter plus de flexibilité et de contrôle aux applications par le biais des techniques de composition de protocoles.

Enfin, en ce qui concerne l'aspect protocole de communication, nous montrons que la présence d'options dans les en-têtes de paquets n'est pas contraire à l'obtention de bonnes performances. A la notion trop rigide d'en-têtes de taille fixe nous substituons celle d'en-têtes de taille prédictible.

Si l'on se place à un niveau d'abstraction plus élevé, il ressort deux notions clés de ces travaux :

- la *simplicité* et
- la *flexibilité*.

Pour chaque problème existe habituellement une solution naturelle, à la fois simple et efficace. Malheureusement, lorsque toutes les pièces formant le système de communication sont assemblées, plusieurs contraintes et incompatibilités apparaissent. Des compromis sont alors nécessaires, ce qui conduit à compliquer et à rendre moins efficaces les différents composants. C'est la raison pour laquelle certaines techniques hautes performances s'avèrent moins efficaces qu'il n'a été prévu. C'est le cas du parallélisme ou de ILP dont l'utilisation a conduit à augmenter la complexité générale du système de communication pour des gains parfois réduits.

A l'opposé, notre architecture démultiplexée a permis d'implémenter simplement et efficacement des mécanismes de gestion des ressources fondés sur des notions de saturation de l'adaptateur ou de QoS. Le synchronisme des traitements est également un facteur important de simplification et d'efficacité au sein du système de communication.

Mais la complexité ne limite pas seulement les performances, elle limite également les fonctionnalités du système de communication. Il est actuellement très difficile de contrôler de façon stricte tous les aspects de qualité de service en raison de la complexité de l'environnement de communication et des modèles de QoS.

Par conséquent nous pensons que les considérations de simplicité sont essentielles et se doivent de guider les choix de conception. C'est le seul moyen de limiter "l'entropie système" (Section

2.1.3.2) et de fournir des services satisfaisants.

La notion de flexibilité est souvent opposée à celle d'efficacité. Nous avons rencontré plusieurs exemples d'une telle opposition :

- dans le système d'exploitation : les notions de communication par passage de messages et d'indépendance des composants sont sources de problèmes qui sont évités dans les architectures prônant l'intégration des composants.

- dans la méthode d'accès : la méthode d'accès flexible que nous proposons est indéniablement plus coûteuse dans le cas général que les Sockets qui intègrent API et méthode d'accès en une unique couche.

- entre l'application, le système de communication et le réseau : les mécanismes utilisés pour l'adaptation mutuelle des ces trois composants (informations de rétroaction, couches de gestion de la QoS, composition de protocoles intégrés à l'application, etc.) ont tous un coût.

- dans les protocoles : la présence d'options rend le traitement des paquets plus complexe que dans le cas d'en-têtes de taille fixe.

Le prix de la flexibilité provient de l'impossibilité de tirer profit d'hypothèses simplificatrices propres à une situation particulière. L'avantage est de pouvoir s'adapter à un environnement changeant, et ainsi d'éviter une rapide obsolescence.

Flexibilité et efficacité ne sont donc pas nécessairement antinomiques. Pour chacun des quatre points mentionnés ci-dessus, les bénéfices apportés par la flexibilité compensent largement, même d'un point de vue performances, les surcoûts qu'elle occasionne. Ainsi :

- la flexibilité de Streams est indispensable pour concevoir une implémentation démultiplexée et bénéficier de ses avantages,

- la méthode d'accès flexible proposée offre la possibilité d'ajouter aisément de nouveaux appels systèmes aux fonctionnalités très évoluées,

- les mécanismes d'adaptation des applications, système de communication et réseau voient leur coût largement compensé par les bénéfices retirés, et enfin

- la présence d'options permet d'apporter de nouvelles fonctionnalités aux protocoles sans compromettre la compatibilité.

Les travaux que nous avons menés nous ont permis d'aborder deux tendances importantes dans le domaine des réseaux : d'une part les systèmes ouverts et leurs environnements standards d'exécution de protocoles, qui prônent l'indépendance et la portabilité des composants, et pour lesquels la compatibilité ascendante est un critère essentiel; d'autre part des techniques avancées, parfois encore expérimentales, qui repensent totalement le système de communication. Parmi les

voies suivies se trouvent les nouveaux protocoles et systèmes d'exploitations permettant le support du multimédia, les nouvelles piles de communication créées sur mesure afin de donner un plus grand contrôle aux applications et utilisateurs - ce sont eux qui ont la meilleure connaissance des caractéristiques des flux de données et du service désiré -, les applications intelligentes pouvant s'adapter au réseau etc.

Nous nous sommes efforcés dans ces travaux de montrer comment ces deux axes peuvent cohabiter, voir fusionner le cas échéant, mais de nombreux points restent ouverts.

Bibliographie

[Abbott 92] M. Abbott, L. Peterson, “Automated integration of communication protocol layers”, Research Report, TR 92-24, Departement of Computer Science, University of Arizona, Tucson, décembre 1992.

[Ahlgren 93] B. Ahlgren, B. Lyles, “Avoiding copying data in ATM host/network interfaces”, unpublished draft, mars 1993.

[Ahlgren 94] B. Ahlgren, P. Gunningberg, “A minimal copy network interface architecture supporting ILP and ALF”, HIPPARCH’94, Sophia-Antipolis, décembre 1994.

[Ahlgren 95] B. Ahlgren, M. Bjorkman, K. Moldeklev, “The performance of a no-copy API for communication”, Third IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems (HPCS’95), août 1995.

[Anderson 95] T. Anderson, D. Culler, D. Patterson, “A case for NOW (Networks of Workstations)”, IEEE Micro, URL: http://now.cs.berkeley.edu/, février 1995.

[Bailey 94] M. Bailey, B. Gopal, M. Pagels, L. Peterson, “PATHFINDER: a pattern-based packet classifier”, USENIX Operating Systems Design and Implementation (OSDI), California, novembre 1994.

[Bhatti 95] N. Bhatti, R. Schlichting, “A system for constructing configurable high-level

protocols", ACM SIGCOMM'95, août 1995.

[Biersack 94] E. Biersack, E. Rütsche, T. Unterschütz, "Demultiplexing on the ATM adapter: experiments with Internet protocols in User space", HIPPARCH'94, Sophia-Antipolis, France, décembre 1994.

[Bjorkman 93a] M. Bjorkman, P. Gunningberg, "Locking effects in multiprocessor implementations of protocols", ACM SIGCOMM'93, septembre 1993.

[Bjorkman 93b] M. Bjorkman, "Architectures for high performance communication", Doctoral dissertation, Department of Computer Systems, Uppsala University, PO Box 325, S-751 05 Uppsala, Sweden, septembre 1993.

[Bolot 94] J-C. Bolot, T. Turletti, I. Wakeman, "Scalable feedback control for multicast video distribution in the Internet", SIGCOMM'94, septembre 1994.

[Boykin 90] J. Boykin, A. Langerman, "Mach/4.3 BSD: a conservative approach to parallelization" , Computing Systems, USENIX Winter'90, janvier 1990.

[Braun 92] T. Braun, M. Zitterbart, "Parallelism in XTP" , Transfer Magazine, mai-juin 1992.

[Braun 94] T. Braun, "PATROCLOS: a flexible and high-performance transport subsystem", Proceedings of the IFIP fourth Workshop on Protocols For High Speed-Networks (PfHSN'94), Vancouver, Canada, août 1994.

[Braun 95a] T. Braun, C. Diot, "Protocol implementation using Integrated Layer Processing", ACM SIGCOMM'95, août 1995.

[Braun95b] T. Braun, C. Diot, A. Hoglander, V. Roca, "Experimental evaluation of TCP in user space", Technical Report, INRIA Sophia-Antipolis, 1995.

[Campbell 91] M. Campbell, R. Barton, J. Browning, D. Cervenka, B. Curry, ... "The parallelization of UNIX System V Release 4.0", USENIX Winter'91, Dallas, janvier 1991.

[Campbell 94] A. Campbell, G. Coulson, D. Hutshinson, "A quality of service architecture", ACM SIGCOMM Computer Communication Review, Vol 24 No 2, avril 1994.

[Castelluccia 94] C. Castelluccia, W. Dabbous, "Modular communication subsystem implementation using a synchronous approach", USENIX, High-Speed Networking, août 1994.

[Castelluccia 95] C. Castelluccia, P. Hoschka, "A compiler based approach to protocol optimization", Third IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems (HPCS'95), août 1995.

[Chesson 91] G. Chesson, "An introduction to the XTP/PE project", Interop'91, 1991.

[Chesson 94] G. Chesson, “Experiments with lock-free data structures and parallel communication software”, Invited speaker during HPN’94, Grenoble, France, juin 1994.

[Chrisment 94] I. Chrisment, “Impact of ALF on communication subsystems design and performance”, HIPPARCH’94, Sophia-Antipolis, France, décembre 1994.

[Clark 89] D. Clark, V. Jacobson, J. Romkey, H. Salwen, “An analysis of TCP processing overhead” , IEEE Communication Magazine, juin 1989.

[Clark 90] D.D. Clark, D.L. Tennenhouse, “Architectural Considerations for a New Generation of Protocols” , SIGCOMM’90, septembre 1990.

[Cole 95] R. Cole, D. Shur, “IP over ATM: a framework document”, Internet Draft, draft-ietf-ipatm-framework-doc-02, avril 1995.

[Crowcroft 92] J. Crowcroft, I. Wakeman, Z. Wang, “Layering considered harmful”, IEEE Network, Vol 6 No 1, janvier 1992.

[Crowcroft 95] J. Crowcroft, “TCP very slow start and freeway avoidance”, discussion on the end2end mailing list initiated by the message <899.812190626@cs.ucl.ac.uk>, septembre 1995.

[Deering 95] S. Deering, “Internet Protocol, version 6 (IPv6) specification”, Internet Draft, draft-ietf-ipv6-spec-02.txt, juin 1995.

[Diaz 95] M. Diaz, K. Drira, A. Lozes, C. Chassot, “On the definition and representation of the quality of service for multimedia applications”, HPN’95, Palma, Spain, septembre 1995.

[DiGenova 95] P. Di Genova, G. Ventre, “Efficiency comparison of real-time transport protocols”, HPN’95, Palma, Spain, septembre 1995.

[Diot 94] C. Diot, R. de Simone, C. Huitema, “Communication protocols development using ESTEREL”, HIPPARCH’94, Sophia-Antipolis, France, décembre 1994.

[Diot 95a] C. Diot, C. Huitema, T. Turletti, “Multimedia applications should be adaptative”, Third IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems (HPCS’95), août 1995.

[Diot 95b] C. Diot, I. Chrisment, A. Richards, “Application Level Framing and automated implementation”, HPN’95, Palma, Spain, septembre 1995.

[Druschel 93] P. Druschel, L. Peterson, “Fbufs, a high bandwidth cross-domain transfer facility”, Proceedings of the 14th ACM Symposium on Operating Systems Principles”, décembre 1993.

[Druschel 94] P. Druschel, L. Peterson, B. Davie, “Experience with a high-speed network

adaptor: a software perspective", SIGCOMM'94, septembre 1994.

[Edwards 94] A. Edwards, G. Watson, J. Lumley, D. Banks, C. Calamvokis, C. Dalton, "User-space protocols deliver high performance to applications on a low cost Gb/s LAN", SIGCOMM'94, septembre 1994.

[Edwards 95] A. Edwards, S. Muir, "Experiences implementing a high performance TCP in user-space", ACM SIGCOMM'95, août 1995.

[Feldmeier 90] D. Feldmeier, "Multiplexing issues in communication system design", ACM SIGCOMM'90, septembre 1990.

[Feldmeier 93a] D. Feldmeier, "A survey of high performance protocol implementation techniques" , Research Report, Bellcore, février 1993.

[Feldmeier 93b] D. Feldmeier, "A framework of architectural concepts for high-speed communication systems", IEEE Journal on Seclected Areas in Communications, mai 1993.

[Feldmeier 93c] D. Feldmeier, "A data labelling technique for high-performance protocol processing and its consequences", ACM SIGCOMM'93, septembre 1993.

[Ferrari 92] D. Ferrari, A. Banerjea, H. Zhang, "Network support for multimedia - a discussion of the Tenet approach", Technical Report TR-92-072, ICSI, Berkeley, octobre 1992.

[Fdida 94] S. Fdida, K.L. Thai, E. Anique, "Architectures de communication pour applications réparties", Actes du Xème congrès: De Nouvelles Architectures pour les Communications, Paris, mai 1994.

[Garg 90] A. Garg, "Parallel Streams: a multiprocessor implementation", USENIX Winter'90, janvier 1990.

[Ghosh 94] A. Ghosh, J. Crowcroft, M. Fry, M. Handley, "Integrated layer video decoding and application layer framed secure login: general lessons from two or three very different applications", HIPPARCH'94, Sophia-Antipolis, décembre 1994.

[Gnuplot 93] T. Williams, C. Kelley, R. Lang, D. Kotz, J. Campbell, G. Elber "GNUPLOT 3.4", GNU software, juin 1993.

[Gunningberg 91] P. Gunningberg, C. Partridge, T. Sirotkin, B. Victor, "Delayed evaluation of gigabit protocols", Proceedings of the second MultiG Workshop, juin 1991.

[Heavens 92] I. Heavens, "Experiences in fine grain parallelisation of Streams-based communication drivers" , Technical OpenForum, Utrecht, Netherlands, novembre 1992.

[Hutchinson 89] N.C. Hutchinson, S. Mishra, L.L. Peterson, V.T. Thomas, "Tools for

Implementing Network Protocols” , Software - Practice and Experience, Vol 19 No 9, septembre 1989.

[Hutchinson 91] N.C. Hutchinson, L.L. Peterson, “The x-Kernel, an architecture for implementing network protocols” , IEEE Transactions on Software Engineering, vol 17 No 1, janvier 1991.

[IBM 92] “AIX version 3.2 for RISC System/6000TM: Kernel extensions and device support programming concepts”, IBM technical documentation, SC23-2207-01, janvier 1992.

[IBM 93] “High speed networking technology: an introductory survey”, Document number GG24-3816-01, Raleigh, juin 1993.

[IBM 94] “The PowerPC architecture: a specification for a new family of RISC processors”, IBM, Morgan Kaufmann Publishers Inc., San Francisco, 1994.

[Jacobson 90] V. Jacobson, “4BSD TCP header prediction”, ACM Computer Communication Review, Vol 20 No 2, avril 1990.

[Jacobson 93] V. Jacobson, “TCP receive packet processing in 30 instructions”, News sent on the comp.protocols.tcp-ip list, septembre 1993.

[Jain 94] P. Jain, N. Hutchinson, S. Chanson, “A framework for the non-monolithic implementation of protocols in the x-kernel”, USENIX, High-Speed Networking, août 1994.

[Kay 93b] J. Kay, J. Pasquale, “The importance of non-data touching processing overheads in TCP/IP”, ACM SIGCOMM’93, septembre 1993.

[Kleiman 92] S. Kleiman, J. Voll, J. Eykholt, A. Shivalingiah, D. Williams, M. Smith, S. Barton, G. Skinner, “Symetric Multiprocessing in Solaris 2.0” , COMPCON, California, printemps 1992.

[Kleinpaste 95] K. Kleinpaste, P. Steenkiste, B. Zill, “Software support for outboard buffering and checksumming”, ACM SIGCOMM’95, août 1995.

[Maeda 92] C. Maeda, B. Bershad, “Networking performance for microkernels”, Proceedings of the 3rd Workshop on Workstation Operating Systems (WWOS-3), disponible par FTP à mach.cs.cmu.edu sous /doc/published/netperf.ps, avril 92.

[Maeda 93] C. Maeda, B. Bershad, “Protocol service decomposition for high-performance networking”, 14th ACM Symposium on Operating Principles, disponible par FTP à mach.cs.cmu.edu sous /doc/published/user.level.protocols.ps, décembre 1993.

[McCanne 93] S. McCanne, V. Jacobson, “The BSD packet filter: a new architecture for user-

level packet capture", USENIX Winter'93, janvier 1993.

[Mogul 91] J. Mogul, A. Borg, "The effect of context switches on cache performance", Architecture Support for Programming Languages and Operating Systems (ASPLOS), California, avril 1991.

[Montz 94] A. Montz, D. Mosberger, S. O'Malley, L. Peterson, T. Proebsting, J. Hartman, "Scout: a communications-oriented operating system", Research Report TR 94-20, Departement of Computer Science, University of Arizona, disponible par FTP à ftp.cs.arizona.edu sous xkernel/Papers/scout.ps, juin 1994.

[Moore 93] C. Moore, "The PowerPC 601 microprocessor", IEEE COMPCON'93, février 1993.

[Mosberger95] D. Mosberger, L. Peterson, S. O'Malley, "Protocol latency: MIPS and reality", Technical Report TR 95-02, University of Arizona, 1995.

[Nahum 94] E. Nahum, D. Yates, J. Kurose, D. Towsley, "Performance issues in parallelized network protocols", USENIX Operating Systems Design and Implementation (OSDI), California, disponible par FTP à gaia.cs.umass.edu sous "/pub/Nahu94:Performance.ps.Z", novembre 1994.

[Narten 95] T. Narten, "Neighbor Discovery for IP version 6 (IPv6)", Internet Draft, draft-ietf-ipngwg-discovery-02.txt, septembre 1995.

[Nicholson 91] A. Nicholson, J. Golio, D. Borman, J. Young, W. Roiger, "High speed networking at Cray Research" , ACM SIGCOMM Computer Communication Review, Vol 21, No 1, janvier 1991.

[Pagels 94] M. Pagels, P. Druschel, L. Peterson, "Cache and TLB effectiveness in processing network I/O", Technical Report TR 94-08, Departement of Computer Science, University of Arizona, disponible par FTP à ftp.cs.arizona.edu sous xkernel/Papers/cache.ps, avril 1994.

[Peterson 90] L. Peterson, N. Hutchinson, S. O'Malley, H. Rao, "The x-kernel, a platform for accessing Internet ressources", IEEE Computer, Vol 23 No 5, mai 1990.

[RFC 817] D. Clark, "Modularity and efficiency in protocol implementation", Request For Comments, juillet 1982.

[RFC 896] J. Nagles, "Congestion control in TCP/IP networks", Request For Comments, janvier 1984.

[RFC 1112] S. Deering, "Host extensions for IP multicasting", Request For Comments, août 1989.

[RFC 1144] V. Jacobson, "Compressing TCP/IP headers for low-speed serial links", Request For Comments, février 1990.

[RFC 1190] "Experimental Internet Stream Protocol, version 2 (ST-II)", Request For Comments, octobre 1990.

[RFC 1191] J. Mogul, S. Deering, "Path MTU discovery", Request For Comments, novembre 1990.

[RFC 1323] V. Jacobson, R. Braden, D. Borman, "TCP Extensions for high performance", Request For Comments, mai 1992.

[RFC 1349] P. Almquist, "Type of Service in the Internet protocol suite", Request For Comments, juillet 1992.

[RFC 1577] M. Laubach, Classical IP over ATM", Request for Comments, janvier 1994.

[RFC 1633] R. Braden, D. Clark, S. Shenker, "Integrated services in the Internet architecture: an overview", Request For Comments, juin 1994.

[RFC 1644] R. Braden, "T/TCP - TCP extensions for fransactions, functional specification", juillet 1994.

[RFC 1809] C. Partridge, "Using the flow label field in IPv6", Request For Comments, juin 1995.

[Richards 95] A. Richards, "The universal transport service, an adaptative end-to-end protocol analysis and design", Thesis Report, University of Technology, Sydney, juillet 1995.

[RMTP 95] "Reliable multicast - quote without comment", end2end mailing list, message <199511062339.AA12443@can.isi.edu>, novembre 1995.

[Robin 94] P. Robin, G. Coulson, A. Campbell, G. Blair, M. Papathomas, D. Hutchinson, "Implementing a QoS controlled ATM based communications system in Chorus", Proceedings of the IFIP fourth Workshop on Protocols For High-Speed Networks (PfHSN'94), Vancouver, Canada, août 1994.

[Roca 93] V. Roca, C. Diot, "XTP and TCP/IP comparison in a Streams environment", Proceedings of the Fourth Workshop on future trends of Distributed Computing Systems, Lisboa, Portugal, septembre 1993.

[Roca 94] V. Roca, C. Diot, "A high performance Streams-based architecture for communication subsystems", Proceedings of the IFIP fourth Workshop on Protocols For High Speed-Networks (PfHSN'94), Vancouver, Canada, août 1994.

[Roca95a] V. Roca, "A networking performance evaluation environment", Internal Bull Technical Report, juillet 1995.

[Roca 95b] V. Roca, T. Braun, C. Diot, "Design of efficient communication architectures for

open systems", ready for submission paper, 1995.

[Romanow 94] A. Romanow, S. Floyd, "Dynamics of TCP traffic over ATM networks", SIGCOMM'94, septembre 1994.

[Rutsche 92] E. Rutsche, M. Kaiserswerth, "TCP/IP on the Parallel Protocol Engine" , Proceedings High Performance Networking (HPN'92), Liege, Belgium, décembre 1992.

[Salehi 94] J. Salehi, J. Kurose, D. Towsley, "Schedulling for cache affinity in parallelized cummunication protocols", Technical Report UM-CS-1994-075, University of Massachusetts, disponible par ftp à gaia.cs.umass.edu sous /pub/ Sale95:Scheduling.ps.Z, octobre 1994.

[Saxena 93] S. Saxena, J.K. Peacock, F. Yang, V. Verma, M. Krishnan, "Pitfalls in multithreading SVR4 STREAMS and other weightless processes", USENIX Winter'93, San Diego, janvier 1993.

[Schmidt 94] D. Schmidt, T. Suda, "A framework for measuring the performance of alternative process architectures for parallelizing communication subsystems", IEEE/ACM Journal of Transactions on Networking (a restricted version also appeared in PfHSN'94, août 1994), URL http://www.cs.wustl.edu/~schmidt, 1994.

[Schulzrinne 95] H. Schulzrinne, Casner, Frederick, Jacobson, "RTP: a transport protocol for real-time applications", Internet draft, draft-ietf-avt-rtp-07.txt, mars 1995.

[Shenker 94] S. Shenker, "Fundamental design issues for the future Internet", preprint submitted to JSAC, 1994.

[Speer 94] S. Speer, R. Kumar, C. Partridge, "Improving UNIX kernel performance using profile based optimization", USENIX Winter'94, janvier 1994.

[Stevens 94] W.R. Stevens, "TCP/IP illustrated volume 1: the protocols", Professional Computing Series, Addison-Wesley Publishing Company, 1994.

[Stevens 95] W.R. Stevens, G. Wright, "TCP/IP illustrated volume 2: the implementation", Professional Computing Series, Addison-Wesley Publishing Company, 1995.

[Streams 90] "UNIX system V release 4; programmer's guide: STREAMS", Prentice Hall, 1990.

[SunOS 93] "Multi-threaded Streams", Chapter 13 of the SunOS 5.2 Streams Programmer's Guide, mai 1993.

[Talbot 95] J. Talbot, "Turning the AIX operating system into an MP-capable OS", USENIX 95, janvier 1995.

[Tantawy 93] A. Tantawy, “Réalisation de protocoles à haute performance”, Proceedings of the CFIP’93 Conference, septembre 1993.

[Thekkath 93a] C. Thekkath, H. Levy, “Limits to low-latency communication on high-speed networks”, ACM Transactions on Computer Systems, Vol 11 No 2, mai 1993.

[Thekkath 93b] C. Thekkath, T. Nguyen, E. Moy, E. Lazowska, “Implementing network protocols at user level”, ACM SIGCOMM’93, septembre 1993.

[Tracey 94] J. Tracey, A. Banerji, “Device driver issues in high-performance networking”, USENIX, High-Speed Networking, août 1994.

[Vessey 90] I. Vessey, G. Skinner, “Implementing Berkeley sockets in System V Release 4”, USENIX Winter’90, Dallas, janvier 1990.

[Wakeman 95] I. Wakeman, A. Ghosh, J. Crowcroft, V. Jacobson, S. Floyd, “Implementing real time packet forwarding policies using Streams”, USENIX, janvier 1995.

[Weaver 94] A. Weaver, “Xpress Transport Protocol version 4”, TRANSFER, Vol 7 No 6, novembre/décembre 1994.

[WindowsNT 93] H. Custer, “Inside Windows NT”, Microsoft Press, 1993.

[Wolman 94] A. Wolman, G. Voelker, C. Thekkath, “Latency analysis of TCP on an ATM network”, USENIX Winter’94, San Francisco, janvier 1994.

[XTI 93] “X/Open Transport Interface (XTI) version 2”, X/Open Company, Ltd., septembre 1993.

[XTP 95] “Xpress Transport Protocol specification, XTP Revision 4.0” , XTP Forum 95-20, mars 1995.

[Yuhara 94] M. Yuhara, B. Bershad, C. Maeda, J. Moss, “Efficient packet demultiplexing for multiple endpoints and large messages”, USENIX Winter’94, janvier 1994.

ANNEXE A

Propositions d'amélioration de Streams et de son usage

Nous avons vu à la Section 5.2.1.1 que Streams ne pouvait être considéré comme étant un environnement hautes performances. Cependant il est possible, grâce à la résolution d'un certain nombre de ses limites et à une utilisation appropriée de ses fonctionnalités, de remédier (partiellement) à la situation. Obtenir un niveau de performances similaire à celui d'une pile BSD est envisageable sur un monoprocesseur (Figure 19 page 75). Nous faisons ici une synthèse des techniques à mettre en oeuvre dans une implémentation Streams performante.

Notre but n'est pas de contester la communication par messages de Streams ni sa méthode d'accès en couches. Les surcoûts induits constituent le prix à payer pour bénéficier des avantages de flexibilité et de portabilité. Mais ces caractéristiques, du fait de leur coût, doivent être utilisées avec clairvoyance.

Résoudre le problème de gestion des buffers de l'application coté réception est par contre le premier impératif. Plusieurs algorithmes sont discutés à la Section 3.5.1.3.

Garantir le synchronisme des traitements est également capital (Section 5.2.2). L'utilisation des routines de service doit être bannie. En effet, ces routines introduisent des délais supplémentaires (fonction de l'environnement Streams utilisé), voir des changements de contextes, en particulier sur les versions multiprocesseurs. La notion utilisée par notre environnement Streams selon laquelle un message peut être traité par n'importe quelle thread, est coûteuse car elle implique que des informations soient copiées dans une structure attachée au message. Il est préférable que la thread applicative effectue les traitements requis lors d'un appel système.

La simplicité des solutions retenues est aussi capitale. Le mécanisme de synchronisation utilisé sur notre environnement en est un contre-exemple.

Enfin nous pouvons résumer nos recommandations concernant l'usage des queues Streams et des routines de services[1] associées comme suit :

- effectuer des traitements asynchrones : OUI, mais c'est d'un intérêt limité dans le cas d'une pile TCP/IP pour laquelle la latence est un soucis majeur.

- réduire le niveau de priorité des traitements suite à une interruption : OUI, cela permet d'effectuer ces traitements à un plus faible niveau de priorité, et donc de préserver les caractéristiques temps-réel de l'OS (Section 4.2.1.3). C'est utile dans le cas des trames Ethernet entrantes et des arrivées à échéance de timers.

- effectuer un contrôle de flux local : NON, le contrôle de flux est relatif à la saturation de la queue, non à l'utilisation systématique des routines de service. Un driver/module peut bloquer un stream en insérant un message dédié, de taille supérieure à la taille maximale de la queue (qui peut dans ce cas être initialisée à un) dans celle-ci. Le déblocage se fait simplement en retirant ce message de la queue[2] (Section 3.5.1.3).

  Cas des requêtes de transmission de données : la plupart des piles TCP/IP sous Streams insèrent systématiquement les TSDUs dans la queue et les traitent dans la fonction de service. Cette stratégie doit être bannie au profit d'un mécanisme de listes privées de TSDUs et de messages bloquants.

- ajout d'un parallélisme vertical : NON, cette forme de parallélisme est connue pour être médiocre (Section 2.2.4).

1. Queues et routines de service sont liées : insérer un message dans une queue déclenche le scheduling de la routine de service associée. Ce scheduling peut toute fois être inhibé par l'usage de la routine Streams `noenable`.
2. Une routine Streams agissant directement sur l'indicateur de saturation de la queue serait ici la bienvenue puisqu'elle éviterait le recours artificiel à ces messages de blocage.