Titre: Title:	Conception, mise en oeuvre et caractérisation de mécanismes matériel/logiciel pour l'interconnexion firewire-ethernet
Auteur: Author:	Mame Maria Mbaye
Date:	2004
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Mbaye, M. M. (2004). Conception, mise en oeuvre et caractérisation de mécanismes matériel/logiciel pour l'interconnexion firewire-ethernet [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/7195/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/7195/
Directeurs de recherche: Advisors:	Yvon Savaria, & Samuel Pierre
Programme: Program:	Génie électrique

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

CONCEPTION, MISE EN ŒUVRE ET CARACTÉRISATION DE MÉCANISMES MATÉRIEL/LOGICIEL POUR L'INTERCONNEXION FIREWIRE-ETHERNET

MAME MARIA MBAYE

DÉPARTEMENT DE GÉNIE ÉLECTRIQUE

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION

DU DIPLÔME DE MAÎTRISE EN SCIENCES APPLIQUÉES

(GÉNIE ÉLECTRIQUE)

AVRIL 2004

Your file Votre référence
ISBN: 0-612-91955-2
Our file Notre référence
ISBN: 0-612-91955-2

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

CONCEPTION, MISE EN ŒUVRE ET CARACTÉRISATION DE MÉCANISMES MATÉRIEL/LOGICIEL POUR L'INTERCONNEXION FIREWIRE-ETHERNET

Présenté par : MBAYE Mame Maria

En vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées

A été dûment accepté par le jury d'examen constitué de :

M. CHAMBERLAND Steven, Ph.D., Président de jury

M. SAVARIA Yvon, Ph.D., directeur et membre

M. PIERRE Samuel, Ph.D., codirecteur et membre

M. BOYER Raymond-François, Ph.D., membre

REMERCIEMENTS

Je tiens à remercier Monsieur Savaria, mon directeur de maîtrise pour sa disponibilité, ses conseils et son analyse qui m'ont permis de mener à terme mes travaux de recherche dans les meilleures conditions. Je tiens à souligner la contribution de Monsieur Pierre, mon co-directeur de maîtrise, qui m'a permis de mieux aborder l'aspect réseautique de mon projet de maîtrise. Je voudrais aussi souligner le support financier de Gennum Corp et de Micronet.

J'en profite pour remercier tous mes camarades du GRM (Groupe de Recherche en Microélectronique) et en particulier S. Regimbal pour son aide dans le domaine de la vérification de design matériel, ainsi que H. Ghattas, O. Mahrez, J. Pepga, A. Chureau, M. Dubois, sans oublier le technicien du groupe de recherche A. Vesey et l'administrateur réseaux R. Lepage.

Je terminerai par remercier mes amis, mes frères et ma sœur et bien sûr, mes parents qui m'ont soutenue moralement tout au long de mes études.

RÉSUMÉ

Au début des années 90, une demande est apparue pour les applications audio/vidéo numériques. Elle coïncidait avec l'utilisation de plus en plus fréquente des algorithmes de compression comme MPEG. Malheureusement, à l'époque, les protocoles de transmission tels que SCSI ou IDE ne pouvaient pas supporter les nouvelles applications vidéo. C'est dans ce contexte qu'un nouveau protocole proposé par la compagnie Apple a vu le jour : Firewire qui est ensuite devenu IEEE 1394.

Le but de ce mémoire est d'étudier des méthodes d'interconnexion possibles entre des réseaux Firewire et Ethernet. Pour cela, il fallait commencer par comprendre les différentes contraintes liées à l'interconnexion de réseaux en général. Pour résoudre certaines de ces contraintes, des chercheurs ont proposé des méthodes d'interconnexion qui sont basées sur les services que les protocoles associés aux réseaux fournissent. Une étude des deux protocoles Firewire et Ethernet a aussi fait ressortir la multitude de services offerts par Firewire et qu'Ethernet ne peut pas fournir tout seul. Selon le type de données à transporter d'un réseau à un autre (AV ou datagramme IPv4), un équipement d'interconnexion devrait supporter des protocoles tels que IPv4, TCP ou UDP.

Dans le cas d'une interconnexion pour le transport de datagrammes IPv4, une norme RFC2734 a déjà été définie. Une solution logicielle basée sur cette norme a été conçue et implémentée. Cette solution a été profilée sur un processeur ARM7TDMI afin d'analyser les performances de ce processeur embarqué pour des applications d'interconnexion. Les résultats de profilage ont montré que le débit maximal d'un pont déployé sur un ARM7 était de 42,5 Mbps. Ces opérations de profilage ont aussi été effectuées pour valider une architecture d'une plate-forme SoC qui avait été proposée par des étudiants du GRM (Groupe de Recherche Microélectronique). La comparaison entre la solution logicielle et l'architecture SoC a fait ressortir la nécessité du parallélisme entre les composantes d'une architecture SoC par exemple.

La plus importante contribution de ce mémoire est la conception d'une passerelle pour le transport de données AV de plusieurs équipements Firewire vers un réseau Ethernet. Une passerelle a été spécifiée, conçue, implémentée et testée. Cette passerelle tient compte des différences d'adressage qui existent entre les deux réseaux. Les problèmes d'adressage ont été résolus en intégrant un nouveau pilote FireoverIP dans le *Kernel* de Linux, ce qui permet à la passerelle d'attribuer des adresses IP aux équipements Firewire. Donc, les clients du réseau Ethernet n'ont qu'à préciser l'adresse IP d'un équipement Firewire pour communiquer avec ce dernier. La méthode de complémentation de protocole a été appliquée pour pallier les différences de formats de données. Une entête virtuelle FireoverIP devra être rajoutée dans les paquets TCP ou UDP adressés à la passerelle. Les tests de la passerelle ont montré l'impact de la transmission des petits messages UDP sur les performances de la passerelle. L'assemblage de blocs AV dans de plus larges messages est nécessaire pour que la passerelle puisse atteindre des performances maximales qui atteignent jusqu'à 29,9 images/s pour un taux théorique de 30 images/s.

ABSTRACT

Public demand for audio-video AV digital applications started in the beginning of 90s. It also started with the wide use of AV compression algorithms like MPEG, and improved storage and processing speed capacity of computer equipments. Unfortunately, AV protocols at that time, like SCSI and IDE, could not provide services required by new AV applications. That is one reason why Apple proposed a new protocol: Firewire that also became IEEE 1394.

The purpose of this thesis is to find out and resolve different cases of Firewire-Ethernet networks interconnection. To achieve this goal, interconnection constraints have been pointed out. Some researchers have proposed methods to resolve these constraints based on services provided by networks protocols. A study about Firewire and Ethernet has shown that Ethernet does not provide services like reservation. In fact, Ethernet higher layers protocols, like TCP, UDP, IP should be supported by any interconnection equipment. This study revealed also 2 cases of interconnection. The first one is based on IPv4 datagrams transport, and the second one relates to AV data transport.

A standard : RFC 2734 (Johansson, 1999), has already been proposed for IPv4 datagrams transport over Firewire network. That lead us to design and implement a software protocols converter based on this RFC. This software has been executed on a ARM7TDMI processor to evaluate its performances for interconnection applications, and to validate a SoC architecture. This architecture was designed by students of the GRM (Microelectronics Research Group). Comparison between the software profiling results and the SoC architecture performances revealed the main necessity of architecture components parallelism.

The main contribution of this thesis is a flexible gateway for AV transport between multiple Firewire equipments and an Ethernet network. This gateway resolved networks addressing issues by adding a new driver in a Linux *Kernel* called FireoverIP. This driver

permits to allocate an IPv4 address to each Firewire equipment. Then, when Ethernet clients want to communicate with a Firewire equipment, they just have to put the equipment's IPv4 address on packets going to the gateway. Differences in data structures were resolved with the protocol complementation method. A virtual layer has been added upon TCP or UDP messages sent to the gateway. The gateway simulations show importance of incoming AV blocks assembly before their transmission in UDP messages. Without concatenation, the gateway cannot send all the received Firewire AV packets. Now with a good number of assembled AV blocks in UDP messages, the gateway can send 29.9 frames/s, out of a possible 30 frames/s limit in the implemented prototype.

TABLE DES MATIÈRES

LISTES DES FIGURES

LISTE DES TABLEAUX

SIGLES ET ABRÉVIATIONS

ARP	: Address Resolution Protocol
AV	: Audio / Vidéo
CSR	: Control and Status Registers
FEC	: Forward Error Correction
LAN	: Local Area Network
MAC	: Medium Access Control
MPEG	: Moving Picture Experts Group
MTU	: Medium Unit
OSI:	: Open Systems Interconnection
SoC	: System on Chip
USB	: Universal Serial Bus
WAN	: Wide Area Network

LISTE DES ANNEXES

CHAPITRE 1

INTRODUCTION

L'interconnexion de réseaux est un problème à résoudre à chaque fois qu'un nouveau protocole voit le jour, étant donné que les utilisateurs veulent profiter des services offerts par les nouveaux protocoles tout en utilisant leurs anciens équipements ou leurs applications courantes. Dans le monde des réseaux, un protocole se caractérise entre autres par sa simplicité de fonctionnement, les débits qu'il permet d'atteindre et la diversité des applications ou des services qu'il peut supporter. Au début des années 1990, une demande s'est créée pour les applications audio/vidéo (AV) afin de palier les faiblesses des services fournis par les protocoles de l'époque (par exemple : SCSI-Small Computer System Interface). C'est dans ce contexte que la compagnie Apple a sorti le protocole Firewire, qui fut ensuite normalisé sous le nom de IEEE 1394. Les avantages de ce protocole sont tels que les chercheurs l'ont étudié afin de voir comment l'exploiter pour le transport de datagrammes IPv4, entre autres.

Dans le cadre de ce mémoire, des techniques d'interconnexion entre des réseaux Firewire et Ethernet seront étudiées afin de faire ressortir tous les problèmes à résoudre durant l'interconnexion de ces réseaux et de proposer des solutions à ces derniers. Ethernet a été sélectionné car c'est le protocole le plus populaire dans les réseaux locaux ou LAN. Dans la suite de ce chapitre, la notion d'interconnexion de réseaux sera définie, ensuite les éléments de la problématique du sujet de recherche seront explicités, suivis des objectifs de cette recherche. Et enfin, le plan du mémoire sera détaillé.

1.1 Définition

L'interconnexion de réseaux est un problème très large qui dépend des réseaux faisant l'objet de l'interconnexion. Un réseau se caractérise par les empilements de protocoles qu'on y retrouve, entre autres. Cependant, un élément fondamental caractérise un réseau : le protocole de bas niveau qui transporte les données. C'est la raison pour laquelle, il y a

plusieurs niveaux d'interconnexion. Généralement, on retrouve plusieurs appellations pour nommer un équipement d'interconnexion de réseaux. Par exemple, un pont est utilisé pour interconnecter des protocoles de bas niveau, un « routeur » pour résoudre des différences d'adressage, une passerelle pour des protocoles de haut niveau. La passerelle traite généralement un empilement de protocoles, alors que le pont gère une paire de protocoles différents. La complexité des traitements à effectuer pour l'interconnexion est très reliée à la classe des protocoles considérés.

Les systèmes embarqués sont des appareils électroniques et informatiques spécialisés pour des tâches bien spécifiques et qui sont intégrés dans un équipement doté d'une autre fonctionnalité. Par exemple, les systèmes embarqués se retrouvent dans les ascenseurs, routeurs Internet, les consoles de jeu, les caméras, etc. Ces systèmes ont évolué de manière fulgurante en relation avec la disponibilité de SoC. Cette méthode est basée sur la réutilisation de modules (IP) et sur le co-design. Cette méthode profite de la capacité en termes de taille et vitesse des nouvelles technologies, pour intégrer un maximum de modules matériels ou logiciels sur une même puce.

1.2 Éléments de la problématique

Les premiers LAN connectaient principalement plusieurs PC entre eux ou des équipements tels qu'une imprimante ou un serveur de fichiers. De plus, ces réseaux étaient localisés dans des bureaux. À l'heure actuelle, le marché des particuliers est en pleine expansion. Ces derniers veulent recevoir de la vidéo, de la musique, leur courrier électronique de la toile. De plus, les distributeurs doivent offrir tous ces services à distance. Les produits électroménagers sont ou seront tous connectés sur un même réseau local, comme le montre la Figure 1-1. Tous les appareils pourront donc communiquer entre eux et être commandés de l'extérieur via Internet.

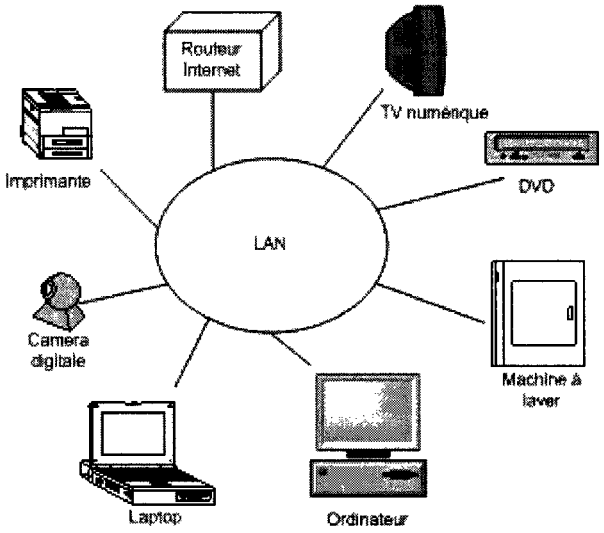

Figure 1-1. LAN maison du futur

Il est à remarquer que ces équipements sont très différents par rapport aux services qu'ils fournissent et à leur fonctionnement. Par exemple, les TV numériques requièrent une très grande bande passante du réseau afin de respecter le transfert de flots vidéo (Boutaba et al., 2002). Par contre, les appareils tels qu'une machine à laver demandent peu de bande passante et transmettent principalement des données de contrôle afin d'exécuter une commande. Le ou les PC se trouvant dans le LAN peuvent avoir, entre autres, un rôle de gardien (Firewall) et de routeur afin de contrôler et de retransmettre les transactions venant d'Internet ou y allant.

Le premier problème qui se pose, concerne la multitude d'applications qui sont exécutées par ces différents équipements. Le second problème est induit par le premier. En effet, ces applications, pour exécuter leurs fonctionnalités, font appel à des protocoles standardisés ou privés. Et c'est cette multiplicité de protocoles qui rend difficile sinon impossible dans certains cas, l'interconnexion de ces différents équipements. Par exemple, les équipements AV standards tels que les caméras numériques ou appareils photos fonctionnent exclusivement avec les protocoles Firewire et USB, alors que les PC

pour s'échanger de l'information utilisent Ethernet et Token Ring afin de transmettre des données issues des protocoles TCP-UDP/IP.

USB 1.0 a connu un grand succès jusqu'au jour où les applications vidéo haut de gamme sont entrées en jeu dans le milieu de l'informatique. C'est à ce moment que les limites de USB sont apparues évidentes. En effet, la vitesse maximale de USB 1.0 qui s'élève à 12 Mbps, est insuffisante pour transporter des flots vidéo qui requièrent des vitesses allant jusqu'à 70 Mbps pour les WebCam, par exemple. Des fabricants tels que IBM, Compaq et d'autres, essayent d'améliorer le protocole. Ainsi en avril 2000, une nouvelle version de USB, le USB 2.0 a été mise sur le marché. Cette version est 40 fois plus rapide que la précédente en atteignant la vitesse nominale de 480 Mbps. Les spécifications du protocole ont été définies depuis. Cependant, USB 2.0 n'est pas encore aussi répandu dans les équipements multimédias. Il faut rappeler que la première version avait pris 3 ans pour se stabiliser et être introduite sur le marché.

Avant l'entrée sur le marché du protocole USB, un autre protocole beaucoup plus performant était déployé dans les laboratoires de Apple : le Firewire. Ce protocole a été normalisé sous le nom IEEE 1394. Ce protocole offrait et offre jusqu'à maintenant des avantages par rapport à USB. Le premier avantage concerne ses vitesses de transmission qui ont commencé à 100 Mbps pour atteindre 800 Mbps en 2002. Le second avantage, non des moindres, est le mode de fonctionnement point à point (peer-to-peer) du protocole. Ce mode permet à deux équipements Firewire de communiquer sans l'intervention d'un ordinateur, ce que USB n'offre pas pour l'instant.

La firme 3d Solutions (2001) différencie les périphériques comme suit : Firewire pour les appareils AV haut de gamme comme les magnétoscopes ou caméras numériques et USB pour les disques durs externes ou encore les CD et DVD.

Dans le milieu des LAN, le protocole le plus populaire est IEEE 802.3 ou Ethernet. Ce protocole, après 20 ans, reste le plus exploité dans les bureaux, les universités ou les maisons. Ce succès s'explique par sa simplicité de déploiement, son coût peu élevé et enfin ses vitesses de transmission qui ont atteint 10 Gbps en 2002. Ethernet est un

protocole qui va de pair avec IPv4 généralement. IPv4 est le protocole le plus utilisé pour le transport des données sur Internet, entre autres sur les LAN.

Ces deux types de protocoles se différencient principalement par le type de données qu'ils transportent, ce qui accentue la différence des services qu'ils offrent. Le transport de données AV nécessite une qualité de service qu'Ethernet ne pourra jamais fournir à cause de ses modes de fonctionnement. C'est la raison pour laquelle, un paquet Ethernet véhicule une multitude d'empilements de protocoles afin de fournir des services plus complexes. Durant la résolution du problème d'interconnexion Firewire-Ethernet, il faudra donc tenir compte de certains empilements de protocoles pour effectuer le transport de données AV. Des travaux ont été effectués pour transmettre des données AV d'un équipement Firewire vers un autre équipement spécifique. Cependant, pour l'instant, aucune technique de communication entre plusieurs équipements AV Firewire avec plusieurs postes connectés sur un réseau Ethernet n'a été proposée.

1.3 Objectifs de recherche

Ce mémoire s'inscrit dans le cadre du développement d'une plate-forme SoC qui a été initiée en 1999 par M. Jean-Marc Tremblay. Le projet initial visait la conversion de protocoles dans un sens général par un système embarqué. Le but de cette recherche est de réaliser des preuves de concept sur lesquelles se baser afin de concevoir et de valider des architectures de plate-forme SoC spécialisées dans la conversion Firewire-Ethernet. Pour cela, il faudra :

- Analyser tous les cas de figure s'appliquant à l'interconnexion Firewire-Ethernet;
- Concevoir et développer des applications logicielles de conversion selon les cas de figure qui ressortiront de l'analyse mentionnée précédemment;
- Analyser les performances de ces applications logicielles dont les résultats seront utilisés durant la conception des architectures de plate-formes SoC proposé par les membres du groupe de recherche.

1.4 Plan du mémoire

Le chapitre 1 de ce mémoire débutera par une brève présentation des contraintes liées à l'interconnexion de réseaux ainsi que certaines méthodes d'interconnexion, suivie d'une description des protocoles Firewire et Ethernet. Durant notre revue de littérature, deux cas de figure d'interconnexion Firewire-Ethernet ont été notées : un pont pour le transport de datagrammes IPv4 entre ces deux types de réseaux et une passerelle pour le transport de données AV. Ces deux cas de figure seront explicités en insistant sur les travaux antérieurs qui ont déjà été développés dans ces domaines. Dans le chapitre 3, tous les travaux concernant la conception et le développement d'une application logicielle pour l'interconnexion Firewire-Ethernet pour le transport de datagammes IPv4 seront présentés. Le chapitre 4 regroupe tous les travaux qui ont été effectués sur le transport de données AV entre un réseau Firewire et un réseau Ethernet. Le dernier chapitre résumera tous les travaux qui ont été effectués sur l'interconnexion. Les limites de ces travaux seront aussi données, suivies d'indications sur les recherches futures.

CHAPITRE 2

INTERCONNEXION, PROTOCOLES ET APPLICATIONS

L'interconnexion de réseaux a toujours été un problème pour les chercheurs. Ces derniers ont déjà établi les contraintes qui complexifient l'interconnexion de réseaux. Dans un premier temps, nous allons parler de ces contraintes en insistant sur la notion de conversion générique.

Ensuite, les protocoles Firewire et Ethernet, seront présentés. Une comparaison de ces protocoles sera aussi effectuée afin de comprendre les difficultés qui seront rencontrées durant l'interconnexion de ces deux types de réseaux. Étant donné que Firewire est un protocole plus ou moins récent, nous allons parler de la place de Firewire dans le monde des réseaux locaux pour montrer l'avenir de ce dernier et son utilisation dans des LAN.

Et enfin, pour terminer, nous présenterons des travaux qui ont été effectués sur la transmission de données AV venant d'un équipement Firewire vers un réseau basé sur IPv4. Ces travaux permettront d'évaluer notre contribution par rapport à l'interconnexion de réseaux Firewire-Ethernet.

2.1 Problèmes à résoudre durant l'interconnexion de réseaux

L'interconnexion de réseaux est un problème qui a été abondamment étudié au début des années 80. Les scientifiques se sont vite rendus compte que la prolifération des protocoles induirait des problèmes d'interopérabilité entre les différents équipements. C'est la raison pour laquelle des modèles de référence ont été introduits, tels que le modèle OSI, afin de classifier les protocoles.

La classification du modèle OSI se base sur les services, à savoir les fonctions qui peuvent être offertes par un protocole. Svobodova et al. (1990) définissent l'hétérogénéité des réseaux selon trois niveaux qui dépendent :

- des éléments matériels ou logiciels qui interviennent dans chacun des réseaux ;
- du lien physique et des protocoles de bas niveau ;
- des protocoles de haut niveau ou de type application.

Le modèle OSI permet de résoudre une partie des problèmes d'hétérogénéité dus aux deux premiers niveaux. Le dernier est le plus complexe à résoudre, puisqu'il dépend des utilisateurs des réseaux. Une définition de la classification du modèle OSI peut être consultée à l'Annexe A.

Le modèle OSI a été proposé après l'introduction de la plupart des protocoles qu'on retrouve à l'heure actuelle dans les réseaux. De plus, la variété des services fournis par les protocoles accentue l'hétérogénéité des réseaux. Par exemple, la couche transport du modèle comporte 5 classes et la couche réseau a un mode non-orienté connexion (CLNS) et un mode orienté connexion (CON). L'interconnexion de réseaux se complexifie quand les empilements de protocoles de chacun des réseaux ne respectent pas le modèle OSI.

Prenons le cas d'un convertisseur générique, ce dernier doit pouvoir recevoir n'importe quel paquet de type X, comme illustré par la Figure 2-1, et convertir ce paquet en un autre paquet de type Y. Il faut se rappeler qu'un paquet peut contenir des données brutes, des données de contrôle ou les deux.

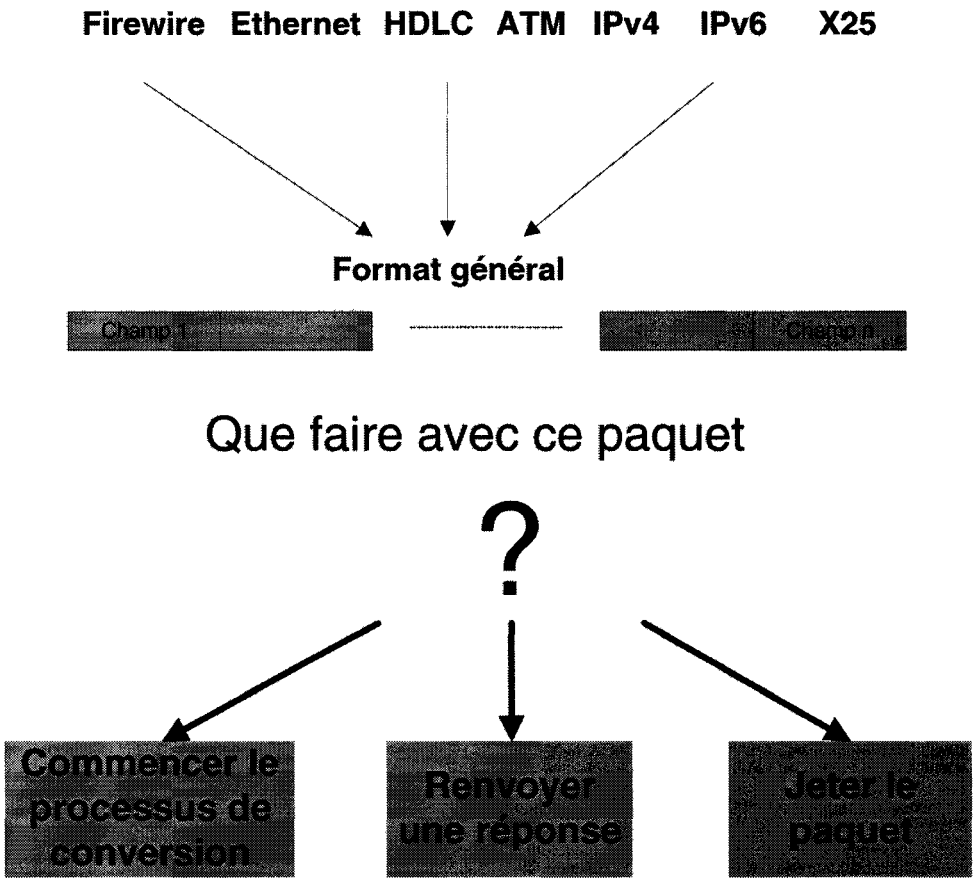

Figure 2-1. Problématique de la conversion de protocoles

Le convertisseur doit d'abord identifier le type de paquet qu'il vient de recevoir, ensuite il doit effectuer toutes les tâches directes de conversion, mais aussi les tâches de contrôle qui sont vitales pour les équipements sources ou destination. Dans la suite de cette sous-section, les contraintes à résoudre durant la conversion seront données. Ainsi, le lecteur pourra comprendre pourquoi la conversion générique est compliquée à effectuer ou sinon impossible dans certains cas.

2.1.1 Format générique des données– Requis syntaxiques

La première différence entre les protocoles concerne la diversité de leurs formats de données (Shu et Liu, 1990). Chaque protocole représente l'information à sa manière (Green, 1986; Probert et al., 1991). Lorsque l'on désire manipuler plusieurs protocoles, il est utile de définir un format général qui permettra de travailler indépendamment des

formats et qui sera réutilisable pour les futurs protocoles à traiter. Ce format générique doit aussi faciliter le traitement matériel des protocoles et la conception des algorithmes de conversion. Même si les protocoles ne sont pas classés sur la même couche ou s'ils n'offrent pas les mêmes services, le but d'un format générique est de stocker les paquets associés de manière régulière et transparente, ce qui permettra de prendre des décisions adéquates.

2.1.2 Représentation des requis sémantiques

La conversion de protocoles implique forcément la connaissance du fonctionnement des protocoles source et destination (Green, 1986 ; Chang et Liu, 1990 ; Shu et Liu, 1990). Ce sont leurs modes d'opération qui permettront principalement de définir la faisabilité des conversions. Vu les classes de services offerts par les différents protocoles et les contraintes liées à ces différentes classes, il serait utile de définir un formalisme qui permettra de comprendre le fonctionnement des protocoles et qui facilitera le processus de conversion de protocoles. Ce formalisme doit être simple et tenir compte des différents modes de fonctionnement afin de pouvoir gérer le plus de protocoles possibles. Par exemple, il faut pouvoir formaliser des services tels que la fragmentation, la gestion de congestion, etc.

2.1.3 Interprétation du contenu des paquets

Les protocoles rajoutent de l'information dans le paquet initial afin d'exécuter leurs tâches correctement. Dans certains cas, le paquet contiendra seulement de l'information propre au protocole. Étant donné que le mode de fonctionnement est déjà connu, il s'agit principalement de fragmenter le paquet selon les requis syntaxiques et de définir le type de paquet par rapport aux modes de fonctionnement du protocole source principalement.

2.1.4 Décisions à prendre

Après avoir interprété le contenu du paquet, une décision doit être prise (Shu et Liu, 1990). Le protocole source attend-t-il une réponse ou un accusé de réception ? Qui doit les émettre : le convertisseur ou la destination ? La question qui se pose est la suivante :

quelle est la place du convertisseur dans le réseau ? Le convertisseur, peut-il être transparent dans le réseau ? Jusqu'à quel point le convertisseur peut-il prendre des décisions ? Une intervention humaine est-elle nécessaire dans certains cas ?

2.1.5 Traduction des adresses

La validité des adresses contenues dans le paquet converti est primordiale. Le bon format de données n'assure pas le bon fonctionnement de l'algorithme de conversion de protocoles. Le contenant, i.e. le format de données, est très important, mais le contenu est aussi ou sinon plus important. Le format des adresses est différent mais la définition des adresses diffère aussi. Les types d'adresses, universelles ou locales, influencent la validité des adresses dans le temps. En effet, les adresses universelles sont généralement fixes, alors que les adresses locales sont associées à une connexion entre les équipements du réseau. Sunshine (1990) présente très bien le problème, à savoir qu'il faut pouvoir identifier les équipements de chaque réseau et être capable d'effectuer la mise à jour des tables d'adresses en tenant compte des différents types d'adressage.

Comme vous pouvez le voir, les contraintes sont multiples et très difficiles à résoudre. Généralement, étant donné que les protocoles se ressemblent très peu, la conversion directe est quasi impossible pour la majorité des paires de protocoles. Ainsi, diverses solutions ont été proposées afin d'interconnecter des réseaux hétérogènes.

2.2 Méthodes d'interconnexion

Comme nous l'avons dit précédemment, la conversion est possible dans certains cas. Cependant, il existe plusieurs méthodes d'interconnexion qui se basent sur les services offerts par les différents protocoles. L'obstacle majeur de la conversion est la diversité des services offerts par les protocoles. Il faut rappeler qu'un service est une fonctionnalité offerte par un protocole (Tohio, 2003). Par exemple, le transport des données, le routage, la liaison de données sont considérés comme des services. Les méthodes d'interconnexion seront présentées sous forme d'illustration en se basant sur des exemples concrets.

2.2.1 Conversion de services

Cette solution est appliquée quand les protocoles présents sont très différents au niveau de leur sémantique et leur syntaxe, ils offrent les mêmes services mais de manières différentes. Dans ce cas, le convertisseur ne s'occupe pas des protocoles en tant que tel, comme le montre la Figure 2-2, mais des services mis en jeu durant l'interconnexion (Bochman et al., 1990).

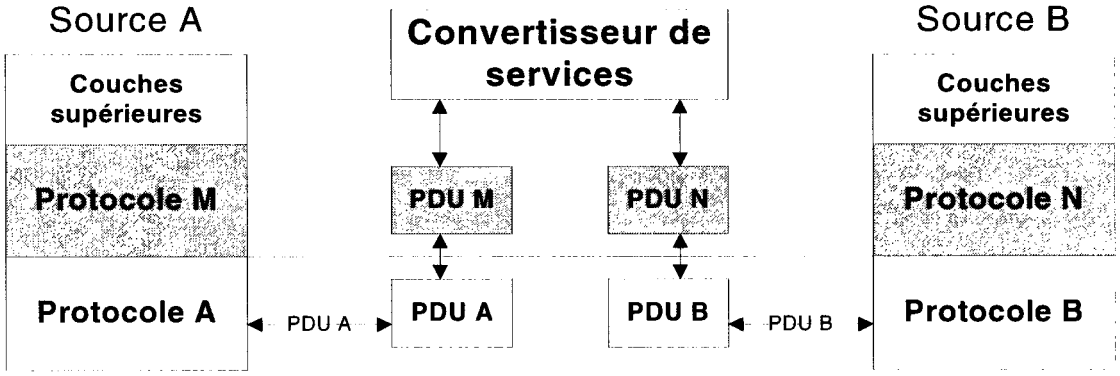

Figure 2-2. Cas d'une conversion de services

Deux types de conversion de services sont proposés (Tohio, 2003):

- La concaténation de services : elle est appliquée quand les protocoles offrent exactement les mêmes services ;

- L'adaptation d'interface de services : cette solution est adoptée quand les protocoles n'offrent pas exactement les mêmes services, les applications qui les utilisent requérant seulement des services qui sont semblables (Bochmann et al., 1990).

2.2.2 Encapsulation

Cette approche est classée dans les méthodes d'interconnexion, mais elle ressemble davantage à une méthode de retransmission de paquets. Cette méthode, illustrée par la Figure 2-3, permet d'interconnecter deux réseaux X et Y, fonctionnant avec le même protocole A, mais reliés par un troisième réseau W, utilisant un protocole B différent. Cette solution nécessite deux équipements d'interconnexion, chacun se trouvant à la périphérie d'un des réseaux X et Y. Ces équipements d'interconnexion sont chargés d'effectuer les tâches d'encapsulation et de décapsulation des paquets et de leur retransmission vers le réseau A ou B, selon le sens de l'interconnexion.

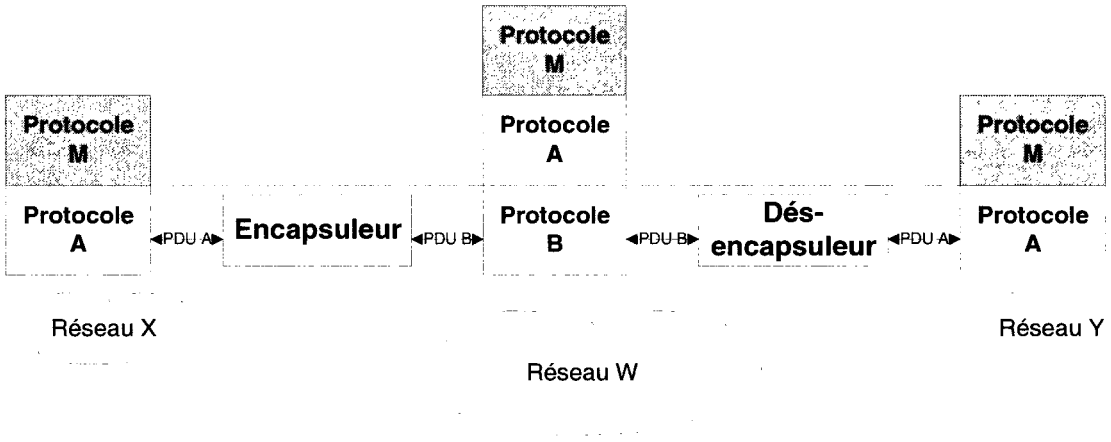

Figure 2-3. Cas d'une encapsulation de protocoles

Généralement, le protocole B offre de meilleurs services que l'autre. Un exemple typique est l'encapsulation de paquet IPv4 dans un paquet X25 (Svobodova, 1990). X25 est un protocole utilisé dans les réseaux WAN. Alors qu'il est classé sur la couche 3 du modèle OSI, au même titre que IPv4, ses services de contrôle sont très garnis, avec en plus deux méthodes d'adressage : par canaux virtuels ou permanents. IPv4 n'offre presqu'aucun des

mécanismes de contrôle offerts par X25. C'est la raison pour laquelle l'encapsulation est utilisée dans le cas de ces deux protocoles, d'autant plus que IPv4 nécessite des services semblables à ceux de X25 dans les WAN.

2.2.3 Complémentation

La complémentation est une solution utilisée quand les services offerts par ces protocoles diffèrent beaucoup. Quand l'un des protocoles offrent plus de services que l'autre, alors Chang et Liu (1990) proposent l'ajout d'une couche virtuelle afin d'interconnecter des réseaux hétérogènes. Cette couche virtuelle sera insérée entre deux couches de protocoles s'il le faut, comme le montre la Figure 2-4. En général, elle est insérée par dessus le protocole le plus pauvre en termes de services à offrir (Bochmann et al., 1990). Cette couche virtuelle comportera des champs qui permettent de rajouter les services manquants. En fait, le convertisseur, en recevant un paquet de type A, consulte l'entête de la couche virtuelle pour retransmettre la charge utile vers le réseau Y. Il se chargera de constituer l'entête de la couche virtuelle lorsqu'il recevra un paquet de type Y.

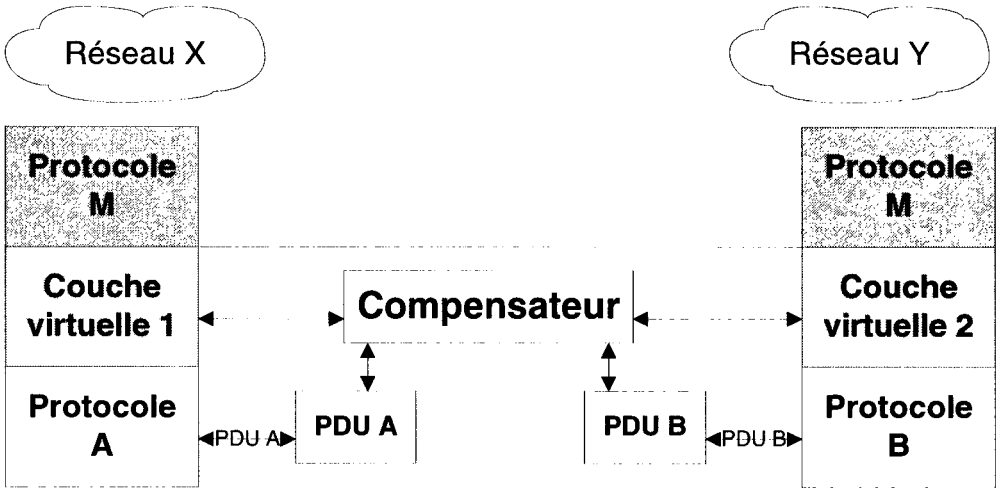

Figure 2-4. Cas d'une complémentation de protocoles

Cette méthode offre une meilleure conversion des services. Cependant, elle n'est pas transparente. Elle nécessite de nombreux changements au niveau des entités matérielles ou logicielles de chaque réseau. En effet, en plus du convertisseur situé entre les deux

réseaux, leurs équipements doivent implémenter une couche virtuelle. Ainsi, des modifications doivent être effectuées sur chaque machine, ce qui peut être très coûteux pour une entreprise, par exemple. Cette méthode a été appliquée pour l'interconnexion de réseaux ATM et IPv4.

2.2.4 Conversion de protocoles

La conversion de protocoles est appliquée lorsque les protocoles sont semblables à quelques différences près. Cette solution permet de traduire les messages reçus (Calvert et Lam, 1989) et elle émule le comportement de chaque protocole. Donc, lorsqu'un paquet de type A est reçu, le convertisseur effectuera toutes les opérations nécessaires à la retransmission de la charge utile dans un paquet B, il renverra tous les paquets de contrôle nécessaires au bon fonctionnement des réseaux. Donc, l'utilisateur de type A aura l'impression de communiquer directement avec l'utilisateur de type B, comme le montre la Figure 2-5.

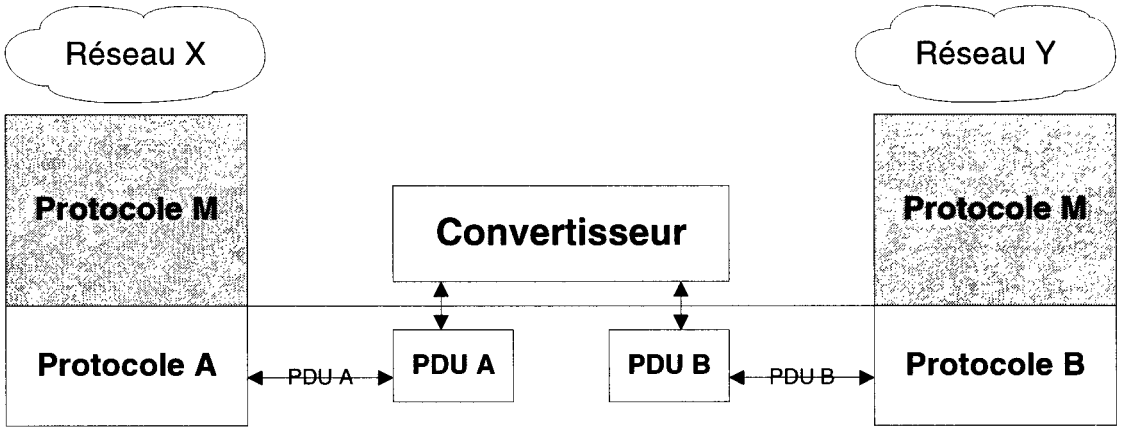

Figure 2-5. Cas d'une conversion de protocoles

Un convertisseur est un élément logiciel ou matériel se trouvant à la frontière des deux réseaux qui effectuent des traitements transparents sur les paquets transitant entre les deux réseaux hétérogènes. Ces traitements sont transparents vis-à-vis des utilisateurs des deux réseaux, aucune modification ne doit être effectuée sur les postes des usagers pour faciliter l'interconnexion.

Un exemple de conversion est la conversion Firewire - Ethernet pour le transport de datagrammes IPv4. Ces deux protocoles se ressemblent assez pour que la conversion soit possible sans grande difficulté, comme nous pourrons le voir dans la suite du mémoire.

2.3 Description des protocoles

Dans cette section, nous allons présenter brièvement les deux principaux protocoles traités dans ce mémoire : Firewire et Ethernet. Ensuite, une comparaison entre Firewire et USB sera donnée. Cette comparaison permettra de faire ressortir les avantages de Firewire. Et enfin, une comparaison entre Firewire et Ethernet sera aussi établie en insistant sur les différences entre leurs modes d'adressage et les services que ces deux protocoles offrent.

2.3.1 Firewire

Firewire est un terme introduit par la compagnie Apple, la première à avoir introduit le protocole. Cependant, le protocole est plus connu sous le nom de IEEE 1394 avec des variantes 1394a et 1394b (la plus récente).

La différence entre ces deux variantes concerne principalement la vitesse de transmission des données. La norme IEEE 1394a se limite à une vitesse de 400 Mbps avec une taille maximale d'un paquet égale à 2048 octets, tandis que IEEE 1394b peut atteindre une vitesse de 3 200 Mbps avec une taille maximale égale à 32 768 octets (*IEEE Std. 1394, 2002*).

Un réseau Firewire supporte une multitude d'appareils, comme le montre la Figure 2-6. Ces équipements se diversifient par leur fonctionnement et leur application. Cependant, un réseau Firewire est optimisé pour des environnements supportant de la vidéo et de l'audio.

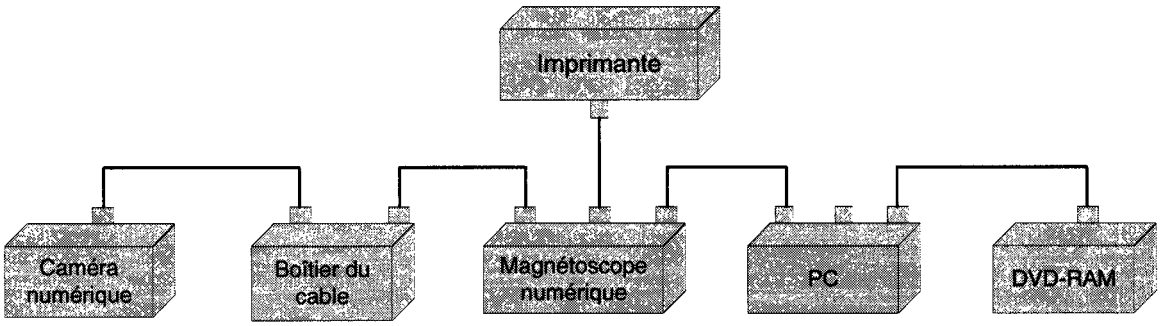

Figure 2-6. Topologie d'un réseau Firewire

Firewire permet aux équipements de communiquer en mode point-à-point. Ce type de communication implique que les appareils aient au moins les mêmes capacités et qu'une session de communication soit établie avant chaque session. Firewire implémente une architecture Maître/Esclave. Ce protocole est très complexe, le format de l'entête varie selon le mode de transmission des données qui peut être asynchrone ou isochrone.

Les paquets isochrones sont des paquets émis vers tous les équipements du réseau Firewire. Un seul équipement peut émettre sur un canal donné et tous les autres équipements peuvent recevoir et utiliser les données transmises sur ce canal isochrone. Par exemple, une caméra émettra son flot vidéo sur un canal qui a été défini auparavant, et tous les équipements qui sont branchés sur le réseau tels qu'un PC ou une télévision, auront le droit de capter les images transmises et de les afficher.

Selon le standard IEEE1394-1995, le mode asynchrone de Firewire comporte une dizaine de types de paquets. Ces types dépendent principalement de la transaction qui s'effectue à savoir une écriture ou une lecture. Les transactions asynchrones permettent généralement d'accéder entre autres au CSR (Control and Status Register) du nœud Firewire adressé. Pour mieux comprendre la variété de types de paquet, il est utile de présenter le système d'adressage de Firewire qui est illustré par la Figure 2-7.

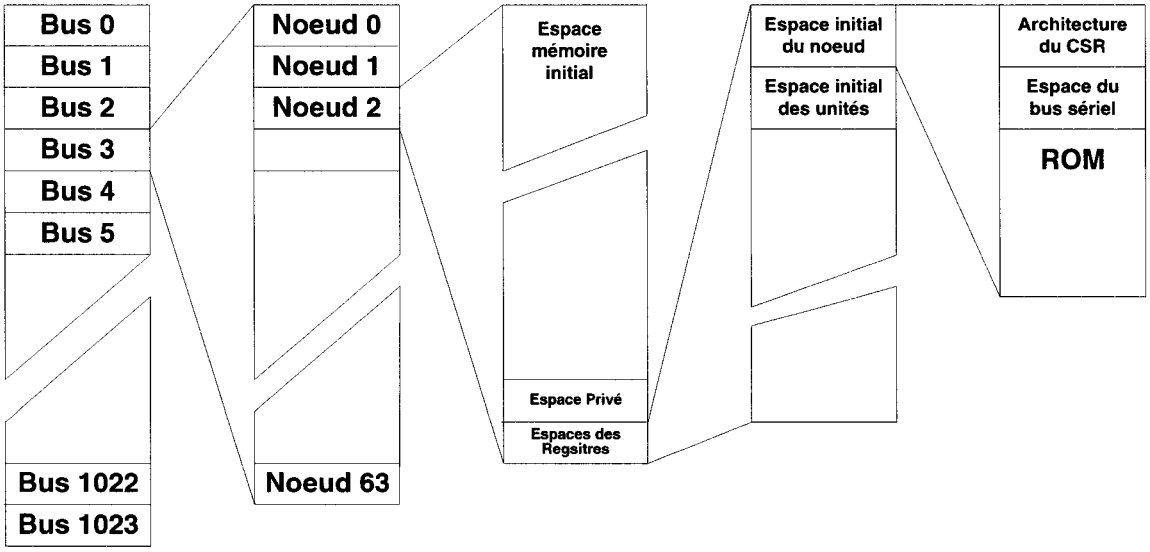

Figure 2-7. Système d'adressage de Firewire (*IEEE Std 1394-1995, 1995*)

C'est un système d'adressage très hiérarchisé basé sur :

- le nombre de bus du réseau ;
- le nombre de nœuds branchés sur chaque bus, où chaque nœud comportant entre autres un espace réservé aux registres ;
- l'espace de registres qui comporte un espace initial contenant : le CSR, les données du bus et la ROM.

Tous les équipements Firewire ont un CSR qui respecte en majorité la norme IEEE 1212. Cette norme propose une organisation des propriétés associées à chaque équipement. Par exemple, on retrouvera les registres suivants : identifiant de l'équipement (node ID), reset (Reset start), etc. La ROM renferme des informations propres à l'équipement telles que son numéro de série, le fabricant de la carte, etc. Pour consulter ou régler certaines propriétés d'équipements Firewire tels qu'une caméra, les usagers ont seulement à effectuer des lectures et des écritures dans les registres associés à ces propriétés. L'utilisateur précisera l'adresse du registre dans le champ « destination offset » du paquet asynchrone.

On retrouvera un gestionnaire de bus et un gestionnaire de ressources isochrones dans un réseau Firewire. Ce ne sont pas tous les équipements Firewire qui peuvent remplir ces fonctions. Le gestionnaire de bus est la racine, c'est le nœud qui effectue l'arbitrage durant l'attribution des adresses physiques entre autres. Le gestionnaire de ressources isochrones est le nœud responsable des réservations de bande passante et de canaux, entre autres.

2.3.2 Ethernet

Ethernet est un des protocoles les plus populaires dans le monde des réseaux locaux (LAN). Contrairement à Firewire, les données sont véhiculées sur le même bus, comme le montre la Figure 2-8. C'est la raison pour laquelle Ethernet opère selon le mécanisme : CSMA/CD (Carrier Sense Multiple Access/Collision Detection), afin de détecter les collisions dues aux envois simultanés de données sur le bus par différents équipements.

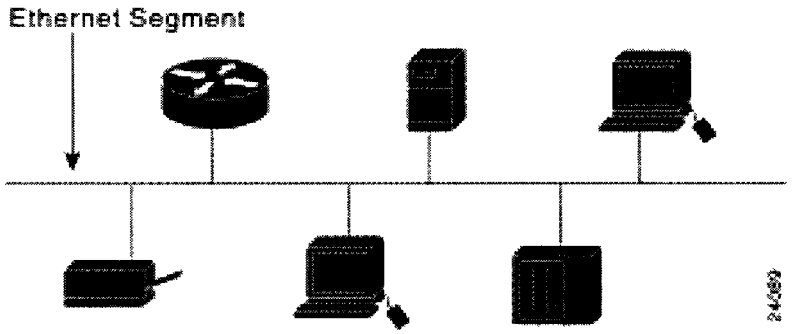

Figure 2-8. Topologie d'un réseau Ethernet

Il faut rappeler qu'une des versions d'IEEE 802.3 est aussi communément appelée Ethernet giga bits. En effet, ce protocole permet d'atteindre des vitesses qui dépassent 1 giga bits par seconde et qui peuvent atteindre 10 giga bits par seconde. Ces vitesses permettent évidemment de transmettre de la vidéo ou de l'audio sur ce type de réseau.

IEEE 802.3 supporte deux sous-couches dans un réseau local : MAC (Medium Access Control) et LLC (Logical Link Control). Cette dernière sous-couche permet entre autres

de définir les caractéristiques du protocole qui a fait appel aux services d'Ethernet et du protocole qui va utiliser les données à transmettre. La sous-couche LLC encapsule un paquet de type IEEE 802.2. Par exemple, on retrouvera des paquets de type SNAP (Sub Network Access Protocol). Dans certains réseaux, ce type de paquet est le plus utilisé pour envoyer des paquets IP sur un réseau local.

Ethernet est un protocole qui offre peu de services par rapport à Firewire. C'est une des raisons pour lesquelles on retrouve autant de protocoles par-dessus Ethernet, comme le montre la Figure 2-9.

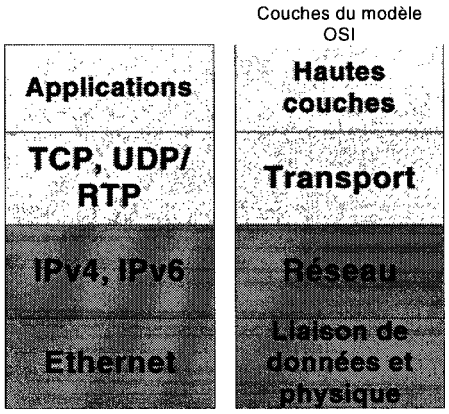

Figure 2-9. Empilements de protocoles au-dessus de Ethernet

Étant donné que Ethernet est un protocole assez connu, nous allons nous arrêter ici. Pour plus de renseignements, le lecteur pourra se procurer la norme IEEE-802.3.

2.3.3 Comparaison des protocoles

Dans la suite de cette sous-section, nous allons comparer Firewire à USB, ensuite à Ethernet. Ces comparaisons permettront de mieux faire ressortir les atouts de Firewire.

2.3.3.1 Comparaison entre Firewire et USB

Il serait utile de situer IEEE 1394 par rapport à USB, le protocole le plus populaire qu'on retrouve dans le monde des PC. Ces deux protocoles offrent à peu près les mêmes services, comme le montre le Tableau 2-1.

Ce tableau de comparaison est basé sur USB 2.0 qui est la version la plus récente. Les premières cartes USB 2.0 sont sorties en juillet 2001. Cette version est celle qui peut vraiment être comparée à Firewire, car les performances de la version USB 1.1 sont nettement inférieures à celles de Firewire. USB 2.0 est le grand concurrent de Firewire à l'heure actuelle. Le succès de USB par rapport à Firewire est dû à des raisons économiques. En effet, Intel a misé sur USB depuis juin 2002 avec l'intégration de chipset USB 2.0 sur des cartes mères Intel (référence). On retrouve généralement des ports Firewire sur les équipements Sony et Apple, et sur certaines machines Compaq (3D solutions).

Tableau 2-1. Comparaison Firewire-USB

	IEEE 1394	USB 2.0	Commentaires
Allocation de bande passante	Oui	Oui	-
Allocation de canaux	Oui	Oui	-
Mode asynchrone	Oui	Oui	-
Mode isochrone	Oui	Oui	-
Distance du câble	4,5 m	5 m	La spécification de USB 2.0 définit une longueur de 5 m, mais en pratique, elle se limite à 3 m.
Vitesse	800 Mbps	480 Mbps	-
Maître	Tous	PC	Tout équipement Firewire qui offre les services d'un maître peut être maître du bus, alors que dans un réseau USB seul un PC peut l'être.
CSR	Oui	Oui	-

2.3.3.2 Comparaison entre Firewire et Ethernet

Firewire et Ethernet sont très différents, comme le montre le Tableau 2-2. La technique d'adressage dynamique de Firewire constitue une différence majeure. Firewire définit les adresses des équipements durant la réinitialisation du bus, alors que les adresses MAC

d'Ethernet sont statiques. Un des avantages d'Ethernet concerne l'ajout et la suppression d'équipements dans le réseau. Ces événements impliquent une réinitialisation automatique du bus et une redistribution des adresses de chaque équipement dans le réseau Firewire.

Firewire propose l'implantation d'un CSR (Control and Status Registers). Ces registres contiennent toute l'information utile du réseau, allant de la bande passante disponible aux types de données vidéo transmises par l'équipement (s'il y a lieu). Dans un réseau Ethernet, ce sont les protocoles des couches supérieures qui fournissent ces informations.

Tableau 2-2. Comparaison des fonctionnalités de Firewire et Ethernet

	IEEE 1394	IEEE 802.3
Allocation bande passante	Oui	Non
Allocation de canaux	Oui	Non
Contrôle du délai	Non	Non
Gestion d'erreurs	Oui	Non
Mode asynchrone	Oui	Oui
Mode isochrone	Oui	Non
Réinitialisation du bus	Oui	Non
Adressage dynamique	Oui	Non
Distance du câble	4,5 m	100 m
CSR	Oui	Non

Un autre point non des moindres concerne la différence des débits atteignables par ces deux protocoles. Firewire version IEEE1394a atteint à l'heure actuelle des vitesses de 400 Mbps alors que celles LAN Ethernet (à la maison) dépassent à peine 100 Mbps. Cette différence pourrait causer la perte de données lors de l'interconnexion de ces réseaux.

La distance du câble est un des inconvénients majeurs de Firewire avec une longueur de 5 m. La version 1394b pallie ce problème avec une distance de 50 m pour des câbles en fibre optique plastique et de 100 m pour des câbles en fibre optique de verre.

Les différents modes de transmission de paquets chez Firewire poseront un problème durant la conversion de protocoles. En effet, les services offerts par ces différents types

de paquet ne sont pas fournis par Ethernet. Il faudra faire appel aux protocoles des couches supérieures d'Ethernet pour offrir les mêmes services.

Firewire offre un mécanisme de gestion d'erreurs pour les paquets asynchrones. Ce mécanisme permet de renvoyer un accusé après la réception d'un paquet et ensuite de renvoyer une réponse relative à l'exécution d'une tâche donnée telle qu'une écriture dans le CSR.

Dans la suite de cette section, nous allons discuter plus en détails des différences concernant les modes d'adressage de ces deux protocoles et sur les services qu'ils fournissent.

2.3.3.2.1 Mode d'adressage

Il existe plusieurs différences concernant les systèmes et méthodes d'adressage de ces deux protocoles. La première et non des moindres concerne la longueur des adresses. Une adresse MAC comporte 48 bits alors qu'une adresse physique Firewire est codée sur 64 bits. Ces 64 bits sont répartis comme suit : 10 bits pour identifier le bus, 6 bits pour définir un nœud du bus et 48 bits pour préciser un espace mémoire du CSR. L'adresse MAC est définie lors de la fabrication de la carte réseau Ethernet, alors que l'adresse physique Firewire est définie durant le processus d'auto-identification des équipements Firewire. Ce processus se déroule après le démarrage du réseau Firewire ou après l'ajout ou le retrait d'un équipement du réseau (Std IEEE 1394-1995, 1995).

La seconde différence concerne les méthodes d'adressage. Comme nous l'avons dit précédemment, Firewire opère selon deux modes de transmission, asynchrone ou isochrone et chacun de ces modes a sa propre méthode d'adressage. Le mode asynchrone fonctionne selon l'adresse physique alors que le mode isochrone utilise tout simplement un numéro de canal qui est codé sur 6 bits. Un seul équipement peut émettre sur un canal donné, les autres équipements peuvent seulement recevoir les paquets émis sur ce canal.

2.3.3.2.2 Services fournis

Après l'insertion d'équipements et une réinitialisation du bus, les équipements peuvent établir des canaux de communication, en réservant de la bande passante et en créant des canaux. Ethernet n'offre aucun de ces mécanismes de réservation de ressources.

Firewire permet de réserver de la bande passante sur le bus. La bande passante est codée sur 32 bits. Lorsqu'un usager veut réserver de la bande passante, il doit d'abord lire la valeur actuelle du registre, soustraire à cette valeur la quantité nécessaire, et enfin, le restant est écrit dans le registre contenu dans le CSR du gestionnaire de ressources isochrones.

Dans un réseau Ethernet, ce sont les couches supérieures qui fournissent des mécanismes de réservation de bande passante. Le protocole RSVP est celui qu'on retrouve dans les réseaux TCP/IP. Mauthe et al. (2001) définissent RSVP comme un protocole qui permet d'améliorer la qualité de service d'un réseau par le contrôle des requêtes allant vers les différents routeurs et enfin en établissant et tenant à jour l'état du réseau, ce qui facilite l'exécution du service. RSVP se retrouve très peu dans les réseaux locaux. Il est plutôt utilisé dans les WAN. Il faut préciser que la capacité de réserver de la bande passante est un critère de qualité de service, surtout quand il s'agit de transport de données audio/vidéo.

2.4 Firewire dans les LAN

Vu toutes les possibilités de Firewire, une demande s'est créée pour transporter des datragrammes IPv4 dans un réseau Firewire. Dans cette sous-section, nous allons parler de la présence de Firewire dans les LAN, en insistant sur les différentes solutions qui existent à l'heure actuelle. Puis, la norme sera présentée ; cette dernière a été proposée pour faciliter le transport de datagrammes IPv4 dans un réseau Firewire.

2.4.1 Utilisation de Firewire dans les LAN

Sur le marché, la compagnie Unibrain, propose un pilote Firenet qui permet de transporter des datagrammes IPv4 sous Firewire. Les tests de ce logiciel sont très concluants. Le #transport de datagrammes TCP/IP sous Firewire grâce à ce pilote est beaucoup plus efficace par rapport au transport sous Ethernet 100baseT (UnibrainS.A., 2002).

Les tests d'Unibrain (2002) étaient basés sur un échange de messages TCP/IP entre deux PC comportant des Pentium 4 (1,7 GHz, 512 RAM). En mode écriture, les tests montrent que la vitesse effective s'élève à 18 Mbps sous Firewire, à 7,6 Mbps en 100baseT et 23,4 Mbps avec du Ethernet 1 Giga bits / sesonde. Donc, le transport de datagrammes sous Firewire est deux fois plus rapide que celui de datagrammes sous 100baseT. Cependant, le réseau Ethernet 1 Gbps est plus rapide, mais cela se comprend étant donné que le réseau Firewire fonctionne à 400 Mbps. Ces résultats sont encourageants d'autant plus, qu'en fin 2002, les premières cartes Firewire à 800 Mbps ont été mises sur le marché. Donc, il faudrait s'attendre à obtenir de meilleures performances dans le futur.

Cependant, il y a un problème de taille : Firenet/Unibrain est basé sur un protocole privé. La méthode d'encapsulation n'est pas publique ; donc, pour avoir un LAN sous Firewire, il faut acheter des licences pour chaque PC, de plus, l'interconnexion avec un réseau Ethernet par exemple sera impossible. Les gestionnaires de LAN qui veulent passer sous Firewire devront changer les cartes Ethernet pour des cartes Firewire et ils devront se procurer des licences.

La solution d'Unibrain est efficace; cependant, elle met cette société en situation de monopole. En effet, seules les machines fonctionnant avec Firenet pourront s'échanger des messages TCP/IP, de plus les coûts de remplacement des équipements et des logiciels sont élevés.

Windows XP et Windows Me supportent le transport de messages TCP/IP sous Firewire en respectant le RFC 2734. Sous Linux, la communauté propose des pilotes du RFC qui effectuent aussi l'échange de messages TCP/IP sous Firewire.

2.4.2 RFC 2734

Pour transporter des datagrammes IPv4, il faut rajouter une couche virtuelle entre l'entête IEEE1394 et le datagramme (Santamaria R., 2000). La charge utile du paquet Firewire sera donc constituée d'une pseudo-entête et du datagramme. Le format du datagramme IPv4 (Postel et al., 1980) ne sera pas défini dans cette section, mais il est fourni à l'Annexe B. Les datagrammes IPv4 pourraient être fragmentés afin de respecter la limite maximale de la charge utile d'un paquet Firewire. Il faut préciser que cette limite dépend de la vitesse de transmission Firewire. Par exemple, à 200 Mbps, la limite est de 1024 octets alors qu'à 400 Mbps, elle est de 2048 octets. La pseudo-entête varie selon le type de fragments. On différencie les fragments selon leur position, à savoir est-ce le premier ou un de suivants?

Nous commencerons par présenter les différents champs de l'entête virtuelle, ce qui permettra de comprendre pourquoi le RFC a été proposé sous cette forme.

Tableau 2-3. Champs de l'entête virtuelle proposée par le RFC 2734

Champs	Taille (bits)	Commentaires
Lf	2	POSITION DU FRAGMENT
RSV	14	Réservé.
Ether_type (Ipv4)	16	Type du paquet transporté.
Dgl	16	Numéro d'identification du datagramme. Il apparaît dans les paquets appartenant à un datagramme fragmenté.
Datagramme Ipv4	-	La charge utile

Le premier champ : « lf » permet d'offrir un service qui n'est pas fourni par Firewire à savoir la fragmentation, ainsi que le champ « dgl ». La fragmentation est due au fait que les tailles maximales des charges utiles de ces protocoles sont différentes. Les gros datagrammes (taille : 65 536 octets) IPv4 ne pourront pas être contenus dans un paquet

IEEE1394. Il faut rappeler que la taille maximale de la charge utile d'un paquet 1394a opérant à 400 Mbps est de 2048 octets.

La seconde raison d'être de l'entête virtuelle est la suivante : dans les architectures basées sur IP, plusieurs types de paquets circulent dans les réseaux. Lorsqu'un paquet est reçu par les pilotes de bas niveau, ils doivent être capables de reconnaître rapidement le type du paquet pour l'envoyer au pilote associé. Firewire transporte des données, mais il ne donne aucun renseignement sur le type des données transmises, alors que ce type est primordial pour les pilotes, ces derniers doivent savoir si c'est un paquet IPv4 ou ARP par exemple. Pour résoudre ce problème, la couche virtuelle comporte le champ : « ether_type », qui précise le type du paquet.

Dans les réseaux IPv4 fonctionnant avec Ethernet, on retrouve des paquets ARP. Le RFC 2734 propose d'implémenter une version de ARP dans un réseau Firewire. ARP est un protocole qui résout l'adressage entre les adresses PHY_ID des cartes réseaux des équipements et l'adresse IP associé à ces derniers. Il faut rappeler que, dans un réseau IP, chaque équipement a une adresse IP unique. ARP est un protocole très simple. Chaque machine comporte une table ARP qui tient une correspondance entre chaque adresse IP et une adresse PHY_ID associée. Prenons un exemple simple pour expliquer le fonctionnement de ARP : une machine A ayant l'adresse IP X veut parler à une autre machine B ayant l'adresse IP Z. Malheureusement, elle ne trouve pas l'adresse PHY_ID de B. Dans ce cas, elle va envoyer une requête à toutes les machines se trouvant sur le réseau pour demander à celle qui à l'adresse IP Z de lui renvoyer son adresse PHY_ID. Dès que la machine A recevra la réponse de la machine B, elle mettra à jour sa table et elle pourra commencer à envoyer des paquets IPv4 à la machine B. Nous reparlerons plus en détail du fonctionnement de ARP dans la section 4.1.2.

Des études de performance ont été effectuées par Keville et Tompkin (2002) pour évaluer la performance de IPv4 sous Firewire (IP1394) par rapport à Ethernet sous Firewire (Eth1394). Eth1394 consiste à encapsuler des trames Ethernet dans un paquet Firewire. Les tests ont été effectués sous linux avec une carte OHCI/1394. En fait, le débit de

IP1394 pouvait même atteindre un maximum de 135,5 Mbps pour des paquets ayant une taille de 25000 octets, alors que Eth1394 atteignait une vitesse maximale de 100 Mbps pour une taille de 10000 octets. La seconde observation notée par Keville est l'effet de la fragmentation sur le débit des deux architectures. Il montre une irrégularité de la bande passante lorsque la taille du paquet est comprise entre 2000 et 3000 octets. Cette irrégularité prouve que l'algorithme de fragmentation commence à être efficace lorsque la taille du paquet dépasse la taille du MTU.

2.5 Requis de la vidéo

Le marché des appareils AV numériques a été stimulé par l'introduction de la formule cinéma maison et des DVD. En effet, la qualité de l'image et du son étant meilleure, les particuliers optent de plus en plus pour cette formule.

Le transport de vidéo demande beaucoup de ressources en termes de vitesse de transmission et de fiabilité des données afin d'obtenir la qualité de l'image souhaitée. Un des défis à l'heure actuelle est la transmission d'AV à travers le Web et plus généralement les applications en temps réel telles que: la vidéo-conférence, la téléphonie sur IP, TV et Radio (en direct), le vidéo surveillance, etc. Cependant, certains problèmes se posent car les réseaux à l'heure actuelle ne permettent pas de respecter la qualité de service nécessaire au transfert d'AV via Internet. Selon Grace et al. (2001), IPv4 n'est pas le meilleur protocole pour transmettre des données multimédia. En effet, IPv4 ne garantit pas une bande passante élevée aux applications en temps réel et il n'assure pas le transfert des données dans un temps abordable (Conti et al., 2002).

En fait, le transport d'AV implique la notion de QoS (Quality of Service) et IPv4 n'a pas été conçu pour offrir la QoS requise par l'AV. La notion de QoS pour le transport d'AV est généralement basée sur trois critères (Coulouris et al., 2001 ; Mathy et al., 1999 ; Wu et al., 2001) qui seront présentés dans la suite de cette section.

2.5.1 Bande passante

Ce paramètre caractérise la vitesse de transmission des flots. Elle est évaluée au début de la communication avant la transmission des données brutes. Cette condition implique qu'il faut avoir un mécanisme de contrôle de la bande passante disponible sur le réseau. La bande passante est reliée aux équipements qui consommeront l'AV et aux types de données vidéo. Comme le montre le Tableau 2-4 (Boutaba et al., 2002 ; Apostolopoulos et al., 2001), les vitesses de transfert à respecter varient énormément selon les applications. Les algorithmes de compression ont beaucoup évolué, ce qui permet d'atteindre des taux de compression de 50 :1 dans certains cas. En fait, les données vidéo non-compressées nécessitent une large bande passante; par exemple, les applications HDTV non-compressées requièrent 1,5 Gbps de bande passante alors que celles compressées en format MPEG en demandent dans le meilleur des cas 20 Mbps. C'est la raison pour laquelle Boutaba et al. (2002) proposent d'insérer des codeurs et des décodeurs dans la chaîne de transmission et de traitement des flots afin de diminuer la bande passante requise.

Tableau 2-4. Vitesse de transmission selon les applications

Application	Caractéristique	Bande passante
Videoconference	H.261	64 Kbps à 2 Mbps
VCR quality	MPEG-1	1.2–1.5 Mbps
TV quality	Compressée MPEG-2 4:2:2	15-60 Mbps
HDTV	Compressée Non-compressée	19.2–60 Mbps 1.5 Gbps
DVD	MPEG-2	Max. 9.8 Mbps

2.5.2 Taux d'erreurs

Ce taux définit le pourcentage de données reçues mais qui ne sont pas valides. Ce taux permet d'évaluer la qualité des données et de la transmission. Les taux sont calculés par la destination et généralement la source n'est aucunement responsable de la dégradation

du réseau. Le Tableau 2-5 donne les taux d'erreurs (BER) associés à certaines applications; la qualité de la vidéo est considérée inacceptable si les BERs sont supérieurs à ceux recommandés (Boutaba et al., 2002).

Tableau 2-5. Taux d'erreurs (BIT) requis par certaines applications

Application	Taux d'erreurs
Vidéoconférence	< 1e-6
Qualité TV	< 6e-11
HDTV	< 2e-11

2.5.3 Délai et latence

Le délai est surtout lié au temps de transmission du paquet en incluant le temps de traitement du paquet. Le délai varie en fonction de chaque paquet. Il dépend principalement du temps de traitement des équipements qui contribuent à la transmission du paquet de la source à la destination et de la taille des files de traitement. L'imposition d'une borne au délai permet de limiter la variabilité du taux de réception au niveau du client par exemple. Le Tableau 2-6 donne le délai acceptable d'un paquet dans le réseau (Zhu et al., 2000). Et comme on peut le voir, les applications HDTV sont les plus exigeantes encore une fois.

Tableau 2-6. Délai requis par certaines applications lors de la transmission sur un réseau

Applications	Délai
Vidéoconférence	< 400 ms
Qualité TV	< 100 ms
HDTV	< 50 ms

Ces trois critères sont interdépendants. Par exemple, si la bande passante dépasse celle requise, cela pourrait remplir les tampons des équipements. Le remplissage des tampons entraîne généralement un délai de traitement plus long qui impliquera la violation d'une échéance temporelle, d'où l'augmentation du taux d'erreurs.

2.6 Transport de vidéo dans un LAN

Il y a deux modes de livraison de la vidéo (Hong et al., 2002) : le téléchargement, qui est celui qu'on rencontre le plus souvent, et la transmission de flots continus en direct (real time video streaming). Ces deux modes de livraison n'ont pas les mêmes requis de qualité de service. Pour le mode téléchargement, tous les flots doivent d'abord être reçus avant que les opérations de décodage et d'affichage puissent démarrer. Alors que le mode continu permet de décoder et d'afficher les images au fur et à mesure qu'elles arrivent.

À l'air du direct, le mode téléchargement est révolu, les opérateurs veulent envoyer aux usagers des films en direct, par exemple. Les applications en temps réel sont multiples et le mode téléchargement fait appel à des protocoles qui ont déjà fait leurs preuves et qui assurent une intégrité et une qualité des images. C'est une des raisons pour lesquelles dans la suite de ce mémoire, nous parlerons seulement du mode continu.

L'envoi de flots vidéo sur un réseau se fait grâce à plusieurs unités de traitement. La Figure 2-10 donne un exemple des différentes unités qui peuvent intervenir durant l'envoi d'un flot vidéo.

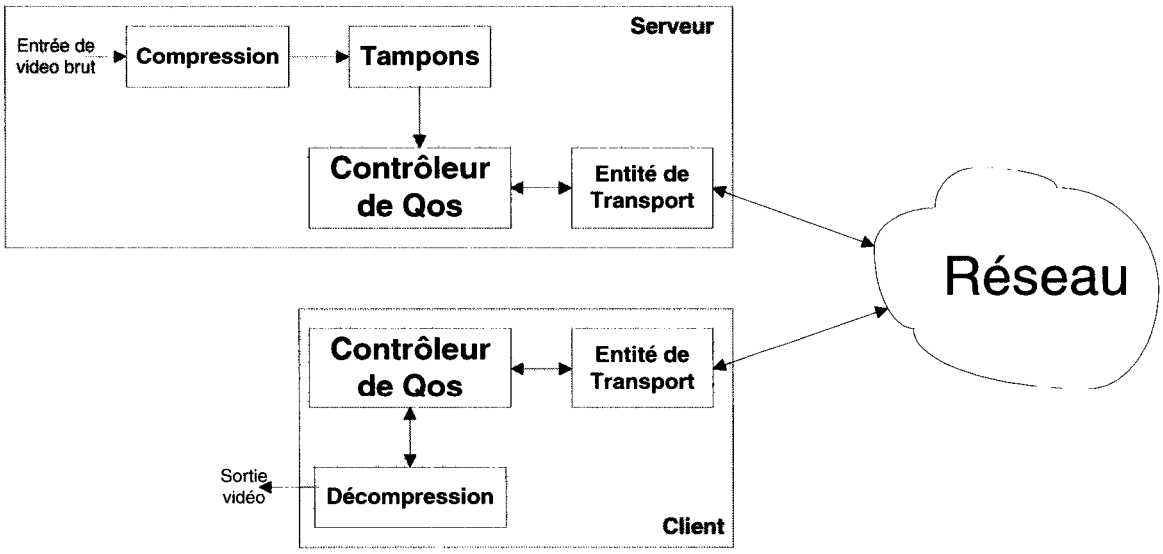

Figure 2-10. Architecture générale d'un système d'envoi de flots vidéo

Généralement, le processus d'envoi commence avec la capture de l'image venant d'un appareil tel une caméra. Selon la caméra, les données peuvent être compressées ou non. Dans le cas de certaines WebCam par exemple, les données AV ne sont pas compressées. Dans un cas pareil, Hong et al. (2002) proposent que les données soient compressées afin de diminuer la charge de données à retransmettre sur le réseau. Les données compressées sont stockées au fur et à mesure dans un tampon en attendant qu'elles puissent être traitées par le contrôleur de qualité de service. Ce dernier envoie ensuite les données vers le protocole de transport qui se chargera de tous les traitements nécessaires à la transmission des paquets sur le réseau.

Le module de contrôle de qualité de service est facultatif. Ce module permet d'améliorer la qualité de la transmission. Deux techniques de contrôle sont proposées : le contrôle de congestion du réseau et le contrôle d'erreurs. La technique de la congestion permet de réduire l'impact des pertes de paquets et du délai dû à la congestion du réseau. La technique de contrôle d'erreurs réduit l'impact des contraintes de bande passante, de pertes de paquets et des délais sur la qualité des données reçues. Hong et al. (2002) montrent bien qu'en intégrant un module de contrôle de la qualité, le taux d'erreurs diminue considérablement. Leur module est basé sur une approche de transfert de la correction d'erreur (FEC). Dans le cas contraire, c'est-à-dire sans module de contrôle, le nombre de paquets perdus varie irrégulièrement, ce qui ne permet pas d'assurer un flot vidéo acceptable.

L'élément qui opère toutes les tâches de réception des données vidéo est communément appelé un serveur d'audio/vidéo continu ou DVP. Il repose sur une architecture client/serveur. Ce module comporte généralement 3 entités :

- Communicateur : les clients font appel à ce serveur pour recevoir des flots ;
- Système d'exploitation : il permet de gérer à haut niveau les différentes requêtes et de réaliser les différentes tâches durant le processus de transmission ;
- Unité de stockage : elle contiendra les différentes queues.

Le Tableau 2-7 (Boutaba et al., 2002) récapitule très bien les services qui seraient utiles à un DVP et ceux que certains protocoles offrent.

Tableau 2-7. Résumés des services offerts par certains protocoles à un DVP

	MPEG	AAL1	RTP	UDP
Indicateur de temps	Oui	Oui	Oui	No
Détection d'erreurs par le CRC	Non	Seulement sur l'entête	Non	Optionnel
Numéros de séquence	Oui	Oui	Oui	Non
FEC (correction d'erreur directe)	Optionnel	Oui	Possible	Non
Multiplexage	Oui	Non	Oui	Oui
Encapsulation	Oui	Oui	Non	Oui

Boutaba et al. (2002) précisent bien que les protocoles sont quelques fois redondants. Par exemple, MPEG et RTP définissent tous les deux un indicateur de temps et des numéros de séquences. Ainsi, en encapsulant des paquets MPEG dans du RTP, les traitements de contrôle sont répétés deux fois. Cette redondance fait ressortir le problème du choix d'un empilement de protocoles adéquat lorsqu'il s'agit de transporter des données audio/vidéo.

Dans la littérature, les auteurs parlent souvent des traitements effectués sur la vidéo en tant que tel, mais ils ne parlent jamais des traitements de routage qui influencent aussi la qualité du flot. Ces derniers sont très importants, d'autant plus qu'ils ont un impact sur le temps total de transmission. Boutaba et al. (2002) parlent des services que les protocoles peuvent offrir à un serveur de flot. Ces derniers insistent plutôt sur la notion de multiplexage qui permet d'intégrer des paquets appartenant à un flot qui doivent être regroupés dans un seul gros paquet pour ensuite les envoyer vers le ou les mêmes destinataires.

2.7 Transport de vidéo d'un réseau Firewire vers un autre réseau

Des développements ont été effectués pour retransmettre de l'AV d'un équipement Firewire vers un réseau IPv4. Dans cette section, nous allons vous présenter les solutions qui ont été proposées par différents chercheurs, permettant ainsi de découvrir les travaux concrets qui ont été consacrés à Firewire.

2.7.1 Conception et implémentation d'un flot video numérique (DV) sur Internet

Ogawa et al. (1999) ont développé et implémenté un système d'envoi de flots DV sur Internet. Leur système, qui est illustré par la Figure 2-11, reçoit des paquets 1394 contenant de l'AV d'une caméra numérique (DV camcoder), il effectue certaines manipulations telles que l'encapsulation du paquet dans un format privé qui sera contenu dans un message UDP transmis sur Internet. Le destinataire à son tour effectue les opérations contraires pour récupérer le paquet 1394 encapsulé et pour l'envoyer vers un autre équipement Firewire qui pourra afficher l'image par exemple. Comme vous pouvez le voir, cette solution se base sur la méthode d'encapsulation présentée à la section 2.2.2.

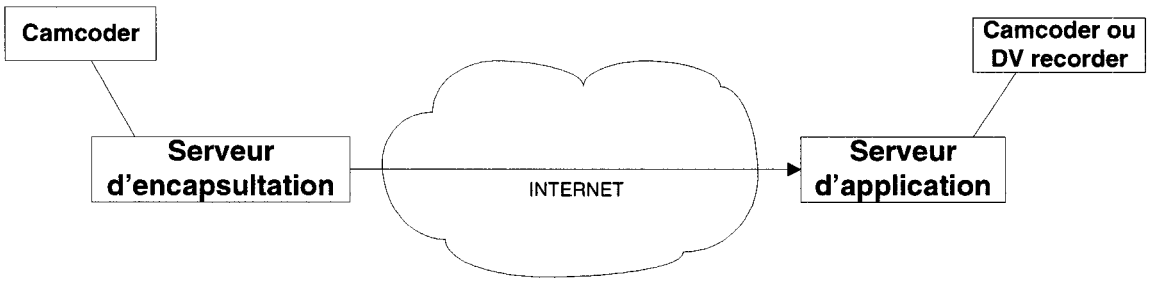

Figure 2-11. Système de Ogawa et al. (1999)

Ogawa et al. (1999) n'utilisent pas RTP, ils proposent une autre solution qu'ils trouvent plus économique. Ils conçoivent une entête privée qui est illustrée par la Figure 2-12. Cette entête permet d'effectuer un contrôle de la qualité de la transmission, étant donné qu'UDP ne garantit aucune qualité. D'après Ogawa et al. (1999), la compression n'influence pas la qualité du transfert. Il faut préciser que ces chercheurs ont utilisé les fonctions offertes par le routeur COMET. Ce routeur (Koga, 2000) a été développé dans un groupe de recherche d'un laboratoire de Fujitsu.

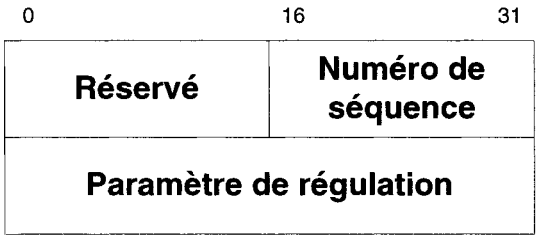

Figure 2-12. Format de l'entête proposé par Ogawa et al. (1999)

2.7.2 Wireless gateway

Saito et al. (2001) proposent une passerelle qui interconnecte un équipement IEEE1394 et un réseau mobile 802.11b à 2.4 GHz. Cette application peut relier une caméra et une télévision sans fil par exemple. La particularité de cette solution est qu'elle se base sur RTP. En effet, comme le montre la Figure 2-13, leur système reçoit des flots vidéo via une interface Firewire; ces flots sont compressés en format MPEG2; des contrôles sont effectués par des unités spéciales; et ensuite les paquets compressés sont encapsulés par RTP pour être transmis sur le réseau sans fil.

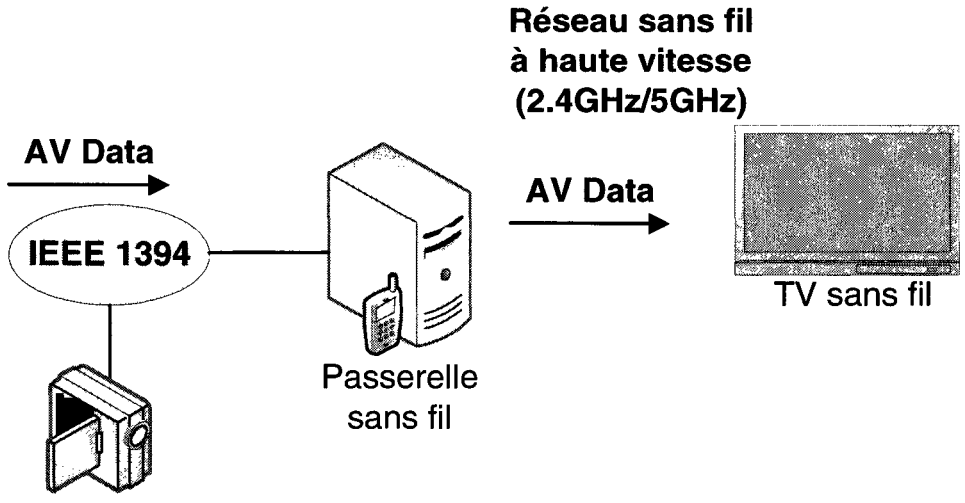

Figure 2-13. Architecture logicielle de la passerelle de Saito et al. (2001)

L'autre aspect important de cet article est que leur passerelle joue aussi le rôle d'un proxy. Il faut rappeler qu'un proxy est un ordinateur qui relie un réseau privé à Internet. Ils proposent une auto-configuration de la passerelle pour que cette dernière sache les

services offerts par les équipements de chaque réseau. De plus, ces équipements ne connaissent pas ceux qui sont connectés sur l'autre réseau. Les équipements d'un réseau sont identifiés par les services qu'ils offrent. Donc, pour communiquer avec un équipement de l'autre réseau, il faut faire appel à un service offert par ce dernier.

Cette solution est très séduisante, mais les traitements au niveau de la passerelle sont énormes. Le processus d'auto-configuration induit que la passerelle reconnaisse les équipements de chaque réseau et les types de services qui peuvent être offerts par ces derniers par exemple. Cette étape peut prendre beaucoup de temps surtout dans le cas de grands réseaux. Saito et al. (2001) ont effectué leurs premiers tests sur un réseau restreint, mais ils sont conscients des difficultés à rencontrer lorsque les réseaux comportent un grand nombre de machines.

CHAPITRE 3

TRANSPORT DE DATAGRAMMES IPv4

La fiabilité et l'efficacité de Firewire sont telles que de plus en plus de LAN seront implantés avec Firewire et non avec Ethernet, d'autant plus que des équipements tels que des imprimantes, des disques de stockage de données et des scanners opèrent avec Firewire. Cependant, les LAN Ethernet existants ne seront probablement pas remplacés par des LAN Firewire. Les coûts de remplacement des cartes réseaux Ethernet seraient trop dispendieux. Il faudra donc trouver un moyen de relier ces deux types de réseaux à un coût moindre.

Deux contraintes ont été définies pour résoudre le problème d'interconnexion. Elles concernent la longueur des paquets et la limitation de la vitesse de transmission de Firewire. En effet, la longueur maximale d'un paquet Firewire dépend de la vitesse de transmission. Firewire définit des fenêtres de transmission ayant une période de 125 µs. Ainsi, la taille maximale d'un paquet est définie en fonction de la vitesse de transmission et de cette période (Norme IEEE1394-1995).

Dans la première partie de ce chapitre, la première solution logicielle proposée pour résoudre le problème d'interconnexion sera explicitée. La seconde partie de cette section comporte les résultats de profilage qui ont été collectés durant l'exécution des différentes solutions logicielles sur le processeur ARM7TDMI. Une analyse de performance sera aussi présentée afin de noter l'impact des communications sur ce type d'architecture logicielle, plus précisément sur le débit du système. Dans la troisième partie, l'architecture de la plate-forme SOC du convertisseur de protocoles sera présentée. Nous insisterons sur les modules de l'architecture matérielle que nous avons eus à spécifier et à implémenter. Les performances de cette architecture seront exposées et une comparaison sera effectuée afin de montrer les gains de performance obtenus par l'architecture matérielle par rapport à ceux de la solution logicielle optimisée. Enfin, le chapitre finira

par une discussion sur l'architecture matérielle afin de proposer des améliorations à cette dernière.

3.1 Problématique de l'interconnexion LAN Ethernet – LAN Firewire

La flexibilité de Firewire lui permet de transporter des datagrammes IPv4 à haut débit avec de bien meilleures performances que Ethernet. De nouveaux LAN Firewire voient le jour de plus en plus, cependant, dans certains cas, ces derniers devront co-existaient avec des LAN Ethernet. Un scénario possible est illustré par la Figure 3-1. Un disque de stockage de données qui est branché sur un réseau Firewire et un PC est dans un LAN Ethernet. Le premier équipement se trouve au premier étage et le second au 10^{e} étage du même bâtiment. Nous supposons dans cet exemple qu'un LAN Ethernet est déjà implanté dans le bâtiment.

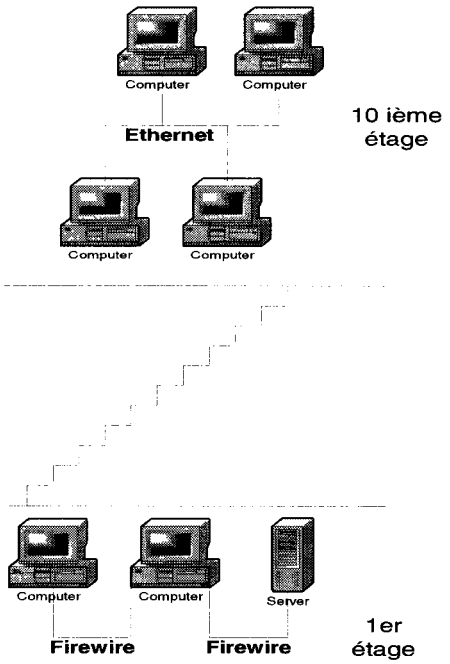

Figure 3-1. Cas d'interconnexion entre un PC Ethernet avec un serveur Firewire

La première solution qui s'offre consiste à rajouter des répéteurs entre ces deux équipements pour que la communication soit établie. La seconde solution consiste à :

- encapsuler le paquet Firewire dans un paquet IP/Ethernet ;

- transporter ce paquet Firewiresur le LAN Ethernet ;
- décapsuler le paquet;
- retransmettre ce paquet sur un mini-réseau Firewire vers le PC concerné.

La seconde solution nécessiterait l'ajout de deux PC: l'un à la périphérie du réseau Firewire et l'autre à la périphérie du PC concerné. Chacun de ces PC devrait exécuter un programme effectuant l'encapsulation et la décapsulation des paquets Firewire.

Ces deux solutions sont très complexes et coûteuses. La solution que nous proposons serait basée sur l'ajout d'un pont juste à la périphérie du réseau Firewire. Ce pont serait connecté au réseau Ethernet déjà en place. Cette solution, illustrée à la Figure 3-2, aurait comme principal avantage d'être peu coûteuse en termes de câblage et frais d'installation, puisqu'un seul équipement serait ajouté.

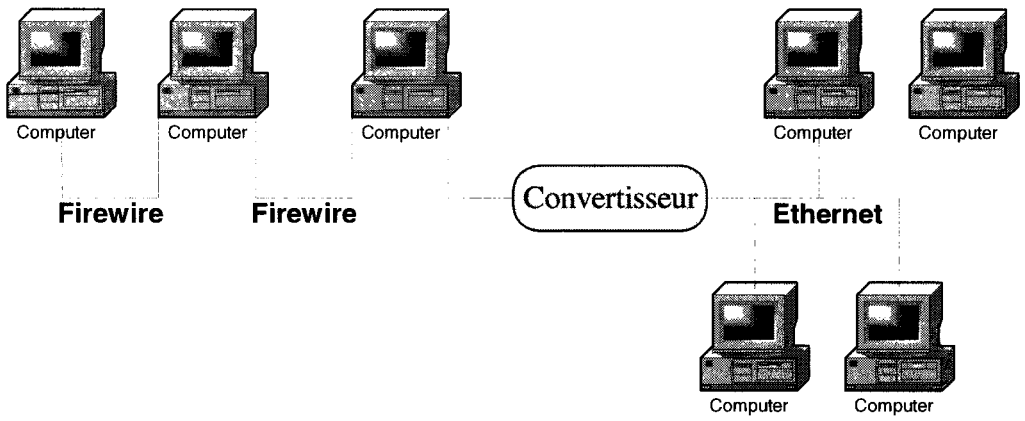

Figure 3-2. Scénario d'une communication entre un LAN Firewire et LAN Ethernet

Cette solution peut être basée sur le RFC 2734 (Voir Section 1.4.2) la norme la plus fiable et la plus utilisé pour transporter des datagrammes IPv4 dans un réseau Firewire. Comme le montre la Figure 3-3, l'empilement de protocoles qui entre en jeu durant l'interconnexion implique entre autres : Firewire, la couche virtuelle, IPv4 et Ethernet. Il faut rappeler que la couche virtuelle a été rajoutée afin de compléter l'information fournie par Firewire, permettant ainsi la fragmentation de datagrammes IPv4 durant la communication.

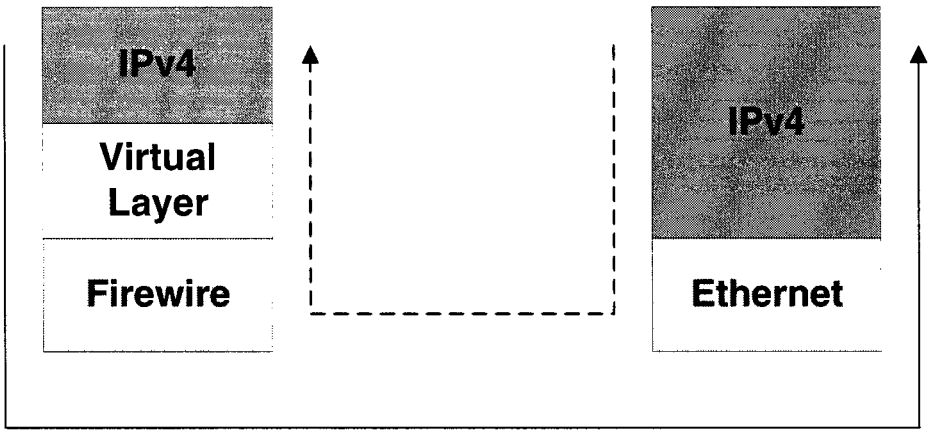

Figure 3-3. Empilement des protocoles

Ce pont pourrait être conçu de manière logicielle ou matérielle. Ainsi, dans la suite de ce chapitre, les deux modes de résolution seront présentés. La solution logicielle a été déployée sur un ARM7TDMI. Ce processeur a été choisi pour sa popularité dans le milieu des systèmes embarqués. Cet exercice permettra ainsi d'évaluer les performances de ce type de processeurs pour des applications réseaux comme l'interconnexion.

3.2 Solution logicielle

Comme nous l'avons dit précédemment, deux solutions logicielles ont été proposées. La première solution est basée sur une approche orientée objet et sera appelée la solution non optimisée. La seconde solution constitue une solution optimisée.

3.2.1 Analyse et conception de la solution non optimisée

L'objectif initial visé était de proposer une solution pour la conversion de protocoles dans un sens général, à savoir convertir tout paquet de type Protocole A en un paquet de type Protocole B. C'est dans ce contexte que nous avons proposé cette solution orientée objet. Pour commencer, une vue comportementale du système sera présentée afin de définir les différents intervenants du système, ainsi que les principaux scénarios qui se déroulent durant la conversion. Ensuite, l'aspect structurel sera explicité, pour montrer les différentes classes qui ont été implémentées et les relations qui existent entre elles.

3.2.1.1 Vue comportementale

Comme le montre la Figure 3-4, la conversion de protocoles implique l'intervention de 8 acteurs au total. L'acteur primordial qui configure l'environnement de conversion est le Gestionnaire de système.

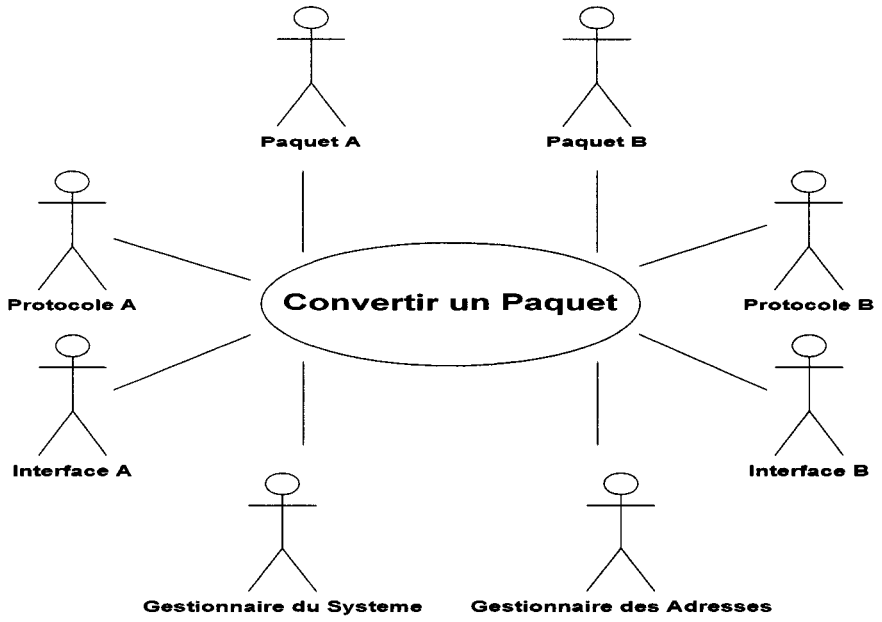

Figure 3-4. Diagramme de scénarios général de la conversion de 2 protocoles : A et B

Cet acteur intervient durant la phase d'initialisation du système, illustrée par la Figure 3-5. Durant cette phase, le gestionnaire précise au système les fichiers de configuration qui comportent la définition de chaque protocole. Un protocole peut être composé de plusieurs types de paquets, ces types étant tous définis dans un fichier. Un exemple de fichier peut être consulté à l'Annexe C. À la fin de cette étape, les acteurs Protocoles A et B sont initialisés.

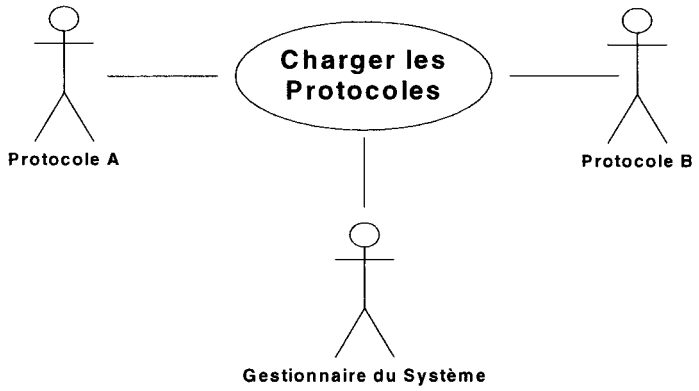

Figure 3-5. Initialisation du système

Après l'initialisation du système, ce dernier est maintenant capable de recevoir les paquets fournis par une des interfaces A ou B, comme le montre la Figure 3-6. Nous avons défini les paquets comme des acteurs afin de différencier les types de données qui sont manipulés durant les opérations de validation et d'extraction de champs.

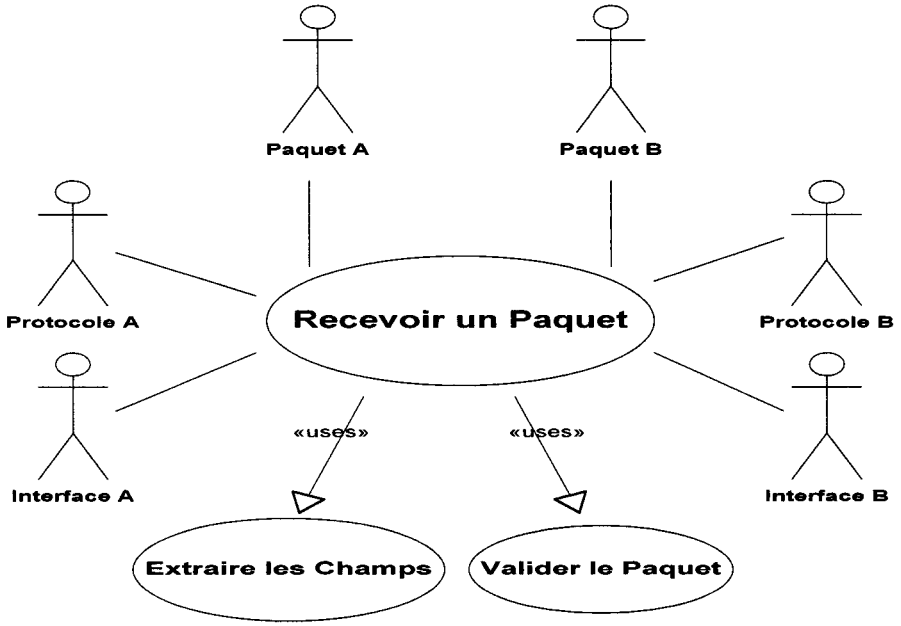

Figure 3-6. Scénario de réception d'un paquet

L'opération d'extraction de champs met en jeu les acteurs Protocole et Paquet selon le sens de la conversion. L'extraction de champs permettra entre autres d'identifier les types des paquets reçus et de valider les valeurs de certains champs en cas de restrictions propres à chaque champ. Le scénario « Valider le Paquet » vérifie si le paquet reçu est conforme aux restrictions générales liées à un type de paquet. Par exemple, la taille du paquet respecte-t-elle la taille maximale précisée dans le fichier de configuration?

Une des phases les plus importantes concerne le routage des paquets. En effet, il faudrait que le système soit capable de retransmettre le paquet reçu vers le bon équipement destination. Pour cela, il faut retrouver les adresses de type B correspondant aux adresses de type A contenues dans le paquet A reçu. C'est dans le cadre de cette recherche qu'intervient l'acteur Gestionnaire d'adresses, comme illustré par la Figure 3-7.

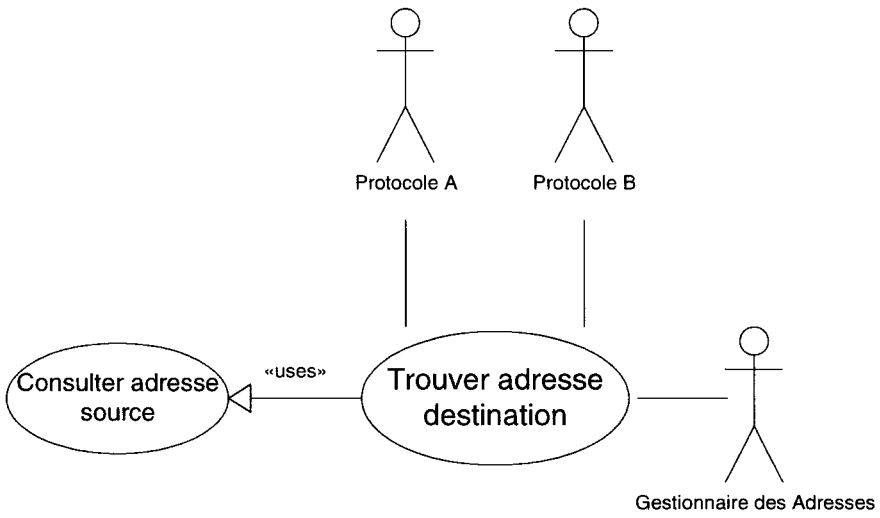

Figure 3-7. Recherche des adresses destination

Le Gestionnaire d'adresses peut être considéré comme un autre système à part entière. En effet, en raison de sa complexité, la gestion d'adresses s'opère en parallèle avec le processus de conversion. Cette répartition des tâches a été proposée afin de différencier le traitement de données brutes et celui de données de contrôle. Le gestionnaire d'adresses a

besoin de l'adresse de type A ou B pour débuter la recherche de correspondance. Cette adresse est fournie par un des acteurs de type Protocole.

Le dernier scénario concerne la constitution du paquet destination à retransmettre vers une des interfaces A ou B selon le sens de la conversion, comme illustré par la Figure 3-8. Cette opération utilise trois sous-opérations pour effectuer sa tâche. La première consiste à trouver la taille du paquet destination afin de réserver un espace mémoire dans lequel sera contenu le paquet destination. Ensuite, il faut mettre à jour les différents champs de l'entête de ce paquet. Enfin, il faudra récupérer la charge utile contenue dans le paquet source et la concaténer au paquet destination.

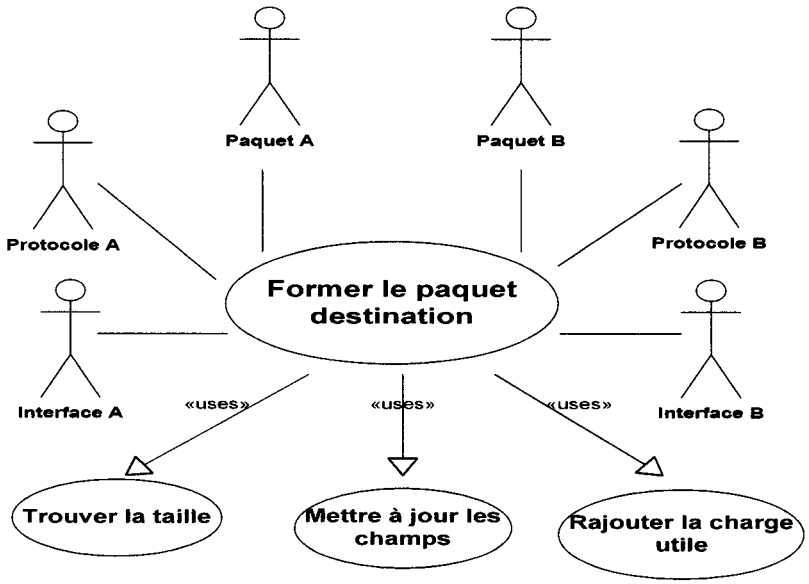

Figure 3-8. Formation du paquet destination

Les interfaces physiques sont des équipements qui reçoivent des paquets sous forme numérique et qui effectuent toutes les opérations nécessaires à la retransmission du paquet sous forme analogique sur le lien physique qui peut être un câble. Selon les protocoles, les interfaces physiques effectueront le calcul de checksum des paquets sortant ou la vérification de checksum. Certaines interfaces effectuent aussi des opérations de filtrage. Par exemple, les interfaces physiques Ethernet n'accepte que les

paquets ayant comme adresse MAC celle de la carte ou les messages systèmes (broadcast).

3.2.1.2 Vue structurale

Grâce à l'étude préliminaire des protocoles Firewire et Ethernet, différentes classes et leurs propriétés ont pu être établies. Le diagramme de classes de la Figure 3-9 présente les différentes structures qui ont été établies et les relations qui existent entre ces structures. Pour simplifier ce diagramme, nous avons omis de préciser les différentes méthodes (fonctions) associées à chaque classe.

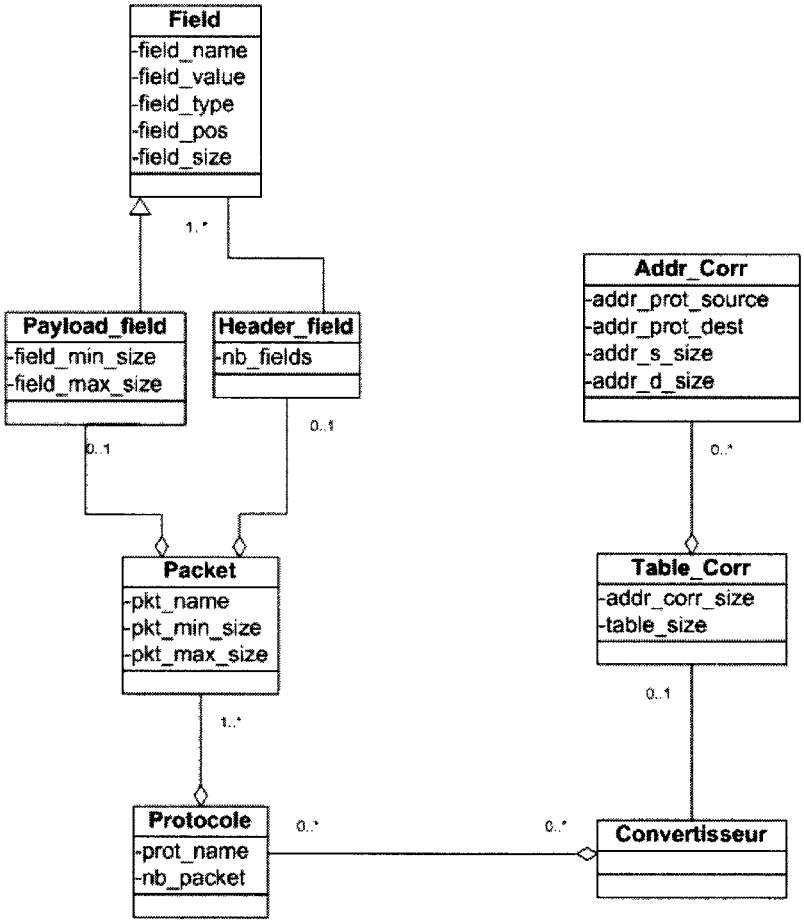

Figure 3-9. Diagramme de classes du convertisseur semi-générique

Dans la suite de ce mémoire, nous allons décrire les classes du diagramme en insistant sur leurs attributs. Ensuite, nous montrerons le fonctionnement du système basé sur ces classes.

A. Field

Chaque champ a plusieurs propriétés telles que son nom. Un champ très important a été rajouté : «type». En examinant le format des protocoles, nous avons vu qu'un paquet était composé d'une entête et d'une charge utile. Dans l'entête, nous pouvons observer différents types de champs : des champs d'adresses, de contrôle, de validité, de synchronisation. Le type du champ sera utile surtout quand il s'agira d'effectuer les opérations de fragmentation d'un paquet entrant et celles de conversion d'adresses.

B. Payload_Field

Cette classe hérite des attributs de la classe Field. En fait, la charge utile est un champ avec des propriétés supplémentaires telles que la taille minimale et maximale du champ. Ces champs sont utilisés pour valider la charge utile du paquet. Si un paquet reçu ne respecte pas ces paramètres, le système devra le jeter.

C. Header_Field

Cette structure comporte plusieurs objets de type « Field » comme indiqué par la relation d'agrégat entre cette classe et la classe « Field ». Il s'agit de tous les champs contenus dans l'entête plus le dernier champ du paquet (communément appelé la queue) s'il y a lieu. Selon le type du protocole, ce champ pourrait ne pas exister. Et enfin, cette classe comporte le nombre total de champs de contrôle associés au paquet : « nb_fields ».

D. Packet

Un paquet est constitué d'une entête et d'une charge utile, d'où la relation d'agrégat entre la classe « Packet » et les classes « Payload_Field » et « Header_Field ». De plus, le paquet se caractérise par son nom et certaines propriétés telles que sa taille

minimale et sa taille maximale. Ce sont les deux premières vérifications à effectuer avant l'extraction des champs associés à un paquet.

E. Protocol

Un protocole est constitué d'un ou de plusieurs types de paquet. Firewire, par exemple, comporte 10 types de paquets. Donc, en plus de ces objets de type « Packet », cette classe se caractérise par son nom et le nombre de paquets qui lui sont associés. Cette classe comporte des méthodes génériques qui permettent de consulter, modifier ou même supprimer des champs d'un type de paquet donné ou les valeurs de ceux-ci.

F. Addr_Corr

Un enregistrement de cette classe comporte principalement une adresse source (de type A par exemple) et une adresse destination (de type B par exemple). De plus, la taille des adresses en bits est aussi précisée.

G. Table_Corr

La table de correspondance est constituée de plusieurs enregistrements de type « Addr_Corr ». Pour effectuer les opérations de recherche, il faut garder la taille de la table, raison pour laquelle l'attribut « tab_size » a été rajouté. Cependant, l'utilité principale de cette classe concerne le parcours des différents enregistrements pour la recherche de l'adresse correspondante à une adresse donnée en paramètre.

H. Conversion

C'est la classe qui assemble l'ensemble des entités. Il faut d'abord définir les objets de type « Protocol » qui entreront en jeu durant la conversion, et la table de correspondance qui est primordiale pour le routage des paquets à retransmettre. Enfin, c'est dans cette classe que les opérations telles que la réception de paquet et la formation du paquet destination sont implémentées. C'est cette classe qui rend la

solution orientée objet semi-générique, car ces dernières opérations ne sont pas automatisées.

Pour le cas de la conversion Firewire-Ethernet, 4 objets de type Protocole ont été générés, comme le montre la Figure 3-10. Cette figure illustre l'enchaînement des opérations qui s'effectuent durant la conversion Firewire à Ethernet.

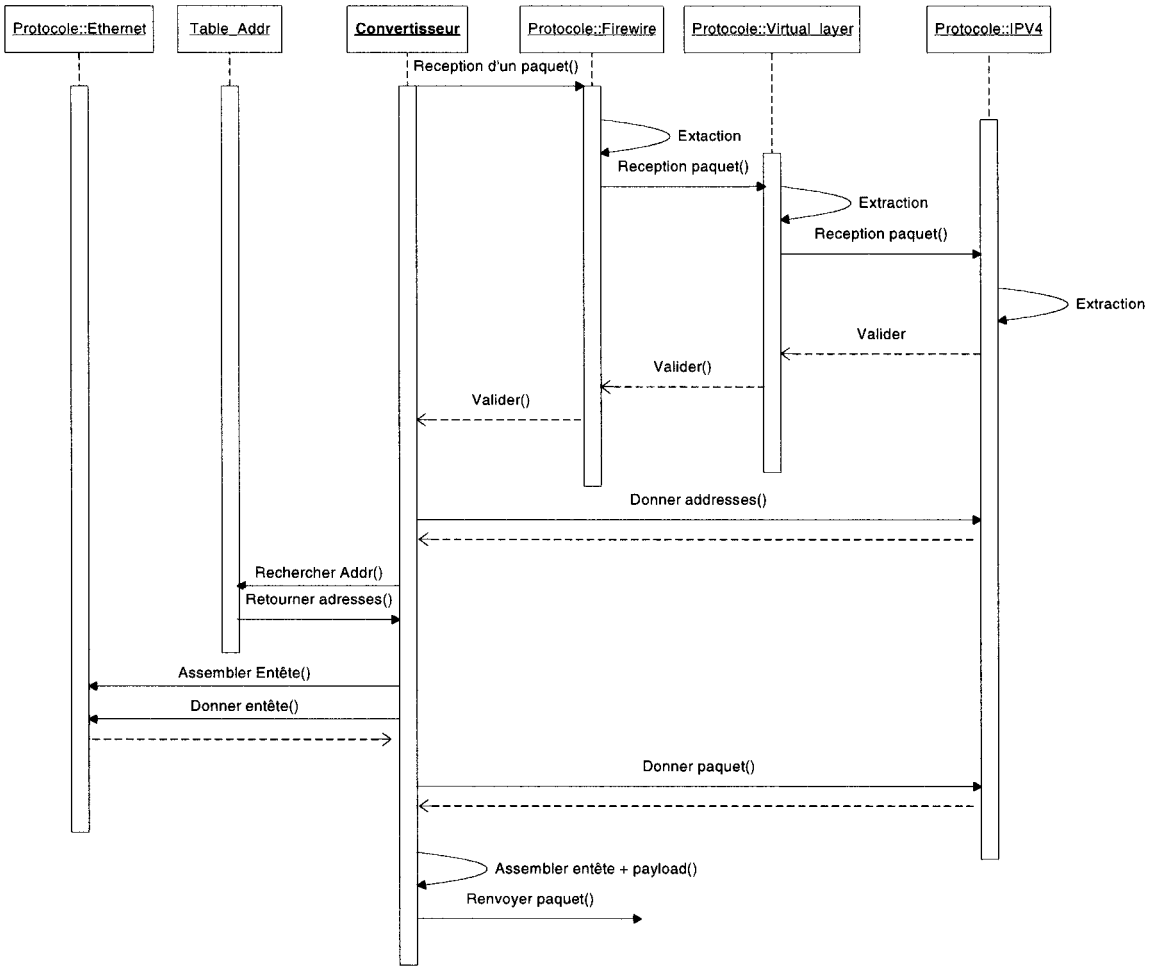

Figure 3-10. Diagramme de séquence de la conversion Firewire / Ethernet

Comme on peut le voir, des extractions successives sont effectuées afin de vérifier que le paquet reçu transporte bien un datagramme IPv4. Ensuite, les différentes opérations pour la recherche d'adresses sont effectuées. De plus, le convertisseur utilise l'adresse IPv4 destination pour retrouver l'adresse Ethernet de l'équipement destination. Ensuite, le

convertisseur demandera à l'objet IPv4 de lui transmettre tout le paquet afin d'intégrer la charge utile dans le paquet Ethernet à retransmettre.

La présentation de la solution orientée objet s'arrête à ce point pour ne pas s'étendre dans les détails d'implémentation qui ont été développés dans le cadre de cette maîtrise.

Une première application, la conversion de protocoles Firewire et Ethernet, a été testée en se basant sur ce système de conversion. Les résultats de profilage issus des premiers tests seront exposés plus loin. Nous expliquerons aussi pourquoi cette solution orientée objet a été mise de côté au profit d'une solution spécifique à la conversion de Firewire et d'Ethernet uniquement. Le code qui a été implémenté peut être consulté à l'Annexe D.

3.2.2 Analyse et conception de la solution optimisée

La méthode orientée-objet, bien que séduisante, est très lourde en termes de temps d'exécution. En effet, la gestion des objets nécessite l'ajout d'instructions afin de contrôler et de manipuler ces derniers. Après analyse des résultats de la solution orientée-objet, une approche fonctionnelle, s'appuyant sur les traitements propres aux deux protocoles seulement, a été adoptée.

La Figure 3-11 donne les différentes étapes de la conversion de Firewire à Ethernet et vice versa. Généralement, trois faits peuvent faire échouer la conversion de protocoles :

1. La charge utile ne contient pas un datagramme IPv4;
2. La longueur du paquet est supérieure à la longueur maximale définie;
3. La recherche des adresses Firewire ou Ethernet (selon le sens de la conversion) correspondantes aux adresses IPv4 contenues dans le paquet source a échoué.

Comme le montre la Figure 3-11, les traitements durant les deux conversions sont assez semblables. Seules les positions des champs à consulter diffèrent ainsi que l'enchaînement des opérations pour l'assemblage du paquet à retourner.

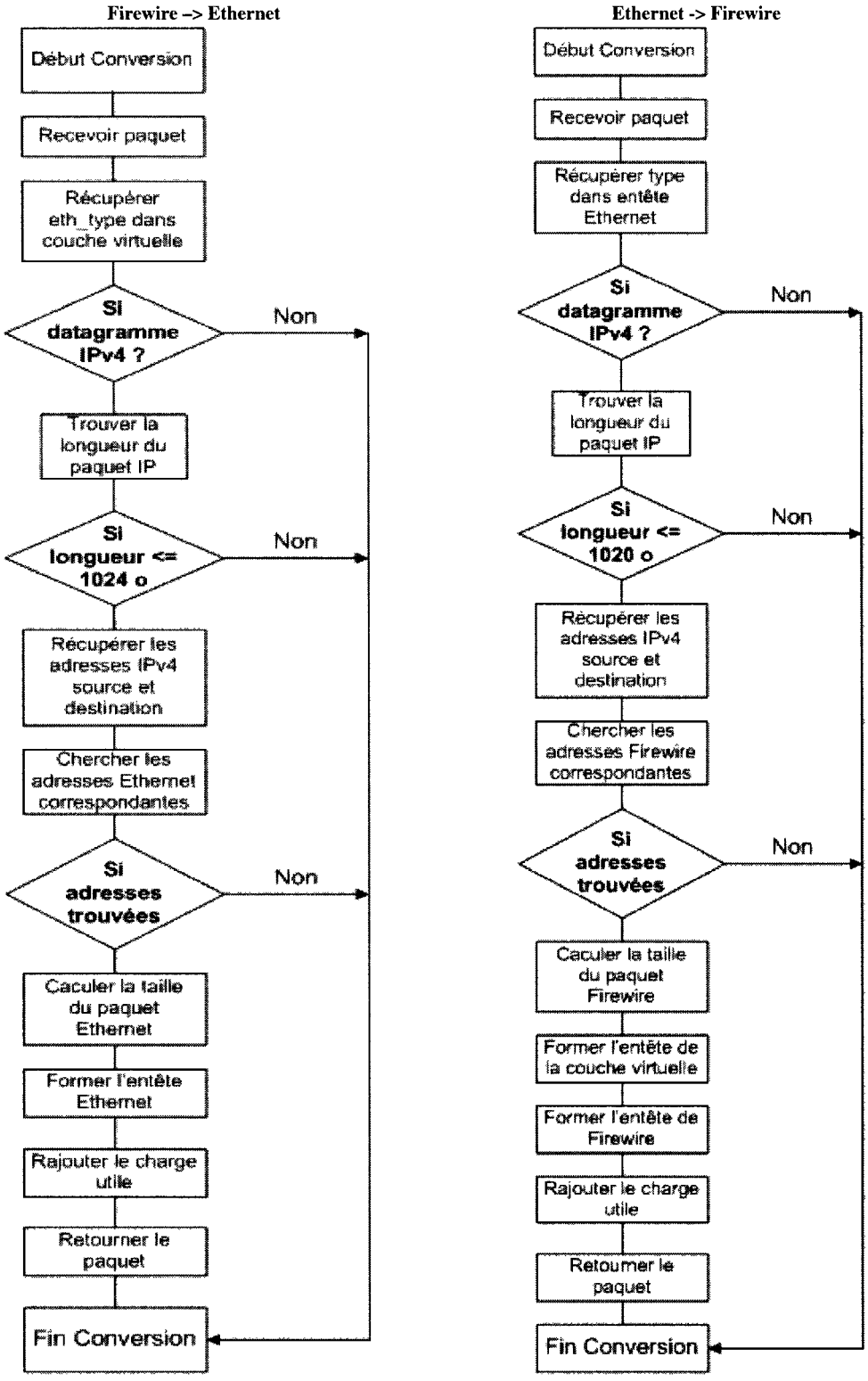

Figure 3-11. Organigramme de la conversion Firewire - Ethernet

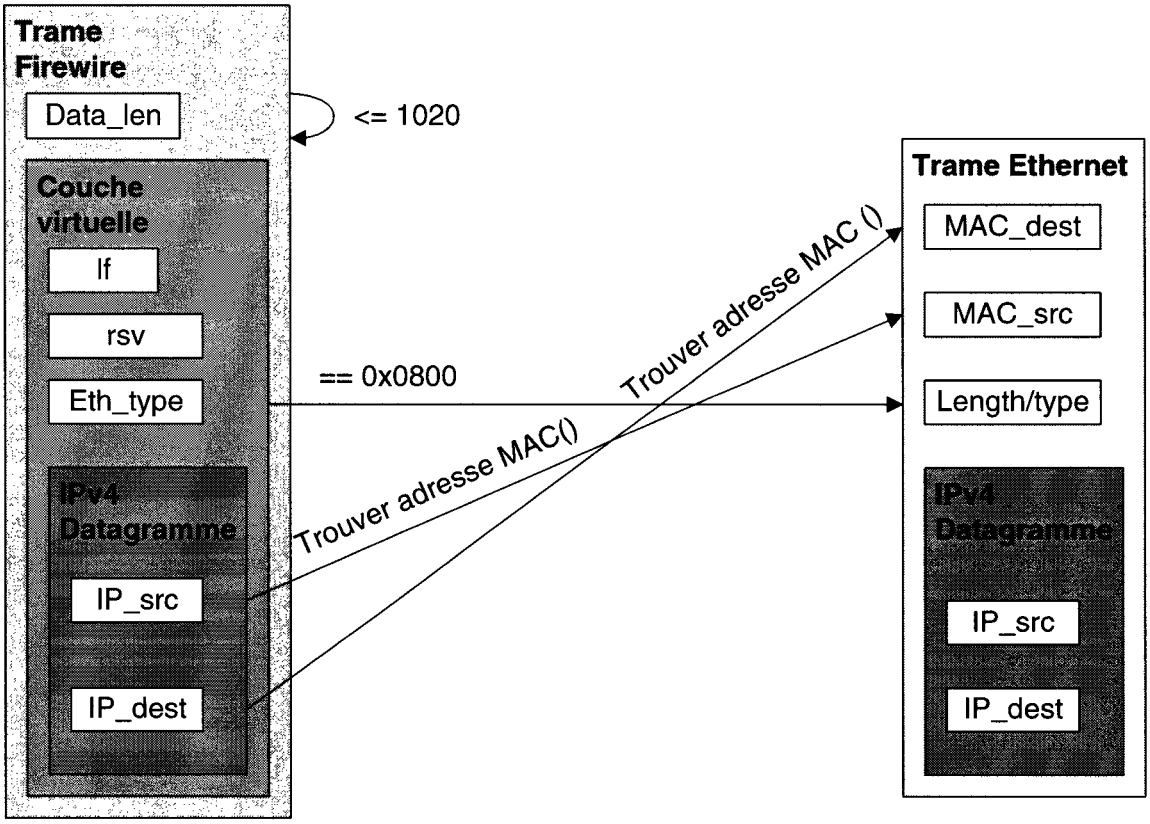

Figure 3-12. Correspondances entre les champs

Comme il a été dit précédemment, Ethernet est un protocole très simple contrairement à Firewire. Comme le montre la Figure 3-12, peu de champs du paquet Firewire sont utilisés pour effectuer la conversion. Seul le champ « data_len » du paquet Firewire est consulté afin de vérifier que la longueur du paquet est conforme, pour éviter la fragmentation du paquet.

Ensuite, le champ « eth_type » contenu dans l'entête de la couche virtuelle est consulté afin de vérifier que la charge utile du paquet Firewire contient un datagramme IPv4. Le protocole IPv4 est caractérisé par la valeur 0x0800. Il faut préciser que lorsque le champ « length/type » de l'entête Ethernet a une valeur supérieure à 1500, alors cette valeur indique le type de paquet contenu dans la charge utile du paquet Ethernet.

Enfin, les champs IP_scr et IP_dest sont exploités afin de retrouver les adresses MAC source et destination correspondantes aux équipements adressés. Il faut rappeler que chaque équipement Firewire envoyant des datagrammes IPv4 est identifié par une adresse IPv4. C'est cette adresse qui est contenue dans les datagrammes IPv4. Donc, le convertisseur devra tenir une table de correspondance afin d'associer à une adresse IPv4 une et une seule adresse MAC ou Firewire. Il faudrait s'assurer qu'une adresse IPv4 n'indexe pas deux équipements se trouvant dans les deux réseaux.

3.3 Évaluation de performance du ARM7TDMI

Après avoir développé les codes associés à chacune de ces solutions logicielles (Annexe D), des activités de profilage ont été effectuées sur un ARM7TDMI fonctionnant à une vitesse de 66 MHz. Ces activités avaient pour but de noter les traitements coûteux en termes de temps d'exécution et d'évaluer les performances de la conversion sur ce type de processeur. Donc, dans la suite de cette sous-section, la charge de chaque tâche durant le processus de conversion sera présentée. Ensuite, les performances de chacune des applications seront évaluées. Et enfin, le débit de la solution optimisée sera évalué en fonction des opérations d'entrée/sortie.

3.3.1 Répartition des tâches

Les résultats de profilage ont été obtenus grâce à l'ARM7TDMI Emulator. Bien que ce soit juste les entêtes qui sont manipulées et que la charge utile ne soit pas touchée durant les extractions successives, ces dernières deviennent le goulot d'étranglement de l'application avec 46.9% du temps d'exécution. Le Tableau 3-1 illustre la charge de chaque tâche durant le processus de conversion. Il faut préciser que la réception et la transmission des paquets ne sont pas représentées. Ces tâches sont effectuées à bas niveau.

Tableau 3-1. Répartition des tâches Firewire à Ethernet

Tâches	Non optimisé		Optimisé	
	%	Nb-cycles	%	Nb-cycles
Extraire les différents champs (entête) des paquets : 1- Firewire 2 - Couche virt. 3 - IPv4	46.9	1 977 932	0.45	**1 905**
Récupérer les adresses source et destination IPv4	9.8	409 746		
Rechercher les adresses Ethernet correspondantes à ces adresses Ipv4	3.0	128 045	7.75	**32 820**
Former l'entête Ethernet	4.7	204 873	91.8	**388 759**
Rajouter la charge utile dans le paquet	35.6	1 438 633		
Total	**100**	**2 685 52**	**100**	**423 484**

Les premiers résultats sont issus de la solution semi-générique. Il faut rappeler que les définitions des protocoles étaient chargées automatiquement et l'extraction de champs était principalement basée sur des manipulations de chaînes de caractères. Des fonctions de bibliothèques fournies par ARM ont été exploitées afin d'effectuer ces manipulations. Vu le temps excessif pris par ces fonctions, nous avons décidé d'optimiser le code pour une application bien spécifique : la conversion de Firewire à Ethernet. En conséquence, aucune fonction n'est appelée, excepté une fonction d'assemblage de chaînes d'entiers.

3.3.2 Modèle de performance du ARM7TDMI

Une comparaison a été effectuée entre le code optimisé et le code semi-générique. Le Tableau 3-2 montre, sur un échantillon de paquets, le gain qui a été obtenu grâce à la spécialisation du code. La manipulation de chaînes de caractères sur un ARM est déconseillée.

Tableau 3-2. Temps de traitement entre l'application optimisée et la semi-générique

Taille des paquets (octets)	Temps de traitements des applications (nb cycles)		
	Opt	Non-Opt	Diff
80	2711	53571	50860
88	2785	53659	50874
96	2859	53733	50874
144	3303	54177	50874
176	3599	54473	50874
224	4043	54917	50874
244	4228	55293	51065
280	4561	55626	51065
308	4820	55694	50874

Note : Opt : Optimisé, Diff : Différence

Le temps additionnel requis pour ce traitement est considérable, soit environ 50874 cycles de plus en moyenne. Cependant, avec la manipulation d'entiers, les performances notées sont bien meilleures et le temps de traitement demeure acceptable. Ceci montre qu'il est plus efficace d'optimiser les applications devant être exécutées sur un ARM, d'autant plus que le ARM7TDMI est limité en termes de vitesse de traitement : 66 MHz dans le meilleur des cas et 45 MHz pour le pire cas.

Un échantillon de paquets a été traité par les applications, et les Figure 3-13 et Figure 3-14 illustrent l'évolution du temps de traitement en fonction de la taille des paquets de chacune des solutions. Le temps de traitement varie de manière linéaire avec la taille des paquets. Cela s'explique par le fait que les tâches durant le processus de conversion sont constantes, exceptées la tâche de rajout de la charge utile qui varie selon la taille de cette dernière. On peut remarquer que les deux applications évoluent presque sur la même pente. Nous avons fait le choix de reformer le paquet sortant au complet, entre autres, parce que les DMA d'effectuent pas d'assemblage des différentes parties d'un paquet. Bien sur, nous aurions pu concevoir un co-processeur d'assemblage qui formerait les paquets sortant à partir des entêtes converties et de la charge utile initiale en les envoyant au fur et à mesure à l'un des DMA.

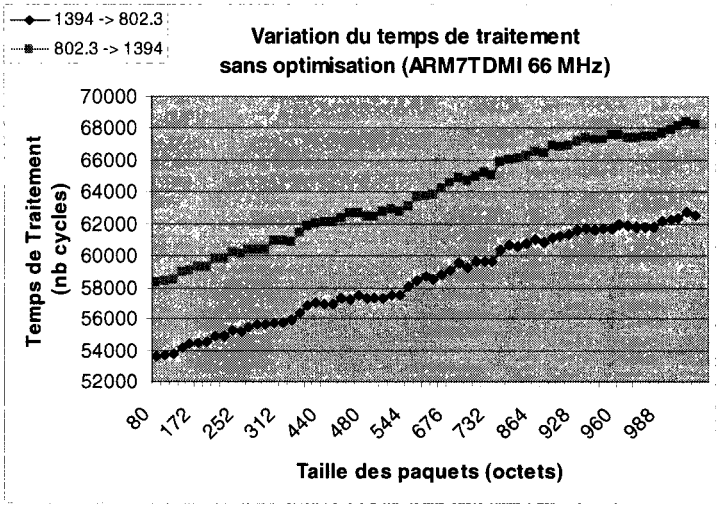

Figure 3-13. Temps de traitement de la solution non-optimisée

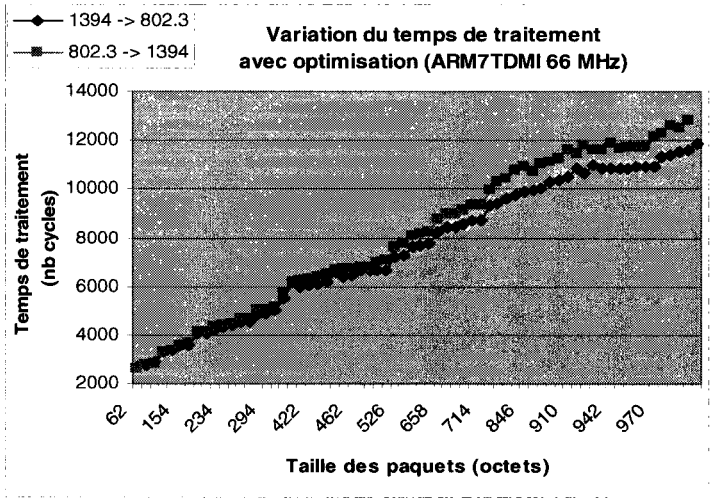

Figure 3-14. Temps de traitement de la solution optimisée

3.3.3 Impacts de l'environnement du ARM

Différentes composantes entrent en jeu durant le processus de conversion. En effet, outre le processeur qui exécute le programme de formatage, une mémoire interviendra durant le processus ainsi que des DMA, comme le montre la Figure 3-15. La vitesse des DMA

inclus dans les contrôleurs OHCI (Open Host Controller Interface) et MAC, ainsi que celle du bus de données définiront principalement les limites de la solution.

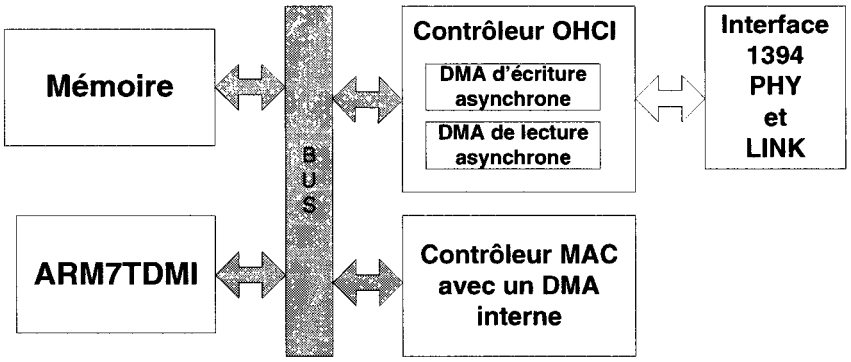

Figure 3-15. Architecture de la solution logicielle

Dans le cas du ARM7TDMI, le bus de données sera un AMBA/AHB. Ce bus a une vitesse de transmission de 66 MHz. De plus, il opère une écriture en 2 coups d'horloge et une lecture en 3 coups d'horloge. La vitesse des contrôleurs OHCI varie selon les modèles, par exemple le modèle TSB12LV23 (Texas Instruments, 1999) fonctionne avec une horloge maximale de 33 MHz. La vitesse du contrôleur MAC d'Atmel (Atmel, 2002) s'élève à 50 MHz.

La Figure 3-16 illustre le chemin de traitement de l'information. La donnée doit d'abord être reçue via un DMA et transférée en mémoire. Ensuite, les opérations de formatage sont effectuées sur le paquet et enfin ce dernier est déchargé de la mémoire pour être retransmis via un DMA de sortie.

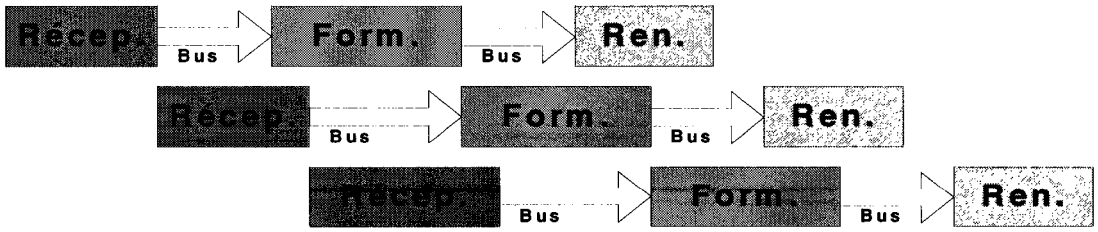

Figure 3-16. Vue d'ensemble du chemin des traitements

Le temps de traitement de chaque étape de l'application optimisée est formulée comme suit :

$$Tréc(s) = \left(\frac{1}{Vdma\_in} + \frac{2}{Vbus}\right) \times \left(\frac{pkt\_size}{4}\right) \quad (3.1)$$

$$Tren(s) = \left(\frac{1}{Vdma\_out} + \frac{3}{Vbus}\right) \times \left(\frac{pkt\_size}{4}\right) \quad (3.2)$$

$$Tfor_{1394->802.3} = 1.77*10^{-7} \times pkt\_size + 2.96*10^{-5} \quad (3.3)$$

$$Tfor_{802.3->1394} = 1.83*10^{-7} \times pkt\_size + 2.74*10^{-5} \quad (3.4)$$

Dans ces relations, Tréc désigne le temps de réception d'un paquet, Tfor le temps de formatage et Tren le temps de renvoi. La taille des paquets : pkt_size est mesurée en octets.

Une solution multi-thread a été adoptée afin d'augmenter le débit du système. Ainsi, pendant l'étape de réception d'un paquet $x+1$, un paquet x est en phase de formatage et un paquet $x-1$ sera retransmis. Cependant, le débit dépendra du temps de formatage et celui de retransmission d'un paquet. Il faut rappeler que le temps de réception du paquet $x-1$ est inclus dans le temps de formatage du paquet x.

La taille du paquet définit les performances du logiciel. Le Tableau 3-3 montre qu'avec de petits paquets, le logiciel devient le goulot d'étranglement du système, ce qui limite le débit absolu (incluant les entêtes) du système. En fait, le formatage prend le dessus tout le temps, ce qui limite le débit du système dans le meilleur des cas à 42,5 Mbps pendant que les I/O plafonnent avec 15,37 % du temps total de la conversion Firewire à Ethernet.

Tableau 3-3. Temps de conversion

Cas (pkt_size)	Temps Trait.	Tréc. (%)	Tren. (%)	Tfor. (%)	Vitesse (Mbps)
Firewire à Ethernet					
Pire (68 o)	43µs	2.38	2.04	98.5	12.89
Moyen (252 o)	72.9 µs	5.2	5.3	89.5	25.2
Meilleur (1048 o)	213 µs	7.44	7.93	84.6	42.5
Ethernet à Firewire					
Pire (54 o)	40 µs	1.67	3.18	98.15	13.4
Moyen (252 o)	73.6 µs	4.06	5.18	89.5	28.54
Meilleur (1034 o)	222 µs	5.83	7.12	84.6	39.9

3.4 Solution matérielle – Architecture version 2

Une architecture matérielle a été proposée par (Tremblay et al., 2001). Elle est illustrée à la Figure 3-17. Cette dernière devait résoudre le problème de la conversion de protocoles.

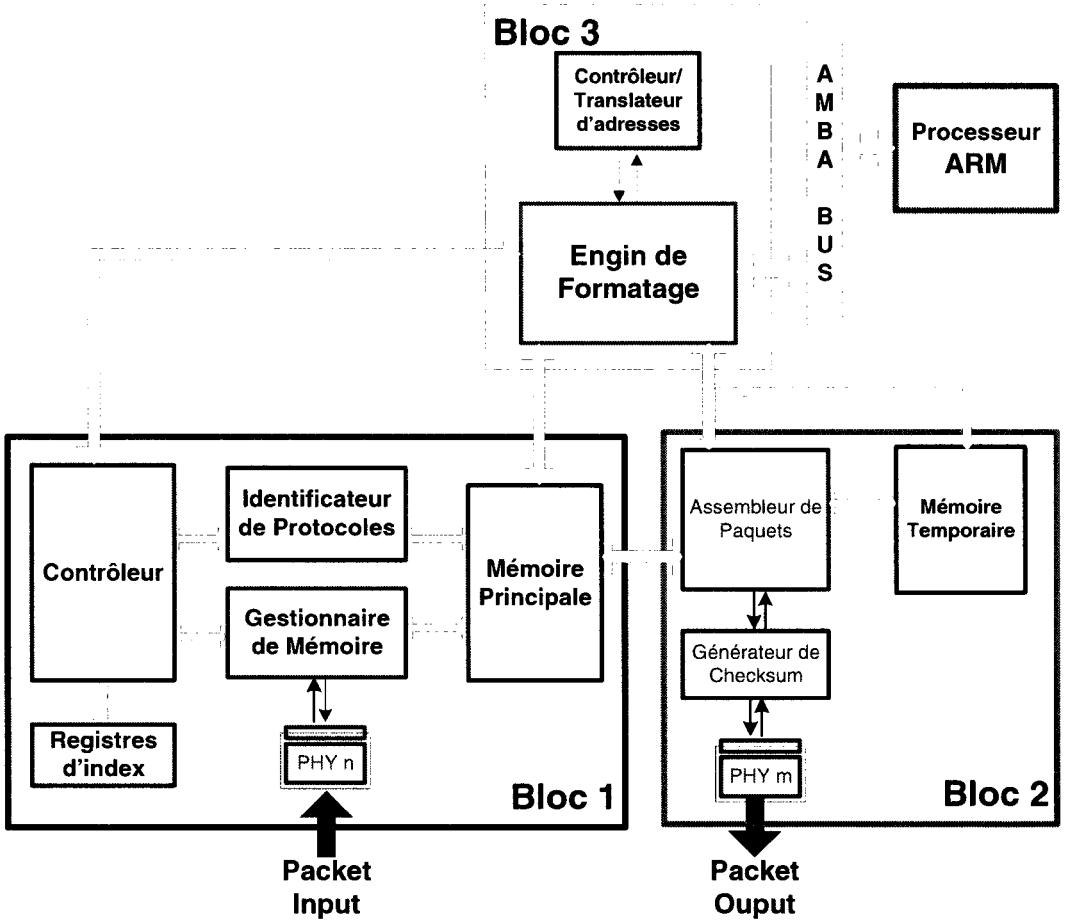

Figure 3-17. Architecture d'une plate-forme SOC pour la conversion de protocoles

Il faut préciser que cette architecture permet le traitement d'un flot de données d'un seul protocole de bas niveau.

Le convertisseur de protocoles a été subdivisé en trois blocs. Le premier bloc (Bloc1) est spécialisé dans la réception du paquet et les traitements préliminaires qui permettent de préparer la tâche de conversion proprement dite. Les paquets arrivent via une interface physique de transmission de données. Cette interface communique avec le gestionnaire

de mémoire, en lui transmettant les données et certains signaux de contrôle qui permettent d'activer le début d'une réception, entre autres. Le gestionnaire de mémoire stocke les données au fur et à mesure dans la mémoire principale et transmet au CONTROLEUR les caractéristiques du paquet en transmission. Ces caractéristiques sont : l'adresse de début du paquet dans la mémoire principale, le protocole de bas niveau et la taille totale du paquet. À la fin de la transmission, le CONTROLEUR averti par le GESTIONNAIRE DE MEMOIRE, va créer un vecteur d'étiquettes qui contiendra ces caractéristiques. Le CONTROLEUR distribue des tâches à l'IDENTIFICATEUR DE PROTOCOLES et à l'ENGIN DE FORMATAGE. Tous les paquets reçus doivent être traités par l'IDENTIFICATEUR DE PROTOCOLES afin que ce dernier puisse identifier l'empilement de protocoles contenus dans le paquet reçu. Après traitement, l'IDENTIFICATEUR DE PROTOCOLES renvoie une réponse au CONTROLEUR. Selon cette réponse, le paquet peut être converti ou ce dernier devra être supprimé via le GESTIONNAIRE DE MEMOIRE. La réponse de l'identificateur de protocoles se matérialise par la modification du champ protocole. En fait, ce champ sera considéré durant la conversion de protocoles. Après l'identification de protocoles, le CONTROLEUR pourra émettre un ordre de conversion vers l'engin de formatage selon la disponibilité de ce dernier. Comme vous pouvez le voir, le Bloc 1 est primordial dans l'architecture. Le bon fonctionnement de ce bloc conditionne le succès du convertisseur de protocoles.

Après avoir préparé les caractéristiques du paquet, le Bloc 3 se chargera de la conversion proprement dite. Ce bloc est composé de l'ENGIN DE FORMATAGE et du convertisseur d'adresses (AC). Rappelons que l'ENGIN DE FORMATAGE est un mini-processeur qui effectue des opérations arithmétiques et des instructions de chargement par exemple. La conversion de protocoles inclut la conversion d'adresses étant donné que, selon les protocoles, le format des adresses diffère. Durant la conversion logicielle, le programme fera appel au service du module matériel : le CONVERTISSEUR D'ADRESSES. Ce co-processeur devra être composé d'une table de correspondance. Cette table contiendra les adresses du protocole source et celles qui correspondent au protocole destination. La

conversion de protocoles consiste en la création de nouvelles entêtes selon l'empilement de protocoles destination. Ces entêtes sont stockées dans la MEMOIRE TEMPORAIRE. À la fin de la conversion logicielle, le Bloc 3 fera appel aux services du Bloc 2.

Le Bloc 2 est spécialisé dans la transmission des paquets convertis en respectant l'ordre des formats de données. L'ENGIN DE FORMATAGE émet un ordre de transmission à l'ASSEMBLEUR DE PAQUETS, en précisant les adresses et les tailles des entêtes ainsi que celle de la charge utile. L'ENGIN DE FORMATAGE fournit aussi des informations concernant le calcul de checksum. Selon le protocole destination de la couche liaison de données (Modèle OSI), l'algorithme de calcul du checksum pourrait varier. L'ASSEMBLEUR DE PAQUETS se charge de lire les entêtes dans la MEMOIRE TEMPORAIRE ainsi que la charge utile contenue dans la MEMOIRE PRINCIPALE et de les transmettre au GENERATEUR DE CHECKSUM. Ce dernier effectue le calcul du checksum et le rajoute à la queue du paquet. Il faut préciser que cette opération de calcul est facultative. Le générateur enverra les mots du paquet au fur et à mesure vers l'interface physique de sortie en commençant par l'entête de la couche liaison de données.

3.4.1 Modules de l'architecture

Nous avons spécifié et conçu certains modules de la plate-forme. Nous avons également développé un algorithme de conversion Firewire-Ethernet qui devait s'exécuter sur l'engin de formatage. Les opérations de synthèse ont été effectuées grâce à l'outil Synplify de la compagnie Xilinx. Il faut préciser que la plate-forme a été déployée sur un FPGA : XCV1000 de la famille Virtex E.

Le Tableau 3-4 récapitule les performances atteintes par les modules que nous avons eu à implémenter. Le Gestionnaire de Mémoire est le plus lent, mais cette vitesse est satisfaisante. Les études sur les interfaces physiques (Atmel, 2002; Texas Instrument, 1999) ont montré que ces dernières transmettent les données à une vitesse de 33 MHz à 66 MHz pour Firewire et 50 MHz pour Ethernet. Cette contrainte oblige l'Assembleur de

Paquets ainsi que le Gestionnaire de Mémoire à restreindre leur vitesse de transmission à la même vitesse que les interfaces physiques selon le sens de la conversion.

Tableau 3-4. Performances des modules conçus et implémentés

Modules	Vitesse	Espace	
		Type (LUTs)	Pourcentage
GESTIONNAIRE DE MEMOIRE	66.4 MHz	166	1 %
CONTROLEUR	72.4 MHz	499	2 %
ASSEMBLEUR DE PAQUETS	80 MHz	541	2 %
MEMOIRE PRINCIPALE	74 MHz	16835	68 %
MEMOIRE TEMPORAIRE	176.4 MHz	130	1 %

Une différence entre les deux mémoires est notée. Elle est due à l'outil de synthèse. En effet, la vitesse des mémoires varie (en décroissance) en fonction de leurs tailles. Il faut préciser que la mémoire temporaire ne contient que trois entêtes qui occupent au total 128 octets alors que la mémoire principale peut contenir 4 paquets, chacun ayant une taille maximale de 1048 octets. Il faut préciser que l'outil de synthèse pour créer une mémoire multiport avec un port d'écriture et plusieurs ports de lecture, génère plusieurs mémoires à un port de lecture et un port d'écriture : RAM dual port. L'outil a généré 3 mémoires, comme le montre la Figure 3-18. Celles-ci contiennent chacune des RAMs de type RAM_B4_S4 et elles sont de taille égale.

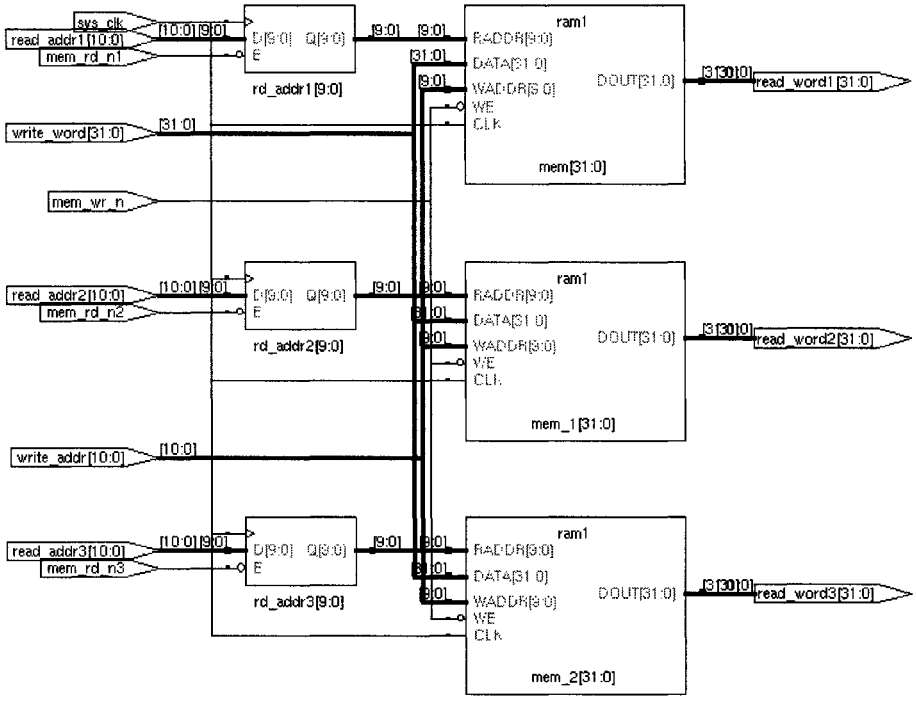

Figure 3-18. Mémoire principale générée par Synplify

Nous avons placé la mémoire principale dans les blocs RAM du FPGA, comme le montre la Figure 3-19. Malheureusement, il n'y avait plus d'espace pour la mémoire temporaire, mais comme nous l'avons vu, cette dernière nécessite peu de LUTs.

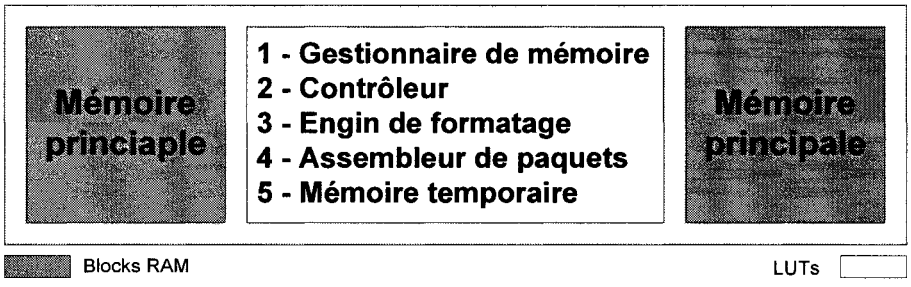

Figure 3-19. Placement des modules dans le FPGA Xilinx Virtex XCV1000

3.4.2 Performances de l'architecture matérielle

Étant donné que l'architecture est déjà subdivisée en fonction des traitements (stockage, formatage et assemblage), le temps de stockage (5) des données comprendra le temps de réception des paquets plus la latence du gestionnaire de mémoire et celle du contrôleur

(6). Le temps de formatage comprend juste le temps d'exécution de l'algorithme de formatage. Ce temps sera constant quelle que soit la taille du paquet. Enfin, le temps d'assemblage (7) sera constitué de la latence due à l'assembleur de paquets (8) et au contrôleur de MAC. Les formules de calcul de ces différents temps de traitements sont donnés ci-après :

$$Tstock = Tréc + Tbloc1 \quad (3.5)$$

$$Tbloc1 = Tmm + Tctr = \frac{3}{MMFréquence} + \frac{3}{CTRFréquence} \quad (3.6)$$

$$Tassem = Tpa + Tren \quad (3.7)$$

$$Tpa = \frac{3}{PAFréquence} \quad (3.8)$$

$$\mathbf{Tfor_{1394->802.3} = 69 / Fréquence\ du\ GF} \quad (3.9)$$

Noter que les formules de calcul n'intègrent pas les temps de traitement de l'identificateur de protocoles et celui du générateur de checksum. Le temps de l'identificateur n'a pas été considéré car, dans le cas de cette conversion, le système ne reçoit que des datagrammes IPv4. De plus, les vérifications sont effectuées par l'engin de formatage. Les interfaces physiques Firewire et Ethernet effectueront le calcul de checksum, raison pour laquelle le temps de traitement du générateur de checksum n'a pas été considéré.

Comme le montre ces formules, l'architecture matérielle n'accélère pas les opérations d'entrée/sortie. Ces formules ont permis de calculer les débits absolus (incluant les entêtes) de l'architecture matérielle selon la taille des paquets à recevoir. Le Tableau 3-5 donne ces débits. On peut remarquer que le débit maximal atteint une vitesse de 318 Mbps pour un petit paquet de 80 octets. Ce débit est considérable quand on sait que celui de l'architecture logicielle atteint à peine 14,75 Mbps pour un paquet de 80 octets. Il faut constater l'accélération du temps de formatage, qui a permis d'atteindre de telles performances. Ce temps est tellement minime que le goulot d'étranglement du système est maintenant dû au temps de stockage, d'assemblage et de renvoi. Ici, les paquets

prennent maintenant plus de temps à être reçus, stockés, assemblés et retransmis, contrairement à la solution logicielle.

Tableau 3-5. Temps de conversion de l'architecture matérielle

Cas (pkt_size)	Temps Trait.	Tstock. (%)	Tassem. (%)	Tfor. (%)	Vitesse (Mbps)
Firewire à Ethernet					
Pire (80 o)	2.01 µs	30.95	23.38	45.66	318.9
Moyen (252 o)	4.14 µs	46.49	31.35	22.14	154.66
Meilleur (1048 o)	14.14 µs	56.22	37.30	6.47	45.23

L'architecture matérielle accélère le temps de formatage, qui était le goulot d'étranglement de la solution logicielle. La Figure 3-20 montre les variations de débits selon la solution. Contrairement à la solution logicielle, le débit décroît avec l'augmentation de la taille des paquets. Cela s'explique par le fait que la plate-forme ne commence à formater un paquet qu'à la fin de la réception de ce dernier, donc plus le paquet est gros, plus les temps de réception et d'assemblage seront élevés. Les temps de stockage et d'assemblage deviennent prédominants. Le formatage n'occupe plus que 6.47 % du temps total de conversion pour un paquet de 1048 octets. Ces différents temps diminuent considérablement le débit de l'architecture matérielle pour de gros paquets.

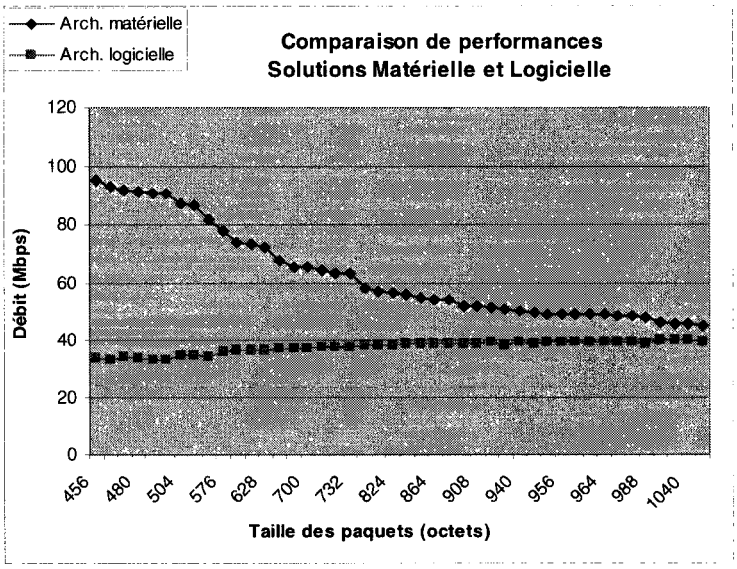

Figure 3-20. Variation du débit de l'architecture matérielle

Le débit de la solution matérielle s'approche de celui de la solution logicielle pour de gros paquets. Cela s'explique par le fait que le débit moyen dû au temps de formatage de la solution logicielle augmente avec la taille des paquets, alors que celui de la solution matérielle est tellement minime qu'il n'a pas d'impact significatif sur son débit total. Ceci s'explique par le manque de parallélisme de l'architecture matérielle. Il faut rappeler que cette dernière ne traite qu'un paquet à la fois, tandis que la solution logicielle grâce à la présence d'unités DMA, permet d'effectuer les traitements de réception, formatage et renvoi s'effectuent en parallèle et en même temps.

3.4.3 Améliorations et Discussion

Durant la révision de cette architecture, nous avons eu à apporter quelques modifications à cette dernière, afin qu'elle soit capable de traiter un flot de données. La première modification concerne le contrôleur et l'assembleur de paquets. L'assembleur de paquets n'avertissait pas le contrôleur afin que ce dernier puisse mettre à jour la table d'allocation d'adresses. Donc, dès que le convertisseur recevait 4 paquets, le système était bloqué et il rejetait tous les nouveaux paquets venant de l'interface d'entrée. Pour régler ce problème, nous avons rajouté une liaison entre le contrôleur et l'assembleur de paquets.

La seconde modification concerne le générateur de checksum. Durant l'étude des interfaces physiques, nous avons noté que ces derniers effectuaient déjà le calcul des checksum des paquets Firewire et Ethernet qui sont basés sur l'algorithme du CRC32. Aussi, nous avons proposé de retirer ce générateur de l'architecture et d'envoyer les données de l'assembleur de paquets vers l'interface de sortie directement.

Une autre proposition concerne le module d'identification de protocoles. Ce dernier commence l'identification à la fin de la réception du paquet. Il serait utile que le processus d'identification démarre durant la réception du paquet. Ainsi, dès que l'identificateur de protocoles détectera un problème sur le paquet (protocole ou le type du paquet incorrect), le paquet pourra être rejeté durant la réception, ce qui permettra d'économiser du temps et des ressources.

Le dernier problème concerne le contrôleur. En effet, ce dernier est très sensible par rapport aux modifications qui peuvent être apportées à certains modules de l'architecture. De plus, son rôle est vital dans celle-ci. Il serait utile de partitionner l'exécution de certains traitements du contrôleur en logiciel et en matériel. Par exemple, la gestion des adresses dans la mémoire principale et l'attribution des tâches de l'engin de formatage seraient exécutées en partie par le processeur ARM d'une part et le reste par le contrôleur matériel.

D'autres propositions ont été émises, telles que le remplacement de la mémoire multiport par une seule mémoire à deux ports afin d'augmenter la capacité de stockage du convertisseur. On peut noter que la mémoire principale serait une mémoire simple à un seul port de lecture et un autre port d'écriture. Elle serait branchée sur un BUS. Ainsi, tous les modules y accéderaient. La mémoire temporaire a été supprimée, les entêtes seront maintenant contenues dans la mémoire principale.

Nous proposons aussi que le générateur de checksum soit un co-processeur de l'engin de formatage. Il effectuera le calcul de checksum des protocoles de couches supérieures tels que TCP ou IP.

En appliquant les propositions qui viennent d'être émises, nous obtenons l'architecture de la Figure 3-21. Cette nouvelle architecture permettra de palier le manque de parallélisme de l'architecture v2 et allégera les tâches de Contrôleur et rendra les co-processeurs moins dépendants du Contrôleur. Cette architecture permettra d'effectuer les tâches de d'identification, de réception, de formatage, d'assemblage et de renvoi simultanément.

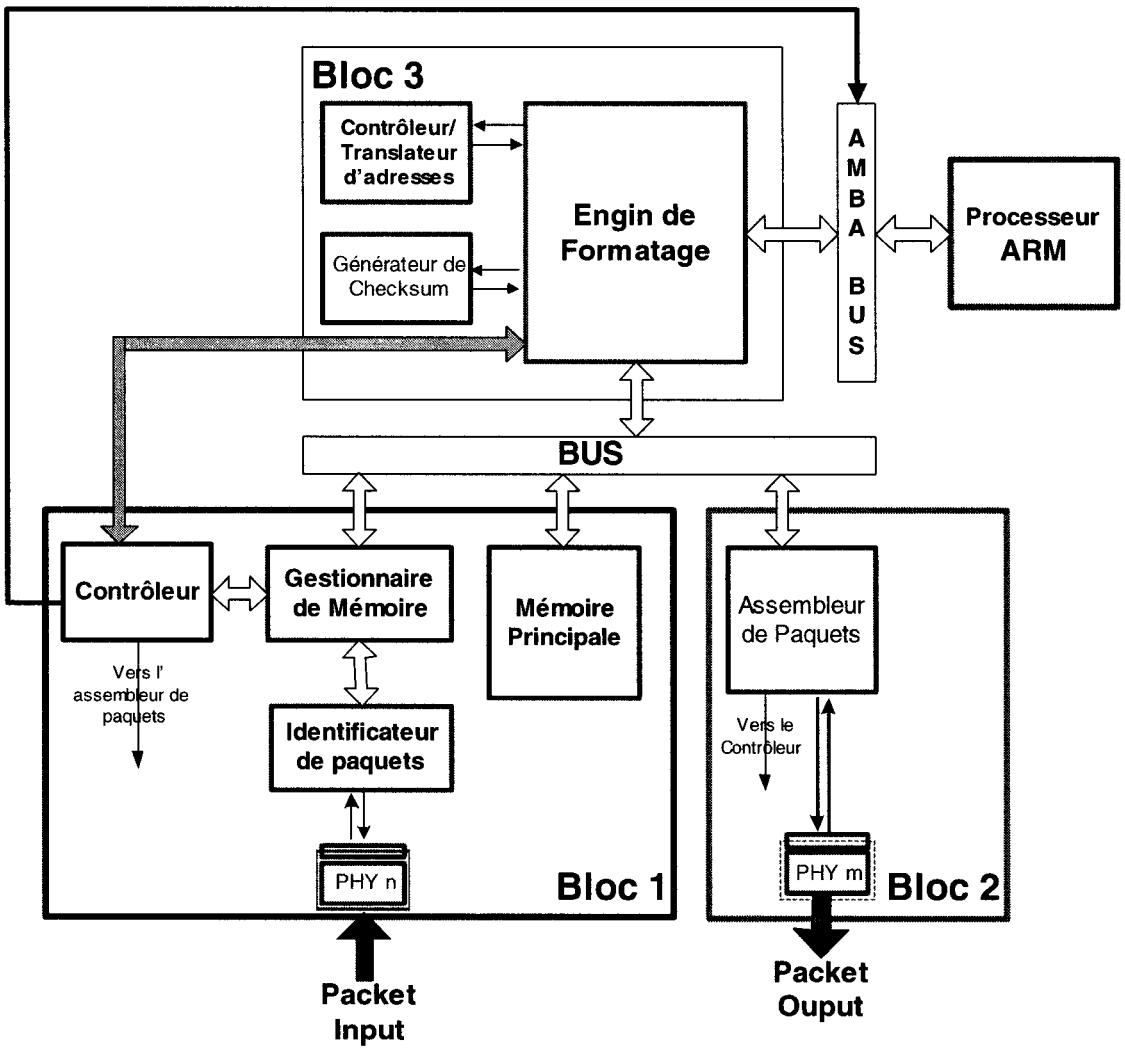

Figure 3-21. Proposition d'une nouvelle architecture pour la conversion de protocoles

CHAPITRE 4

TRANSPORT DE DONNEES AUDIO/VIDEO

Dans le Chapitre 3, nous nous sommes focalisés sur le traitement de données brutes en négligeant les aspects de routage de la conversion Firewire-Ethernet. En fait, dans les réseaux locaux TCP/IP, il y a un autre protocole qui entre en jeu pour la résolution des adresses : c'est ARP. Dans le cas du transport de datagrammes IPv4 dans un réseau Firewire, le RFC 2734 propose aussi une implémentation de ARP, raison pour laquelle nous n'avions pas insisté sur la résolution d'adresses, puisqu'une solution existe déjà.

Pour transporter des données AV d'un équipement Firewire vers un réseau TCP/IP, la résolution d'adresses n'est pas aussi simple que cela, étant donné que les équipements AV n'implémentent pas TCP/IP. Ces derniers s'attendent à recevoir un paquet Firewire dans lequel est encapsulé des données de contrôle. Lorsqu'il y a plusieurs équipements AV dans le réseau Firewire, comme illustré à la Figure 4-1, il faudrait que les PC inclus dans le réseau Ethernet soient capables de communiquer de manière transparente avec une *WebCam*, par exemple.

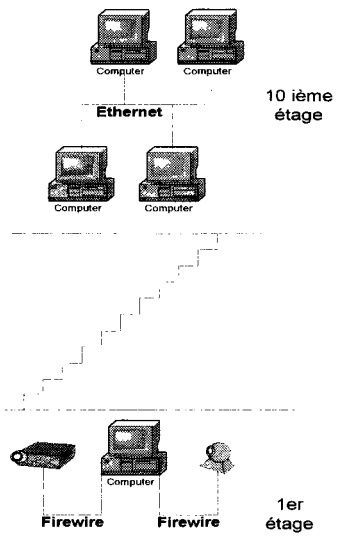

Figure 4-1. Exemple d'application AV

Le problème qui se pose concerne la multitude de protocoles gérés par les différents équipements. La Figure 4-2 montre, à titre d'exemple, les empilements de protocoles qu'une passerelle pourrait avoir à traiter. Cette dernière devra prendre la place des équipements Firewire pour recevoir les messages venant des PC et les renvoyer à l'équipement AV adressé et vice versa. Dans cet exemple, on voit que la passerelle devra gérer des connexions TCP-UDP/IP, étant donné qu'ils sont des protocoles standards dans les LAN. Le protocole le plus populaire sur Internet pour transporter des données AV est RTP (Real Time Protocol), cependant ce dernier peut être remplacé par un protocole privé.

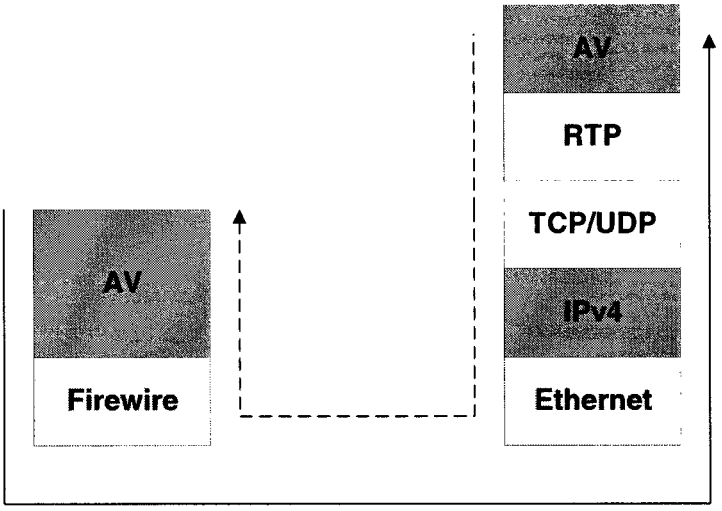

Figure 4-2. Empilement de protocoles d'une caméra numérique à un PC

Pour interconnecter ces deux types de réseaux, il faut résoudre les problèmes d'adressage et les différences entre les formats de données de chacun des protocoles. Donc, dans la suite de ce chapitre, nous proposons une méthode d'adressage permettant d'attribuer à chaque équipement Firewire une adresse IP, ce qui permettra de les identifier uniquement en faisant abstraction de leur adresse physique Firewire. Ensuite, l'architecture du système complet pour le transport de données AV sera exposée afin de montrer comment le module de transport de données AV communique avec le module de résolution d'adresses.

Enfin, les performances du système au complet seront analysées pour montrer le temps associé à chacun de ses modules et le débit de la passerelle

4.1 Algorithme de résolution d'adresses pour l'interconnexion Firewire-Ethernet

Étant donné que la solution logicielle sera déployée sous Linux, il faudrait comprendre l'architecture d'un système d'exploitation. L'architecture du *Kernel* de Linux (Tanuan, 1998) est illustrée par la Figure 4-3. Comme nous le constatons, le *Kernel* comporte différents gestionnaires, chacun étant spécialisé pour fournir un service unique. Le gestionnaire qui nous concerne est le gestionnaire de services réseaux. Ce dernier est accessible via l'interface d'appel système. De plus, les services offerts par le gestionnaire réseau sont prédéfinis par les pilotes TCP/IP dans ce cas.

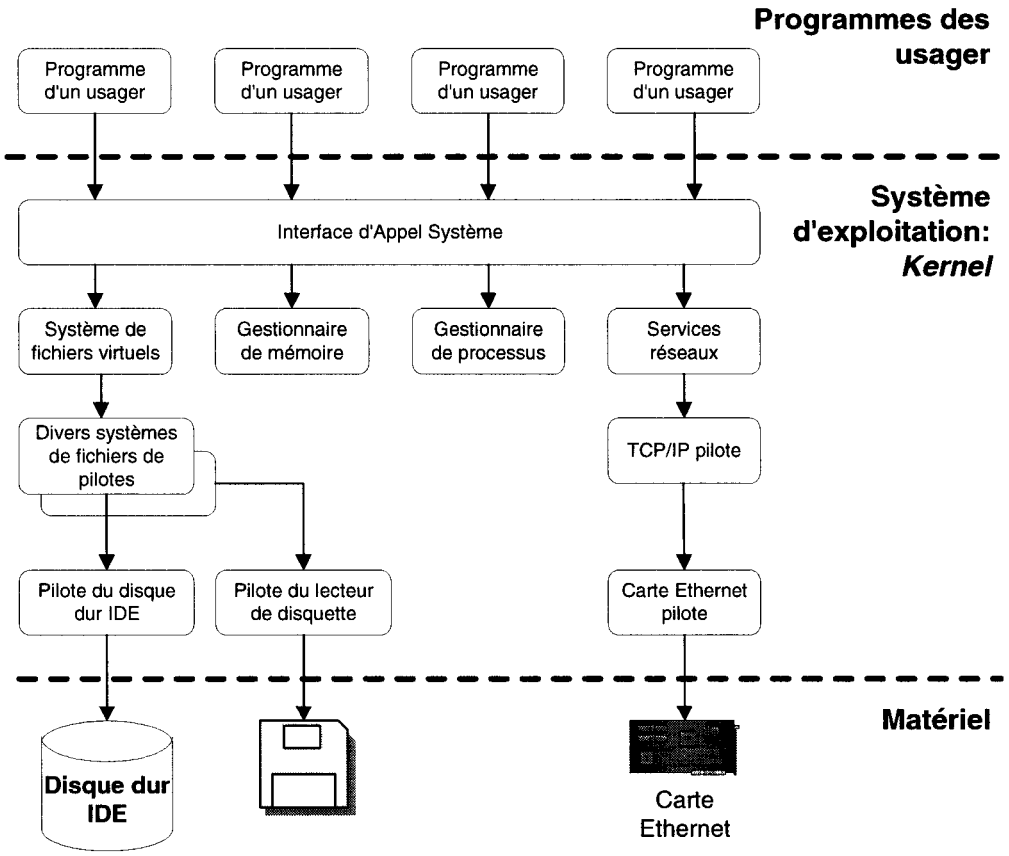

Figure 4-3. Architecture générale du *Kernel* de Linux

Donc pour effectuer le routage de paquets d'un réseau Ethernet vers Firewire, la passerelle devra utiliser des fonctions fournies par les pilotes TCP/IP et rajouter des fonctionnalités au gestionnaire de services réseaux. Il faut retenir qu'un pilote FireoverIP est inséré dans le *Kernel* de la passerelle comme le montre la Figure 4-4. Ce pilote comportera toutes les nouvelles fonctions nécessaires à la retransmission des paquets et toutes les tables d'adresses qui seront utilisées pour le routage des paquets.

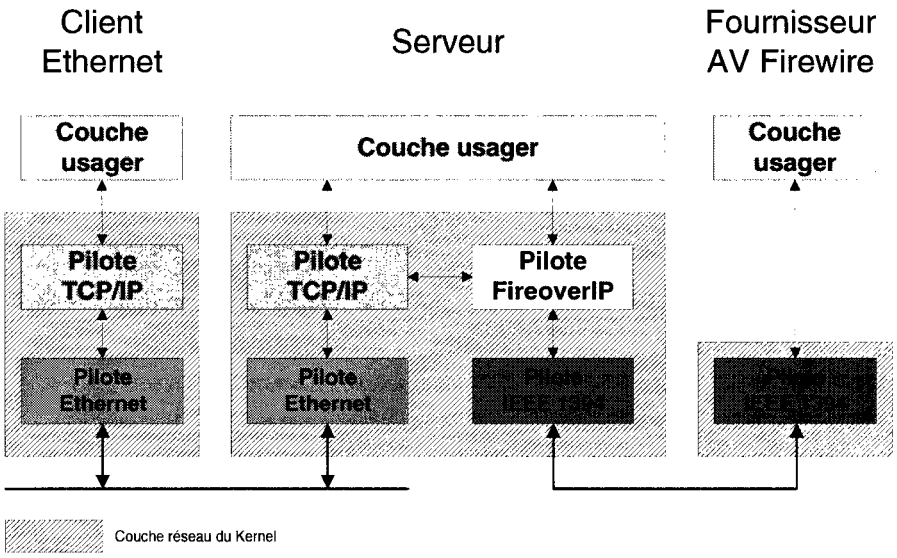

Figure 4-4. Place de la passerelle dans l'architecture d'un système d'exploitation

Le paragraphe qui suit donne une brève description des tâches du module d'adressage. Elles peuvent être décrites comme suit :

1. La première tâche du pilote FireoverIP consiste à collecter les adresses physiques de tous les équipements Firewire et à les stocker dans sa table de correspondance (elle associe une adresse physique Firewire et une adresse IPv4) ;
2. La seconde tâche consiste à attribuer une adresse IPv4 à chaque adresse physique Firewire, les adresses IPv4 sont transmises par la couche usager, la Figure 4-5 illustre l'état virtuel des deux réseaux après l'allocation des adresses IPv4 aux équipements Firewire ;

3. Après l'allocation, la passerelle est capable de répondre aux requêtes ARP ou UDP destinés aux équipements Firewire et d'accepter les demandes de connexions TCP.

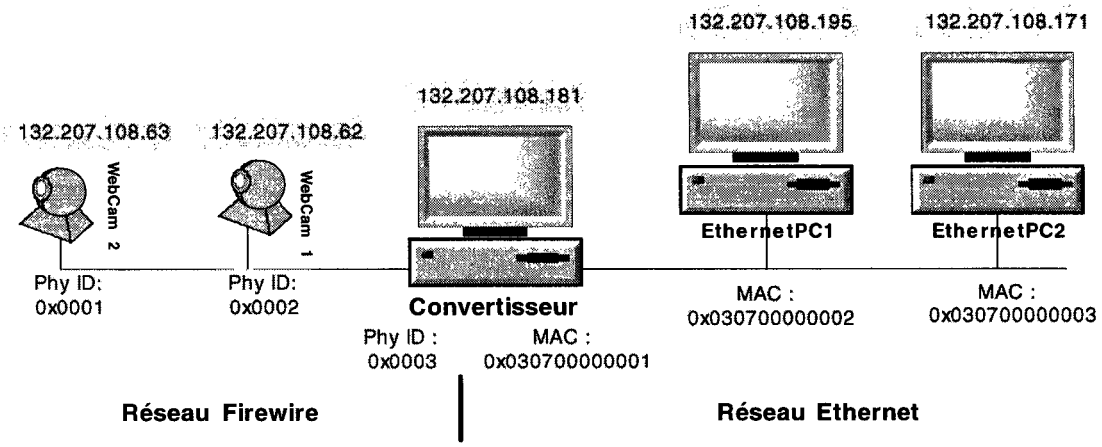

Figure 4-5. État des réseaux après l'étape d'initialisation

L'Annexe E peut être consultée pour voir plus en détails les différentes tâches du module d'adressage.

Il faut préciser que chaque équipement Firewire comporte un numéro de série, donc les clients du réseau Ethernet devront envoyer des requêtes de lecture pour consulter les registres internes des équipements Firewire afin de les identifier et de connaître leurs caractéristiques.

4.2 Algorithme pour le transport de données AV

Il y a deux types de trafic qui transitent entre les différents réseaux : contrôle et données brutes. Le trafic de contrôle sera contenu dans des paquets TCP comme le montre la Figure 4-6. Ces paquets TCP véhiculeront généralement des requêtes ou des réponses de lecture ou d'écriture, alors que les paquets UDP transporteront des flots vidéo à proprement parler.

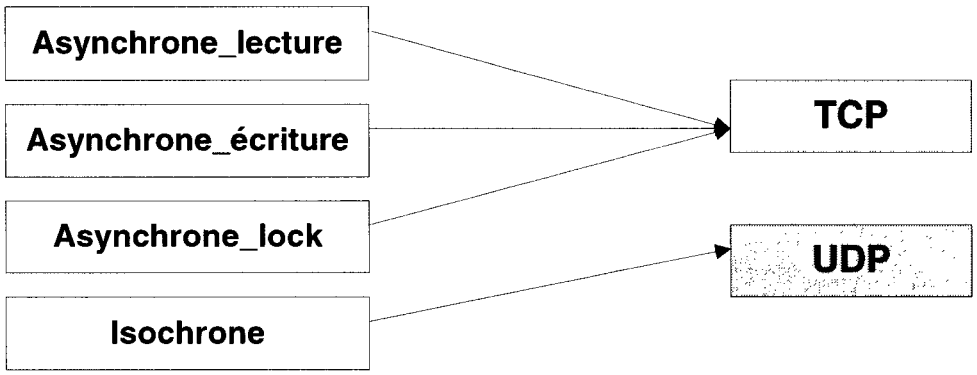

Figure 4-6. Correspondance entre les trafics et les types de paquets

Ce choix s'explique facilement. En effet, TCP (Postel et al. Aug 1980) offre à peu près les mêmes services que ceux supportés par les paquets Firewire asynchrones. Par paquet asynchrone, on entend les paquets de types : lecture, écriture ou lock. Ces derniers renferment un numéro de transaction qui peut être comparé au numéro de séquence contenu dans le paquet TCP. Ils nécessitent l'envoi d'un accusé de réception simultané dès la réception du message, si ce dernier n'est pas reçu à temps, le paquet sera renvoyé comme le fait TCP. Si un champ du paquet ne respecte pas certaines règles; par exemple, si l'adresse du registre est incorrecte, un message d'erreur sera envoyé à la source.

Les paquets isochrones, qui renferment généralement des données AV, nécessitent peu de contrôle. Les services requis par les paquets isochrones sont identiques à ceux offerts par le protocole UDP. Par exemple, il est prévu d'effectuer une simple vérification de l'intégrité du paquet en calculant son checksum et si ce dernier ne correspond pas à celui contenu dans le paquet, le traitement de ce paquet est interrompu.

4.2.1 Couche virtuelle : FireoverIP

Nous avons expliqué précédemment qu'une caméra est mise à jour grâce à un mécanisme d'écriture dans ses registres internes. Les registres sont identifiés par une adresse de 48 bits. Pour consulter ou modifier les paramètres de la caméra, le client TCP devra préciser cette adresse dans le paquet envoyé. C'est une des raisons pour lesquelles nous avons utilisé la technique de complémentation, pour interconnecter un réseau TCP/IP à un réseau Firewire qui comporte des équipements AV, comme le montre la Figure 4-7.

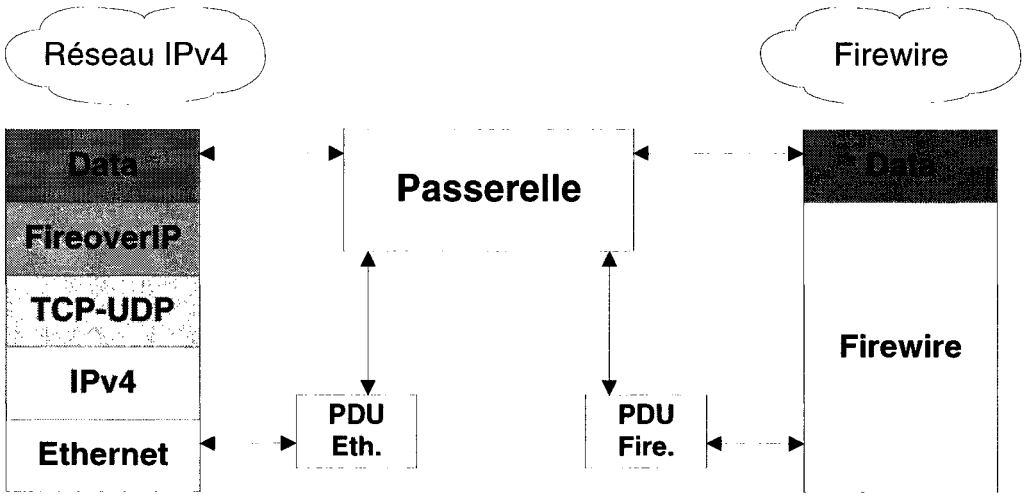

Figure 4-7. Protocoles présents durant l'interconnexion de LAN TCP/IP et Firewire

La couche virtuelle FireoverIP comporte toutes les informations facilitant le transfert d'une requête venant du réseau Ethernet vers l'équipement Firewire adressé. Le format des informations associées à cette couche est illustré à la Figure 4-8.

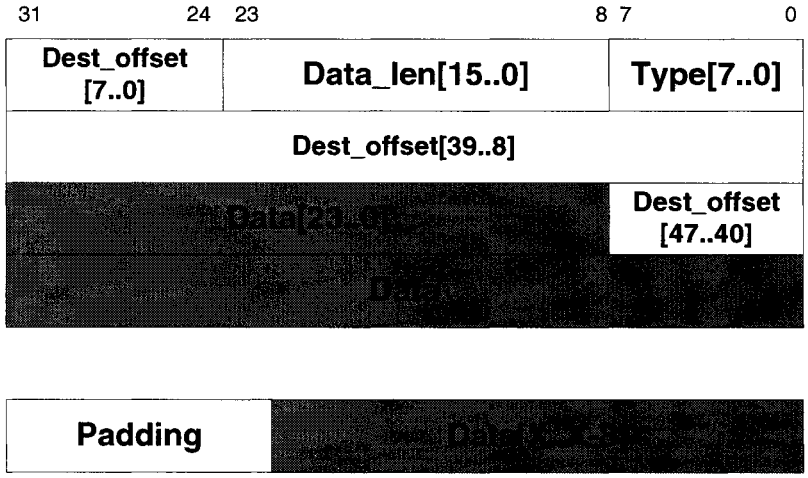

Figure 4-8. Format de l'entête de la couche virtuelle FireoverIP

La passerelle doit juste vérifier la validité de certains champs de la couche virtuelle et ensuite interpréter son contenu pour savoir s'il s'agit d'une requête. Les contrôles effectués sur l'entête de la couche virtuelle sont explicités dans la suite de ce chapitre.

Les champs de la couche FireoverIP sont :

- ***Type :***

Ce dernier renferme la nature du paquet à retransmettre. Il indique s'il s'agit d'un paquet asynchrone d'écriture ou de lecture, ou d'un paquet isochrone. Il est codé sur 8 bits.

- ***Data_len :***

Ce champ précise la taille de la charge utile du paquet, à savoir la taille effective en octets du champ « data ». Il est composé de 16 bits.

- ***Dest_offset :***

Il indique l'adresse du registre sur lequel l'opération de lecture ou d'écriture doit être effectuée. Il comporte 48 bits.

- ***Data :***

Ce champ contient la vraie charge utile du paquet transmis. Il comportera les octets à écrire dans le CSR, ou les résultats de lecture, ou les données AV quand il s'agira d'un paquet isochrone. Du rembourrage peut être effectué si la taille en bits de la charge utile n'est pas un multiple de 32.

Maintenant que le format des données qui s'échangent est établi, une description de l'architecture globale du système sera donnée.

4.2.2 Architecture globale du système

Une architecture de type *multi-thread* a été adoptée comme le montre la Figure 4-9. Un *thread* que nous nommerons serveur est associé à chaque équipement Firewire et c'est ce dernier qui établit les connexions TCP avec les clients du réseau Ethernet. Les *threads* se partagent l'accès sur la carte Firewire de la passerelle.

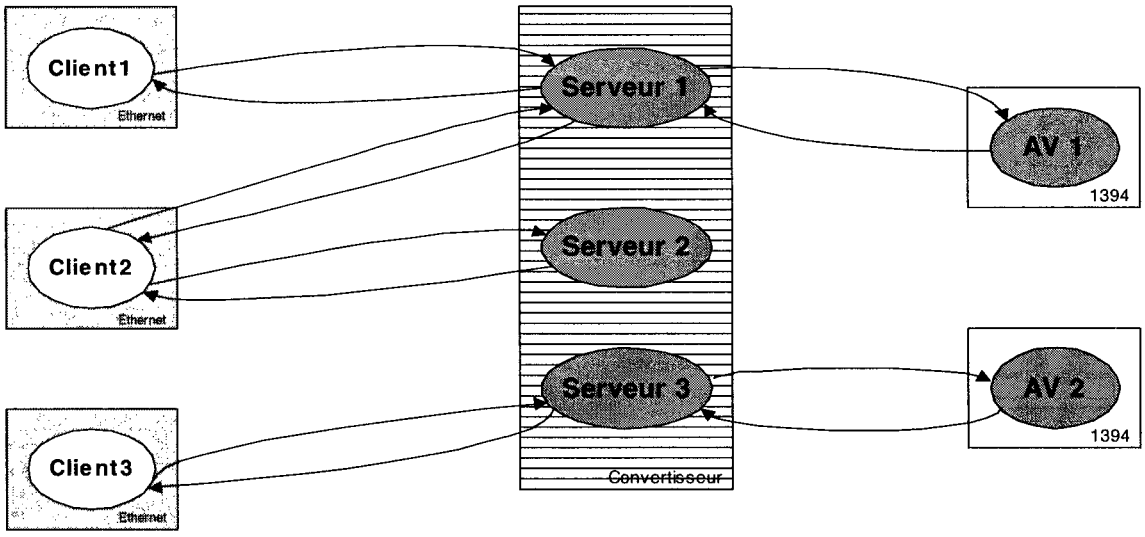

Figure 4-9. Architecture du système pour le transport de données AV

Les serveurs écoutent sur des ports différents les demandes de connexion provenant des clients Ethernet. Après avoir établi une connexion, le serveur peut commencer à exécuter les requêtes d'écriture ou de lecture provenant du client. Le nombre de serveurs dépend du nombre d'équipements Firewire branchés sur le bus et tous les serveurs exécutent le même code exécutable. Un *thread* est aussi associé à la passerelle. En effet, ce dernier est considéré comme un équipement Firewire à part entière. Dans certains cas, la passerelle pourra jouer le rôle du gestionnaire de ressources isochrones. Dans ce cas, les clients Ethernet devront communiquer avec ce serveur afin d'effectuer les opérations de réservation de canaux isochrones ou de bande passante. Chaque *thread* peut envoyer sur un port UDP donné les flots AV envoyés par la caméra, comme nous allons le voir dans la suite de ce chapitre.

4.2.3 Architecture de la passerelle

L'architecture globale du système d'interconnexion a été définie dans la section précédente. Dans celle-ci, nous allons décrire le fonctionnement de la passerelle. Nous commencerons par décrire son processus d'initialisation, les fonctionnalités du *thread* serveur, et enfin les mises à jour internes à effectuer en vue de retransmettre les flots AV.

4.2.3.1 Initialisation du module d'adressage (Couche usager)

La Figure 4-10 montre les différentes étapes du processus. Ce dernier commence par consulter le nombre d'équipements Firewire connectés sur le bus Firewire sur lequel la passerelle est branchée. Ensuite, les adresses de tous ces équipements, excepté la passerelle, sont mises à jour. Pour cela, la passerelle effectue une interruption pour appeler une fonction du pilote FireoverIP : UDPDATE_ENTRY. Un *thread* : serveur_1394, est alors créé, ce dernier joue le rôle d'un serveur associé à cet équipement Firewire. Les fonctionnalités de ce *thread* sont explicitées à l'Annexe F.

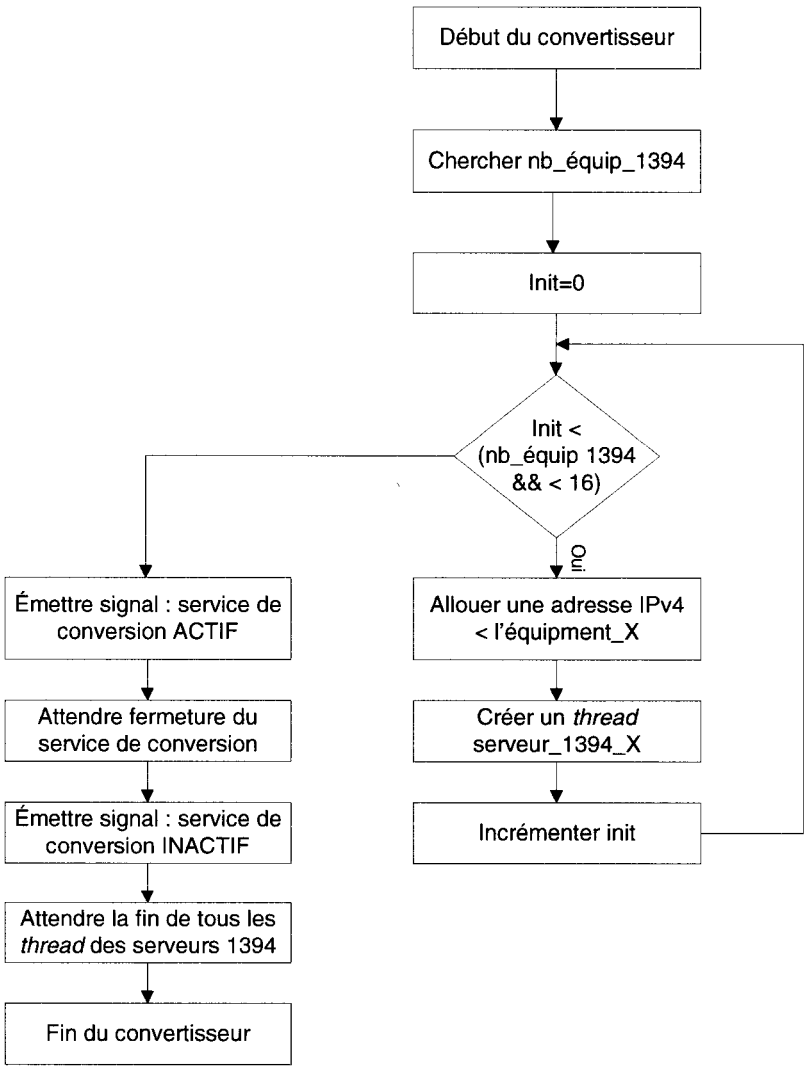

Figure 4-10. Organigramme de la passerelle

Le nombre d'équipements a été limité à 16 pour l'instant. Cette restriction a été imposée afin de ne pas saturer le système avec un nombre de *threads* trop élevé. L'allocation des adresses IPv4 se fait de manière statique. Pour ce faire, l'administrateur fournit des adresses IPv4, et il doit s'assurer de la validité et de l'unicité de celles-ci dans le LAN IPv4 courant.

Après la mise à jour des adresses IPv4 de ces équipements, un signal *sig_serveur_start* est émis pour signifier que le service d'interconnexion est maintenant actif et que les *threads* créés auparavant peuvent maintenant accepter des connexions.

L'administrateur du système pourra arrêter le service d'interconnexion à n'importe quel moment. Dès que ce dernier voudra l'interrompre, un signal *sig_serveur_stop*, sera émis pour que tous les *threads serveur_1394*, s'arrêtent normalement. La passerelle collectera le résultat de tous les *threads* et ensuite le service sera fermé.

4.2.3.2 Retransmission des flots AV

Avant de commencer la retransmission des flots AV, la passerelle doit détecter les ordres de mise à jour suivants :

- Initialisation du numéro de canal sur lequel l'équipement Firewire émettra les flots AV ;
- Début de la transmission des flots.

Ces deux ordres sont transmis en envoyant des ordres d'écriture sur les registres : REG_CAMERA_ISO_DATA et REG_CAMERA_ISO_EN.

C'est durant l'étape d'interprétation que la passerelle détectera un de ces ordres. Lorsque la passerelle capte un ordre d'initialisation du numéro de canal, elle effectue une interruption pour mettre à jour la table des canaux implémentée au niveau *Kernel*. Il faut rappeler que cette table permet d'effectuer une correspondance entre un numéro de canal, l'adresse IPv4 de l'équipement qui émet sur ce dernier et la liste des clients qui écoutent

des flots transmis sur ce canal donné. L'interruption permet de rajouter l'adresse du client dans cette dernière liste.

Généralement, si l'initialisation du numéro de canal s'est bien déroulée, le client enverra un ordre d'écriture pour que l'équipement commence l'envoi de flots AV. Dans notre cas, la passerelle doit mettre à jour sa carte Firewire pour que cette dernière écoute et accepte (do_listen) tous les paquets isochrones entrants et associés au canal défini précédemment. Après cette mise à jour interne de la passerelle, elle pourra effectuer des lectures successives sur la file de réception de chaque canal réservé et les paquets lus seront envoyés en mode UDP continu (broadcast) sur le réseau Ethernet.

Pour finir, le client, par une autre opération d'écriture, demandera à interrompre la transmission de flots AV.

4.2.4 Architecture du client

Le client Ethernet qui désire recevoir des flots AV venant d'une caméra doit d'abord effectuer quelques traitements sur cette dernière. Ces traitements consistent à initialiser certaines propriétés de la caméra. Sans ces opérations, la caméra ne sera pas apte à envoyer des données AV. L'Annexe G explicite en détail le comportement d'un client Ethernet désirant recevoir des images venant d'un équipement Firewire.

Maintenant que les modules principaux de notre architecture ont été explicités, nous allons présenter dans la suite de ce chapitre les performances de notre système.

4.3 Résultats

Notre environnement de test, illustré par la Figure 4-11, était constitué des éléments suivants :

- Un premier PC sur lequel le programme de la passerelle était exécuté ; ce dernier ayant bien sûr une carte Firewire ;
- Une WebCam de type ADS Pyro 1394 WebCam, cette dernière offre une résolution maximale de 480x640 avec un taux d'envoi maximal de 30 images par secondes selon le format des images (ADSTechnologies, 2002) ;
- Deux postes qui jouent le rôle de clients Ethernet ; il faut préciser que les clients et bien sûr la passerelle fonctionnent sous Linux (ReadHat 9).

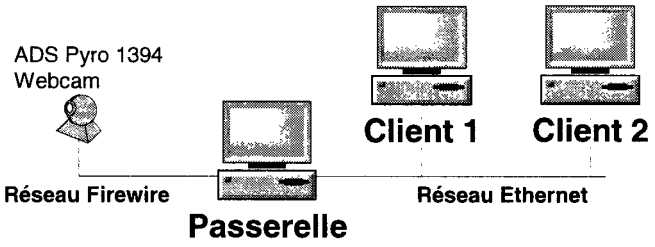

Figure 4-11. Environnement de test de la passerelle

Les tests ont été effectués sur le réseau Ethernet du GRM (groupe de recherche en Microélectronique). Ce dernier a un débit maximal de 100 Mbps. Le but de ces tests était d'observer l'impact de la retransmission des flots AV sur la qualité des images envoyées par les clients. Il faut rappeler que la qualité des images est caractérisée par trois critères : délai, bande passante et taux d'erreurs. Dans notre cas, nous allons évaluer la qualité de l'image selon ces deux derniers critères.

Les performances d'un système dépendent bien sur de son design et de son implémentation. Cependant, les recherches ont montré que les performances observées sont aussi fonction de la charge de travail traitée par le système et des métriques (aussi appelées facteurs) qui influenceront le comportement du système à évaluer (Feitelson, 2003 ; Tanenbaum, 1994). Donc il faut que l'évaluateur puisse établir les facteurs qui influencent le système et qu'ils puissent établir les différentes valeurs (aussi appelées niveaux) de chacun de ces facteurs, afin de ne pas rencontrer de surprise durant la phase de simulation.

Dans la suite de cette section, nous allons d'abord commencer par établir notre plan d'expérimentation. Il s'agit de définir les différentes charges de travail qui seront imposées à notre passerelle, ensuite les différents facteurs qui devraient avoir un effet sur cette dernière seront définis. Par la suite, les résultats de simulation seront présentés.

4.3.1 Plan d'expérience

Pour définir les facteurs qui affectent le comportement du système, il faut établir le chemin de traitement des blocs isochrones, à savoir de leur réception à leur renvoi vers le réseau Ethernet. Ces facteurs dépendent principalement des différents processus qui entrent en jeu durant le traitement d'un bloc isochrone. La Figure 4-12 donne les différents processus de traitement qui manipulent un bloc isochrone de sa réception à son renvoi vers le réseau Ethernet.

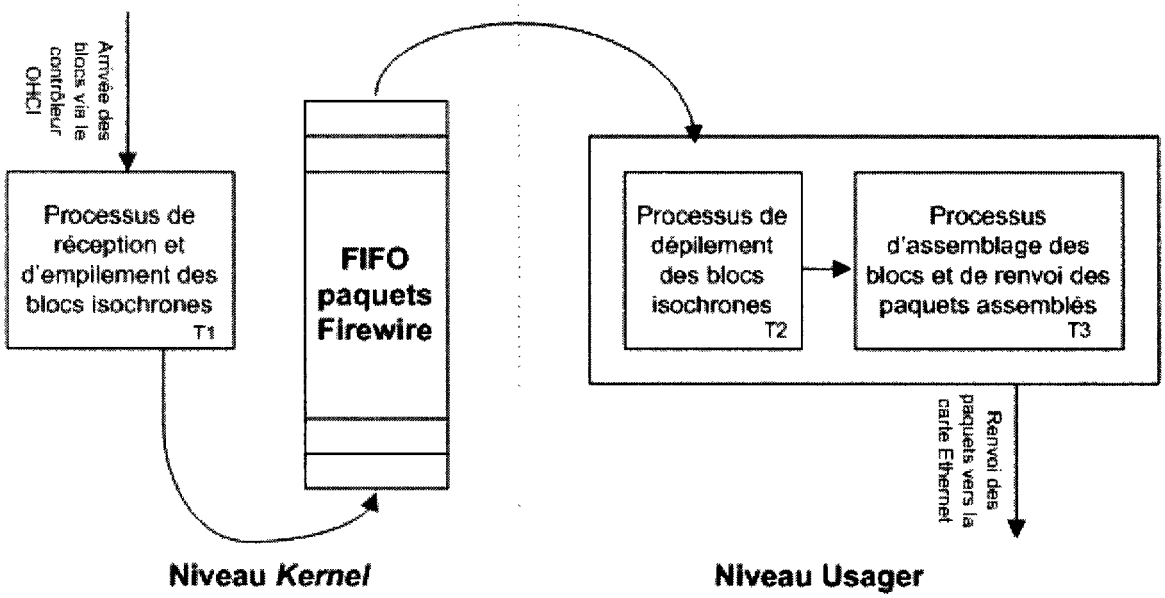

Figure 4-12. Processus de traitements des blocs isochrones

Le premier processus : réception et empilement des blocs isochrones, dépend principalement du taux d'arrivée des données et du temps d'empilement d'un bloc dans la FIFO. La profondeur de la FIFO peut aussi avoir un impact sur ce processus, mais nous négligerons ce facteur afin de simplifier les sessions de simulation.

- Le taux d'envoi des images définit la charge de travail traitée par la passerelle:
    - CT (7.5, 15 et 30 images/s).
- Le temps d'empilement dépendra généralement de la taille des blocs isochrones en sachant que cette taille dépend du taux d'envoi. Les niveaux de ce deuxième facteur : F1, sont :
    - F1(324, 644, 1284).

Nous avons considéré que la charge de travail était constante pour simplifier nos calculs, cependant il faut savoir que la charge de travail de la passerelle dépend aussi des autres processus propres au système d'exploitation et aux autres applications qui s'exécutent sur le poste de la passerelle en parallèle.

Le second processus de *dépilement des blocs isochrones* est affecté par les échanges entre le niveau usager et le pilote Firewire. Le dépilement des blocs nécessite une interruption au niveau usager qui peut coûter très cher comme le rappelle (Tanembaum, 1994). Le facteur associé à ce processus est le même que celui qui affecte le temps d'empilement, à savoir la taille des blocs isochrones.

Le facteur associé au dernier processus d'*assemblage et renvoi des paquets assemblés* (concaténation de blocs isochrones) dépend de la taille des blocs isochrones, mais surtout de la taille totale du paquet. Nous avons établi les niveaux de ce facteur comme suit :

- F2 {324*N1, 644*N2, 1284*N3}

avec N1 [1..202], N2 [1..101] et N3 [1..51].

La taille totale du paquet assemblé est limitée par le protocole UDP en sachant que la taille maximale en théorie d'un message UDP est égale à 65507 octets, ainsi les valeurs limites de N1, N2 et N3 dépendent de cette taille maximale. Par exemple, la limite de N2 est calculée comme suit :

N2 = (65507 - 14 (taille entête FireoverIP)) \ 644 (taille d'un bloc au taux de 15 im/s) = 101.6

Après avoir défini les facteurs affectant le comportement du système, nous pouvons maintenant définir les différentes sessions de simulation. Ces sessions sont établies en faisant varier les niveaux de chacun des facteurs. Nous effectuerons des variations de type hybride, à savoir que nous ferons varier F1 et F2 en même temps, en sachant que F2 dépend de F1. Le Tableau 4-1 illustre la suite des variations des différents niveaux associés à chaque facteur.

Tableau 4-1. Exemple d'un échantillon de simulations

CT	F1	F2	N_X
7.5 images/s	324 o.	324	1
		1620	5
		65548	202
15 images/s	644 o.	644	1
		65044	101
30 images/s	1284 o.	1284	1
		65484	51

4.3.2 Simulation et analyse

Dans la suite de cette section, nous allons présenter les résultats recueillis pour les 3 taux d'images : 7.5, 15 et 30 images/s, en sachant que la caméra envoie des images de type : 480x640, 8 bits/pixel, en tons de gris). Ensuite, nous allons effectuer une comparaison des résultats associés à chacun de ces taux d'images, afin de voir comment les performances varient et si elles se dégradent en fonction de la vitesse d'envoi. Nous caractériserons le pourcentage d'octets perdus en fonction de la taille des paquets.

4.3.2.1 Taux d'images : 15 images/s

Nous avons commencé par ce taux, en sachant qu'en théorie, le taux d'envoi des images nécessiterait au maximum une bande passante de 37,0944 Mbps, ce qui est largement supportable par notre réseau Ethernet.

Le recueil des résultats de simulation consistait à noter la variation de 4 paramètres qui ont un impact sur la qualité de la retransmission des flots AV par la passerelle. Ces paramètres qui varient selon le nombre de blocs isochrones assemblés dans un paquet Ethernet, sont les suivants :

- Le nombre de blocs isochrones reçus et dépilés et le nombre de paquets envoyés ;
- L'occupation des processus de la retransmission : réception, renvoi et assemblage ;
- La vitesse moyenne d'envoi des flots AV ;
- Le pourcentage d'octets perdus durant la retransmission.

4.3.2.1.1 Nombre de paquets envoyés et reçus

La Figure 4-13 montre la variation du nombre de paquets envoyés en fonction du nombre de blocs encapsulés dans chaque message UDP. Pour cette expérience, 64 simulations successives ont été effectuées pour chaque taille de message et chaque envoi de flots AV durait 20 s en moyenne. Pour un taux d'images de 15 images/s, on remarque que la passerelle a de la peine à envoyer les blocs AV les uns à suite des autres. Au départ, quand N est égal à 1, elle envoie au entre 14384 et 22367 messages UDP de taille 660 octets (il ne faut pas oublier la couche virtuelle). Avec l'envoi des paquets isochrones au fur et à mesure qu'ils arrivent, la passerelle ne pourra jamais s'approcher du débit de 15 images/s. Par exemple, le maximum de 22367 blocs AV en 20 s correspond dans le meilleur des cas à un taux d'images maximal de 2,32 images/s (22367/480(=nombres de blocs AV par image)), ce qui est très en deçà des objectifs.

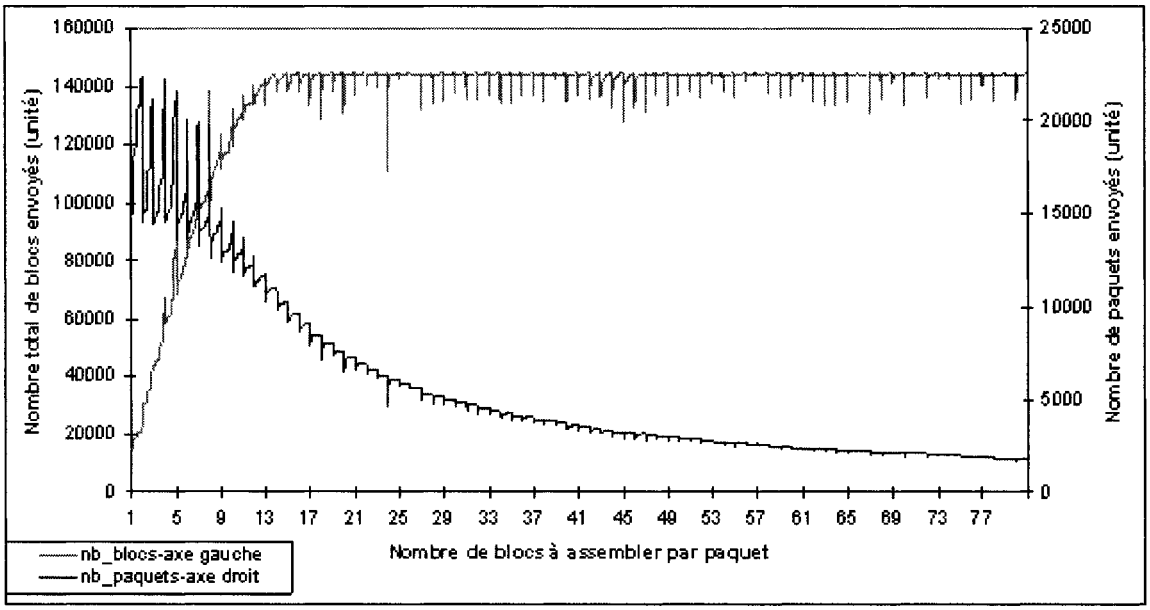

Figure 4-13. Variation du nombre de paquets envoyés pour un taux de 15 images/s

En fait, la passerelle perd trop de temps à renvoyer les petits paquets, donc, elle n'a pas le temps de capturer tous les paquets que lui envoie la caméra. Cependant, quand le nombre de blocs assemblés dans un message UDP est supérieur ou égal à 13, le nombre de blocs envoyés au total se stabilise et plus aucun gain n'est noté, la passerelle capture à peu près le même nombre de blocs AV.

4.3.2.1.2 Occupation des processus de la conversion : réception, renvoi et assemblage

La Figure 4-14 explique le comportement de la passerelle. En fait, étant donné que les processus de dépilement et d'envoi de messages UDP, sont séquentiels, la passerelle n'a pas le temps de dépiler les blocs isochrones, elle passe tout son temps à envoyer les petits messages UDP. C'est quand N est supérieur ou égal à 13 que la passerelle atteint sa vitesse de croisière et qu'elle arrive à dépiler presque tous les paquets contenus dans la FIFO de réception. Comme le montre la Figure 4-15, la tâche d'assemblage des messages UDP est négligeable par rapport aux deux super-tâches.

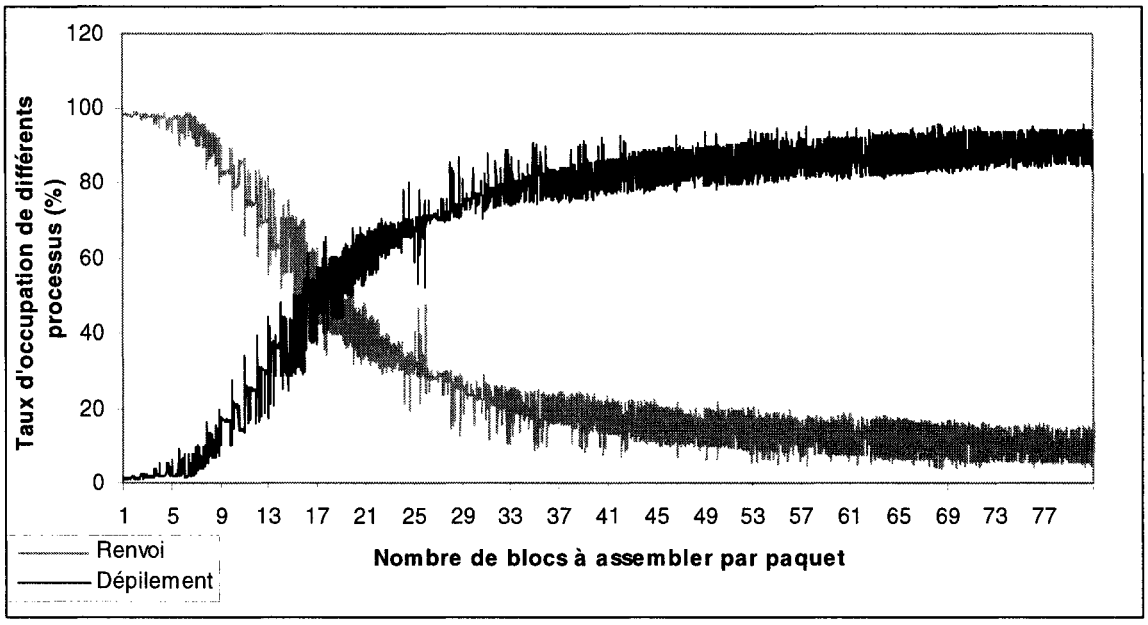

Figure 4-14. Variation de l'occupation des processus de la conversion

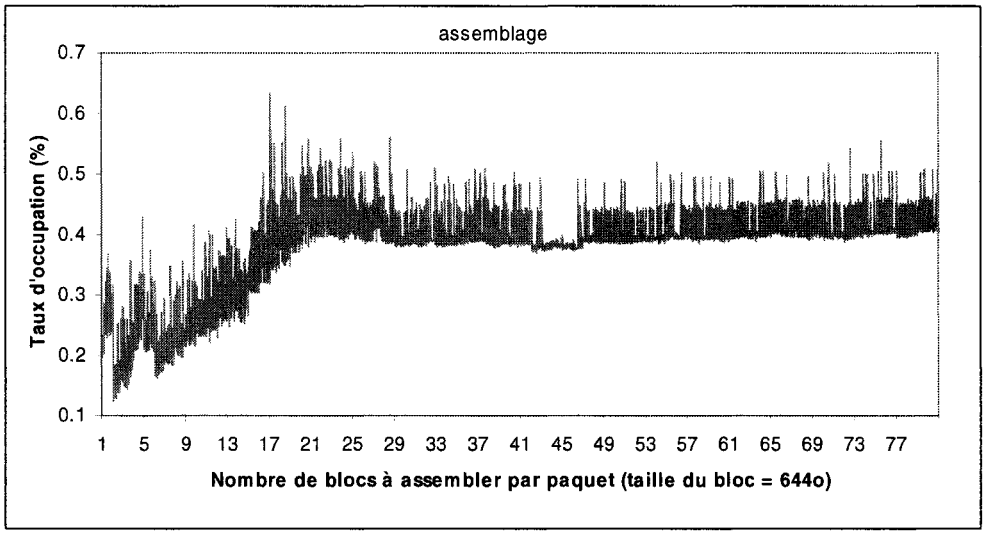

Figure 4-15. Variation de l'occupation du temps d'assemblage

4.3.2.1.3 Vitesse moyenne d'envoi des flots AV

La Figure 4-16 montre la variation du débit d'envoi de la passerelle en fonction du nombre de blocs isochrones assemblés dans un paquet Ethernet. Le débit de la passerelle se stabilise pour atteindre un maximum de 37,14 Mbps. La seconde courbe, sans

conversion, donne la variation de la vitesse de réception des paquets sans le processus de conversion, à savoir sans les opérations de retransmission sur le réseau Ethernet.

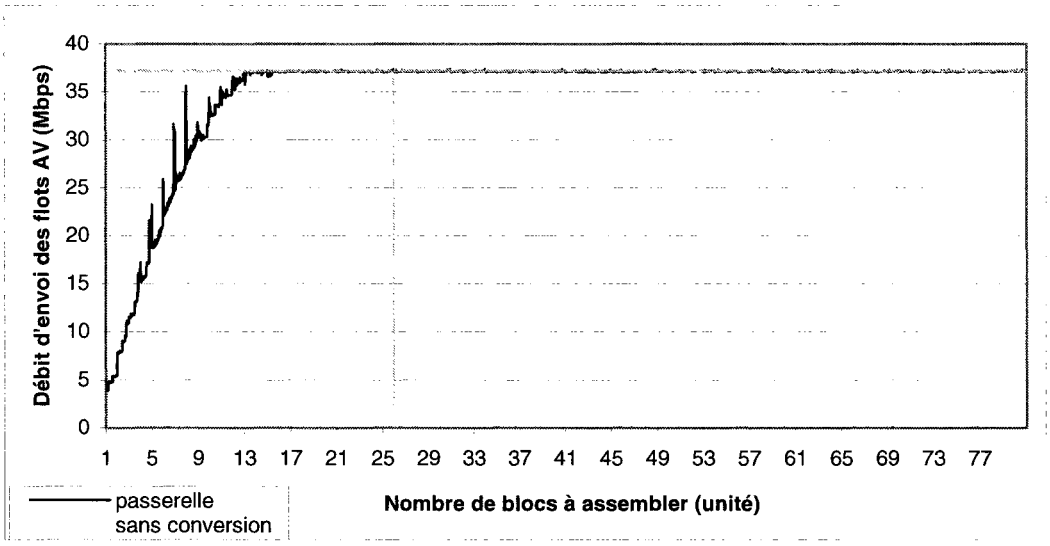

Figure 4-16. Variation de la vitesse moyenne d'envoi des flots AV pour 15 im/s

4.3.2.1.4 Pourcentage d'octets perdus

Le dernier paramètre à évaluer est le taux d'erreurs ou plus précisément le nombre de paquets perdus durant la transmission. La Figure 4-17 donne la variation du pourcentage d'octets perdus durant les simulations. Comme on peut le voir, les taux de perte sont très instables. On remarque que le pourcentage d'octets perdus n'est pas constant en fonction du nombre de blocs par paquet. La différence concerne la taille des messages ; lorsque celle-ci est élevée, le taux augmente considérablement avec cette dernière. Par exemple, le pic observé durant l'envoi d'un message contenant 72 blocs AV avec une perte de 12.39 % des octets envoyés, correspond à la perte de 248 messages UDP, alors que le pic de 11.16 % correspond à une perte de 618 messages UDP. Donc, l'envoi de très gros paquets doit être évité, car le taux de perte augmente rapidement avec la taille des paquets.

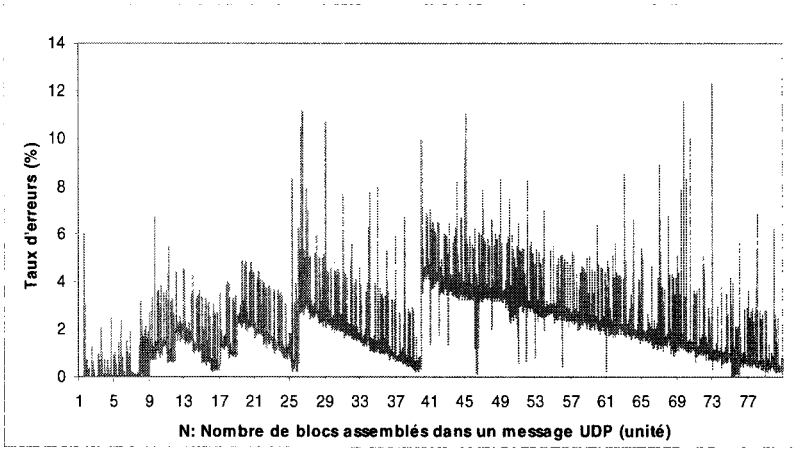

Figure 4-17. Variation du pourcentage d'octets perdus

Mis à part ces pertes dispersées et assez faibles, le nombre moyen de messages UDP perdus est très faible, et cela est même surprenant en sachant qu'en moyenne 10 % des messages sont perdus en mode UDP. On remarque une tendance sur le graphe de la Figure 4-17, nous ne possédons pas d'explication claire à cette tendance et ce sujet mériterait d'être approfondi.

4.3.2.2 Taux d'images : 7,5 et 30

Le passage d'un taux d'images à un autre est caractérisé par l'incrémentation de la taille des paquets isochrones envoyés par la caméra et bien sûr par celle de la vitesse d'envoi et de réception des flots AV. Pour un taux de 7,5 images/s, la taille des paquets isochrones s'élève à 324 o et la vitesse théorique à 18,5472 Mbps. Pour un taux de 30 images/s la taille des paquets isochrones s'élève à 1284 o et la vitesse théorique à 74,1888 Mbps.

En fait, pour ces deux autres vitesses d'envoi, on observe presque les mêmes comportements que ceux qui avaient été notés pour un taux de 15 images/s. Une première différence concerne la vitesse de réception des paquets isochrones. Cette dernière double quand on passe à 30 images/s avec un maximum de 74,008 Mbps.

L'Annexe H inclut les graphes représentant les performances associées à ces deux taux. Le Tableau 4-2 récapitule les performances les plus pertinentes associées à ces taux.

Comme on s'y attendait, la bande passante est divisée par deux par rapport à celle obtenue précédemment (maximum de 37.14 Mbps pour 7,5 images/s).

Tableau 4-2. Récapitulatif des performances notées pour des taux de 7,5 et 30 im/s

	7,5 images/s		30 images/s	
	min	**max**	**min**	**max**
Nombre messages envoyés	739 (145 blocs)	24682 (1 bloc)	2758 (40 blocs)	20571 (1 bloc)
Nombre de blocs envoyés avec un seul bloc encapsulé (octets)	12182 (1 bloc)	145575 (45 blocs)	13210 (1 bloc)	145620 (30 blocs)
Bande passante (Mbps)	1,57 (1 bloc)	18,86 (80 blocs)	6,78 (1 bloc)	74,008 (40 blocs)
Nombre de paquets perdus	-	600 (15 blocs)	-	448 (15 blocs)

4.3.2.3 Synthèse des résultats

Il est intéressant de mesurer les taux d'images effectivement envoyées par la passerelle. La Figure 4-18 présente l'évolution du taux d'images envoyées par seconde selon les trois débits de la caméra.

Pour un taux théorique de 7,5 images/s, la passerelle envoie une moyenne maximale de maximum 7,44 images/s ; pour celui de 15 : 14.9 images/s et enfin pour celui de 30 : 29.9 images/s. Les variations du nombre d'images envoyées par seconde avec un débit théorique de 30 images/s s'accentuent un peu plus par rapport au débit de 15 images/s.

Ces petites différences entre les taux d'envoi théoriques et les taux réels sont sûrement dues au temps d'initialisation de la passerelle. Il faut préciser que la caméra envoie réellement les images selon le taux théorique initialisé, à 30 Mbps par exemple. Il s'agit du temps de démarrage du *thread* de renvoi des flots isochrones, du temps de mise à jour de la carte Firewire, ainsi que du temps de création et d'ouverture d'un canal UDP sur lequel les flots seront émis. Ainsi, même si le client définit une fenêtre de transmission de 20 s, la passerelle a dû effectuer le dépilement durant 19,7 s par exemple.

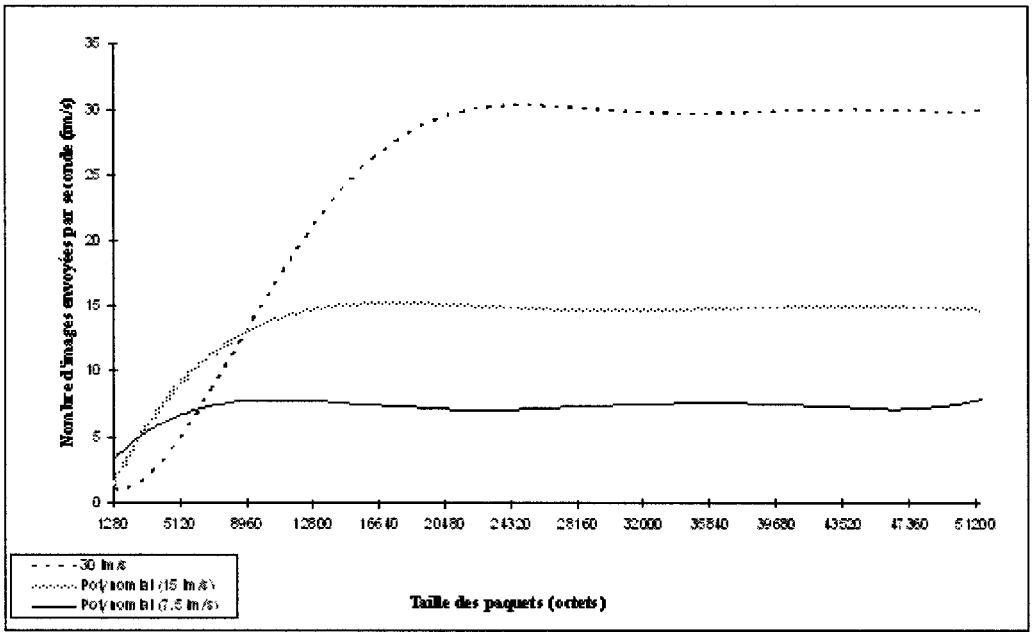

Figure 4-18. Évolution du nombre d'images envoyées

Durant les simulations, nous avons fait varier la taille des messages UDP transmis vers les clients en sachant que cette taille varie selon le nombre de blocs isochrones assemblés dans les messages. Cependant, dans un environnement réel, le client devrait être capable de recevoir un paquet Ethernet de même taille quelque que soit le taux d'envoi de la Webcam. Le graphe précédent permet d'établir la taille minimale des messages UDP à envoyer vers les clients quelque soit le taux d'images envoyées par la ou les WebCam. D'après le graphe, la taille de 20480 octets semble être la taille minimale qui permet d'atteindre les meilleures performances, quelque que soit le débit d'envoi de la WebCam.

Dans l'ensemble, les résultats obtenus sont très satisfaisants. Cependant, il ne faudrait pas se réjouir trop tôt. En effet ces simulations ont été effectuées durant les fins de semaine. Donc le trafic sur le réseau du laboratoire était très faible. Aussi, il faudrait s'attendre à des résultats inférieurs si le réseau est congestionné. Un autre facteur qui pourrait influencer ces bons résultats est le nombre d'applications qui s'exécutent sur la passerelle. Pour l'instant, à part les services habituels du système d'exploitation, peu de processus autres que ceux de la passerelle s'exécutent sur le système utilisé. Si d'autres

applications fonctionnaient sur le système et qu'elles requéraient, elles aussi des accès sur le réseau, il faudrait s'attendre à ce que les performances de la passerelle diminuent. Le dernier facteur est bien sûr la puissance de calcul du système qui exécute la passerelle. Cette dernière s'exécute sur un processeur Pentium IV 2.2 GHz.

4.3.2.4 Modèle de performance

À partir des résultats recueillis, on pourrait dériver un modèle de performance. Ce dernier est basé sur le chemin de traitements précédemment présenté et illustré à la Figure 4-19.

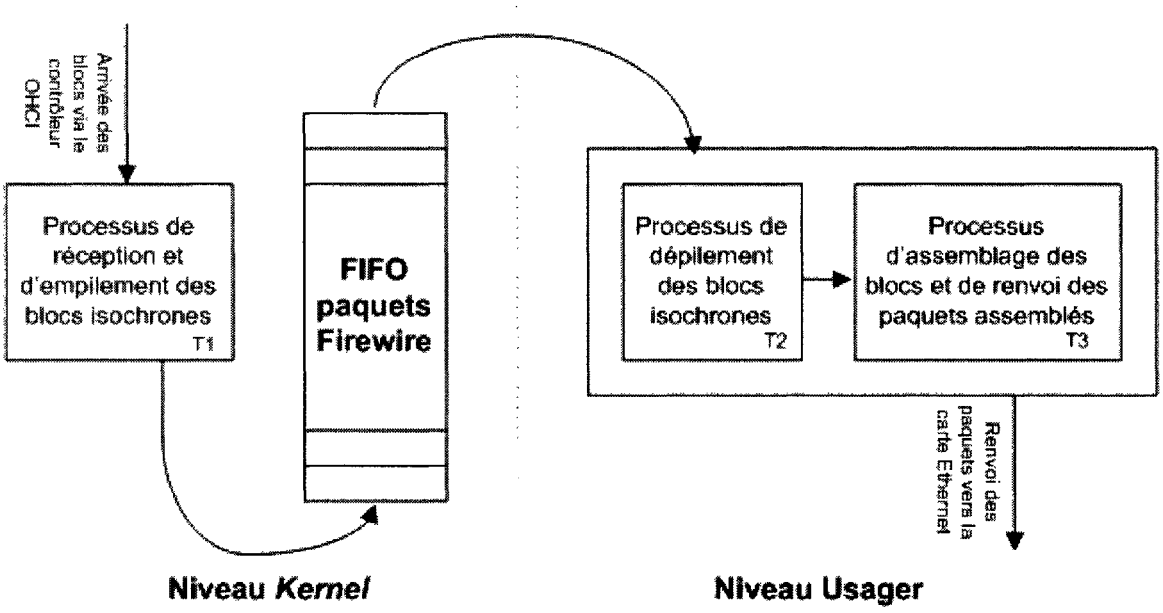

Figure 4-19. Processus de traitements des blocs isochrones

Le débit du système dépend principalement du nombre de blocs isochrones dépilés de la FIFO de réception. Or, on a vu que ce nombre varie selon la taille des paquets assemblés et renvoyés vers le réseau Ethernet. Dans la suite de cette section, nous allons essayer de modéliser le nombre de blocs dépilés, ce qui permettra de trouver le débit moyen de la passerelle en fonction de la taille des paquets assemblés et du taux de réception des blocs. Le temps total de la conversion (égal au temps total d'envoi des blocs isochrones) est défini par le client, ce temps est réparti comme suit:

$$Tot = T2 + T3$$

où

- T2 = temps total de dépilement des blocs isochrones
- T3 = temps total d'assemblage et de renvoi des paquets.

Il est difficile de calculer le temps moyen de réception d'un bloc isochrone. Ce temps dépend de processus concurrents qui n'ont pas les mêmes priorités en termes d'ordonnancement des différents processus. C'est la raison pour laquelle le temps total de réception des blocs isochrones a été réparti entre T2 et T3, en sachant que la priorité du processus de réception est bien plus élevée que celle des deux autres processus.

Le temps de dépilement des blocs isochrones peut être formulé comme suit :

$$T2 = x*T1 + y*(T21+T22)$$

où

- T1 = temps de réception et d'empilement d'un bloc isochrone au niveau *Kernel* ;
- T21 = temps de dépilement d'un bloc fixe ;
- T22 = temps d'attente avant de pouvoir dépiler un bloc ;
- x = nombre de blocs reçus durant le dépilement ;
- y = nombre de blocs dépilés.

Le temps d'assemblage et de renvoi des paquets regroupant des blocs isochrones est formulé comme suit :

$$T3= z*T1 + y* T31 +w* T32$$

où

- T31 = temps d'assemblage d'un bloc
- z = nombre de blocs reçus durant l'assemblage et de renvoi
- w = nombre de messages UDP envoyés qui correspond au nombre de blocs dépilés divisé par le nombre de blocs encapsulés dans un message
    - w = y / N

- T32 = temps de renvoi d'un message UDP assemblé
- N = nombre de blocs encapsulés dans un message UDP

À partir de ces formulations, le nombre de blocs dépilés peut être libellé comme suit :

$$y=\frac{Ttot-LT1}{T21+T22+T31+\frac{T32}{N}}$$

où
L = x+y = nombre de blocs envoyés par la caméra pendant la simulation

Le nombre total de blocs isochrones envoyés par la caméra pendant la simulation au total dépend du temps de la simulation et du taux d'images envoyées par la WebCam. Il faut préciser que le nombre de blocs/s envoyés par une caméra pour des images de 480*640 avec 8 bits par pixel est égal à 7200 blocs/s, en sachant que la taille des blocs varie en fonction du taux d'envoi. Par exemple, pour une simulation de 20 s, avec un taux de 15 images par seconde (la camera envoie 480 blocs isochrones par seconde), la passerelle devrait recevoir 144000 blocs, (ce qui équivaut à : 7200 * 20).

La formule étant définie, il s'agit maintenant de trouver les inconnues : T21, T22, T31, T32. Le temps d'attente, T22, dépend du taux d'envoi des blocs isochrones et des temps T21, T22, T31 et T32. Durant l'envoi de paquets vers le réseau Ethernet, la passerelle reçoit des blocs isochrones en parallèle, ce qui fait que lors du dépilement d'un bloc, elle n'a pas à attendre la réception d'un bloc pour le dépiler. Lorsque le temps d'envoi moyen est faible, la passerelle a plus de temps pour dépiler un bloc et c'est à ce moment que le taux d'envoi des blocs a un impact significatif sur le temps de dépilement. La passerelle est contrainte de se réajuster par rapport au taux d'envoi, elle ne peut pas dépiler plus de blocs que la caméra n'en envoie. T22 est défini comme suit :

$$T22 = 0 \qquad \textit{quand } TEB \leq T21 - T31 - \frac{T32}{N}$$

$$T22 = \left(\frac{1}{TEB} - T21 - T31 - \frac{T32}{N}\right) \qquad \textit{quand } TEB > 21 - T31 - \frac{T32}{N}$$

avec TEB = taux d'envoi des blocs isochrones

Pendant nos simulations, nous avons caractérisé les temps T1, T21, T31 et T32. Pour nous assurer de la précision des temps recueillis, nous avons utilisé les valeurs du compteur de coups d'horloge du processeur de la passerelle. En effet, les processeurs de la famille Intel Pentium implémentent un compteur qui s'incrémente à chaque coup d'horloge du système. Ce compteur est très utile, surtout quand on sait que les fonctions d'évaluation de temps telles que : « gettimeofday » de Linux ont une précision très faible (de l'ordre des milli-secondes). Nous avons réutilisé la fonction assembleur rdtsc(), proposée par le professeur François-Raymond Boyer. Cette dernière lit et renvoie la valeur du registre compteur de coups d'horloge. Le temps d'exécution d'une séquence d'instructions est déduit avec le nombre de cycles d'horloge associé à la séquence et la fréquence d'horloge du processeur sur lequel l'algorithme de la passerelle s'exécute.

1. Temps de réception des différents types de blocs isochrones : T1

Pour trouver T1, nous avons noté le temps de réception moyen des différents blocs isochrones (324, 644, 1284). Ces temps sont récapitulés dans le Tableau 4-3.

Tableau 4-3. Temps de réception des blocs isochrones

F1	Temps moyen
324	0.618 µs
644	0.764 µs
1284	1.081 µs

2. Temps de dépilement des différents types de blocs isochrones : T21

Pour trouver le temps de dépilement fixe quand le temps d'attente est nul, nous avons sélectionné le temps moyen quand N est égal à 1, en sachant que c'est le meilleur des cas. Le Tableau 2-7 récapitule les différents temps que nous avons notés.

Tableau 4-4. Temps de dépilement des blocs isochrones

F2	Temps moyen
324	13.608 µs
644	17.248 µs
1284	17.805 µs

3. Temps d'assemblage des différents types de blocs isochrones : T31

Le Tableau 4-5 énumère les différents temps qu nous avons recueillis en fonction des différents taille de blocs, étant donné que le temps d'assemblage aussi dépend de la taille des blocs.

Tableau 4-5. Temps d'assemblage des blocs isochrones

F1	Temps moyen
324 o	0.225 µs
644 o	0.253 µs
1284 o	0.426 µs

4. Temps de renvoi des paquets vers le réseau Ethernet : T32

Le temps de renvoi des paquets est un peu plus difficile à mesurer. En effet il n'est pas constant et dépend du trafic du réseau étant donné que l'envoi des paquets UDP est bloquant dans notre cas. Cependant, nous avons constaté un comportement : le surplus dû à la taille des paquets est pour ainsi dire nul. Le temps moyen d'envoi varie selon la taille initiale d'un bloc. Au lieu d'établir une fonction de degré 1, nous avons préféré noter les temps moyens associés à chaque taille de bloc, ces temps sont récapitulés dans le Tableau 4-6.

Tableau 4-6. Temps moyen d'envoi des paquets Ethernet

F1	Temps moyen
324 o	1,301 ms
644 o	1,432 ms
1284 o	1,696 ms

Maintenant que les différents temps ont pu être caractérisés, nous pouvons comparer les estimations trouvées avec le modèle et les résultats réels de simulation. La Figure 4-20 montre l'évolution du nombre de blocs dépilés selon la formule de notre modèle et selon les vrais résultats de simulation pour un taux de 15 images par seconde (TEB = 480*15

blocs/s). Comme on peut le voir, les deux courbes ont du mal à se rejoindre quand N est compris entre 1 et 6. Cette différence doit être due aux variations incessantes du temps d'envoi (selon le trafic courant du réseau) et au temps d'envoi du modèle, qui est un temps moyen. Apparemment, le temps d'envoi est inférieur au temps moyen quand N est compris dans cet intervalle. Par contre, on voit que la saturation à 144000 blocs est atteinte en même temps à N égal à 13. C'est une des raisons pour lesquelles nous n'avons pas modifié le temps moyen pour améliorer le modèle quand N est compris entre 1 et 6.

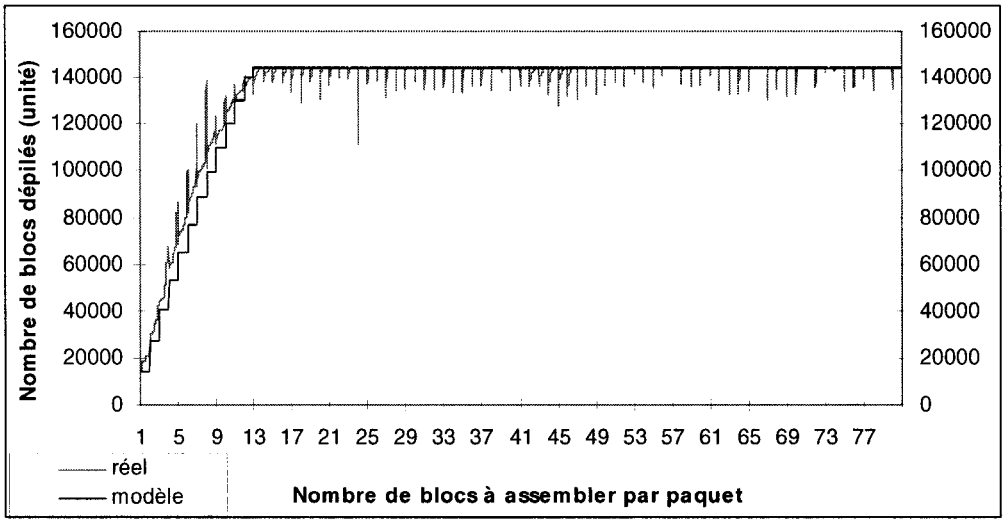

Figure 4-20. Variation du nombre des blocs dépilés

L'Annexe I fournit des résultats qui caractérisent l'évolution du nombre de blocs dépilés selon le modèle et les simulations réelles pour les taux de 7.5 et 30 images à la seconde. Il faut remarquer que les points de saturation sont différents selon la taille initiale d'un bloc. Il est égal à 11 pour une taille de 324 o et 14 pour 1284 o.

La Figure 4-20 valide le modèle de performance qu'on a établi. On remarque une petite différence quand le nombre de blocs isochrones assemblés est inférieur à 13, mais ensuite la courbe du modèle suit très bien celle de la simulation réelle. Il faut rappeler que le modèle permet de prédire le nombre de blocs isochrones (y) que la passerelle pourra dépiler de sa file de réception et envoyer vers le réseau Ethernet. Le modèle est basé sur l'équation suivante :

$$y=\frac{Ttot-LT1}{T21+T22+T31+\frac{T32}{N}}$$

Il faut rappeler que le paramètre N de l'équation dépend du concepteur de la passerelle et c'est le paramètre qui variait durant les simulations. Le paramètre L est en fonction du taux d'envoi des blocs isochrones envoyés par la caméra.

Les paramètres de l'équation du modèle qui dépendent de la passerelle, de la capacité du réseau Ethernet et de la taille des blocs isochrones envoyés par la caméra sont T21 (temps moyens de réception d'un bloc isochrone), T31 (temps d'assemblage d'un bloc isochrone dans un message UDP), T22 (temps de dépilement d'un bloc isochrone) et T32 (temps de retransmission). Les valeurs possibles de ces paramètres ont été recueillies durant des simulations réelles, ces valeurs sont récapitulées au Tableau 4-7 selon la taille initiale d'un bloc isochrone.

Tableau 4-7. Récapitulatif des temps de réception, d'assemblage, de retransmission

	Taille des paquets isochrones (octets)		
	324	644	1284
Temps de réception	13.608 µs	17.2485 µs	17.8057 µs
Temps de dépilement	0.618226026 µs	0.764275928 µs	1.08186605µs
Temps d'assemblage	0.22581 µs	0.253365E µs	0.426834µs
Temps de renvoi	0.001301431s	0.001432659 s	0.001696244 s

Ces temps moyens ont été évalués à partir d'une série de 5120 simulations sans oublier que le modèle ne tient pas compte des variations de débit sur le réseau dues aux autres utilisateurs, ce qui expliquerait entre autres les petites différences entre le modèle et les simulations réelles. Ces différences pourraient être résolues si le réseau de notre laboratoire était plus performant. Par exemple, si le réseau avait un débit de 1 Gbps, les temps de renvoi du réseau Ethernet seraient bien inférieurs à ceux qu'on obtient actuellement.

Ce modèle nous permettra de trouver le nombre optimal de blocs à encapsuler dans un message UDP, c'est-à-dire celui qui permet d'atteindre les meilleures performances. Ceci nous évitera d'effectuer toutes ces simulations dans le futur.

Au départ nous voulions effectuer la retransmission des paquets isochrones directement à partir du *Kernel*. Des problèmes techniques nous ont cependant empêché de mener à bien cette solution. Étant donné que la réception d'un paquet isochrone est effectuée dans une routine d'interruption, certains paramètres du *Kernel* empêchaient la retransmission de ces paquets vers la carte Ethernet. Un blocage s'effectuait dans la pile TCP/IP/Ethernet qui entraînait le gel de la passerelle. Après plusieurs essais et modifications, nous avons finalement décidé d'effectuer la réception et la retransmission des paquets isochrones au niveau usager.

Le principal traitement de la passerelle est le processus de retransmission des flots AV. Ce dernier caractérise les performances de la passerelle en termes de taux d'envoi des messages contenant des blocs AV vers le réseau Ethernet. Après avoir recueilli les performances de la passerelle et les valeurs des paramètres qui ont un impact sur les performances de la passerelle, nous avons pu établir un modèle de performance de la passerelle. Ce modèle caractérise assez bien les performances de la passerelle en fonction du nombre de blocs AV assemblés dans un message transmis vers le réseau Ethernet avec toutefois de légères différences. Ces différences sont dues au fait que le modèle ne tient pas compte de la congestion du réseau Ethernet ou aux temps d'exécution des différentes applications qui roulent en parallèle sur le même poste que celui de la passerelle. Ces différences sont toujours notées dans les applications de transmissions de flots AV en temps réel étant donné que les protocoles de transport n'effectuent pas par eux-mêmes de gestion d'erreurs et de congestion de flots. Un sacrifice a été fait : la passerelle n'effectue pas de gestion d'erreurs. La gestion d'erreur coûte très chère en termes de débit d'envoi des messages sur le réseau Ethernet, c'est une raison pour laquelle même les caméras n'effectuent pas de gestions d'erreurs en mode temps réel. La gestion d'erreurs est surtout effectuée en mode téléchargement lorsque l'application nécessite aucune perte de données.

CHAPITRE 5
CONCLUSION

Ce chapitre présente d'abord une synthèse du contenu des trois précédentes parties. Nous discuterons ensuite les limitations de nos travaux, puis nous récapitulerons les améliorations qui pourraient être apportées à nos modèles. Pour finir, des indications pour des recherches futures seront données.

5.1 Synthèse des travaux

La première phase de ce projet de maîtrise consistait à passer en revue les recherches reliées aux manières d'interconnecter des réseaux Firewire et Ethernet. Nous avons observé que le sujet de l'interconnexion de réseaux a été exploré dans les années 80. La problématique a été située dans le contexte du modèle OSI proposé faciliter la compréhension des protocoles et pour aider à résoudre les problèmes d'hétérogénéité des réseaux. Grâce à ces travaux, nous avons pu faire ressortir les obstacles que nous allions rencontrer pour l'interconnexion Firewire/Ethernet. Ces obstacles peuvent être dus : au format générique des données, à la représentation des requis sémantiques, à l'interprétation du contenu des paquets, à la prise de décisions et à la traduction des adresses. Les chercheurs ont aussi proposé des méthodes afin de passer par-dessus ces obstacles, bien qu'il ne soit pas toujours possible de les résoudre. Quatre méthodes d'interconnexion ont été considérées: la conversion de services, l'encapsulation, la complémentation et enfin la conversion de protocoles. Les trois dernières méthodes sont très populaires. L'encapsulation a été appliquée pour l'interconnexion de réseaux X25 et IPv4, par exemple. Une présentation des protocoles Firewire et Ethernet a été donnée. Cette dernière a fait ressortir les avantages de Firewire qui offre bien plus de services qu'Ethernet. Une des nombreuses différences est le service de transport en mode isochrone. Ce service assure l'envoi de flots dans une fenêtre de temps prédéfinie. La

suite de notre revue de littérature a fait ressortir deux cas de figures d'interconnexion Firewire-Ethernet : l'un pour le transport de datagrammes IPv4 et l'autre pour le transport de données AV. En fait, nous avons trouvé l'existence d'une norme pour le transport de datagrammes IPv4 dans un réseau Firewire. Par contre, aucune norme n'existe pour le transport de données AV. Il existe certes des travaux sur ce sujet (Ogawa et al., 1999; Saito et al., 2001), mais ils concernent l'envoi de données AV d'un équipement à un autre, alors qu'un des buts de ce projet est l'échange d'un ou plusieurs flots vers un ou plusieurs clients Ethernet. Ceci nous mène à rappeler les objectifs de cette maîtrise : proposer des preuves de concept des cas de figures de l'interconnexion Firewire-Ethernet en vue de modéliser et de valider des architectures SoC applicables à ce problème. Donc après une présentation de notre revue de littérature, nous avons présenté en détail les travaux que nous avons effectués sur l'interconnexion Firewire-Ethernet pour le transport de datagrammes IPv4 et pour le transport de données AV.

Comme nous l'avons dit précédemment, la norme RFC 2734 propose une méthode pour le transport de datagrammes IPv4 sous Firewire. Donc, nous avons surtout eu à concevoir et à développer une application logicielle afin de recueillir des résultats de profilage sur un processeur ARM7TDMI. Ces activités de profilage avaient pour but de valider une architecture SoC proposée en 1999 par M. J.-M. Tremblay. Cette architecture avait été définie, mais les spécifications de certains de ses modules n'étaient pas disponibles, pas plus que leurs codes VHDL. Donc, nous avons eu à compléter cette architecture SoC en développant certains modules manquants tels que : le gestionnaire de mémoire, le contrôleur, etc. La complétion de cette architecture a permis d'évaluer les performances de cette dernière pour la conversion Firewire-Ethernet. Nous avons pu noter les gains de performances de cette architecture matérielle par rapport à ceux qui avaient été obtenus par l'application logicielle. Cependant, des faiblesses ont pu être notées. Les performances de l'architecture SoC se dégradent lorsque la taille des paquets est élevée. Ce comportement est dû au traitement des paquets en mode séquentiel ; les modules de l'architecture devraient fonctionner en mode *multi-thread*, afin que le convertisseur

puisse recevoir, formater et retransmettre des paquets en parallèle. Une seconde faiblesse qui a pu être notée est l'accès à la mémoire principale de l'architecture. Cette dernière est une mémoire multi-port, ce qui en augmente considérablement la taille et diminue le nombre de paquets que le convertisseur peut recevoir. Toutes les remarques que nous avons notées en analysant les performances du convertisseur nous ont poussé à proposer une nouvelle architecture qui serait bien plus performante que la précédente.

Dans la dernière partie de ce mémoire, tous les travaux qui ont été effectués sur le transport de données AV entre un réseau Firewire et un réseau Ethernet ont été détaillés. Deux problèmes devaient être résolus pour permettre l'envoi et la réception des flots AV : les techniques d'adressage dans les deux réseaux et les formats de données reconnus et traités par les équipements de ces réseaux. Étant donné que cette solution devait être déployée sous un environnement Linux, nous avons eu à rajouter un pilote dans le *Kernel* de ce système d'exploitation : FireoverIP. Ce pilote permet d'allouer des adresses IPv4 aux équipements du réseau Firewire, étant donné que ces derniers ont chacun une adresse physique qui est définie durant le processus d'auto-identification du réseau Firewire. Le pilote comporte une table de routage. Ainsi, les clients Ethernet qui envoient un message adressé à un équipement Firewire ont juste à préciser l'adresse IPv4 de ce dernier ainsi que l'adresse MAC de la passerelle. La passerelle à la réception d'un paquet, doit d'abord valider l'adresse IPv4 et ensuite elle pourra retransmettre le paquet vers l'équipement Firewire adressé.

Pour la résolution du format de données, nous avons appliqué la technique de la complémentation avec l'ajout d'une couche par-dessus les entêtes des protocoles TCP ou UDP que nous avons aussi nommé entête FireoverIP. Comme nous l'avons dit, Firewire offre bien plus de services qu'Ethernet, pour retrouver les mêmes services que Firewire, il faut des empilements de protocoles tels que TCP/IP/Ethernet. Ces empilements sont encore insuffisants pour reconstituer les services de Firewire, raison pour laquelle nous avons rajouté cette entête. Notre passerelle se base sur une application client/serveur, ainsi, l'architecture complète de notre système a été exposée, en incluant les

fonctionnalités d'un client. En fait, les équipements AV doivent être initialisés avant qu'ils puissent envoyer des flots AV. Ces séquences d'initialisation s'effectuent grâce à l'envoi d'ordres d'écriture en mode asynchrone vers l'équipement Firewire concerné. Après avoir développé toutes les fonctionnalités de la passerelle et celle des clients, nous avons recueilli les performances de la passerelle. Cette caractérisation des performances avait pour but de noter les vitesses de réception des flots vidéo en sachant que les clients pouvaient communiquer avec une WebCam Pyro de la compagnie ADS. Nous avons constaté que le convertisseur ne pourrait jamais atteindre des performances raisonnables en effectuant la retransmission des paquets isochrones au fur et à mesure qu'ils sont reçus. Il faut d'abord les assembler dans des messages UDP avant de les retransmettre vers le réseau Ethernet. Nous avons aussi eu à noter la taille optimale des messages UDP qui permet d'atteindre les meilleures performances, quel que soit le taux d'envoi des images venant d'une WebCam. Les résultats atteints sont satisfaisants, bien que certaines fonctionnalités manquent encore à la passerelle.

5.2 Limitations des travaux

Dans le chapitre 3, nous avons traité l'interconnexion Firewire-Ethernet pour le transport de datagrammes IPv4 entre ces deux types de réseaux. Étant donné qu'une norme existe déjà, nous nous sommes focalisés sur le traitement des datagrammes afin de valider l'architecture SoC du convertisseur de protocoles. C'est ainsi que nous avons négligé le traitement des données de contrôle et les opérations de mise à jour des tables de routage qui permettent d'identifier les équipements des deux réseaux. Notre modèle logiciel pourrait être enrichi en rajoutant les fonctions de routage afin de traiter des paquets de type ARP venant des deux réseaux.

Dans le chapitre 4, nous nous sommes focalisés sur le transport de données AV d'un réseau Firewire vers Ethernet. Nous voulions que plusieurs clients Ethernet puissent recevoir des flots AV venant de plusieurs équipements Firewire en même temps. Cependant, la flexibilité des équipements Firewire est telle que cela devient un

inconvénient de temps en temps. En effet, n'importe quel client peut changer les propriétés d'un équipement AV (arrêt de l'envoi de flots, taux d'images reçues par seconde, format des images) alors que celui-ci envoie déjà des flots. Il serait peut être utile que la passerelle ait un système de contrôle pour que seul le premier client Ethernet qui s'est connecté sur un équipement AV ait des droits de lecture et d'écriture sur ce dernier et que les équipements qui suivent aient juste des droits de lecture simple.

Durant nos tests, nous avons remarqué que le délai de traitements de renvoi des réponses de requêtes asynchrones était très important quand la passerelle recevait en même temps des flots AV. C'est ainsi que nous nous sommes rendus compte que le pilote de Firewire sous Linux comporte une seule FIFO qui contient les réponses de requêtes asynchrones et les blocs AV isochrones reçus durant la communication. Donc, le délai de traitement d'une réponse asynchrone augmente considérablement, puisque la passerelle doit dépiler tous les paquets isochrones reçus avant de récupérer la réponse et de la retransmettre sur le réseau Ethernet. Il serait utile d'avoir deux FIFO : une première contenant les paquets isochrones et une seconde pour les paquets asynchrones.

La dernière limitation que l'on peut citer concerne la compression des flots AV qui n'est pas effectuées par la passerelle. Ce choix peut s'expliquer simplement. Ce ne sont pas tous les équipements AV Firewire qui envoient des flots non-compressés comme c'est le cas pour la WebCam avec laquelle nos tests ont été effectués. Il faudrait que la passerelle connaisse les propriétés de tous les équipements AV pour savoir quels sont les flots qu'il devra compresser et ceux qu'il devra retransmettre sans traitement. Il faut préciser que l'insertion d'un module de compression dans la passerelle ne serait pas compliquée à effectuer. Cette initialisation de la passerelle affaiblirait beaucoup sa flexibilité, donc il faudra faire un choix entre la flexibilité ou la diminution du trafic causé par la passerelle.

5.3 Indications de recherches futures

Une des différences qui n'a pas pu être résolue, entre les réseaux Firewire et les LAN Ethernet concerne le service de réservation de bande passante. Certes, RSVP existe,

cependant ce protocole se retrouve plutôt dans les WAN. Des travaux pourraient ainsi être effectués pour proposer un protocole de réservation de bande passante dans les LAN Ethernet flexible et facile à implémenter.

Une recherche future permettrait aussi de rajouter plusieurs équipements Firewire pour noter le comportement de la passerelle et proposer s'il le faut une architecture multiprocesseur pour accélérer sa puissance de calcul. En effet, étant donné que la passerelle pour l'instant traite seulement un seul flot de données AV, elle n'a pas de difficulté à supporter le débit de réception du flot. Cependant, avec plusieurs flots, il serait intéressant d'analyser ses performances et voir si elle arrive à supporter la réception de plusieurs flots et si ses performances ne se dégradent pas en fonction du nombre de flots à traiter.

La dernière indication concerne la proposition d'une architecture d'une plate-forme SoC pour le transport de flots AV d'un réseau Firewire/Ethernet. La passerelle pour l'instant est déployée sur un PC et comme nous l'avons vu, elle consomme presque toute la puissance de calcul de ce dernier durant la réception et la retransmission des flots AV. Donc, au lieu de monopoliser un PC, il serait plus économique de placer une passerelle matérielle à la périphérie des deux réseaux.

BIBLIOGRAPHIE

[1] 1394 TRADE ASSOCIATION. 1998.*1394-based Digital Camera Specification, Version 1.20.*

[2] APOSTOLOPOULOS, J., WEE, S. 2001. *Compressed-Domain Video Processing.* Streaming Media Systems Group, Hewlett-Packard Laboratories. [en ligne]. http://www.hpl.hp.com/techreports/2002/HPL-2002-282.pdf. (Page consultée en octobre 2003).

[3] ADS. 2002. *ADS PYTO 1394 WEBCAM.* [en ligne]. http://www.adstech.com/products/intro/products.asp. (Page consultée en février 2002)

[4] ATMEL CORPORATION. 2002. *Embedded Asic Core Peripherical Ethernet MAC.* [en ligne]. www.atmel.com/atmel/acrobat/doc1794.pdf. (Page consultée en juin 2002).

[5] BOCHMANN, G.V., MONDAIN-MONVAL, P. 1990. « Design Principles for Communication Gateways ». IEEE Journal on selected areas in communications, 8:1. 1298-1300.

[6] BOUTABA, R., REN N.N., RASHEED, Y. 2002. Leon-Garcia A. « Distributed Video Production: Tasks, Architecture and QoS Provisionning ». Multimedia Tools and Applications, Kluwer Academic Publishers. 16. 99-136.

[7] CALVERT, K. LAM, S. 1989. « Deriving a protocol converter: a top-down method ». ACM SIGCOMM Computer Communication Review. Symposium Proceedings on Communications Architectures and Protocols. 19:4, P.247-258.

[8] CHANG, J. LIU, M.T. 1990. « An Approach to Protocol Complementation for Internetworking ». Systems Integration '90. Proceedings of the First International Conference on Systems Integration. P.205-211.

[9] CONTI, M. KUMAR, M. DAS, S. SHIRAZI, B. 2002. « Quality of service Issues in Internet Web Services ». IEEE Transactions on Computer. 51:6. 593-594.

[10] COULOURIS, G. DOLLIMORE, J. KINDBERG, T. 2001. *Distributed Systems: Concepts* and Design. 3th ed. Addison-Wesley. 772p.

[11] GRACE, A. COX, J. JACOBS, R. MORRISON, G. 2000. « Streaming multimedia for the internet ». BT Techno J. 18:1. 89-90.

[12] GREEN, P.E. 1986. « Protocol Conversion », IEEE Transactions on Communications, 34:3. P.257-268.

[13] FEIRELSON, D.G. 2003. « Metric and Workload Effects on Computer Systems Evaluation », IEEE Computer Magazine. P.18-25.

[14] HONG, G.Y. FONG, B. FONG, A.C.M. 2002. « QoS Control for Internet Delivery of Video Data », Proceedings of the International conference on Information technology: coding and computing (ITCC'02). P.458-461.

[15] INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS. INC. 1996. IEEE Std 1394-1995 : *IEEE standard for a high performance serial bus.*

[16] INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS. INC. 2002. *IEEE Std. 1394b - 2002 IEEE Standard for a High-Performance Serial Bus - Amendment 2.*

[17] Institute of Electrical and electronics Engineers. Inc. 1998. *Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications.*

[18] JOHANSSON, P. 1999. *IPv4 over IEEE 1394.* IETF. RFC 2734. [en ligne]. http://www.ietf.org/rfc/rfc2734. (Page consultée en septembre 2001).

[19] KEVILLE, K.L., TOMPKIN, R. 2001. « IEEE 1394 and RFC 2734; a viable HSI for hypercubes », Proceedings of the 2001 IEEE International Conference on Cluster Computing. P.155-157.

[20] KOGA, H. JINZAKI, A. 2000. *DV Stream Transmission over World Wide Internet by Video Frame Recognition.* [en ligne]. http://www.comet-can.jp/PDSflab/comet/paper/pv2000.pdf. (Page consultée en juin 2002)

[21] MATHY, L. EDWARDS, C. HUTCHISON, D. 1999. « Principles of Qos in group communications ». Telecommunications Systems. J.C Baltzer Ag, Science Publishers. 59-84.

[22] MAUTHE, A. GARCIA, F. HUTCHISON, D. YEADON, N. 2001. « QoS Filtering and Reservation in an Internet Environment ». Multimedia Tools and Applications. Kluwer Academic Publishers. 13. 285-306.

[23] OGAWA, A. KOBAYASHI, K. SUGIURA, K. NAKAMURA, O. ET MURAI J. 1999. « Design and Implementation of DV Stream over Internet ». IEEE Internet Workshop. P.255-260.

[24] PLUMMER, D.C. 1982. *An Ethernet Address Resolution Protocol.* IETF. RFC 826. [en ligne]. http://www.ietf.org/rfc/rfc826. (Page consultée en septembre 2001).

[25] POSTEL, J. 1980. *DOD Standard : Internet Protocol.* USC/Information Sciences Institute. IETF. RFC 826. [en ligne]. http://www.ietf.org/rfc/rfc826. (Page consultée en septembre 2001).

[26] POSTEL, J. 1980. *Transmission Control Protocol.* USC/Information Sciences Institute. IETF. RFC 761. [en ligne]. http://www.ietf.org/rfc/rfc826. (Page consultée en septembre 2001).

[27] POSTEL, J. 1980. *User Datagram Protocol*, RFC 768, USC/Information Sciences Institute. IETF. RFC 761. [en ligne]. http://www.ietf.org/rfc/rfc826. (Page consultée en septembre 2001).

[28] PROBERT, R.L. SALEH, K. 1991 « Synthesis of Communication Protocols : Survey and Assessment », IEEE Transactions on Computers. 40:4. 468-476.

[29] SAITO, T. TOMODA, I. TAKABATAKE, Y. TERAMOTO, K. FUJIMOTO, K. 2001. « Wireless Gateway for Wireless Home AV Network and its Implementation ». IEEE Transactions on Consumer Electronics. 47:3. 496-501.

[30] SANTAMRAIA, R. 2000. « IP over IEEE-1394: Using the Internet Protocol over a high-performance serial bus ». Dedicated Systems Magazine. 19-21.

[31] SHU, J.C. LIU, M. 1990. « Protocols Conversion between Complex Protocols ». IEEE Proceedings. Ninth Annual International Phoenix Conference on Computers and Communications. P.584-590.

[32] SUNSHINE, C. 1990. « Network interconnection and gateways ». IEEE Journal on Selected Areas in Communications. 8:1. 4-11.

[33] SVOBODOVA, L. JANSON, P. MUMPRECHT, E. 1990. « Heterogeneity and OSI ». IEEE Journal on Selected Areas in Communications. 8:1. 67-79.

[34] TANUAN, M.C. 1998. *An Introduction to the Linux Operating System Architecture*. University of Waterloo. [en ligne]. http://se.uwaterloo.ca/~mctanuan/cs746g/LinuxCA.html. (Page consultée en mars 2003)

[35] Texas Instruments. 1999. *TSB12LV23 OHCI-Lynx PCI-Based IEEE 1394 Host Controller (Rev. A)*. [en ligne]. http://focus.ti.com/lit/ds/slls328a/slls328a.pdf. (Page consultée en mars 2002)

[36] TOHIO, B. 2003. «Mémoire de Maîtrise :Aspects théoriques de la convertibilité des protocoles de communication ». Montréal. École Polytechnique Montréal. 135p.

[37] TREMBLAY, J-M. BA, A. BERTOLA, M. PEPGA-BISSOU, J. DONZEL, G. NGONGANG, J.P. PLANTE, P. 2001. « Conception d'un modèle exécutable d'un convertisseur de protocoles : Rapport final ». Cours ELE 4304. École Polytechnique Montréal.

[38] 3D SOLUTIONS. 2001. *USB 2.0 versus Firewire 1394*. [en ligne]. http://www.3dsolutions.co.uk/infoUSB2_Firewire.htm. (Page consulté en janvier 2003)

[39] UNIBRAIN, S.A. 2002. *FireNET 2.5*. [en ligne]. www.unibrain.com/evaluations/firenet.htm. (Page consultée en juillet 2002)

[40] WU, D. HOU, Y.T. ZHU, W. ZHANG, Y-Q. PEHA, J.M. 2001. « Streaming Video over the Internet: Approaches and Directions ». IEEE transactions on circuits and systems for video technology. 11:3. 282-300.

[41] YONG, ZHU. 2000. A Survey of Network Requirements to Support Current and Future Data Streams. [en ligne]. http://www.cc.gatech.edu/%7Eyongzhu/Cs7001/cs7001_1_report.html. (Page consultée en juin 2003)

ANNEXES

Annexe A
Modèle OSI

A.1 Description des couches du modèle OSI

Chaque couche du modèle OSI remplit une fonction bien définie concernant le traitement des paquets. Le Tableau A-1 décrit les couches du modèle et donne, en exemple, des protocoles associés à chaque couche.

Tableau A-1. Description des couches du modèle OSI

Couche	Description	Exemples
Application	Elle concerne les protocoles qui ont étaient conçus afin de respecter les besoins de communication spécifiques à certaines applications, souvent ils définissent des interfaces de service.	HTTP, FTP, SMTP
Présentation	Les protocoles de cette couche, transmettent les données selon une représentation orientée réseau, indépendante du format de données propre aux applications. Les opérations d'encryption sont effectuées sur cette couche, si nécessaire.	SSL
Session	A ce niveau les opérations de fiabilité et d'adaptation sont opérées, telles que la détection de défauts et la correction automatique.	
Transport	La dernière couche sur laquelle on parle de message, non de paquet. Les messages sont adressés par des ports de communication associés aux processus. Les protocoles peuvent être orienté-connexion ou non orienté-connexion.	TCP, UDP, RTP
Réseau	Elle transfère les paquets entre ordinateurs dans un réseau spécifique. Dans un réseau WAN ou une interconnexion cela implique la génération d'une route via des routeurs. Dans un réseau local(LAN) , le routage n'est pas nécessaire.	IP, ATM circuits virtuels, X25
Liaison de données	Responsable de la transmission des paquets entre les nœuds, qui sont directement liés par un lien	Ethernet MAC, ATM cell transfer,

	physique. Dans un WAN, il s'agit de transmission entre un routeur et un autre routeur ou un hôte. Dans un LAN, c'est entre une paire d'hôtes.	PPP, LAP-B
Physique	Les circuits et le matériel qui conduisent les signaux de transmission. A ce niveau, l'information est sous forme de signaux analogiques. Des opérations de modulation d'amplitude ou de fréquence sont effectuées sur les signaux électriques (câbles), lumineux (fibres optiques) ou électromagnétiques(radio) par exemple.	Signalisation Ethernet, ISDN

Selon le modèle OSI, chaque couche rajoute de l'information à la charge utile lors de l'émission. À la réception, chaque couche récupère l'information qui lui est destinée. La Figure A-1 montre comment la charge utile chemine de la source à sa destination.

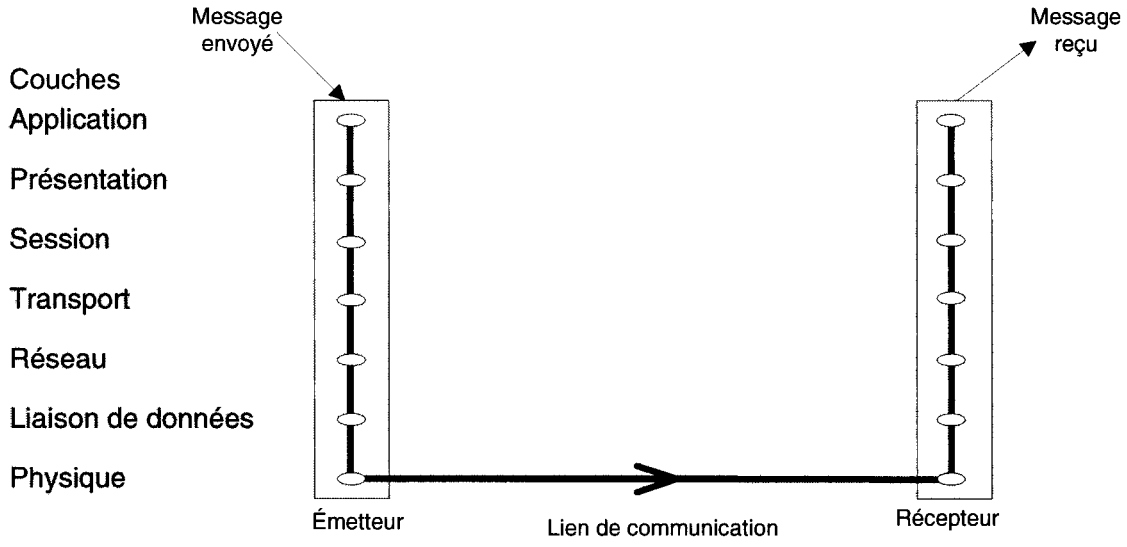

Figure A-1. Cheminement d'un message selon le modèle OSI

Les couches supérieures ne sont pas supposées connaître les protocoles des couches inférieures, en fait une couche inférieure offre des services à la couche qui lui est directement supérieure.

Annexe B
Format d'un datagramme IPv4

B.1 Description des champs du protocole IPv4

Le protocole le plus populaire dans domaine des réseaux à l'heure actuelle. Ce protocole par sa simplicité permet de fragmenter les données de la couche transport et à différents réseaux de communiquer selon leur adresse. Dans un réseau IP, les équipements sont identifiés selon une adresse unique, communément appelée : adresse IP. La figue 2.7 illustre le contenu d'une entête d'un paquet IP.

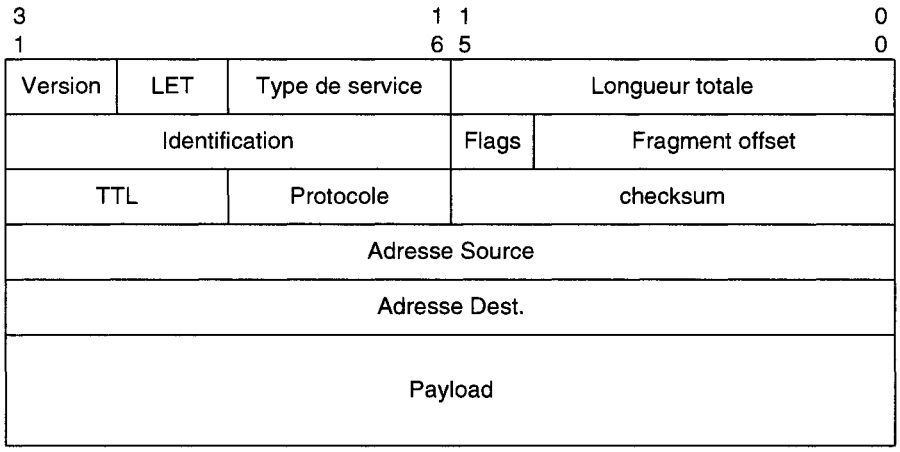

Figure B-1. Format de l'entête d'un paquet IP

Les champs de l'entête IPv4 sont définis comme suit:

- Version : il indique la version IP du paquet qui est égale 0x4 pour IPv4;
- LET : La longueur de l'entête IP qui est égale à 0x5;
- Type de service : Il permet d'indiquer quelle type de service est demandé, selon la pertinence des données transmises par exemple : est-ce des données prioritaires, par exemple ?
- Longueur totale : Elle comprend la longueur de l'entête IP et celle de la charge utile du paquet;

- Identification : Il détermine l'appartenance du paquet dans le cas ou l'information a dû être fragmentée, les paquets appartenant à la même information auront la même identification;
- Flags : Elles donnent de l'information concernant les fragments de données, par exemple, lorsque le bit 2 de ce champ est égal à 0 cela signifie que ce paquet est le dernier paquet d'une suite de fragments appartenant à la même information;
- Fragment Offset : les paquets IP ne suivent pas le même chemin durant la transmission, donc ce champ indique la position du paquet dans le message qui a été fragmenté;
- Durée de vie : vu le nombre d'hôtes dans un réseau Internet, le paquet peut prendre du temps avant d'arriver à destination, donc ce champ permet de limiter le nombre de sauts dans le réseau, à chaque saut la valeur de ce champ est décrémentée et lorsqu'elle est égale à 0, le paquet est détruit;
- Protocole : Il indique le protocole qui est sur la couche supérieure, des codes prédéfinis existent afin d'identifier ces protocoles (TCP, UDP, etc.);
- Checksum de l'entête : Le checksum de l'entête entièrement y compris les champs qui suivent, le résultat de la somme de tous les 16 bits de l'entête en complément à un;
- Adresse Source : l'adresse de l'hôte source;
- Adresse Destination : l'adresse de l'hôte destination;
- Options : des information liées à la sécurité ou aux routeurs sont contenues dans ce champ;
- Bourrage : Si les options ne forment pas un multiple de 32, on rajoute des bits de bourrage à la suite.

Il faut préciser que la taille maximale d'un paquet IP s'élève à 1536 octets.

Annexe C

Format des fichiers de configuration des protocoles

Chaque fichier de configuration comporte :

Sur la première ligne :

- Prot : Nom du protocole

Sur la seconde ligne :

- NBPKT : nombre de type de paquets

REPETER

Sur la troisième ligne

- Pkt :
- le nom du premier type de paquet
- la taille minimale en bits d'un paquet
- la taille maximale en bits d'un paquet
- le nombre de champs contenus dans un paquet
- le nombre de champs appartenant à l'entête du paquet

Description des champs de ce type de paquet

- numéro du champ
- type du champ (mot réservé)
    - HAD : Header Address Destination
    - HAS : Header Address Source
    - HLP : Header Length or Protocol
    - HCX : Header Control
    - PXX : Payload
    - HCH : Header Checksum Header
    - HPC : Header Payload Checksum
- La taille en bits du champ pour les champs de l'entête ou la taille minimale et maximale de la charge utile

JUSQU'À la fin de la description de tous les types de paquets

Exemple du fichier de configuration du protocole Ethernet :

```
Prot: Ethernet_SNAP
NBPKT: 1
Pkt: NORMAL 432 12112 4 3
1 HAD Destination_address 48
2 HAS Source_address 48
3 HLP Length 16
4 PXX data 320 11936
```

Exemple du fichier de configuration du protocole Firewire :

```
Prot: FIREWIRE
NBPKT: 1
Pkt: ASYNC_WRITE_S200 544 8384 12 11
1 HAD Destination_ID 16
2 HCX tl      6
3 HCX Rt      2
4 HCX Tcode         4
5 HCX Pri     4
6 HAS Source_ID   16
7 HCX dest_offset 48
8 HLP Data_len 16
9 HCX ext_tcode 16
10 HCH header_CRC 32
11 PXX data 320 8192
12 HPC data_CRC 32
```

Exemple du fichier de configuration du protocole IPv4 :

```
Prot: IPV4
NBPKT: 1
Pkt: IPfragment 320 11904 13 12
1 HCX Version 4
2 HCX LET 4
3 HCX Type_Service 8
4 HLP Longueur_totale 16
5 HCX Identification 16
6 HCX Flags 3
7 HCX Fragment_Offset 13
8 HCX Durée_vie 8
9 HCX Protocole 8
10 HCH Checksum_entête 16
11 HAS Adresse_Source 32
12 HAD Adresse_Destination 32
13 PXX DATA 160 11744
```

Annexe D

Code C++ de la solution non-optimisée et optimisée

Cette annexe comprend les codes C++ du convertisseur de protocoles logiciel pour le transport de datagrammes IPv4. Tableau D-1 énumère la liste des fichiers qui seront donnés dans la suite de cette annexe.

Tableau D-1. Liste des fichiers du convertisseur logiciel

Section	Num. pages	Fichier	Description
D-1	117	Conv_main.cpp	Il comporte le programme principal qui instancie les objects nécessair à la conversion, en autres.
D-2	120	Converison.h Conversion.cpp	Ils comportent la description des attributs et méthodes de la classe Conversion. Cette classe effectue la convesion Firewire vers Ethernet et vice versa.
D-3	128	Protocoles.h Protocoles.cpp	Ils comportent la description des attributs et méthodes de la classe Ptotocole. Cette classe définit de manière générique les différents types de paquets d'un protocole.
D-4	133	Packet.h Packet.cpp	Ils comportent la description des attributs et méthodes de la classe : Packet. Cette classe définit de manière générique la structure d'un type de paquet en général.

D.1 Conv_main

```
/*****************************************************************************/
/* Title       : Main                                                        */
/* Project     : Convertisseur de protocoles: transport de datagrammes IP  */
/*****************************************************************************/
/* File        : conv_main.cpp                                               */
/* Author      : Mame Maria Mbaye                                            */
/* Created     : 07-06-2002                                                  */
/* Last modified : 24-02-2003                                                */
/*****************************************************************************/
/* Description :  Programme principale du convertisseur de protocoles      */
/*****************************************************************************/
#include "Conversion.h"
#include <string.h>
#define FIRtoETH 0
#define nb_pkt 66
#define test1 1

#if ARM
#include <fstream>
#include <iostream>
#include <time.h>
#else
#include <fstream.h>
#include <iostream.h>
#include <conio.h>
#include <time.h>
#include <stdio.h>
#endif


// convertit une chaine de caraceteres en un mot de 32 bits non signe
Quadlet conv_char_quadlet(char *st)
{
 if (st!= NULL)
  {
   Quadlet conv=0;
   int len = strlen(st);
   for (int i = 0; i<len; i++)
       {
        if (st[i]<48 && st[i]>57)
         {
          cout<<"Fichier INPUT"<<" corrompu !"<<endl;
          return 0;
         }
        conv = (conv * 10) + (st[i]-48);
       }

   return conv;
  }
 else
  return 0;
}

// cette fonction lit les paquets stockes dans un fichier
// parametres
// pkt_nb = renvoie le nombre de paquet lu
// pkt_size = renvoie un tableau de in contenant la taille de chaque paquet
// fich_size = le chemin du ficheir dans lequel est stocke les paquets
Quadlet **lire_pkt_quad(char *fich_in, int *pkt_nb)
{
 ifstream mm_mp(fich_in, ios::in);
 char temp [100];
 int li, col;
 Quadlet **pkt;

 if (!mm_mp)
  return 0;

 mm_mp>>temp;

 pkt = new Quadlet*[nb_pkt];
 for (int i=0; i<nb_pkt; i++)
  pkt[i] = new Quadlet[263];

 li = -1;
 while (!mm_mp.eof())
  {
   if ((temp[0] != ',') && (temp[0]!= '}') && (temp[0]!= '{'))
       {
#if ARM == 0
        pkt[li][col] = (unsigned long)atoi(temp);
#else
        pkt[li][col] = conv_char_quadlet(temp);
#endif
        col ++;
       }
   else if (!strcmp(temp, "{"))
       {
        li ++;
```

```
            col = 0;
           }
     else if ((!strcmp(temp, "}")) && (li == (nb_pkt-1)) )
          break;

     mm_mp>>temp;
    }

  mm_mp.close();
  *pkt_nb = li +1;
  return pkt;
}

/* Programme principal */
int main(void){
  Quadlet **quad;
  int nb_quad[1];
  int i, j ;

  quad = NULL;

// Selon le sens de la conversion
// Ethernet to Firewire
#if sauv_fir_eth == 1 && ARM == 0 && FIRtoETH == 0
  ofstream fir_pkt("../pkt_patrons/fir_pkt_nopt.txt", ios::out);

  if (!fir_pkt)
    {
      printf("les fichiers de sortie ne peuvent etre crees\n");
      return NULL;
    }
#endif

// Firewire to Ethernet
#if sauv_fir_eth == 1 && ARM == 0 && FIRtoETH == 1
  ofstream eth_pkt("../pkt_patrons/eth_pkt_nopt.txt", ios::out);

  if (!eth_pkt )
    {
      printf("les fichiers de sortie ne peuvent etre crees\n");
      return NULL;
    }
#endif

#if FIRtoETH == 1
  Quadlet *res_1_2;
#else
  Quadlet *res_2_1;
#endif

  Conversion Conv1;
  char name2[100], name1[100];

#if ARM == 0
  strcpy(name1, "./../protocoles/firewire_aw.txt");
  strcpy(name2, "./../protocoles/eth_8023.txt");
#else
  strcpy(name1, "./../../../protocoles/firewire_aw.txt");
  strcpy(name2, "./../../../protocoles/eth_8023.txt");
#endif
  if (!Conv1.load_source_prot(name1) ||
      !Conv1.load_dest_prot(name2))
    {
      cout<<" PB DURANT LOAD PROT"<<endl;
      return 0;
    }

#if FIRtoETH == 1
#if ARM == 0
  strcpy(name2, "../pkt_patrons/mm_mp_quad.txt");
#else
  strcpy(name2, "./../../../pkt_patrons/mm_mp_quad.txt");
#endif
  // lire l ensemble des paquets contenus dans le fichier
  quad = lire_pkt_quad(name2, nb_quad);
#else
#if ARM == 0
  strcpy(name2, "../pkt_patrons/eth_pkt.txt");
#else
  strcpy(name2, "./../../../pkt_patrons/eth_pkt.txt");
#endif
  quad = lire_pkt_quad(name2, nb_quad);
#endif

  int ip_len;
  int nb_treat = 0;
  // Appel de la fonction de conversion
  if (quad!=NULL)
    {
      for ( i = 0; i<nb_quad[0]; i++)
           {
// Conversion Firewire to Ethernet
```

```
#if FIRtoETH == 1
        res_1_2 = Conv1.convert_1_2(quad[i]);
        if (res_1_2 != NULL)
         {
          nb_treat++;
          // sauvegarder le paquet converti dans fichier
#if sauv_fir_eth == 1 && ARM == 0
          eth_pkt<<"{ ";
          ip_len = (res_1_2[4] >> 16) + 14;
          cout<<i<<" - ip_len "<<ip_len<<endl;
          if ((ip_len%4) != 0)
              ip_len = (ip_len/4) +1;
          else
              ip_len = (ip_len/4);

          for( j=0; j<ip_len; j++)
              eth_pkt<<" , "<<res_1_2[j];

          eth_pkt<<" } "<<endl;
#endif
         }
        else
         cout<<"res_1_2 PKT ERROR "<<i<<endl;
#endif

// Conversion Ethernet to Firewire
#if FIRtoETH == 0
        res_2_1 = Conv1.convert_2_1(quad[i]);
        if (res_2_1 != NULL)
         {
          nb_treat++;
          // sauvegarder le paquet converti dans fichier
#if sauv_fir_eth == 1 && ARM == 0
          fir_pkt<<"{ ";
          ip_len = (res_2_1[3] >> 16) + 20;
          cout<<i<<" - ip_len "<<ip_len<<endl;
          if ((ip_len%4) != 0)
              ip_len = (ip_len/4) +1;
          else
              ip_len = (ip_len/4);

          for( j=0; j<ip_len; j++)
              fir_pkt<<" , "<<res_2_1[j];
          fir_pkt<<" } "<<endl;
#endif
         }
        else
         cout<<"res_2_1 PKT ERROR "<<i<<endl;
#endif
       }
   }

// fermer les fichiers
#if sauv_fir_eth == 1 && ARM == 0 && FIRtoETH == 0
  fir_pkt.close();
#endif
#if sauv_fir_eth == 1 && ARM == 0 && FIRtoETH == 1
  eth_pkt.close();
#endif

  printf("FINI %d\n", nb_treat);
  getchar();
  return 1;
}
```

D.2 Conversion

```
/*****************************************************************************/
/* Title       : Conversion                                                  */
/* Project     : Convertisseur de protocoles: transport de datagrammes IP    */
/*****************************************************************************/
/* File        : conversion.h                                                */
/* Author      : Mame Maria Mbaye                                            */
/* Created     : 07-06-2002                                                  */
/* Last modified : 24-02-2003                                                */
/*****************************************************************************/
/* Description :  declaration des attributs et methodes de la                */
/*             classe Conversion                                             */
/*****************************************************************************/
/* Generated by Together */

#ifndef CONVERSION_H
#define CONVERSION_H
#include "Protocole.h"
#include  "Table_Corr.h"

class Conversion {
 public:
  Conversion(void);
  ~Conversion(void);
  bool load_source_prot(char* file_name);
  bool load_dest_prot(char* file_name);
  Quadlet *convert_1_2(Quadlet pkt[]);
  Quadlet *convert_2_1(Quadlet pkt[]);
  Quadlet *convert_1_2_nopt(Quadlet pkt[]);
  Quadlet *convert_2_1_nopt(Quadlet pkt[]);

  char* get_trans_label();
  char* get_pad(char* tmp);
  char* get_virtual_layer();
  char* get_length(char* pkt);
  char* get_CRC(char* data);
  bool is_conv_valid();
  void extract(Quadlet pkt[]);
  void rech_addr(void);
  void assembl(Quadlet pkt[]);

 private:
  /** @link aggregation
   * @clientCardinality 2..2
   * @supplierCardinality 0..**/
  Protocole lnkProtocole[2];
  Protocole Prot_ipv4;
  Protocole Prot_sublayer;
  Protocole Prot_eth;
  Table_Corr *lnkTable_Corr;
  unsigned int TL_conv;
};
#endif //CONVERSION_H


/*****************************************************************************/
/* Title       : Conversion                                                  */
/* Project     : Convertisseur de protocoles: transport de datagrammes IP    */
/*****************************************************************************/
/* File        : conversion.cpp                                              */
/* Author      : Mame Maria Mbaye                                            */
/* Created     : 07-06-2002                                                  */
/* Last modified : 24-02-2003                                                */
/*****************************************************************************/
/* Description :  module comporte toutes les fonctions de base qui permettent*/
/*             de convertir un paquet Firewire contenant un datagrammes      */
/*             IP en un paquet Ethernet et vice versa                        */
/*****************************************************************************/
/* Generated by Together */
#include <string.h>
#include "utils.h"
#if ARM
#include <fstream>
#include <iostream>
#else
#include <iostream>
#include <conio.h>
#endif

#define len_without_fragment 1020
#define write_block_req_tcode 0x01
#define destination_offset 0xffffffffffff
#include "Conversion.h"

// Constructeur de la classe
Conversion::Conversion(void)
{
  Quadlet temp[2];
  // Creation des tables d adressage
  lnkTable_Corr = new Table_Corr[2];
```

```
 lnkTable_Corr[0].set_sizeTable_Corr(2);
 lnkTable_Corr[1].set_sizeTable_Corr(2);
 lnkTable_Corr[0].set_addrpar(1,2,1,1);
 lnkTable_Corr[1].set_addrpar(1,2,1,2);


 // remplir les tables par des valeurs par defaut
 temp[0] = 0x00000000;
 lnkTable_Corr[0].set_addr(0,0,0,temp);
 temp[0] = 0x0000ffff;
 lnkTable_Corr[0].set_addr(0,0,1,temp);
 temp[0] = 0xffffffff;
 lnkTable_Corr[0].set_addr(0,1,0,temp);
 temp[0] = 0x00000000;
 lnkTable_Corr[0].set_addr(0,1,1,temp);

 temp[0] = 0x00000000;
 lnkTable_Corr[1].set_addr(0,0,0,temp);
 temp[0] = 0xffffffff;
 temp[1] = 0xffffffff;
 lnkTable_Corr[1].set_addr(0,0,1,temp);
 temp[0] = 0x00000000;
 lnkTable_Corr[1].set_addr(0,1,0,temp);
 temp[0] = 0x00000000;

 lnkTable_Corr[1].set_addr(0,1,1,temp);

 // charger la description des prototocoles IPv4,
 // couche virtuelle Firewire
#if ARM == 0
 Prot_ipv4.addpkt("./../protocoles/IPV4.txt");
 Prot_sublayer.addpkt("./../protocoles/SUB_IPV4_1394.txt");
#else
 Prot_ipv4.addpkt("./../../../protocoles/IPV4.txt");
 Prot_sublayer.addpkt("./../../../protocoles/SUB_IPV4_1394.txt");
#endif

 TL_conv = 1;
}

// Destructeur de la classe
Conversion::~Conversion(void)
{
 if (lnkTable_Corr != NULL)
   delete []lnkTable_Corr;
}
```

```
// Fonction qui charge le protocole source selon la description
// fournie dans le fichier passe en parametre
bool  Conversion::load_source_prot(char* file_name)
{
 return  lnkProtocole[0].addpkt(file_name);
}

// Fonction qui charge le protocole destiantion selon la description
// fournie dans le fichier passe en parametre
bool Conversion::load_dest_prot(char* file_name)
{
 return lnkProtocole[1].addpkt(file_name);
}


// Fonction qui convertit un paquet de type protocole source
// en protocole destination selon la solution non-optimisee
Quadlet *Conversion::convert_1_2_nopt(Quadlet pkt[])
{
 // Premiere partie : extractions
 char pkt_name[100];


 // Extraction de Firewire
 strcpy(pkt_name,"ASYNC_WRITE_S200");
 if (lnkProtocole[0].extract_pkt_h(pkt, pkt_name, 0, 0) == true){
  // Extraction de la couche virtuelle
  Quadlet *fir_data_size;
  fir_data_size = lnkProtocole[0].getfield_value(pkt_name, 8); // a MAJ
  if (fir_data_size!= NULL && fir_data_size[0] <= 1024){
   strcpy(pkt_name,"NOfragment");
   if (Prot_sublayer.extract_pkt_h(pkt, pkt_name, 5, 0) == true){
        // Extraction de l entete IPv4
        strcpy(pkt_name,"IPfragment");
        if (Prot_ipv4.extract_pkt_h(pkt, pkt_name, 6,0 ) == true) {
         // Recherche des adresses eth source et destination
         Quadlet *ip_addrs;
         Quadlet *ip_addrd;
         ip_addrs = Prot_ipv4.get_addr_s(pkt_name);
         ip_addrd = Prot_ipv4.get_addr_d(pkt_name);

         Quadlet *eth_addrs;
      Quadlet *eth_addrd;

        if (ip_addrs != NULL) {
```

```
  eth_addrs = lnkTable_Corr[1].get_addr(ip_addrs, 1,0);
}
else {
  cout<<"addr IPv4 src non trouve dans firewire pkt"<<endl;
  return NULL;
}

if (ip_addrd != NULL){
  eth_addrd = lnkTable_Corr[1].get_addr(ip_addrd, 1,0);
}
else {
  cout<<"addr IPv4 dest non trouve dans firewire pkt"<<endl;
  return NULL;
}


if (eth_addrs == NULL)  {
  eth_addrs = new Quadlet[2];
  eth_addrs[0] = 0xffffffff;
  eth_addrs[1] = 0x0000ffff;
}

if (eth_addrd == NULL) {
  eth_addrd = new Quadlet[2];
  eth_addrd[0] = 0xffffffff;
  eth_addrd[1] = 0xffffffff;
}

// Former l entete ethernet correspondante

//eth_len ou eth_type : 0x8000
lnkProtocole[1].setfield_value2(0, eth_addrs, 1);
lnkProtocole[1].setfield_value2(0, eth_addrs, 2);
Quadlet eth_type[1];
eth_type[0] = 0x8000;
lnkProtocole[1].setfield_value2(0, eth_type, 3);

// rajouter la charge utile du paquet ethernet cad datagramme IPV4
int len = (fir_data_size[0] + 10);
if ((len % 4) != 0)
  len = (len /4)+ 1;
else
  len = len /4;

Quadlet *pkt_dest;
pkt_dest = new Quadlet[len];
```

```
        // header
        strcatQuadlet(pkt_dest,lnkProtocole[1].get_header_q(0),
                0, 0, 0, 0, 112);
        // payload
        strcatQuadlet(pkt_dest, pkt,  6, 3, 0, 16, (fir_data_size[0]*8) - 32);
        return pkt_dest;
       }
       else
        return NULL;
     }
     else
       return NULL;
   }
   else
     return NULL;
 }
 else
   return NULL;
}

// Fonction qui convertit un paquet de type protocole
// destination en protocole source  selon la solution non-optimisee
Quadlet *Conversion::convert_2_1_nopt(Quadlet pkt[])
{
  // Premiere partie : extractions
  char pkt_name[100];

  // Extraction de ETHERNET
  strcpy(pkt_name,"NORMAL")
    ;
  if (lnkProtocole[1].extract_pkt_h(pkt, pkt_name, 0, 0) == true){
    // Extraction de la couche virtuelle
    Quadlet *eth_type;
    eth_type = lnkProtocole[1].getfield_value(pkt_name, 3); // a MAJ
    if ((eth_type!= NULL) && (eth_type[0] == 0x8000)){
      // Extraction de l entete IPv4
      strcpy(pkt_name,"IPfragment");
      if (Prot_ipv4.extract_pkt_h(pkt, pkt_name, 3, 16) == true){
        // ip_len quadlet
        Quadlet *ip_len;
        ip_len =  Prot_ipv4.getfield_value(pkt_name, 4);
        if ((ip_len != NULL) && (ip_len[0] > 1020)){
          cout<<"ERREUR IP len > 1020"<<endl;
          return NULL;
        }
```

```
// Recherche des adresses eth source et destination
Quadlet *ip_addrs;
Quadlet *ip_addrd;
ip_addrs = Prot_ipv4.get_addr_s(pkt_name);
ip_addrd = Prot_ipv4.get_addr_d(pkt_name);

Quadlet *fir_addrs;
Quadlet *fir_addrd;

if (ip_addrs != NULL)
  fir_addrs = lnkTable_Corr[0].get_addr(ip_addrs, 1,0);


if (ip_addrd != NULL)
  fir_addrd = lnkTable_Corr[0].get_addr(ip_addrd, 1,0);

if (fir_addrs == NULL){
  fir_addrs = new Quadlet[1];
  fir_addrs[0] = 0x0000ffff;
}

if (fir_addrd == NULL){
  fir_addrd = new Quadlet[1];
  fir_addrd[0] = 0x0000ffff;
}
// Former l entete ethernet correspondante
// fir adresses
lnkProtocole[0].setfield_value2(0, fir_addrd, 1);
lnkProtocole[0].setfield_value2(0, fir_addrs, 6);

// remplir les petits champs
Quadlet temp[2];
temp[1] = 0x0;
temp[0] = TL_conv; // a revoir tl - write request block data

// tl
//strcatQuadlet(pkt_dest, temp,0,0,0,16,6);
lnkProtocole[0].setfield_value2(0, temp, 2); //
// rt
temp[0] = 0x01;
lnkProtocole[0].setfield_value2(0, temp, 3); //

// tcode
temp[0] = write_block_req_tcode;
lnkProtocole[0].setfield_value2(0, temp, 4); //

// pri
temp[0] = 0;
lnkProtocole[0].setfield_value2(0, temp, 5); //

//destination_offset
temp[0] = 0x0000000;
lnkProtocole[0].setfield_value2(0, temp, 7); //
lnkProtocole[0].setfield_value2(0, temp, 9); //
lnkProtocole[0].setfield_value2(0, temp, 10); //

// data len + extended_code
temp[0] = ip_len[0] + 4;
lnkProtocole[0].setfield_value2(0, temp, 8); //

// construire l<entete virtuelle
eth_type[0] = 0x8000;
Prot_sublayer.setfield_value2(2, eth_type, 3);
eth_type[0] = 0x0000;
Prot_sublayer.setfield_value2(2, eth_type, 1);
Prot_sublayer.setfield_value2(2, eth_type, 2);

// rajouter la charge utile du paquet ethernet cad datagramme IPV4
int len = (ip_len[0] + 24);
if ((len % 4) != 0)
  len = (len /4)+ 1;
else
  len = len /4;

Quadlet *pkt_dest;
pkt_dest = new Quadlet[len];

// header
strcatQuadlet(pkt_dest, lnkProtocole[0].get_header_q(0),
        0, 0, 0, 0, 160);
strcatQuadlet(pkt_dest, Prot_sublayer.get_header_q(2),
        0, 5, 0, 0, 32);
// payload
strcatQuadlet(pkt_dest, pkt, 3, 6, 16, 0, (ip_len[0]*8));
return pkt_dest;
}
else
  return NULL;
}
else
  return NULL;
```

```
  }
  else
    return NULL;
  return NULL;
}

// Fonction qui convertit un paquet de type protocole source
// en protocole destination selon la solution optimisee
Quadlet *Conversion::convert_1_2(Quadlet pkt[])
{
  char pkt_name2[100];
  Quadlet *pkt_dest;

  pkt_dest = NULL;

#if test_print == 1
  cout<<endl<<"PKT_INT"<<endl;
  cout<<pkt<<endl<<endl<<endl;
#endif

  //lnkProtocole[0].get_payload(pkt_name);
  Quadlet *ip_addrs;
  Quadlet *ip_addrd;

  // rechercher la longueur du datagramme IP
  Quadlet *ip_len;
  ip_len = new Quadlet[1];
  ip_len[0] = pkt[6] ^ 0x0000FFFF;

  //recuperer les adresses source et destination
  ip_addrs =  new Quadlet[1];
  ip_addrs[0] = pkt[9];
  ip_addrd =  new Quadlet[1];
  ip_addrd[0] = pkt[10];

  // appeler les fonctions de recherche dans la table
  // ------------------------------------------------
  Quadlet *eth_addrs;
  Quadlet *eth_addrd;

  eth_addrs = lnkTable_Corr[1].get_addr(ip_addrs, 1,0);
  eth_addrd = lnkTable_Corr[1].get_addr(ip_addrd, 1,0);

  if (eth_addrs == NULL)  {
    eth_addrs = new Quadlet[2];
    eth_addrs[0] = 0xffffffff;
    eth_addrs[1] = 0xffffffff;
  }

  if (eth_addrd == NULL) {
    eth_addrd = new Quadlet[2];
    eth_addrd[0] = 0xffffffff;
    eth_addrd[1] = 0xffffffff;
  }
  // mettre a jour le champ longueur totale
  // --------------------------------------
  strcpy(pkt_name2,"NORMAL");


  // entete LLC
  Quadlet sof[1];
  int pkt_size;
  pkt_size = (ip_len[0] + 22)/4;
  if ((pkt_size%4) != 0)
    pkt_size ++;

  pkt_dest = new Quadlet[pkt_size];
  pkt_dest[0] = 0;

  strcatQuadlet(pkt_dest, eth_addrs,0,0,0,48);
  strcatQuadlet(pkt_dest, eth_addrd,0,1,0,16,48);

  // mettre a jour le champ longueur totale
  ip_len[0] = 0x8000;
  sof[0] = 0xaaaa;
  // entete SNAP
  strcatQuadlet(pkt_dest, ip_len,0,3,16,0,16);
  strcatQuadlet(pkt_dest, sof,0,3,16,16,16);

  sof[0] = 3;
  strcatQuadlet(pkt_dest, sof,0,4,24,0,8);
  sof[0] = 0;
  strcatQuadlet(pkt_dest, sof,0,4,0,24,24);

  sof[0] = 0x0800;
  strcatQuadlet(pkt_dest, sof,0,5,16,0,16);

  // former le paquet Ethernet header + payload
  //moccuper de la charge utile
  strcatQuadlet(pkt_dest, pkt,6,5,0,16,ip_len[0]*8);

#if test_print == 1
```

```
  unsigned int temp1;
  for (int i = 0; i<pkt_size; i++){
   for(int j=0; j< 4; j++){
     temp1 = pkt_dest[i];
     temp1 = temp1 << j*8;
     temp1 = unsigned int (temp1 >> 24);
     printf("%d\n",temp1);
   }
   cout<<endl;
  }
#endif

  return pkt_dest;
}

// Fonction qui convertit un paquet de type protocole
// destination en protocole source
Quadlet *Conversion::convert_2_1(Quadlet pkt[])
{
  //char pkt_name[100];
  //char pkt_name2[100];
  Quadlet *pkt_dest;
  //bool is_frag;
  Quadlet ip_len;
  pkt_dest = NULL;

#if test_print == 1
  cout<<endl<<"PKT_INT ETH to FIR"<<endl;
  cout<<pkt<<endl<<endl<<endl;
#endif

  //lnkProtocole[0].get_payload(pkt_name);
  Quadlet *ip_addrs;
  Quadlet *ip_addrd;

  int pkt_size;
  // verifier que le paquet comporte un datagramme IPv4
  if ((pkt[3]>>16) != 0x0800)
    return NULL;


  // recuperer la taille totale du paquet
  pkt_size = pkt[4]>> 16;

  // verifier que la taille limite des paquets (1020 o) est respectée
  if(pkt_size >= len_without_fragment)
    return NULL;

ip_len = pkt_size;

pkt_size = pkt_size + 24; // 24 = vitual layer len + firewire header len

if ((pkt_size%4) != 0)
  pkt_size ++;

pkt_dest = new Quadlet[pkt_size];
pkt_dest[0] = 0;



//recuperer les adresses source et destination
ip_addrs =  new Quadlet[1];
ip_addrs[0] = (pkt[6] << 16) && (pkt[7] >> 16);
ip_addrd =  new Quadlet[1];
ip_addrd[0] = (pkt[7] << 16) && (pkt[8] >> 16);

// appeler les fonctions de recherche dans la table
// ------------------------------------------------
Quadlet *eth_addrs;
Quadlet *eth_addrd;

eth_addrs = lnkTable_Corr[0].get_addr(ip_addrs, 1,0);
eth_addrd = lnkTable_Corr[0].get_addr(ip_addrd, 1,0);


if (eth_addrs == NULL)
  {
    eth_addrs = new Quadlet[1];
    eth_addrs[0] = 0xffff;
    eth_addrs[1] = 0xffff;
  }

if (eth_addrd == NULL)
  {
    eth_addrd = new Quadlet[1];
    eth_addrd[0] = 0xffff;
    eth_addrd[1] = 0xffff;
  }

strcatQuadlet(pkt_dest, eth_addrs,0,0,16,0,16);
strcatQuadlet(pkt_dest, eth_addrd,0,1,16,0,16);
```

```
// mettre a jour le champ longueur totale
// ---------------------------------------
Quadlet sof[1];
sof[0] = TL_conv; // a revoir tl - write request block data
/*TL_conv++;
  if (TL_conv == 64)
  TL_conv = 0;
*/
// tl
strcatQuadlet(pkt_dest, sof,0,0,0,16,6);
// rt
sof[0] = 0;
strcatQuadlet(pkt_dest, sof,0,0,0,10,2);
// tcode
sof[0] = write_block_req_tcode;
strcatQuadlet(pkt_dest, sof,0,0,0,10,4);
// pri
sof[0] = 0;
strcatQuadlet(pkt_dest, sof,0,0,0,10,4);

//destination_offset
sof[0] = 0xffffffff;
strcatQuadlet(pkt_dest, sof,0,1,0,16,16);
strcatQuadlet(pkt_dest, sof,0,2,0,0,32);

// data len + extended_code
sof[0] = ip_len + 4;
pkt_dest[3] = 0x00000000;
strcatQuadlet(pkt_dest, sof,0,3,16,0,16);

//header_crc initialise a 0
pkt_dest[4] = 0x00000000;

//COUCHE VIRTUELLE
sof[0] = 0x00000800;
strcatQuadlet(pkt_dest, sof,0,4,0,0,32);

//moccuper de la charge utile
strcatQuadlet(pkt_dest, pkt,3,6,16,0,ip_len*8);

#if test_print == 1
  unsigned int temp1;
  for (int i = 0; i<pkt_size; i++){
    for(int j=0; j< 4; j++){
      temp1 = pkt_dest[i];
      temp1 = temp1 << j*8;
      temp1 = unsigned int (temp1 >> 24);
      printf("%d\n",temp1);
    }
    cout<<endl;
  }
#endif

  return pkt_dest;
}


void extract()
{
}

void recup()
{
}

void format()
{
}

// Cette fonction retourne le numero de transaction
// qui sera contenu dans le paquet Firewire
char* Conversion::get_trans_label()
{
  return "000001";
}

// cette foncntion calcule la taille d une chaine de caracteres
// envoyee en parametre et elle convertit cette taille en chaine
// de caracteres de 16 car.
char* Conversion:: get_length(char* pkt)
{
  int length;
  length = strlen(pkt);
  return InttoString(length,16);
  //return "1111111111111111";
}

// cette fonction renvoie la valeur par defaut de l entete virtuelle
char* Conversion:: get_virtual_layer()
{
  return "00000000000000000000100000000000";
}
```

```
// retourne les 32 derniers derniers bits d'un paquet
char* Conversion::get_pad(char* tmp)
{
  int i,n,k,l;
  char *pad;
  pad = new char[32];
  n= strlen(tmp);
  k = n % 32;
  l = 32-k;

  for (i=0;i<l;i++)
    pad[i]='0';

  pad[i] = '\0';
  return pad;
}


// cette fonction calcule le checksum d un paquet selon
// l algorithme du CRC 32
char* Conversion:: get_CRC(char* data){
  unsigned int       result;
  int                i,j;
  unsigned char      octet;
  int                QUOTIENT = 0x04c11db7;
  int len = strlen(data);

  if (len < 4) abort();

  result = *data++ << 24;
  result |= *data++ << 16;
  result |= *data++ << 8;
  result |= *data++;
  result = ~ result;
  len -=4;

  for (i=0; i<len; i++)
   {
     octet = *(data++);
     for (j=0; j<8; j++)
      {
           if (result & 0x80000000)
              result = (result << 1) ^ QUOTIENT ^ (octet >> 7);
           else
              result = (result << 1) ^ (octet >> 7);
           octet <<= 1;
      }
   }
     result= ~result;
  return InttoString(result,32);
}

// cette fonction n'est pas encore implementee
bool Conversion::is_conv_valid(){
  return false;
}
```

D.3 Protocole

```
/*****************************************************************************/
/* Title        : Protocole                                                  */
/* Project      : Convertisseur de protocoles: transport de datagrammes IP   */
/*****************************************************************************/
/* File         : protcoles.h                                                */
/* Author       : Mame Maria Mbaye                                           */
/* Created      : 07-06-2002                                                 */
/* Last modified : 24-02-2003                                                */
/*****************************************************************************/
/* Description :   module comporte la declaration des attributs et methodes de*/
/*              la protocole                                                 */
/*****************************************************************************/
/* Generated by Together */

#ifndef PROTOCOLE_H
#define PROTOCOLE_H
#include "Packet.h"
#include "header_field.h"
class Protocole {
 public:
  Protocole(void);
  ~Protocole(void);
  bool addpkt(char *nomfichier);
  bool extract_pkt(Quadlet *pkt_s,char *pkt_name, int h_debut);
  bool extract_pkt_h(Quadlet *pkt_c,char *pkt_name, int h_debut, int shift);
  bool extract_pkt_p(Quadlet *pkt_c,char *pkt_name, int size, int start);
  char *get_payload(char *pkt_name);
  char *get_header_c(char *pkt_name);
  Quadlet *get_header_q(int pkt_name);

  Quadlet *get_addr_s(char *pkt_name);
  Quadlet *get_addr_d(char *pkt_name);
  char *get_checksum(char *pkt_name);
  int  get_lentot(char *pkt_name);
  void convert_addr_s(char *addr);
  void convert_addr_d(char *addr);
  int find_pkt_ind(char *pkt_name);

  void display_protocole(void);
  Packet *get_Packet(char *pkt_name);
  void set_Packet(char *pkt_name,const Packet &P);
  Quadlet *getfield_value(char *pkt_name, int find);
  void setfield_value(char *pkt_name, Quadlet *fvalue, int find);
  void setfield_value2(int pkt_name, Quadlet *fvalue, int find);

 private:
  char* prot_name;

  /** @link aggregation
   * @supplierCardinality 1..**/
  Packet *lnkPacket;
  int nb_packet;
};
#endif //PROTOCOLE_H


/*****************************************************************************/
/* Title        : Protocole                                                  */
/* Project      : Convertisseur de protocoles: transport de datagrammes IP   */
/*****************************************************************************/
/* File         : protcoles.cpp                                              */
/* Author       : Mame Maria Mbaye                                           */
/* Created      : 07-06-2002                                                 */
/* Last modified : 24-02-2003                                                */
/*****************************************************************************/
/* Description :   module comporte les fonctions qui permettent de manipuler */
/*              les champs d un paquet selon uen descirption de protocole    */
/*              donnee                                                       */
/*****************************************************************************/
/* Generated by Together */

#include "Protocole.h"
#include <string.h>
#include "arm.h"

#if ARM
#include <fstream>
#include <iostream>
#else
#include <fstream.h>
#include <iostream.h>
#endif



// Definition des methodes
/************************************************/

// Construction de la classe
```

```
Protocole::Protocole(void){
  nb_packet = 0;
  lnkPacket = NULL;
}

// Desctructeur de la classe
Protocole::~Protocole(void){
  if (lnkPacket != NULL)
    delete []lnkPacket;

}

// Cette fonction renvoie l indice associe a type de paquet
// selon l empilement de protocoles defini
int Protocole::find_pkt_ind(char *pkt_name){
  int ind = -1;
  char pkt[100];

  for (int i=0;i<nb_packet; i++){
    strcpy(pkt,lnkPacket[i].getpkt_name());
    if (!strcmp(pkt_name,pkt)) {
      ind = i;
      break;
    }
  }

  return ind;
}

// Cette fonction charge la description d un protocole
// la description etant contenu dans un fichier dont
// le nom est passe en parametre
bool Protocole::addpkt(char *nomfichier){
  bool success = false;
  ifstream prot_fich(nomfichier, ios::in);


  if (!prot_fich){
    cout<<"erreur durant l<ouverture du fichier";
    return success;
  }

  char c_temp[100];

  int i_temp;
  int pkt_ind,pkt_nbf;

  prot_fich>>c_temp;

  if (!strcmp(c_temp, "Prot:")){
    prot_fich>>c_temp;
    prot_name = new char[strlen(c_temp+1)];
    strcpy(prot_name,c_temp);
  }
  else{
    return success;
  }

  prot_fich>>c_temp;
  if (!strcmp(c_temp,"NBPKT:")){
    prot_fich>>i_temp;
  }

  nb_packet = i_temp;
  if (i_temp > 0){
    lnkPacket = new Packet[i_temp];
    for (int j =0; j<i_temp; j++)
      lnkPacket[j].resetpkt();
  }

  int pos = 0;
  int size = 0;
  int min =0;
  int max = 0;
  char *name;
  name = new char[100];

  prot_fich>>c_temp;
  pkt_ind = -1;
  while(!prot_fich.eof())
    {
      if (!strcmp(c_temp,"Pkt:")){
          pkt_ind ++;
          prot_fich>>c_temp;
          lnkPacket[pkt_ind].setpkt_name(c_temp);
          prot_fich>>i_temp;
          lnkPacket[pkt_ind].setpkt_min_size(i_temp);
          prot_fich>>i_temp;
          lnkPacket[pkt_ind].setpkt_max_size(i_temp);
          prot_fich>>pkt_nbf;
          int pkt_nbh, ind_h, ind_p;
          ind_h = 0;
```

```
        ind_p = 0;

        prot_fich>>pkt_nbh;
        header_field *header;
        header = new header_field(pkt_nbh);
        payload_field *payload = NULL;


        for (int i=0; i<pkt_nbf; i++){
          prot_fich>>pos>>c_temp;

          if ((c_temp!= NULL) && (c_temp[0] == 'H')){
            prot_fich>>name>>size;
            Field f(name,c_temp,pos,size);
            header->addfield(f,ind_h);
            ind_h++;

          }
          else if ((c_temp!=NULL) && (!strcmp(c_temp,"PXX"))){
            prot_fich>>name>>min>>max;
            payload = new payload_field;
            payload->setminsize(min);
            payload->setmaxsize(max);
            Field f(name,c_temp,pos,0);
            payload->setpayloadfield(f);
          }
        }

        lnkPacket[pkt_ind].setpktheader(header);
        lnkPacket[pkt_ind].setpktpayload(payload);

      }
      prot_fich>>c_temp;
    }

  success = true;
  return success;
}

// fonction qui effectue l extraction des champs d un paquet envoye
// en parametre et dont le type est aussi envoye en parametre
bool Protocole::extract_pkt(Quadlet *pkt_s,char *pkt_name, int h_debut)
{
  int pkt_ind = find_pkt_ind(pkt_name);
  bool success = false;

  if (pkt_ind != -1){
    success = lnkPacket[pkt_ind].extract_pkt(pkt_s, h_debut);
  }

  return success;
}

// extraction de l entete du paquet
bool Protocole::extract_pkt_h(Quadlet *pkt_c,char *pkt_name, int h_debut, int
shift){
  int pkt_ind = find_pkt_ind(pkt_name);
  bool success = false;

  if (pkt_ind != -1){
    success = lnkPacket[pkt_ind].extract_pkt_h(pkt_c, h_debut, shift);
  }

  return success;

}

// extraction du payload du paquet
bool Protocole::extract_pkt_p(Quadlet *pkt_c,char *pkt_name, int size, int start){

  int pkt_ind = find_pkt_ind(pkt_name);
  bool success = false;

  if (pkt_ind != -1){
    success = lnkPacket[pkt_ind].extract_pkt_p(pkt_c, size,start);
  }

  return success;

}

// Cette fonciton renvoie la charge utile d un paquet
char *Protocole::get_payload(char *pkt_name)
{
  payload_field pay;
  int i = find_pkt_ind(pkt_name);

  if (i != -1){
    return lnkPacket[i].getpayload_c();
  }
  else {
    return NULL;
```

```
  }
}

// Cette fonction renvoie l entete d un paquet sosu forme de string
char *Protocole::get_header_c(char *pkt_name)
{
  payload_field pay;
  int i = find_pkt_ind(pkt_name);


  if (i != -1){
    return lnkPacket[i].getpkt_header_c();
  }
  else {
    return NULL;
  }
}

// Cette fonction renvoie l adresse source d un paquet
Quadlet *Protocole::get_addr_s(char *pkt_name)
{
  payload_field pay;
  int i = find_pkt_ind(pkt_name);

  if (i != -1){
    return lnkPacket[i].get_addrs();
  }
  else {
    return NULL;
  }
}

// Cette fonction renvoie l adresse destination d un paquet
Quadlet *Protocole::get_addr_d(char *pkt_name)
{
  payload_field pay;
  int i = find_pkt_ind(pkt_name);

  if (i != -1){
    return lnkPacket[i].get_addrd();
  }
  else {
    return NULL;
  }
}
```

```
// cette fonction renvoie le checksum d un paquet
char *Protocole::get_checksum(char *pkt_name)
{
  payload_field pay;
  int i = find_pkt_ind(pkt_name);

  if (i != -1){
    return lnkPacket[i].getpayload_c();
  }
  else {
    return NULL;
  }
}

// Cette fonction renvoie la taille totale d un paquet
int  Protocole::get_lentot(char *pkt_name)
{
  payload_field pay;
  int i = find_pkt_ind(pkt_name);

  if (i != -1){
    return lnkPacket[i].getpkt_len();
  }
  else {
    return 0;
  }
}

// Ces deux prochaines fonctions ne sont pas implementees pour l instant
void Protocole::convert_addr_s(char *addr)
{
}

void Protocole::convert_addr_d(char *addr)
{
}

// cette fonction affiche la description d un protocole
void Protocole::display_protocole(void){
  for (int i = 0; i< nb_packet; i++)
    lnkPacket[i].display_pkt();

}

// Cette fonction renvoie un paquet de type Packet
Packet *Protocole::get_Packet(char *pkt_name){
```

```
  Packet *p;

  p = NULL;
  int i = find_pkt_ind(pkt_name);

  if (i != -1){
    p = new Packet;
    *p = lnkPacket[i];
  }

  return p;

}

// Copie un paquet dans la structure Protocole courante
void Protocole::set_Packet(char *pkt_name,const Packet &P){

  int i = find_pkt_ind(pkt_name);

  if (i != -1){
    lnkPacket[i] = P;
  }
}

// Cette fonction renvoie la valeur d un champ du paquet
Quadlet *Protocole::getfield_value(char *pkt_name, int find)
{
  int i = find_pkt_ind(pkt_name);

  if (i != -1){
    return lnkPacket[i].getfield_value(find);
  }

  return NULL;
}

// Cette fonction intialise la valeur d un champ du paquet
void Protocole::setfield_value(char *pkt_name, Quadlet *fvalue, int find){

  int i = find_pkt_ind(pkt_name);

  if (i != -1){
    lnkPacket[i].setfield_value(fvalue, find);
  }

}
```

```
// Cette fonction intialise la valeur d un champ du paquet
void Protocole::setfield_value2(int pkt_name, Quadlet *fvalue, int find){
  if (pkt_name != -1){
    lnkPacket[pkt_name].setfield_value(fvalue, find);
  }
}

// renvoie le derneir champ de l entete
Quadlet *Protocole::get_header_q(int pkt_name){

  if (pkt_name != -1){
    return lnkPacket[pkt_name].get_header_q();
  }
  else
    return NULL;
}
```

D.4 Packet

```
/*****************************************************************************/
/* Title        : Packet                                                     */
/* Project      : Convertisseur de protocoles: transport de datagrammes IP   */
/*****************************************************************************/
/* File         : Packet.h                                                   */
/* Author       : Mame Maria Mbaye                                           */
/* Created      : 07-06-2002                                                 */
/* Last modified : 24-02-2003                                                */
/*****************************************************************************/
/* Description :  Declaration des attributs et methodes de la classe Packet  */
/*****************************************************************************/
/* Generated by Together */

#ifndef PACKET_H
#define PACKET_H
#include "header_field.h"
#include "payload_field.h"

class Packet {
 public:
  Packet(void);
  Packet(char *name, int min, int max);
  Packet(Packet *P);
  ~Packet(void);
  void setpkt_min_size(int size);
  void setpkt_max_size(int size);
  void setpkt_name(char *name);
  int  getpkt_min_size(void);
  int  getpkt_max_size(void);
  char *getpkt_name(void);
  void setpktheader(header_field *head);
  void addfheader(Field &f, int ind);
  void setpktpayload(payload_field *pay);
  header_field getheader(void);
  payload_field getpayload(void);

  char *getpayload_c(void){return lnkpayload_field->getpayload_c();};

  char *getpkt_c(void);
  char *getpkt_header_c(void);
  Quadlet *get_header_q(void);
  int  getpkt_len(void);
  int  getheader_len(void);
  int  getpayload_len(void);
  bool is_header(void);
  bool is_payload(void);
  void resetpkt(void);
  bool extract_pkt(Quadlet *pkt_c, int h_debut);
  bool extract_pkt_h(Quadlet *pkt_c, int h_debut, int shift);
  bool extract_pkt_p(Quadlet *pkt_c, int size, int start);

  void display_pkt(void);
  Quadlet *get_addrs(void);
  Quadlet *get_addrd(void);
  void set_addrs(Quadlet *addr);
  void set_addrd(Quadlet *addr);
  void setfield_value(Quadlet *fvalue, int find);
  Quadlet *getfield_value(int find);
  void set_lentot_field(int len);
  const Packet &operator = (const Packet &p);


 private:

  /** @link aggregation
   * @clientCardinality 1..**/
  header_field *lnkheader_field;

  int pkt_min_size;
  int pkt_max_size;
  char* pkt_name;

  /** @link aggregation
   * @clientCardinality 1..**/
  payload_field *lnkpayload_field;
};
#endif //PACKET_H



/*****************************************************************************/
/* Title        : Packet                                                     */
/* Project      : Convertisseur de protocoles: transport de datagrammes IP   */
/*****************************************************************************/
/* File         : Packet.cpp                                                 */
/* Author       : Mame Maria Mbaye                                           */
/* Created      : 07-06-2002                                                 */
/* Last modified : 24-02-2003                                                */
/*****************************************************************************/
```

```
/* Description :   Cette classe decrit les manipulations qu on effectue sur  */
/*                 entete et la charge utile d un paquet donne                */
/*****************************************************************************/
/* Generated by Together */

#include "Packet.h"
#include <string.h>
#include "utils.h"
#if ARM
#include <fstream>
#include <iostream>
#else
#include <iostream>
#include <conio.h>
#endif

// Constructeur par defaut
Packet::Packet(void)
{
  // initialiser les attributs de la classe
  pkt_max_size = 0;
  pkt_min_size = 0;
  pkt_name = NULL;
  lnkheader_field = NULL;
  lnkpayload_field = NULL;
}

// Constructeu
Packet::Packet(char *name, int min, int max){
  // initialiser les les attibuts du paquet
  pkt_max_size = max;
  pkt_min_size = min;
  pkt_name = new char[strlen(name)+1];
  strcpy(name,pkt_name);
  lnkheader_field = NULL;
  lnkpayload_field = NULL;
}

// Constructeur Copie
Packet::Packet(Packet *P){
  pkt_max_size = P->pkt_max_size;
  pkt_min_size = P->pkt_min_size;

  // copie le nom du paquet
  if (P->pkt_name != NULL)
    pkt_name = new char[strlen(P->pkt_name)+1];
  strcpy(pkt_name, P->pkt_name);

  // copie l entete
  if (P->is_header() == true){
    header_field h;
    h = P->getheader();
    lnkheader_field = new header_field(h);
  }
  else
    lnkheader_field = NULL;

  // copie la chareg utile
  if (P->is_payload() == true){
    payload_field p;
    p = P->getpayload();
    lnkpayload_field = new payload_field(p);
  }
  else
    lnkpayload_field = NULL;
}

// Destructeur
Packet::~Packet(void)
{
  if (lnkpayload_field != NULL)
    delete lnkpayload_field;

  if (lnkheader_field != NULL)
    delete lnkheader_field ;
}

// Precise si le paquet comporte une entete ou pas
bool Packet::is_header(void){
  if (lnkheader_field == NULL)
    return false;
  else
    return true;
}

// Precise si le paquet comporte une charge utile ou pas
bool Packet::is_payload(void){
  if (lnkpayload_field == NULL)
    return false;
  else
    return true;
```

```
}

// initialise la taille minimale d un paquet
void Packet::setpkt_min_size(int size)
{
  pkt_min_size = size;
}

// initialise la taille maximale d un paquet
void Packet::setpkt_max_size(int size)
{

  pkt_max_size = size;
}

// initialise le nom du type de paquet
void Packet::setpkt_name(char *name)
{
  if (pkt_name != NULL){
    delete [] pkt_name;
  }

  pkt_name = new char[strlen(name)+1];
  strcpy(pkt_name,name);
}

// renvoie la taille minimale d un paquet
int Packet::getpkt_min_size(void)
{
  return pkt_min_size;
}

// renvoie la taille maximale d un paquet
int Packet::getpkt_max_size(void)
{
  return pkt_max_size;
}

// renvoie le nom du type de paquet
char *Packet::getpkt_name(void)
{
  char *temp;
  temp = NULL;

  if (pkt_name != NULL)
    {
      temp = new char[strlen(pkt_name)+1];
      strcpy(temp,pkt_name);
    }
  return temp;
}

// Efface le contenu du paquet
void Packet::resetpkt(void){
  if (lnkpayload_field != NULL)
    delete []lnkpayload_field;
  else
    lnkheader_field = NULL;

  if (lnkheader_field != NULL)
    delete []lnkheader_field ;
  else
    lnkpayload_field = NULL;


  if (pkt_name != NULL)
    delete []pkt_name;
  else
    pkt_name = NULL;
}

// initialise l entete d un paquet
void Packet::setpktheader(header_field *head)
{
  if (lnkheader_field != NULL)
    delete lnkheader_field;

  if (head != NULL)
    lnkheader_field = new header_field(head);
  else
    lnkheader_field = NULL;
}

// initialise la charge utile d un paquet
void Packet::setpktpayload(payload_field *pay)
{
  if (lnkpayload_field != NULL)
    delete lnkpayload_field;


  if (pay != NULL)
    lnkpayload_field = new payload_field(pay);
```

```
  else
    lnkpayload_field = NULL;
}

// renvoie l entete d un paquet
header_field Packet::getheader(void)
{
  return lnkheader_field->getheader();
}

// renvoie la charge utile d un paquet
payload_field Packet::getpayload(void)
{
  payload_field p;
  p = lnkpayload_field->getpayload();
  return p;
}

// renvoie la taille totale d un paquet
int  Packet::getpkt_len(void){

  return (lnkheader_field->getheader_len() + lnkpayload_field->getpayload_len());

}

// renvoie la taille de l entete d un paquet
int  Packet::getheader_len(void){
  return lnkheader_field->getheader_len();

}

// renvoie la taille de la charge utile d un paquet
int  Packet::getpayload_len(void){
  return lnkpayload_field->getpayload_len();
}

// rajoute un champ dans la liste des champs de l entete
void Packet::addfheader(Field &f, int ind){
  lnkheader_field->addfield(f,ind);
}

// recoit uen chaine de caracteres et extrait les valeurs des
// differents champs du paquet
bool Packet::extract_pkt(Quadlet *pkt_c, int h_debut)
{
  bool success = false;
  //header_field *header;
  //payload_field payload;

  int start;
  int size_p;
  if (lnkheader_field != NULL){
    // trouver le debut et la fin de la charge utile
    int len = lnkheader_field->getheader_len();
    success = lnkheader_field->extract_h(pkt_c, h_debut, 0);
    start = h_debut - (lnkheader_field->getheader_len()/32);
    Quadlet *len1;
    len1 = NULL;
    len1 = lnkheader_field->get_lentot_field() ;
    if (len1 != NULL)
      size_p = len1[0] / 8 ;
  }

  if (lnkpayload_field != NULL){
    success = lnkpayload_field->extract_payload(pkt_c, size_p, start);
  }
  return success;
}

// extraction des chamsp de l entete du paquet
bool Packet::extract_pkt_h(Quadlet *pkt_c, int h_debut, int shift)
{
  bool success = false;

  int start, end;
  start = 0;
  end = 0;

  if (lnkheader_field != NULL){
    // trouver le debut et la fin de la charge utile
    success = lnkheader_field->extract_h(pkt_c, h_debut, shift);
  }
  return success;
}

// extraction du payload du paquet
bool Packet::extract_pkt_p(Quadlet *pkt_c, int size, int start)
{
  bool success = false;

  if (lnkpayload_field != NULL){
    // trouver le debut et la fin de la charge utile
```

```
   success = lnkpayload_field->extract_payload(pkt_c, size, start);

  }
  return success;
}

// Affiche les valeurs des differents champs du paquet
void Packet::display_pkt(void){
  if (lnkheader_field != NULL)
    lnkheader_field->display_header();

  if (lnkpayload_field != NULL)
    lnkpayload_field->display_payload();
}


// renvoie l adresse source du paquet
Quadlet *Packet::get_addrs(void)
{
  Quadlet *addr;
  addr = NULL;

  if (lnkheader_field != NULL) {
    addr = lnkheader_field->get_addrs();
  }
  return addr;
}

// initialise l adresse destination du paquet
Quadlet *Packet::get_addrd(void)
{
  Quadlet *addr;
  addr = NULL;

  if (lnkheader_field != NULL) {
    addr = lnkheader_field->get_addrd();
  }
  return addr;
}

// renvoie l adresse source du paquet
void Packet::set_addrs(Quadlet *addr){
  if (lnkheader_field != NULL){
    lnkheader_field->set_addrs(addr);
  }
}
```

```
// renvoie l adresse destination du paquet
void Packet::set_addrd(Quadlet *addr){

  if (lnkheader_field != NULL)   {
    lnkheader_field->set_addrd(addr);
  }
}

// initialise la longueur totale de l entete du paquet
void Packet::set_lentot_field(int len){
  if (lnkheader_field != NULL)   {
    lnkheader_field->set_lentot_field(len);
  }
}

// surcharge l operateur d affectation
const Packet &Packet::operator =(const Packet &p){

  if (this != &p){
    pkt_min_size = p.pkt_min_size;
    pkt_max_size = p.pkt_max_size;

    // copier le nom du paquet
    if (pkt_name != NULL)
      delete pkt_name;

    if (p.pkt_name != NULL)   {
      pkt_name = new char[strlen(p.pkt_name)+1];
      strcpy(pkt_name, p.pkt_name);
    }

    // copier l entete
    if (p.lnkheader_field != NULL)  {
      header_field *f;
      f = new header_field;
      *f = p.lnkheader_field->getheader();
      setpktheader(f);
    }
    else{
      if (lnkheader_field != NULL)
          delete []lnkheader_field;

    }

    // copier la charge utile
```

```
  if (p.lnkpayload_field != NULL)  {
    payload_field *pay;
    pay = new payload_field;
    *pay = p.lnkpayload_field->getpayload();
    setpktpayload(pay);
  }
  else    {
    if (lnkpayload_field != NULL)
        delete []lnkpayload_field;
  }
 }
 return *this;
}

// renvoie un paquet sous forme de chaine de caracteres
char *Packet::getpkt_c(void)
{
 char *temp;
 temp = NULL;

 return temp;
}

// rnevoie l entete du paquet sosu forme de chaine des caraceteres
char *Packet::getpkt_header_c(void)
{
 char *temp;
 temp = NULL;

 if (lnkheader_field != NULL)
  {
   temp = new char[pkt_max_size+1];
   temp[0]= '\0';
  }

 return temp;
}

// renvoie le dernier champ de l entete
Quadlet *Packet::get_header_q(void){

 if (lnkheader_field != NULL)
  {
   return lnkheader_field->getheader_q();
  }
 else
   return NULL;
}

// initialise la valeur d un champ du paquet
void Packet::setfield_value(Quadlet *fvalue, int find)
{
 if (lnkheader_field != NULL)
   lnkheader_field->setfield_value(fvalue,find);

}

// renvoie la valeur d un champ du paquet
Quadlet *Packet::getfield_value(int find)
{
 if (lnkheader_field != NULL)
   return lnkheader_field->getfield_value(find);

 return NULL;
}
```

Annexe E

Fonctionnalités du pilote FireoverIP

E.1 Analyse du module d'adressage

Dans la suite de cette section, nous allons considérer que chaque réseau comporte 2 équipements et que la passerelle est un équipement Firewire à part entière.

Lors de la mise sous tension de la passerelle (nommée convertisseur sur les diagrammes de scénarios), la première opération qui s'opère est le démarrage des services tels que le service réseau. Durant cette étape, la passerelle commence par avertir les PC se trouvant dans le réseau Ethernet qu'elle vient de se connecter sur le réseau. La Figure E-1 montre bien que le démarrage comporte deux scénarios indépendants : l'insertion dans le réseau Ethernet et la réinitialisation du réseau Firewire.

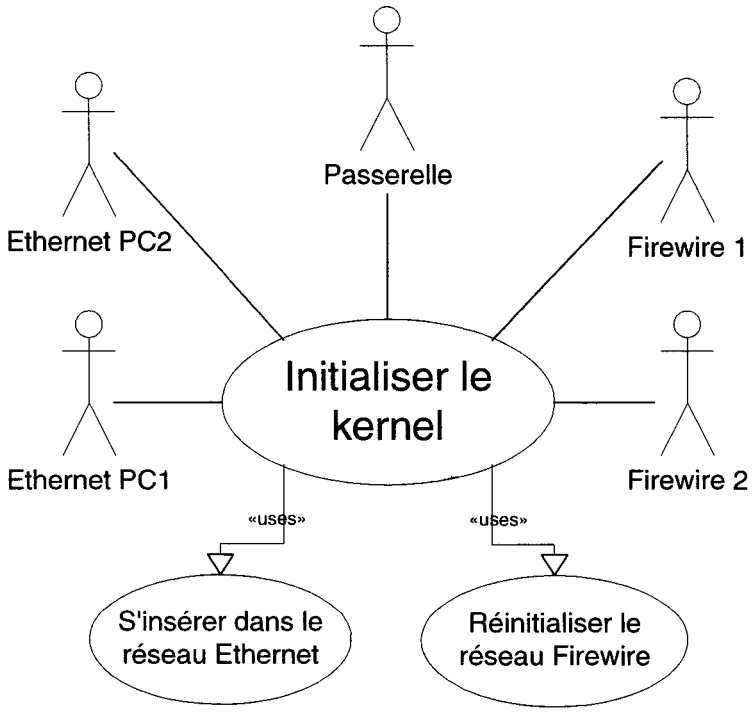

Figure E-1. Scénario: Démarrage de la Passerelle

Le scénario : « réinitialiser le réseau Firewire » comporte de nombreux sous-scénarios. Il faut rappeler que l'ajout ou le retrait d'un équipement Firewire induit une interruption du réseau et une réallocation des adresses physiques de chaque équipement.

Après cette ré-allocation d'adresses physiques Firewire, la passerelle devra en profiter pour attribuer une adresse IPv4 à chaque équipement et mettre à jour ses tables de routage, comme le montre la Figure E-2. Une des phases importantes durant cette phase concerne ces adresses IPv4. **La passerelle devrait avoir une banque d'adresses IPv4 ou les demander par des requêtes au serveur DHCP.**

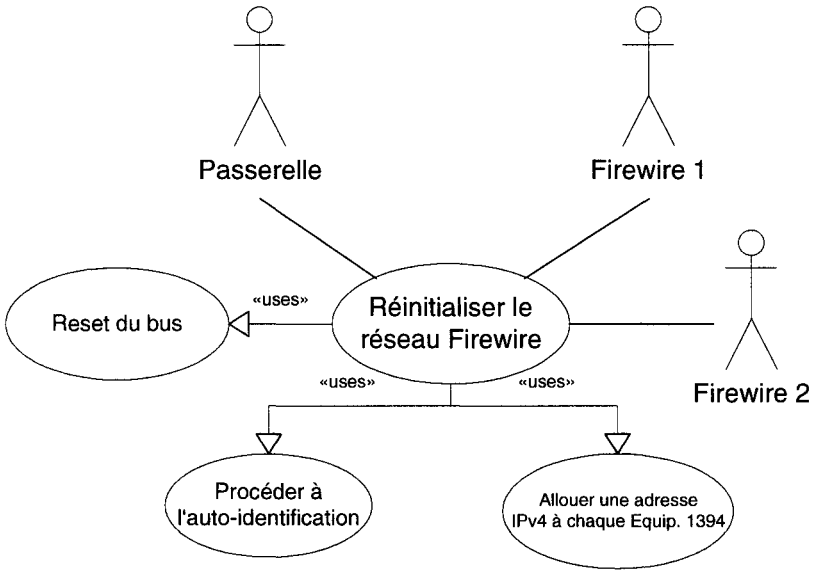

Figure E-2. Détails de la réinitialisation avec le réseau Firewire

Considérons un exemple. À la fin de la séquence d'initialisation, la passerelle aura attribué les adresses 132.207.108.62 à la WebCam1 et 132.207.108.63 à la WebCam2, comme le montre la Figure E-3. Il faut préciser que ces adresses sont virtuelles, étant donné que seule la passerelle sait que ces équipements ont ces adresses. Ces équipements 1394 opèrent juste avec leurs adresses physiques Firewire qui ont été allouées durant le processus d'auto identification de Firewire.

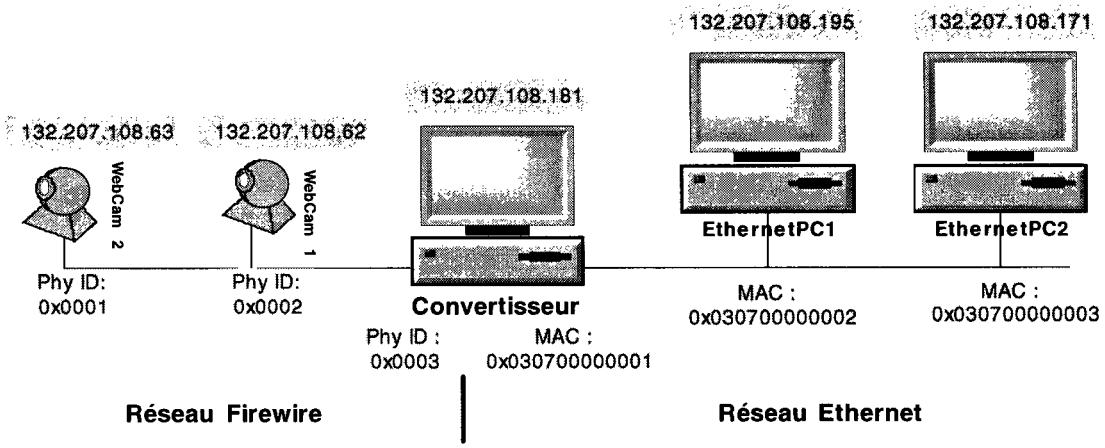

Figure E-3. État des réseaux après l'étape d'initialisation

La passerelle, pour sa part, aura une adresse physique Firewire et une adresse MAC. Les équipements Ethernet, pour envoyer un paquet vers un équipement Firewire, indiqueront l'adresse MAC de la passerelle dans le champ « adresse MAC destination » de l'entête Ethernet, cependant, ils préciseront l'adresse IP de l'équipement Firewire adressé. La passerelle, à son tour, précisera l'adresse physique dans le paquet Firewire constitué.

En plus de l'attribution d'adresses, le module de résolution d'adresses sera consulté par le pilote TCP/IP, dès que ce dernier recevra ou devra transmettre un paquet TCP ou UDP vers le réseau Ethernet.

La Figure E-4 illustre un premier cas durant lequel le pilote TCP/IP fera appel au pilote *FireoverIP* après la réception d'une requête ARP (Plummer, 1982) par le PC1. Ce dernier recherche l'adresse MAC de l'équipement ayant l'adresse : 132.207.108.62. La première opération que le pilote TCP/IP effectue en recevant cette requête est de vérifier que l'adresse IPv4 destination contenue dans le paquet correspond à l'adresse IPv4 de la passerelle. Si cette opération échoue, le pilote TCP/IP fera appel au pilote *FireoverIP*, pour que ce dernier vérifie si cette adresse correspond à celle d'un des équipements Firewire. Si le pilote *FireoverIP* ne trouve pas de correspondance, le pilote TCP/IP ne répondra pas à la requête ARP. S'il trouve l'adresse, le pilote TCP/IP constituera une réponse ARP en y précisant l'adresse MAC de la carte de la passerelle et la renverra sur le lien Ethernet, via la carte réseau.

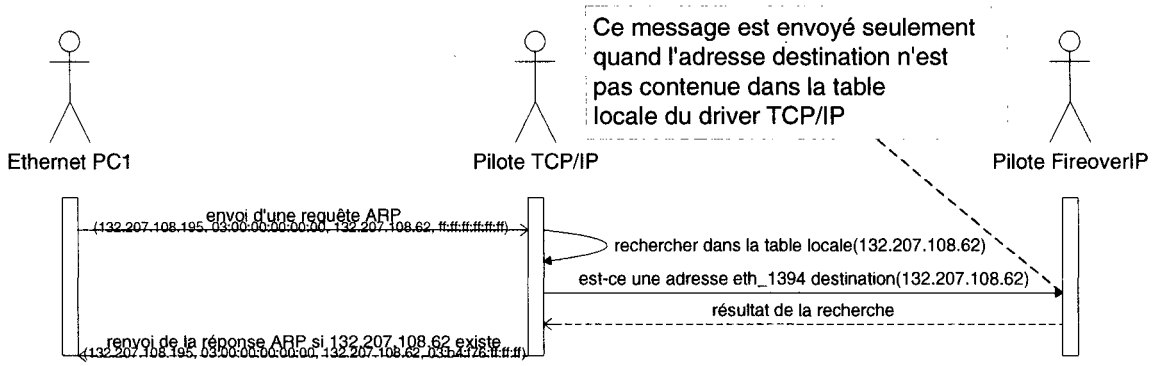

Figure E-4. Séquence d'opérations après la réception d'une requête ARP **caractères trop petits dans les échanges de gauche **

Dans un réseau TCP/IP, on retrouve principalement des communications client/serveur. Ce type de communication commence par une demande d'ouverture de session TCP. La Figure E-5 montre qu'au début de la réception de la demande, le pilote TCP/IP effectue les mêmes tâches, c'est à dire la validation de l'adresse destination contenue dans l'entête du paquet IPv4. Si l'adresse correspond à celle de la passerelle ou à celle d'un des équipements Firewire, le pilote peut commencer à traiter cette demande de connexion. La demande sera acceptée s'il y a un *socket* qui accepte de communiquer sur le port TCP précisé dans la demande TCP. Il faut préciser qu'un *socket* est un objet qui permet d'établir un canal virtuel entre deux ou plusieurs machines, en faisant abstraction des couches TCP/IP. Donc, il faudrait qu'au niveau usager, une application accepte de communiquer avec la machine demandant la connexion. Cette application crée le *socket* et attend généralement les demandes de connexion venant des clients.

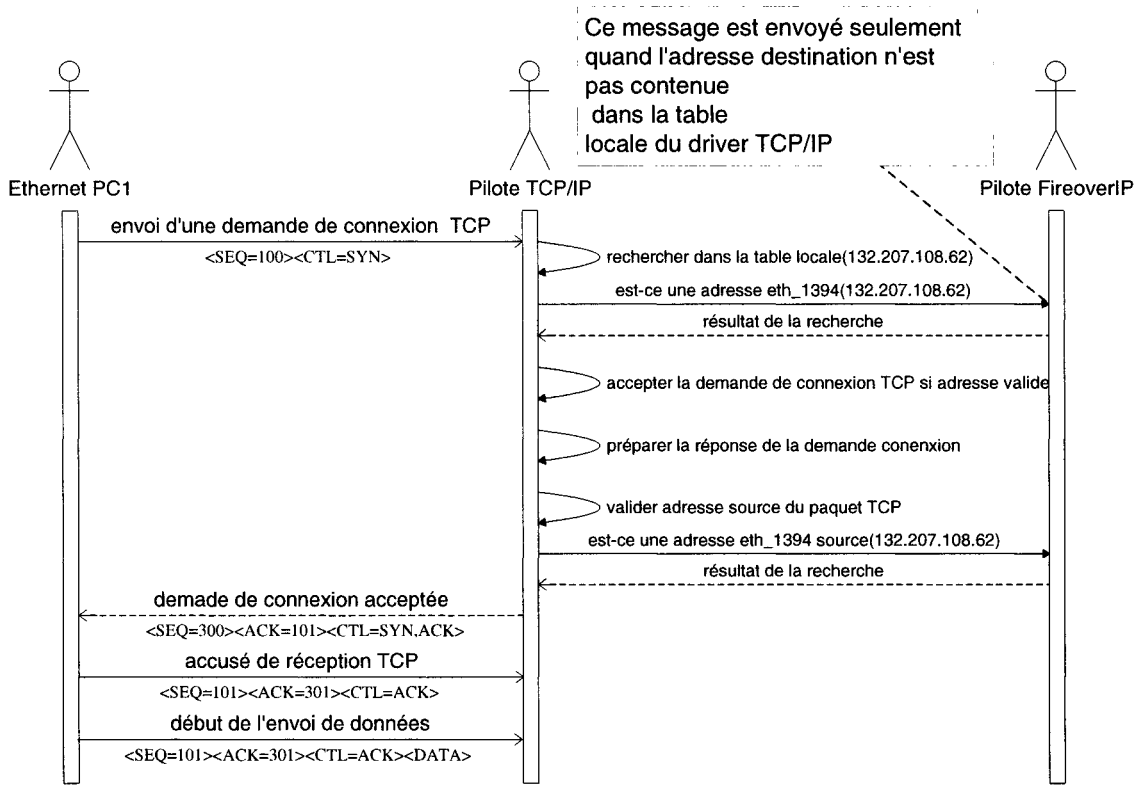

Figure E-5. Séquence d'opérations après la réception d'une demande d'ouverture de session TCP

Cet exemple a été présenté pour faire ressortir un autre service que la passerelle devra offrir. En effet, la communication en mode TCP nécessite la gestion d'erreurs par la passerelle. Le paquet TCP comporte des numéros de séquence permettant de vérifier si des données ont été perdues durant la transmission. Après chaque envoi de données, l'équipement source s'attend à recevoir un accusé de réception. Le PC1 devra envoyer un accusé de réception à la passerelle pour lui signifier qu'il a bien reçu sa réponse. La passerelle devra renvoyer le même paquet si elle ne reçoit pas l'accusé de réception dans un délai raisonnable. Dans l'exemple, le numéro de séquence initial généré par la passerelle était égal à 300. Ainsi la passerelle s'attend à recevoir un accusé de réception dont le champ « Ack number » comportera la valeur 301. Si le prochain paquet ne comporte pas cette valeur, la passerelle devra en conclure qu'une erreur s'est produite durant la transmission. Tous ces détails seront gérés par le pilote TCP.

Il faut remarquer que durant la phase d'établissement de connexion, le pilote IEEE1394 n'est pas utilisé. C'est durant la réception de paquets contenant des données brutes que le

pilote FireoverIP fera appel au pilote IEEE1394. Cet aspect de la communication sera explicité dans la section E.1.1.3.

Après avoir illustré quelque cas de figure durant lesquels le pilote TCP/IP de la passerelle fait appel au pilote FireoverIP, nous allons présenter l'implémentation de la méthode de résolution d'adresses en explicitant les types de données qui ont été implémentés et les différentes méthodes qui sont utilisées par ces pilotes pour valider et retransmettre les paquets.

E.1.1 Implémentation de la méthode de résolution d'adresses

L'analyse du problème de la résolution d'adresses a permis de définir les opérations que le module FireoverIP devra effectuer. Dans cette section, nous allons présenter les détails de conception et d'implémentation pour résoudre ces opérations. Il faut préciser que ces opérations sont effectuées au niveau *Kernel* et que certains pilotes du *Kernel* ont été modifiés afin de tenir compte des requis nécessaires au routage des paquets adressés aux équipements Firewire ou par ces derniers.

Le processus d'initialisation du module s'effectuera selon deux niveaux : *Kernel* et Usager. Nous avons adopté ce mécanisme d'initialisation sur deux niveaux pour faciliter l'allocation des adresses IPv4 des équipements Firewire. Étant donné que les adresses IPv4 dépendront du réseau sur lequel sera déployée la passerelle, l'allocation d'adresses IP se fera à partir du niveau Usager et celles des adresses Firewire au niveau *Kernel*. Si l'allocation des adresses IPv4 s'était effectuée au niveau *Kernel*, les adresses IP à allouer devraient être définies lors de la compilation du *Kernel*. Donc, pour rendre la solution la plus flexible possible, il serait préférable de fournir les adresses IP à partir du niveau usager.

Dans la suite de cette section, les structures qui ont été implémentées seront explicitées ainsi que les mécanismes d'initialisation de ces dernières, ensuite les modifications qui ont été effectuées sur les pilotes du *Kernel* seront explicitées et enfin les mécanismes de communication entre le module FireoverIP et le niveau usager du système seront explicités.

E.1.1.1 Tables de routage

Au départ nous proposions deux tables de routage, une pour acheminer les paquets asynchrones : « rt_1394 » et une autre pour les paquets isochrones : « channels. »

La première table qu'on nommera rt_1394 est de type : « rt_1394_ipv4 » ; comme le montre la Figure E-6, elle hérite de la classe : rt_1394_ipv4_t. En fait, à chaque adresse Firewire correspond une adresse IPv4. La classe : rt_1394_ipv4_t, matérialise cette association.

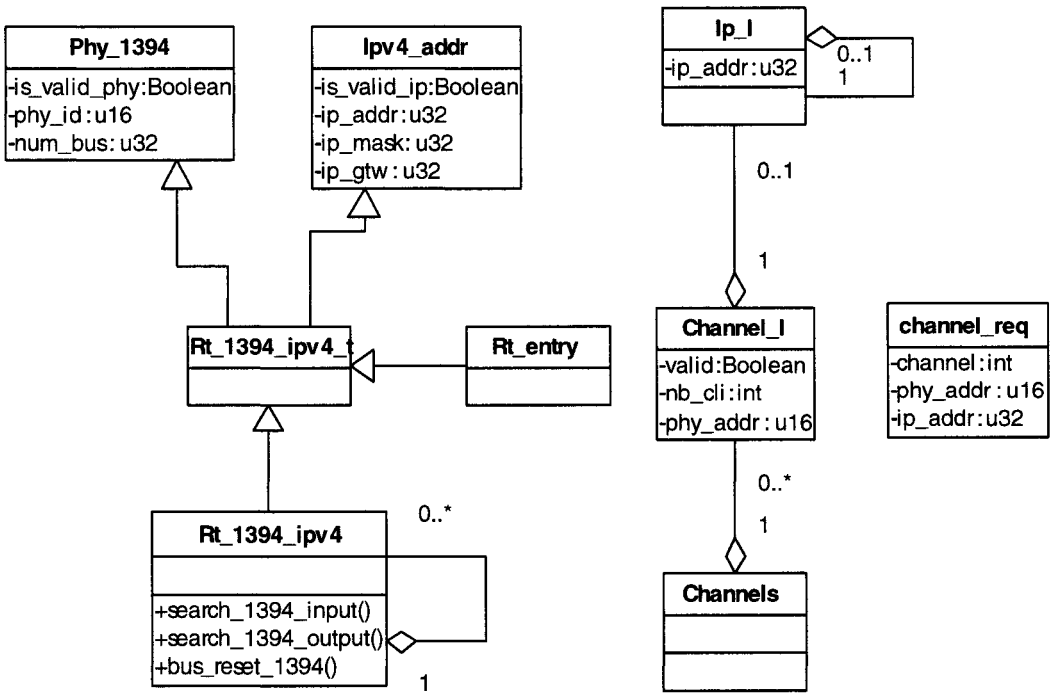

Figure E-6. Diagramme de structures du module de routage FireoverIP

Une adresse Firewire symbolisée par la classe : « phy_addr », est caractérisée par : son identification sur le bus courant (phy_id) et le numéro du bus (num_bus).

Une adresse IPv4 matérialisée par la classe : « ipv4_addr », est caractérisée par trois attributs : une adresse IP à proprement parler (ip_addr), le masque utilisé sur cette adresse pour identifier le sous-réseau auquel appartient ce poste et enfin l'adresse IP de la

passerelle associé à ce sous-réseau. Ces deux derniers attributs sont nécessaires aux fonctions de routage du pilote TCP/IP.

La seconde table est matérialisée par la classe : « Channels ». Cette dernière comporte une liste des adresses physiques Firewire de tous les équipements qui émettent des flots isochrones, ainsi que le nombre de clients qui écoutent les flots venant de chaque équipement associé, ainsi que la liste des adresses IP de ces clients.

Nous avons proposé cette liste des clients afin de contrôler l'arrêt des flots AV par les clients. En effet, ces derniers peuvent indépendamment demander l'arrêt de l'envoi des flots AV d'un équipement Firewire. Cependant, d'autres clients peuvent être en train d'écouter des flots venant de ce même équipement, donc, la passerelle ne devra pas retransmettre cet ordre d'écriture vers cet équipement tant qu'il y aura un client qui écoute ses flots.

La tenue de cette liste permettra à la passerelle de pouvoir effectuer des envois en mode multicast des paquets isochrones vers le réseau Ethernet. Le mode multicast a l'avantage de ne pas saturer les machines du réseau Ethernet qui ne sont pas concernées par ces paquets isochrones.

L'utilité des classes : « rt_entry » et « channel_req », sera explicitée à la section : E.1.1.4

E.1.1.2 Initialisation du module d'adressage

Comme il a été dit précédemment, l'initialisation des tables de routage s'effectue sur deux niveaux : *Kernel* et usager. Dans le premier niveau, les adresses physiques de tous les équipements Firewire sont récoltées et la table rt_1394 est initialisée. En fait, après le processus d'auto-identification, le pilote FireoverIP recevra une liste contenant tous les paquets d'identification transmis par les équipements du réseau Firewire. Il faut préciser que ces paquets contiennent les identifiants de chaque équipement Firewire branché sur le bus Firewire de la passerelle. La passerelle va extraire les identifications de ces équipements et les écrire dans la table : « rt_1394 ». La Figure E-7 illustre le contenu de

cette liste de paquets d'identification et le Tableau E-1 donne l'état de la table rt_1394 après le traitement de ces derniers.

10	0x001	11111111111111111111111111111111111111
10	0x002	11111111111111111111111111111111111111

Figure E-7. Paquets d'auto-identification Firewire

Tableau E-1. État de la table rt_1394 après le traitements des paquets d'identification

Ligne	Is_valid_phy	Phy_id	Num_bus	Is_valid_ip	Ip_addr	Ip_mask	Ip_gtw
0	1	0x0001	0xffc0	0	0.0.0.0	0.0.0.0	0.0.0.0
1	1	0x0002	0xffc0	0	0.0.0.0	0.0.0.0	0.0.0.0

Aucune adresse IPv4 n'est associée à ces adresses physiques Firewire. Les adresses IPv4 seront transmises plus tard, lorsque le service de conversion aura démarré au niveau usager. La seconde table est tout simplement vidée si elle contenait des enregistrements valides.

E.1.1.3 Modifications sur les pilotes TCP/IP et IEEE1394

Il faut principalement retenir que le pilote TCP/IPv4 fait appel aux fonctions « ip_route_input » et « ip_route_output » avant d'accepter les paquets ARP, TCP ou UDP entrant ou sortant (sauf pour ARP). Ces fonctions valident les adresses IPv4 source et destination des paquets reçus ou à transmettre. Donc, ces fonctions ont été étendues afin d'appeler les fonctions : « search_1394_input » et « search_1394_output ». Ces dernières effectueront des recherches dans la table rt_1394 pour vérifier que l'adresse IPv4 source ou destination correspond à celle d'un des équipements Firewire déjà enregistré. Si une correspondance a été trouvée, le pilote TCP/IP devra accepter le paquet entrant et le stocker dans la queue appropriée. S'il s'agit d'un paquet sortant, le processus d'envoi continuera.

La Figure E-8 présente la liste des fichiers qui ont été rajoutés ou modifiés dans le répertoire des fichiers source du *kernel* de linux. Les fichiers « route_1394.h » et

« route_1394.cpp » comportent la déclaration et la description des fonctions du pilote FireoverIP. Le fichier « route.c » comporte toutes les tables et fonctions de routage implémentées par le pilote TCP/IP. Les modifications sur ce fichier consistaient à appeler les fonctions de recherche : « rt_1394_search_input » et « rt_1394_search_output » du pilote FireoverIP dans les fonctions « ip_route_input » et « ip_route_output ». Le fichier « af_inet.c » implémente les fonctions qui effectuent le pont entre le niveau usager du système d'exploitation et le pilote TCP/IP. Donc, les modifications qui ont été effectués sur ce fichier consistaient à appeler la fonction : « rt_1394_ioctl » du pilote FireoverIP.

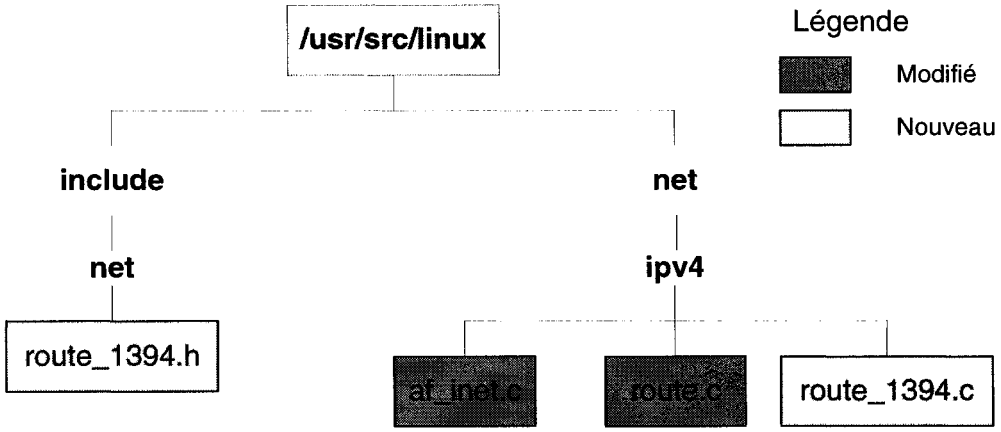

Figure E-8. Fichiers modifiés ou rajoutés dans le répertorie de base des sources du *kernel* de Linux

E.1.1.4 Interactions entre le pilote FireoverIP et le niveau usager

Comme nous l'avons dit précédemment, l'initialisation des adresses IPv4 s'effectue au niveau usager, aussi, nous avons rajouté sur la couche « af_inet », des commandes qui permettent d'appeler des fonctions du module FireoverIP. Il faut rappeler que Linux est conçu d'une telle manière que les usagers peuvent faire appel à certaines fonctions des modules du *Kernel* en passant par des interruptions « ioctl », par exemple. Dans notre cas, les interruptions concernent la couche « af_inet » qui regroupe tous les pilotes associés à IPv4 et FireoverIP entre autres.

Lorsque la couche usager veut mettre à jour l'adresse IP d'équipements Firewire, il lui suffit de remplir un objet de cette classe : « rt_entry », avec les bonnes valeurs et de l'envoyer au pilote via une interruption.

Pour rajouter ou supprimer un élément de la table : « channels », l'usager enverra un objet de type : channel_req, en y précisant l'adresse physique Firewire de l'équipement concerné, l'adresse IP du client concerné par cette transaction et le numéro de canal sur lequel cet équipement est supposé écouter des flots.

Annexe F

Fonctionnalités du *thread* serveur_1394

F.1.1

F.1.2 Fonctionnalités d'un server_1394

Chaque équipement Firewire est émulé par un *thread* serveur_1394 qui jou le rôle du pont entre le l'équipement Firewire associé et les différents clients Ethernet qui désirent établir une connexion avec l'équipement Firewire. Le comportement de ce *thread* est illustré par la Figure F-1.

Durant l'initialisation du *socket*, le serveur précise l'adresse IPv4 associée à l'équipement Firewire. Cette opération est très importante, car durant l'envoi de paquets TCP/IP, c'est cette adresse qui sera contenue dans le champ : adresse source IPv4. Dans le cas contraire, c'est l'adresse courante de la passerelle qui sera indiquée dans le champ.

Dès que le canal est initialisé, le serveur peut commencer à attendre une demande connexion sur le port TCP qui lui est associé. Le nombre de connexion a été limité à 3, ainsi un serveur pourra communiquer avec 3 clients en en même temps.

Lorsqu'une demande de connexion est détectée, cette dernière est acceptée, ensuite un *thread* : conn_1394 est créé. Ce dernier effectue les échanges entre le client et l'équipement 1394 associé. Ensuite, le serveur décrémentera son compteur de *thread* « conn_1394 ». Si 3 clients communiquent avec le serveur, ce dernier doit attendre qu'une des connexions se terminent avant d'accepter une nouvelle demande de connexion.

Pendant tout ce temps, le serveur vérifie que le service de conversion est actif, si ce dernier ne l'est plus, les *threads* de type « conn_1394 » qui avaient été créés sont interrompus et ensuite le serveur est fermé.

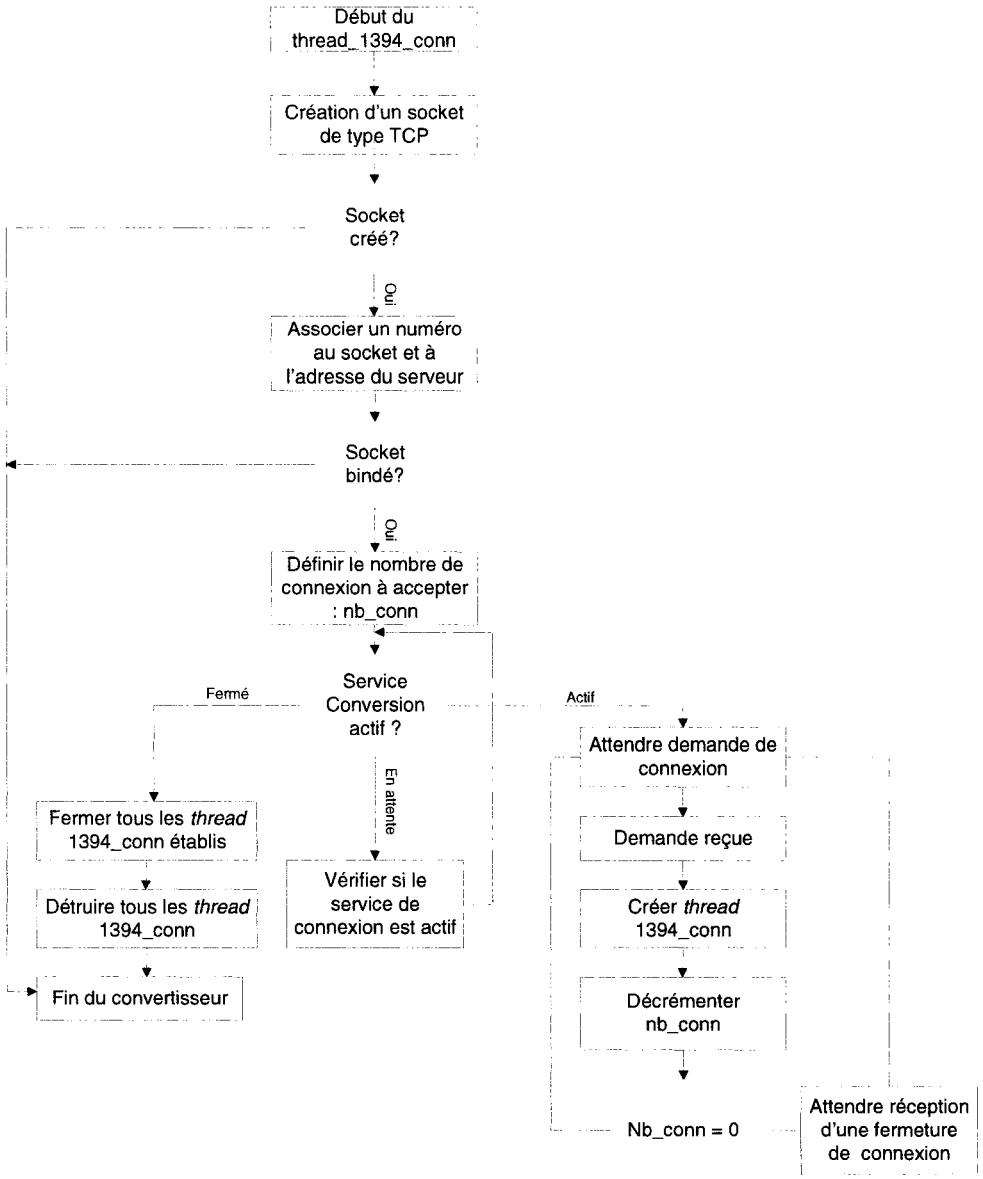

Figure F-1. Organigramme du *thread* serveur_1394

Le *thread* « conn_1394 » effectue les traitements illustrés par l'organigramme de la Figure F-2. Comme vous pouvez le voir, ce *thread* attend des paquets venant du client associé. Lorsqu'un paquet est reçu, le *thread* effectue des contrôles sur les champs de la couche virtuelle. Le premier contrôle concerne le champ type du paquet. Le *thread* traite juste les champs de type : requête de lecture, requête d'écriture et fermeture de la connexion. Cette restriction est due au fait que les équipements Firewire sont pour

l'instant passifs. En effet ils n'initient aucune transaction d'écriture ou de lecture, ils sont juste tenus de répondre à des requêtes de ce type. Le *thread* : conn_1394 est interrompu quand un paquet de type : « close », est reçu. Dans ce cas, l'envoi de flots AV est aussitôt interrompu.

Le serveur doit aussi vérifier la valeur du champ « data_len». Pour les requêtes de lecture, le champ charge utile n'existe pas, donc le champ « data_len» aura la valeur 0. Une réponse de type « échec » sera renvoyée au client si la taille ne respecte pas les limites données, à savoir : égal à 0 pour les lectures et supérieure ou égale à 4 pour les écritures.

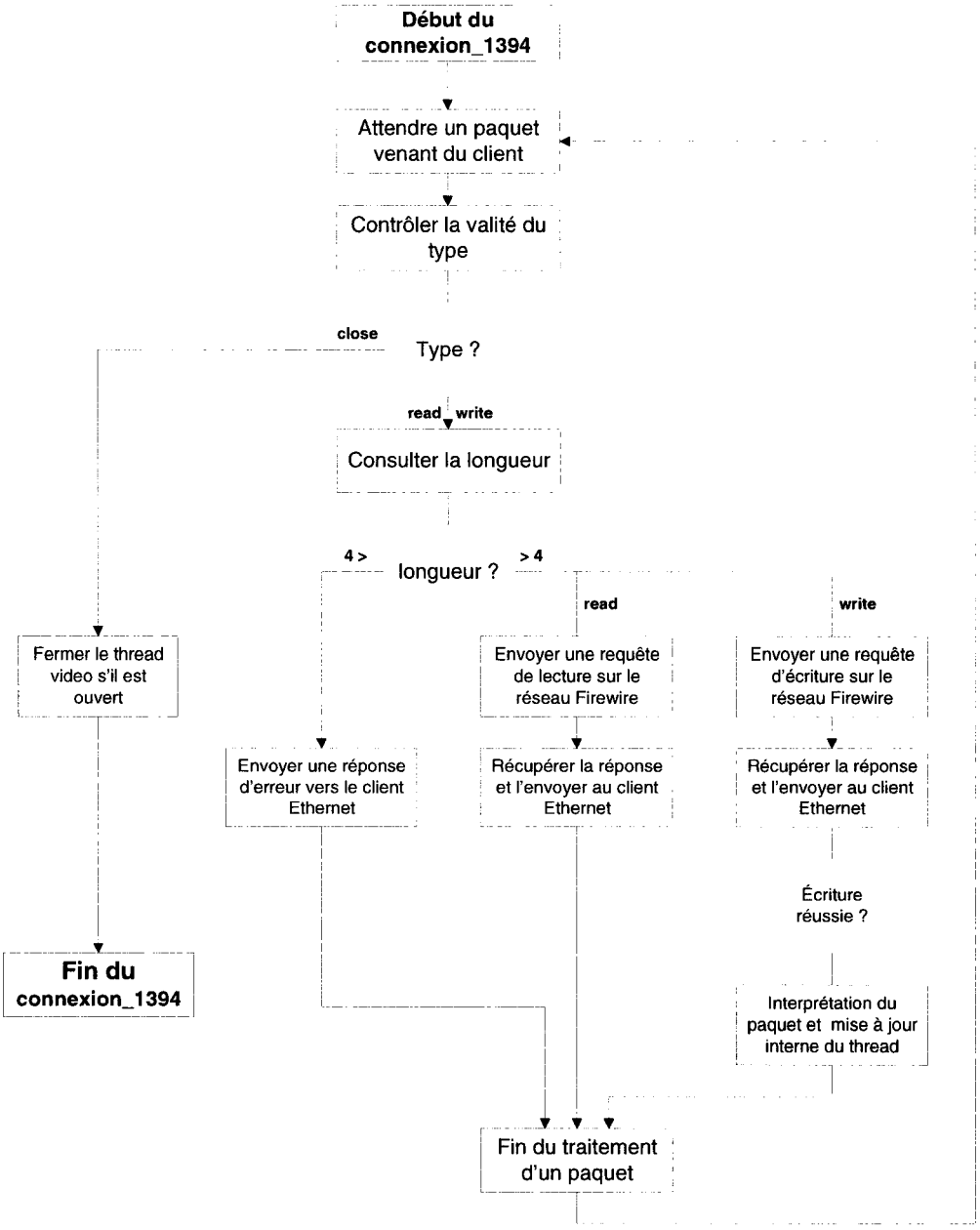

Figure F-2. Organigramme du *thread* conn_1394

La Figure F-3 a donne un exemple du contenu de l'entête virtuelle d'une requête de lecture sur le registre : REG_CAMERA_ISO_EN. Ce dernier indique l'état de la caméra à savoir si cette dernière émet déjà des flots AV. Comme vous pouvez le voir, l'entête comporte juste 14 octets. Si le type et la taille sont conformes le *thread* « conn_1394 » enverra une requête de lecture via la carte Firewire de la passerelle.

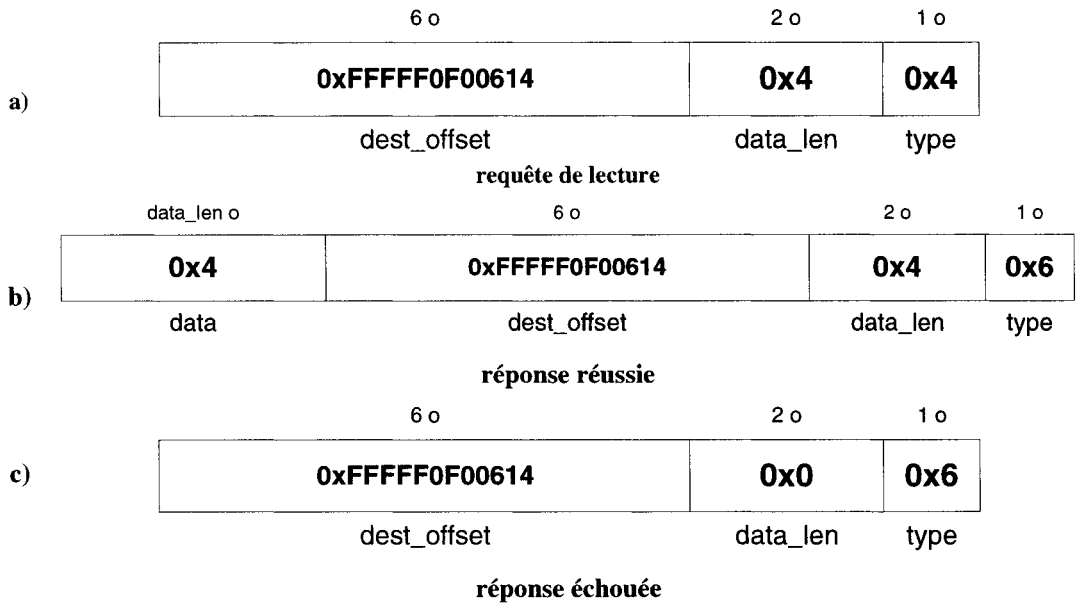

Figure F-3. Contenus de l'entête FireoverIP pour une transaction de lecture

Selon la réponse de l'équipement, le *thread* constituera un paquet de type réponse réussie (Figure 1.19-b) ou échouée (Figure 1.19-c). En effet, il se pourrait que l'équipement Firewire ne soit pas capable de répondre à une requête de lecture ou d'écriture.

Une des erreurs fréquentes est causée par l'invalidité de l'adresse du registre : « dest_offset» sur lequel la transaction doit être effectuée. La passerelle n'effectue pas de contrôle sur cette adresse. Étant donné la multitude d'équipements AV, nous avons préféré laisser ce contrôle aux équipements Firewire destination, puisqu'ils sont les plus à même de savoir sur quels registres des transactions de lecture ou d'écriture peuvent être effectuées. Dans le cas contraire, il aurait fallu que la passerelle sache les restrictions liées à chaque type d'équipement AV, ce qui alourdirait la passerelle en limitant le nombre équipements différents avec lequel elle pourrait communiquer.

Annexe G
Fonctionnalités du client Ethernet

La Figure G-1 explicite le processus d'initialisation d'une caméra par le client.

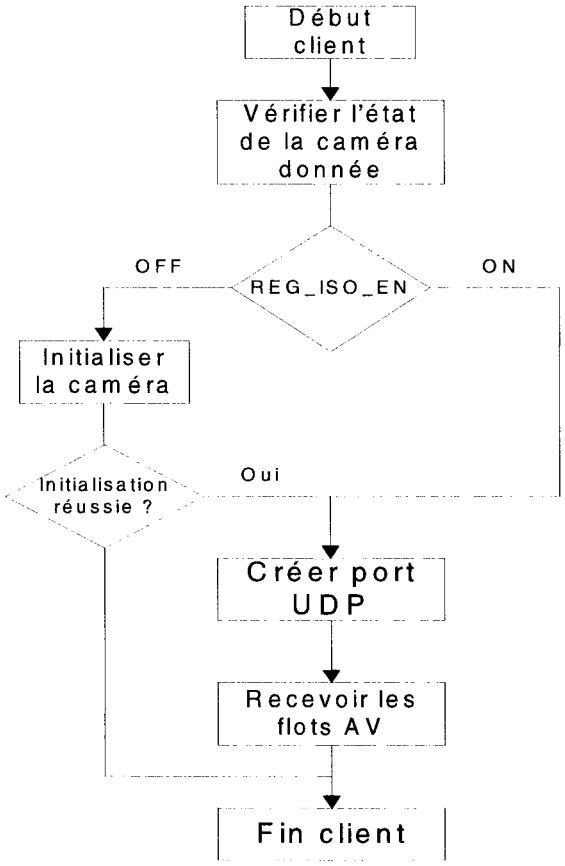

Figure G-1. Processus en vue de recevoir des flots AV

La première étape du processus commence par vérifier l'état de la caméra. Cette étape consiste à effectuer une requête de lecture sur le registre REG_ISO_EN.

Si la caméra émet déjà des flots, le client n'a pas besoin continuer le processus d'initialisation, il pourra commencer à recevoir des flots sur le canal UDP sur le lequel le thread « forward_iso » de la passerelle (associé à la caméra) émet. Il devra pour cela effectuer une autre requête de lecture pour connaître le numéro de canal : num_channel_lu, sur lequel la caméra émet ses flots. En fait, le numéro de port UDP sur

lequel les flots sont transmis correspond à : UDP_SERVEUR_PORT + num_channel_lu. UDP_SERVEUR_PORT est un numéro de base égal à 1600. Ensuite, le client peut créer une socket pour écouter tous les paquets entrants dont le numéro de port correspond à celui défini précédemment.

Si la caméra n'émet pas encore des flots, le client doit effectuer les opérations illustrées par la Figure G-2. La première opération consiste à effectuer une lecture afin de lire le contenu du registre chan_avalaible. À partir de la valeur lue, le client choisira un numéro de canal disponible et il mettra à jour le registre chan_avalaible afin de signifier que ce dernier est réservé. Cette dernière opération n'est pas effectuée sur la caméra mais sur les registres du gestionnaire de ressources isochrones.

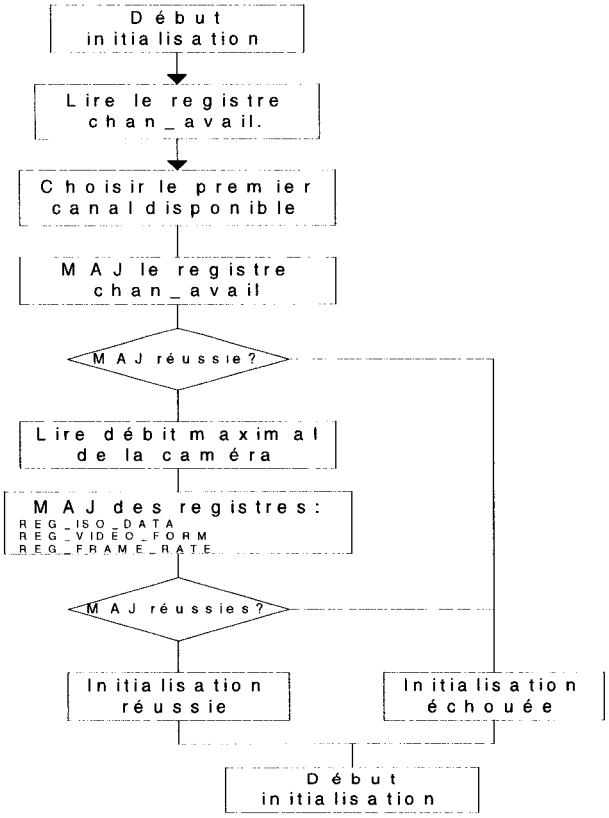

Figure G-2. Étapes d'initialisation d'une caméra

Le client devra ensuite lire le champ REG_ISO_SPEED pour avoir la vitesse à laquelle la caméra peut envoyer ses flots AV. Si ces deux opérations se sont bien déroulées, le client

pourra initialiser le registre REG_ISO_DATA de la caméra afin de signifier à cette dernière le canal sur lequel elle doit émettre ses flots AV (1394 Trade Assocation, 1998).

Comme vous pouvez le remarquer la caméra n'effectue presque aucune opération d'initialisation par elle-même. Après avoir initialisé le champ REG_ISO_DATA, le client a quatre autres opérations à effectuer pour compléter la phase d'initialisation :

- Envoyer une requête d'écriture pour mettre à jour le format vidéo des images à recevoir;
- Envoyer une requête d'écriture pour mettre à jour le taux d'images à transmettre par seconde (frame rate);
- Définir par rapport au format vidéo défini précédemment, la taille en mots de 32 bits d'un paquet contenant des données AV envoyées par la caméra;
- Et enfin, définir le nombre de mots de 32 bits qui constituent une image.

Ces deux derniers paramètres permettent de remplir le tampon de réception du client. Après toutes ces opérations, le client est maintenant prêt à créer une socket UDP et à attendre les messages AV.

Durant le processus de réception des données AV, le client rempli son tampon au fur et à mesure et dès qu'il recevra une image complète, il pourra l'afficher. Chaque paquet envoyé par la caméra comporte une entête de 32 bits. Et les 8 premiers bits de cet entête indiquent si le paquet courant est le premier paquet d'une image. Le client vérifie à chaque paquet reçu l'entête, si c'est le premier paquet d'une image, le tampon est vidé. Pour calculer le nombre d'images perdus, nous utiliserons ce champ de l'entête.

Annexe H

Graphes pour 15 images/s et 30 images/s

Dans cet annexe, nous présentons la variation du nombre de paquets et celle du nombre de blocs envoyés au total durant 20 s de simulation. La Figure H-1 et la Figure H-4 représentent les variations associées aux taux théoriques d'envoi de la caméra de 7.5 im/s et 30 im/s.

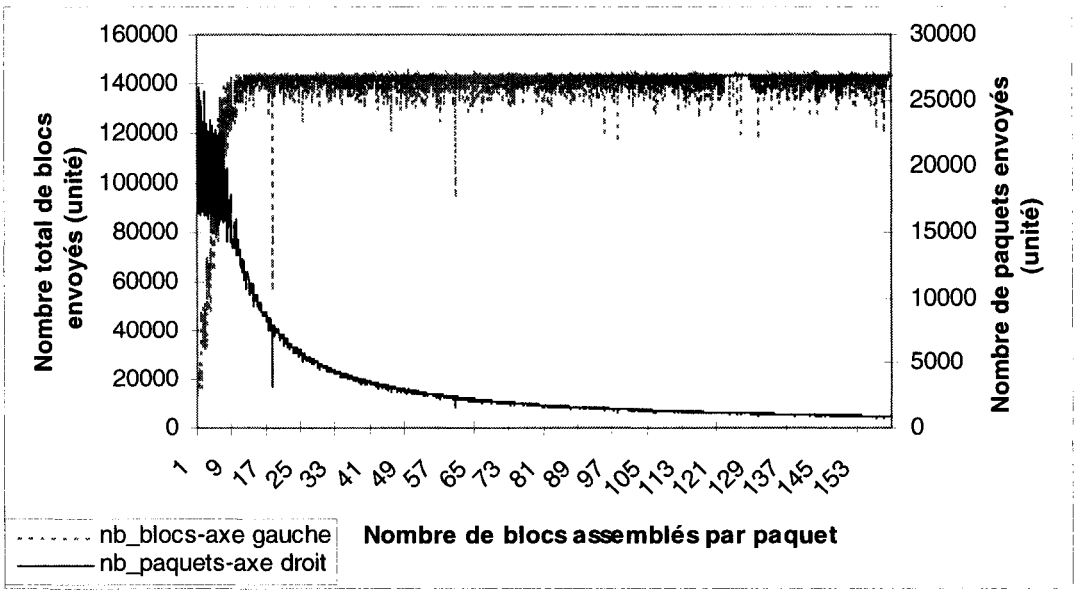

Figure H-1. Variation du nombre de paquets envoyés pour un taux de 7.5 images/s

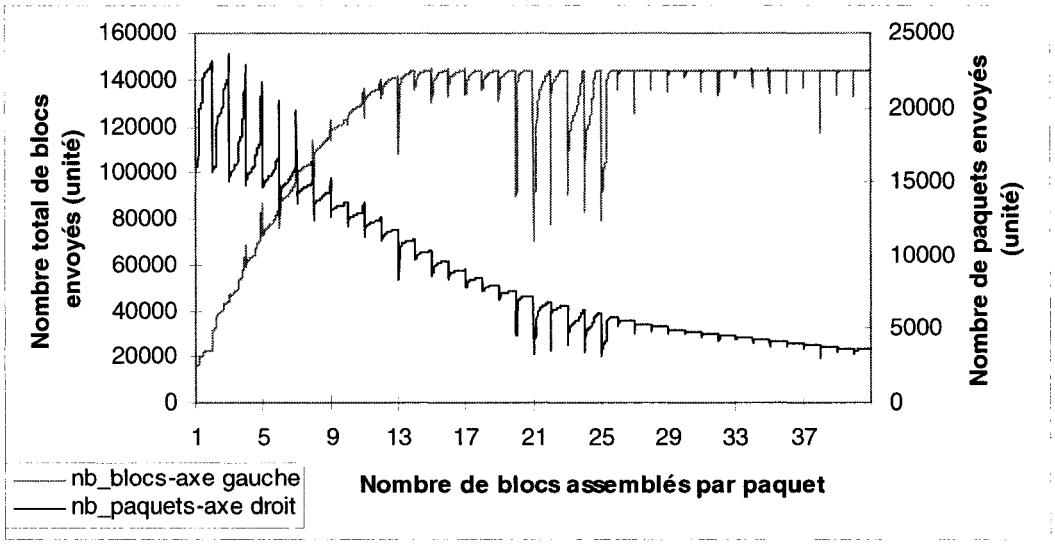

Figure H-2. Variation du nombre de paquets envoyés pour un taux de 30 images/s

La Figure H-3 et la Figure H-4 représentent la variation du débit d'envoi de la passerelle pour des taux théoriques de réception de 7.5 im/s et 30 im/s.

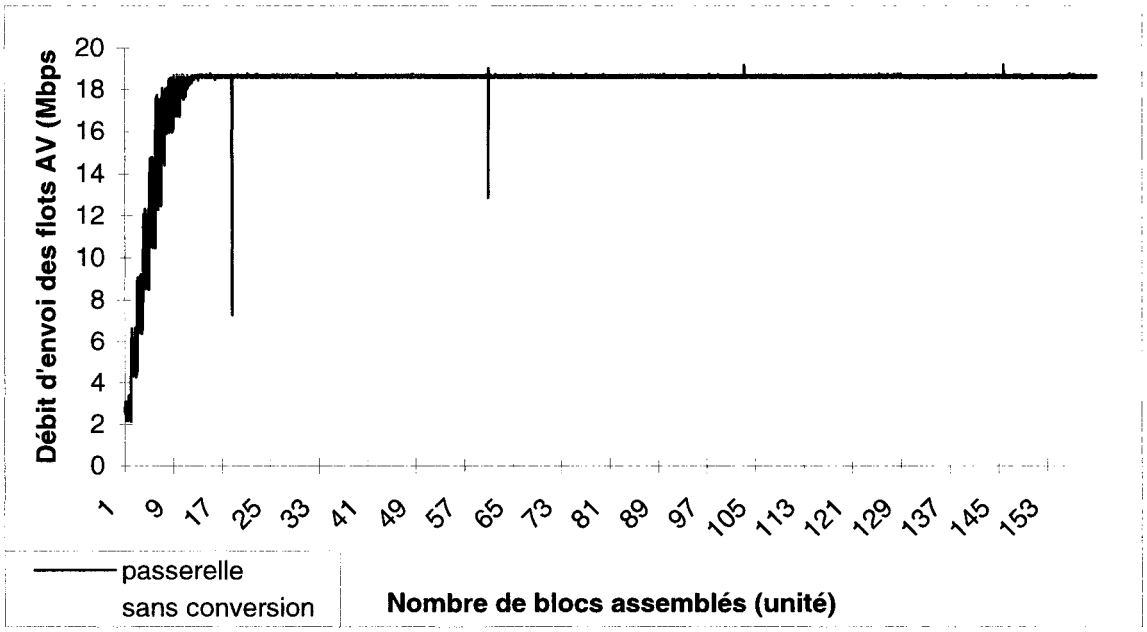

Figure H-3. Variation de la vitesse moyenne d'envoi des flots AV pour 7.5 im/s

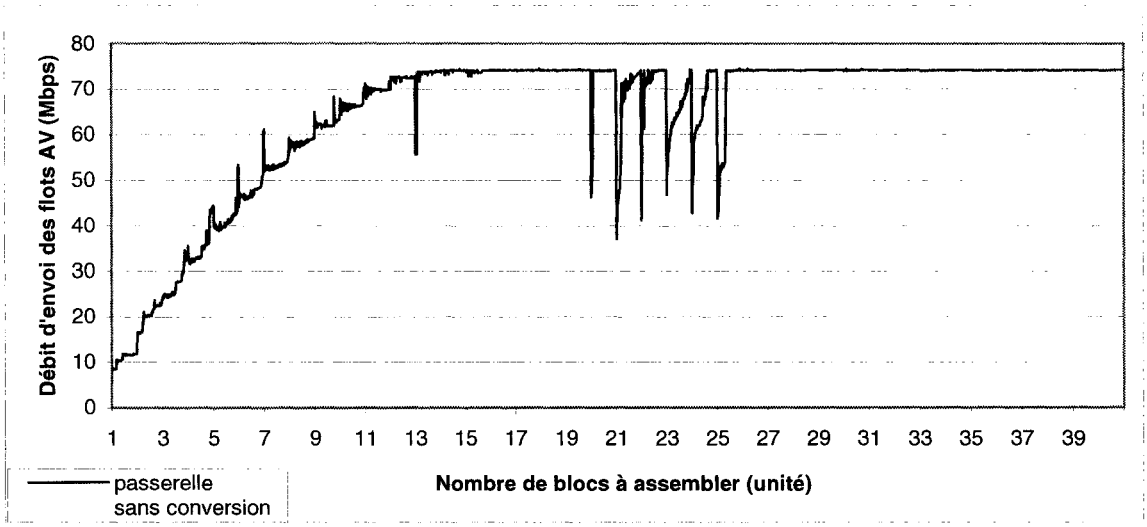

Figure H-4. Variation de la vitesse moyenne d'envoi des flots AV pour 30 im/s

Annexe I

Modèle et résultats de simulations : 7.5 et 30 im/s

Cet annexe récapitule l'évolution du nombre de blocs dépilés selon le modèle et les simulations réelles pour les taux de 7.5 et 30 images à la seconde.

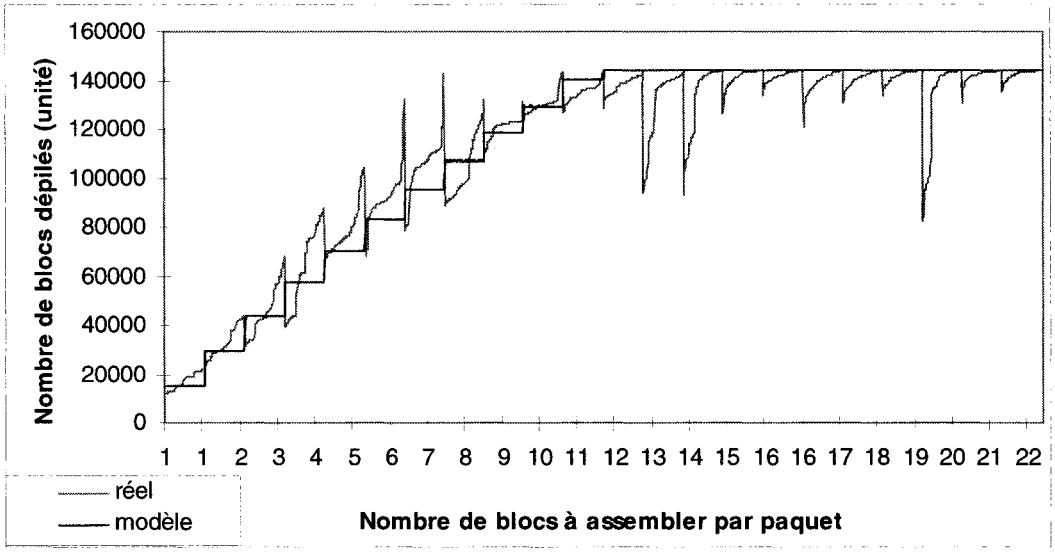

Figure I-1. Variation du nombre des blocs dépilés pour un taux de 7.5 im/s

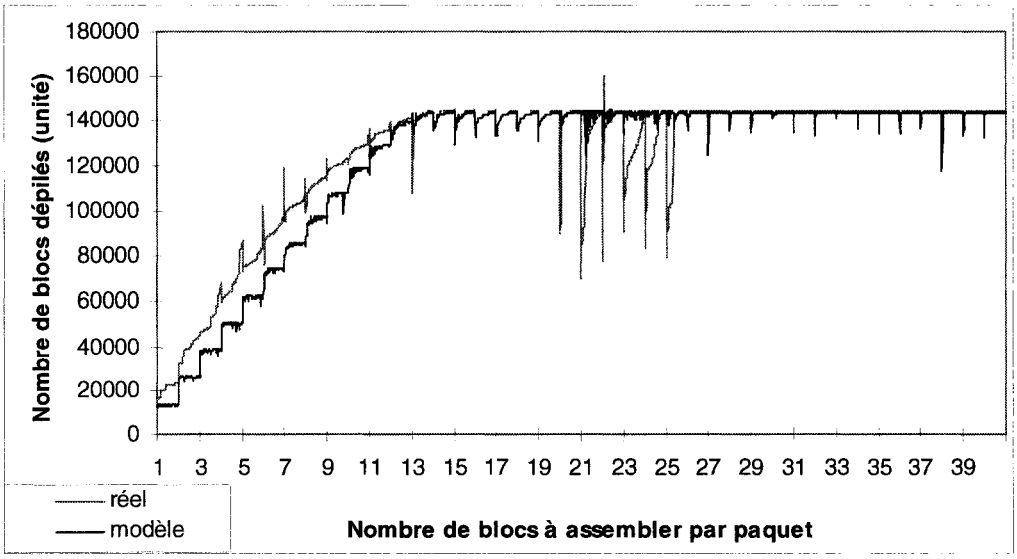

Figure I-2. Variation du nombre des blocs dépilés pour un taux de 30 im/s