UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE FÍSICA DE SÃO CARLOS
DEPARTAMENTO DE FÍSICA E INFORMÁTICA

"COMPARAÇÃO DE DESEMPENHO ENTRE TCP/IP SOBRE ATM E ATM NATIVO"

MARCELO SILVA FREITAS

Dissertação apresentada ao Instituto de Física de São Carlos, Universidade de São Paulo, para obtenção do título de Mestre em Ciências "Física Aplicada – Opção Física Computacional"

USP/IFSC/SBI
8-2-001395

Orientador:
Prof. Dr. Carlos Antonio Ruggiero

Comissão Julgadora:
Prof. Dr. Carlos Antonio Ruggiero (IFSC-USP)
Prof. Dr. Marcos Antonio Cavenaghi (IFSC-USP)
Prof. Dr. Luis Carlos Trevelin (UFSCar)

04

São Carlos – São Paulo
2001

IFSC-USP SERVIÇO DE BIBLIOTECA E INFORMAÇÃO

Freitas, Marcelo Silva

"Comparação de desempenho entre TCP/IP sobre ATM e ATM nativo"/
Marcelo Silva Freitas – São Carlos, 2001

Dissertação (Mestrado) – Instituto de Física de São Carlos da Universidade de São Paulo, 2001 páginas: 149
Área: Física Aplicada-Opção Física Computacional
Prof. Dr. Carlos Antonio Ruggiero

1. TCP/IP, 2. ATM, 3. Desempenho

I. Título

UNIVERSIDADE
DE SÃO PAULO
Instituto de Física de São Carlos

Caixa Postal 369
13560-970 São Carlos, SP
Av. Trabalhador São-carlense, 400,
13566-590 São Carlos, SP

Fone/Fax 16 273 9777
www.if.sc.usp.br
wladerez@if.sc.usp.br

MEMBROS DA COMISSÃO JULGADORA DA DISSERTAÇÃO DE MESTRADO DE MARCELO SILVA FREITAS APRESENTADA AO INSTITUTO DE FÍSICA DE SÃO CARLOS, DA UNIVERSIDADE DE SÃO PAULO, EM 03 DE MAIO DE 2001

COMISSÃO JULGADORA:

Prof. Dr. Carlos Antonio Ruggiero/IFSC-FFI-USP

Prof. Dr. Marcos Antonio Cavenaghi/UNESP

Prof. Dr. Luis Carlos Trevelin/UFSCar

" DE TUDO FICARAM TRÊS COISAS

a certeza de que estamos sempre começando ...

a certeza de que é preciso continuar ...

a certeza de que seremos interrompidos antes de terminar

...

PORTANTO DEVEMOS

fazer, da interrupção, um caminho novo ...

da queda, um passo de dança ...

do medo, uma escada ...

do sonho, uma ponte ...

da procura ... um encontro "

Fernando Sabino

Dedico este trabalho a minha esposa, Delma, que no meio de tantas tribulações, sempre foi meu porto seguro, meu referencial, pelo seu apoio e amor incondicionais.

Agradecimentos

A Deus, por proporcionar todas as condições para que este trabalho fosse concluído com êxito.

A minha família, em especial, minha mãe, pelo amor e incentivo incomensuráveis.

Aos meus amigos, pelas conversas valiosas e compartilhamento de experiências.

Em especial, a Silvana, Zeca e Mariane, por me hospedarem e me tratarem como se fosse membro da família.

A Márcio, Marcos, Cacau, Roberta e Daniel, também pela hospedagem, amizade e apoio.

Ao meu orientador, Prof. Dr. Carlos Antônio Ruggiero, por me aceitar como orientado, acreditando em minha capacidade para o trabalho proposto, por todo apoio e compreensão.

A CAPES pelo apoio financeiro.

Sumário

LISTA DE FIGURAS I

LISTA DE TABELAS II

ACRÔNIMOS III

RESUMO VI

ABSTRACT VII

CAPÍTULO 1 1

INTRODUÇÃO 1

1.1 MOTIVAÇÃO 2

1.2 OBJETIVOS 5

1.3 APRESENTAÇÃO DOS CAPÍTULOS SEGUINTES 6

CAPÍTULO 2 8

HARDWARE, TOPOLOGIA E PROTOCOLOS 8

2.1 ETHERNET *VERSUS* ATM 9

2.2 PILHAS DE PROTOCOLOS 13

2.3 ENCAPSULAMENTO, SEGMENTAÇÃO E REMONTAGEM 15

2.4 QUALIDADE DE SERVIÇO 18

2.5 IP EM REDES ATM 19

2.5.1 *CLASSICAL IP* 19

2.5.2 NSAP E ILMI 21

2.5.3 ARP E SERVIDORES ARP 23

2.5.4 SINALIZAÇÃO ATM 25

2.6 ESPECIFICAÇÃO DO EQUIPAMENTO UTILIZADO 27

CAPÍTULO 3 31

ATM: IMPLEMENTAÇÕES, EXPERIÊNCIAS E DESEMPENHO 31

3.1 DETALHES DE IMPLEMENTAÇÃO DOS *SOCKETS* BSD 32

3.2 A PLATAFORMA HARP 38

3.2.1 O QUE A PLATAFORMA HARP SUPORTA 39

3.2.2 O QUE A PLATAFORMA HARP NÃO SUPORTA 40

3.2.3 VISÃO FUNCIONAL DA PLATAFORMA HARP 41

3.3 OUTRAS IMPLEMENTAÇÕES PARA ATM EM FREEBSD 44

3.3.1 NATM - INTEGRANDO ATM A SISTEMAS BSD 45

3.3.2 UMA CAMADA DE TRANSPORTE PARA ATM NATIVO 46

3.4 PREVISÃO DE DESEMPENHO 48

3.5 TESTES DE DESEMPENHO EM CHAVES ATM FORE 53

CAPÍTULO 4 59

SUBSTITUIÇÃO DAS APIS E TESTES INICIAIS 59

4.1 NETPERF 60

4.1.1 FUNCIONALIDADE 60

4.1.2 IMPLEMENTAÇÃO E FUNCIONAMENTO 61

4.1.3 SUBSTITUIÇÃO DAS APIS 67

4.2 TESTES EM *LOOPBACK* 68

4.3 TEMPOS DE RESPOSTA 70

CAPÍTULO 5 75

MEDIDAS DE TAXAS DE TRANSMISSÃO 75

5.1 TCP/IP SOBRE ATM 76

5.2 UDP/IP SOBRE ATM 86

5.3 ATM NATIVO 90

CAPÍTULO 6 99

CONCLUSÕES, CONTRIBUIÇÕES E TRABALHOS FUTUROS 99

ANEXO A – CÓDIGO DE NETTEST_ATMN.C 102

ANEXO B – CÓDIGO DE NETTEST_ATMN.H 122

REFERÊNCIAS BIBLIOGRÁFICAS 124

Lista de Figuras

FIGURA 2.1 TOPOLOGIAS ETHERNET E ATM 10
FIGURA 2.2 *CHAVEAMENTO* DE CAMINHOS VIRTUAIS (VP) E CANAIS VIRTUAIS (VC) 12
FIGURA 2.3 ENCAPSULAMENTO TCP, IP E ETHERNET 16
FIGURA 2.4 PREPARAÇÃO DAS CÉLULAS PARA TRANSMISSÃO 17
FIGURA 2.5 SINALIZAÇÃO ATM. 26
FIGURA 2.6 ESTRUTURA DE REDE DO LABORATÓRIO. 28
FIGURA 3.1 SUB-SISTEMA DE REDE DO *KERNEL* BSD. 33
FIGURA 3.2 TIPOS DE *MBUF* DO SISTEMA BSD. 35
FIGURA 3.3 ESTRUTURA DOS *BUFFERS* DE *SOCKET* NOS SISTEMAS BSD. 37
FIGURA 3.4 VISÃO FUNCIONAL DA PLATAFORMA HARP 41
FIGURA 3.5 PILHA DE PROTOCOLOS TCP/ATM E ATM NATIVO 48
FIGURA 3.6 RELAÇÃO ENTRE AS CAMADAS ATM E AAL5 50
FIGURA 3.7 ENCAPSULAMENTO LLC/SNAP PARA PACOTES IP SOBRE AAL5 51
FIGURA 4.1 MODELO CLIENTE-SERVIDOR DE NETPERF 61
FIGURA 4.2 ESTRUTURAS DE DADOS QUE CARREGAM INFORMAÇÕES PELA CONEXÃO DE CONTROLE. . 63
FIGURA 4.3 LISTA DOS ARQUIVOS DO PACOTE NETPERF-2.1PL3.TGZ. 64
FIGURA 4.4 FUNCIONAMENTO DA TRANSMISSÃO DE DADOS DO TESTE TCP_STREAM DE NEPERF..... 65
FIGURA 4.5 MENSAGEM ICMP 70
FIGURA 5.1 *THRHOUGHPUT* EM FUNÇÃO DO TAMANHO DA MENSAGEM PARA TESTES TCP/ATM..... 79
FIGURA 5.2 ELIMINAÇÃO DOS PONTOS DE QUEDA DE *THROUGHPUT* NOS TESTES TCP/ATM..... 82
FIGURA 5.3 *THROUGHPUT* EM FUNÇÃO DO TAMANHO DOS *SOCKETS* NOS TESTES TCP/ATM..... 83
FIGURA 5.4 *THROUGHPUT* EM FUNÇÃO DO TAMANHO DOS *SOCKETS* NOS TESTES TCP/ATM..... 84
FIGURA 5.5 *THROUGHPUT* EM FUNÇÃO DO TAMANHO DOS *SOCKETS* NOS TESTES TCP/ATM..... 85
FIGURA 5.6 *THROUGHPUT* EM FUNÇÃO DO TAMANHO DA MENSAGEM NOS TESTES UDP/ATM. 87
FIGURA 5.7 ELIMINAÇÃO DOS PONTOS DE QUEDA NOS TESTES UDP/ATM. 87
FIGURA 5.8 *THROUGHPUT* EM FUNÇÃO DO TAMANHO DA MENSAGEM NOS TESTES UDP/ATM. 89
FIGURA 5.9 *THROUGHPUT* EM FUNÇÃO DO TAMANHO DA MENSAGEM NOS TESTES ATM. 91
FIGURA 5.10 *THROUGHPUT* EM FUNÇÃO DO TAMANHO DA MENSAGEM NOS TESTES ATM. 92
FIGURA 5.16 MENSAGENS DE ERRO NO CONSOLE DA MÁQUINA RECEPTORA. 94
FIGURA 5.12 ESTATÍSTICAS DE UMA TRANSMISSÃO MOSTRADAS PARA VÁRIAS CAMADAS. 95

Lista de Tabelas

TABELA 2.1 CLASSIFICAÇÃO DE SERVIÇOS PARA A CAMADA AAL 12

TABELA 2.2 CONFIGURAÇÃO DA CHAVE LIE-ATM. 28

TABELA 2.3 CONFIGURAÇÃO DA CHAVE IFSC-USP. 29

TABELA 2.4 CONFIGURAÇÃO DAS MÁQUINAS UTILIZADAS NOS EXPERIMENTOS. 30

TABELA 3.1 LIMITES DOS *BUFFERS* DE *SOCKET* PARA PROTOCOLOS DO DOMÍNIO AF_NET. 38

TABELA 3.2 TAXAS DE TRANSMISSÃO MÁXIMAS NO AMBIENTE TCP/ATM 55

TABELA 3.3 TAXAS DE TRANSMISSÃO MÁXIMAS NO AMBIENTE ATM NATIVO 56

TABELA 4.1 DIFERENÇAS ENTRE API FORE E API HARP. 67

TABELA 4.2 RESULTADOS DOS TESTES EM LOOPBACK 69

TABELA 4.3 TEMPOS DE RESPOSTA PARA REQUISIÇÕES ICMP NA REDE ATM. 72

TABELA 4.4 TEMPOS (EM MS) PARA A PRIMEIRA MENSAGEM ICMP DO COMANDO PING. 73

TABELA 5.1 *THROUGHPUT* EM FUNÇÃO DO TAMANHO DOS *SOCKETS* NOS TESTES ATM. 97

TABELA 5.2 *THROUGHPUT* EM FUNÇÃO DO TAMANHO DOS *SOCKETS* NOS TESTES ATM. 97

TABELA 5.3 *THROUGHPUT* EM FUNÇÃO DO TAMANHO DOS *SOCKETS* NOS TESTES ATM. 98

TABELA 5.4 *THROUGHPUT* EM FUNÇÃO DO TAMANHO DOS *SOCKETS* NOS TESTES ATM. 98

TABELA 6.1 RESUMO DOS RESULTADOS DAS TAXAS DE TRANSMISSÃO 99

Acrônimos

AAL – ATM Adaptation Layer

ABR – Avalable Bit Rate

ACK – Abreviação para o termo em inglês *acknowledgement* (reconhecimento)

API – Application Program Interface

ARP – Address Resolution Protocol

ATM – Asynchronous Transfer Mode

B-ISDN – Broadband Integrated Services Digital Network

BSD – Berkeley Software Distribution

CBR – Constant Bit Rate

CCITT – Comite Consultatif Internationale de Telegraphique et Telephonique

CPCS – Common Part Convergence Sublayer

CPU – Central Processing Unit

DCC – Data Country Code

DLPI – Data Link Provider Interface (API de nível 2 OSI)

ESI – End System Identifier

FDDI – Fiber Distributed Data Interface

Gbps – 10^9 (Giga) bits por segundo

HARP – Host ATM Research Plataform

ICD – Internetional Code Designator

ICMP – Internet Control Message Protocol

IEEE – Institute of Electrical and Electronics Engineers

IETF – Internet Engineering Task Force

IGMP – Internet Group Management Protocol

ILMI – Interin Local Management Interface

IP – Internet Protocol

ISDN – Integrated Services Digital Network

ISO/OSI – Internetional Organization for Standardization / Open Systems Interconetion

ITU – Internetional Telecomunications Union

LAN – Local Area Network

LANE – Local Area Network Emulation

LAPB – Link Access Procedure Balanced

LCC/SNAP – Logical Link Control / Sub Network Access Point

LIS – Logical IP Subnetwork

MAC – Media Access Control

MB – 10^6 (Mega) bytes

Mbps – 10^6 (Mega) bits por segundo

MPOA – Multi-Protocol over ATM

MSS – Maximum Segment Size

MTU – Maximum Transmission Unit

NHRP – Next-Hop Resolution Protocol

NNI – Network-to-Network Interface

PDU – Protocol Data Unit

PVC – Permanet Virtual Circuit

QoS – Quality of Service

RAM – Random Access Memory

RFC – Request for Coments

RISC – Reduced Instruction Set Computer

RSVP – Resource Reservation Protocol

RTP – Real Time Protocol

RTT – Round Trip Time

SAAL – Signaling ATM Adaptation Layer

SAR – Segmentation and Reassembly Sublayer

SCP – Switch Central Processor

SCSP – Server Cache Synchronization Protocol

SDH – Synchronous Digital Hierarchy

SDU – Service Data Unit

SNMP – Simple Network Management Protocol

SONET – Synchronous Optical Network

SPANS – Simple Protocol for ATM Network Signalling

SSCF – Service-Specific Coodination Function

SSCOP – Service-Specific Connecion Oriented Protocol

STS-n – Synchronous Transport Signal

SVC – Switched Virtual Circuit

SWS – Silly Window Syndrome

TCP – Transmission Control Protocol

TTCP – Test TCP(UDP) Performance

UBR – Unspecified Bit Rate

UDP – User Datagram Protocol

UNI – User Network Interface

UTP-5 – Unshielded Twisted Pair (categoria 5)

VBR – Variable Bit Rate

VC – Virtual Channel, Virtual Circuit, Virtual Connection

VCC – Virtual Channel Connection

VCI – Virtual Channel Identifier

VPC – Virtual Path Connection

VPI - Virtual Path Identifier

Resumo

Com o recente desenvolvimento de tecnologias de redes de altas taxas de transmissão, tais como Asynchronous Transfer Mode (ATM), o problema da carência por largura de banda foi solucionado. A questão atual é a implementação de sistemas que suportem os protocolos ATM de forma nativa e integral. Atualmente tem-se utilizado aplicativos tradicionais baseados nos protocolos TCP(UDP)/IP no topo da pilha de protocolos ATM. Tal modelo traz redundâncias que implicam diretamente em aumento de *overhead* na comunicação. Muitos modelos têm sido desenvolvidos para levar de forma direta à aplicação os serviços ATM.

O propósito deste trabalho foi comparar o desempenho de transmissões de dados utilizando os protocolos TCP(UDP)/IP sobre ATM com transmissões de dados no modo ATM nativo. Para tal, utilizamos a plataforma de pesquisa HARP do sistema operacional FreeBSD. Tal plataforma implementa o modelo *Classical IP* sobre ATM utilizando os serviços AAL5 e ainda fornece uma API para *sockets* ATM permitindo que uma aplicação tenha acesso direto à camada AAL5. Testes de taxa de transmissão, perda de células, e atraso na rede, foram realizados com base na modificação de parâmetros relacionados à aplicação e sistema operacional, tais como tamanho da mensagem sendo transmitida e tamanho dos *buffers* de *socket*.

Abstract

The solution for the lack of bandwidth was solved by the recent development of high speed networks technologies, such as Asynchronous Transfer Mode – ATM. The question now is the implementation of systems which provide support for ATM protocols in an integral way. Nowadays, legacy network applications, based in TCP/IP protocols have been used on top of ATM protocol stack. Such approach generates redundances that have as consequence an increase in the communication overhead. Many models have been developed to provide ATM services directly to applications.

The goal of this study was to evaluate the performance of data transfers comparing the TCP/IP protocols over ATM with the performance of data transfers in native ATM mode. The Host ATM Research Plataform (HARP) was utilized on the FreeBSD operating system. This plataform implements Classical IP over ATM, utilizing AAL5 services. Furthermore, it provides an API for ATM sockets, allowing that an application directly access the AAL5 services.

Measurements of throughput and investigations about cell loss and delay were carried out altering parameters related to the application and operating system. The parameters adopted were message size and socket buffer sizes.

IESC/UFRJ ... DE BIBLIOTECA INFORMAÇÃO

Capítulo 1

Introdução

Desde a década de 70, computadores têm sido conectados por redes como Ethernet e Token Ring. Tais modelos de rede utilizam meios de transmissão compartilhados e, por isso, não são mais adequados para suportar o contínuo e dramático aumento do volume de dados sendo transmitidos, a demanda por maior largura de banda e a confiabilidade em conexões para transmissão de áudio e vídeo.

Neste cenário, as redes ATM (Asynchronous Transfer Mode) [1,2,3] surgiram solucionando a carência por largura de banda e implementando qualidade de serviço (QoS) nas conexões. ATM é uma tecnologia de rede orientada à conexão que foi definida como o padrão para redes ISDN de banda larga. Suas especificações técnicas já alcançaram um estágio de padronização suficiente para o desenvolvimento de redes operacionais, tanto a nível local (LANs) quanto geograficamente estendido (WANs).

Nos últimos anos, temos visto intenso trabalho sobre tudo que se relaciona a ATM, tanto na área acadêmica quanto na indústria. Entretanto, com a disponibilidade de redes de alta velocidade, a largura de banda não é mais um gargalo. Hoje, o foco

do problema, se deslocou para o estudo das deficiências dos protocolos de camadas mais altas (principalmente TCP/IP) e a integração desses protocolos com redes de alta velocidade.

No presente trabalho, será analisado o desempenho de redes ATM nos seguintes ambientes: TCP/IP sobre ATM, UDP/IP sobre ATM e ATM nativo.

Chamamos de ATM nativo o modo pelo qual aplicações utilizando-se de *sockets* especiais acessam diretamente os serviços ATM.

Medidas de taxa de transferência, atraso e investigações a respeito de perda de células foram realizadas em função de parâmetros como tamanho da mensagem sendo transmitida e tamanho dos *buffers* de *sockets*.

1.1 Motivação

A tecnologia Asynchronous Transfer Mode marcou o início de uma nova era para a integração de redes e transmissão de dados. A tecnologia ATM foi idealizada e implementada para transmissões de alta velocidade de voz, vídeo e dados através de redes de computadores. Além de alta velocidade, outras vantagens sobre tecnologias de rede tradicionais incluem a capacidade de acompanhar a demanda por largura de banda, o crescimento do tamanho da rede de modo bastante flexível e a capacidade de reserva de largura de banda no momento da conexão.

Sendo uma tecnologia tão promissora, sua habilidade de manter compatibilidade com protocolos e aplicações já existentes é fundamental. Infelizmente, a maioria dos *softwares* e aplicações de rede não suportam os

protocolos nativos do ATM. Com a competição de outras tecnologias de rede emergentes, a migração das redes atuais para redes ATM não é garantida, a menos que a tecnologia ATM demonstre que tem vantagens significativas sobre as outras. É indiscutível a importância dos protocolos TCP/IP no mercado atual.

As vantagens que a tecnologia ATM fornece são geralmente aceitas por pesquisadores e administradores. No entanto, existem diferentes maneiras de se fazer uso destas vantagens:

I. Executar aplicações de rede diretamente no topo da pilha de protocolos ATM. Freqüentemente diz-se que as aplicações estão sobre "ATM nativo". No entanto, o principal problema deste modelo é que as APIs (Application Program Interface) fornecidas para programas usando ATM nativo são geralmente incompatíveis com as aplicações de rede existentes (a pilha de protocolos ATM não tem conformidade com o modelo OSI). Portanto, são necessárias modificações nas aplicações atuais [2,5].

II. Emular os outros protocolos no topo da pilha de protocolos ATM. Dois exemplos de padrões como esse são: LANE (LAN Emulation) [6,7] e MPOA (Multiprocol over ATM) [8], ambos provenientes do grupo ATM Forum. Outro grupo, também fundamental na padronização das redes ATM, o IETF (Internet Engineering Task Force), desenvolveu outro padrão conhecido como "Classical IP over ATM" [9,10]. A emulação de protocolos sobre ATM é, definitivamente, menos dispendiosa em comparação com uma migração completa de software e hardware para uma rede ATM. No entanto, este procedimento traz questões de incompatibilidade, tais como a resolução de

endereços IP sobre redes ATM. As camadas de protocolos adicionais introduzem redundâncias e atrasos.

III. Rejeitar a tecnologia ATM como um todo mas prover algumas das funcionalidades do ATM como extensões dos protocolos existentes. Como exemplo, temos os protocolos de qualidade de serviço (QoS) no topo de redes IP, tais como:

- RSVP (Resource Reservation Protocol) [11];
- RTP (Real-Time Transpot Protocol) desenvolvidos pelo grupo IETF [12];
- Intserv (Integrated Sevices) [50];
- Difserv (Differentiated Services) [51,52].

Vários trabalhos mostram que TCP/IP sobre ATM (também conhecido como TCP/IP/ATM), embora tenha desempenho significativamente melhor do que sobre Ethernet, introduz consideráveis *overheads* que impedem que se alcance a largura de banda máxima da rede [21, 36, 37, 39, 43, 44].

No entanto, na maioria destes trabalhos, são utilizadas estações de alto desempenho, em geral com alto custo, e chaves ATM de fabricantes diferentes e em topologias distintas. Em nosso trabalho, queremos saber como se comportam estações mais simples e baratas no ambiente da chave ATM presente em nosso laboratório. Na verdade, a maioria dos computadores encontrados nas redes de computadores, atualmente, são PCs.

Outro problema é que, as APIs que dão suporte ao ATM nativo para o sistema operacional também são todas diferentes umas das outras nos trabalhos encontrados, de modo que fica difícil a comparação entre elas.

Existem muitas implementações de ATM nativo para vários sistemas operacionais. Nós decidimos adotar a plataforma HARP (Host ATM Research Plataform) que dá suporte a ATM no sistema operacional FreeBSD, um sistema operacional de código aberto, muito bem implementado, seguro e estável [15]. A opção pela plataforma HARP justifica-se pelos seguintes aspectos:

1. Trata-se de uma plataforma que possibilita tanto o uso de TCP/IP sobre ATM quanto a utilização de ATM nativo;
2. O ATM nativo é acessado através de uma API de *sockets* muito parecida com a API de *sockets* do BSD tradicional, de tal maneira que a transcrição das chamadas de *socket*, para que um programa passe a ter acesso aos serviços nativos do ATM, torna-se fácil;
3. É uma das poucas opções, e com certeza a que foi melhor projetada e implementada, para a integração de sistemas FreeBSD a redes ATM.

Mais detalhes sobre a plataforma HARP, bem como outros exemplos de suporte a ATM em sistemas BSD serão apresentados no Capítulo 3.

1.2 Objetivos

O objetivo maior deste trabalho é caracterizar o desempenho de três ambientes de transmissão de dados utilizando-se o sistema FreeBSD em redes ATM: TCP/IP sobre ATM, UDP/IP sobre ATM e ATM nativo.

Pretende-se seguir as etapas listadas abaixo:

- Caracterizar o desempenho da arquitetura e do sistema operacional das máquinas utilizadas, isto é, levantar questões de desempenho relacionadas à transferência de dados dentro da estação;
- Estudar o efeito de vários parâmetros no desempenho dos protocolos e, em particular, o acréscimo de *overhead* gerado pelos protocolos TCP/UDP e IP quando utilizados sobre a pilha de protocolo ATM;
- Caracterizar o desempenho da API para *sockets* ATM;
- Testar as ferramentas da plataforma HARP disponíveis para gerenciamento de uma rede ATM ao nível do sistema operacional e discutir a funcionalidade destas.

1.3 Apresentação dos Capítulos Seguintes

Além deste capítulo, onde introduzimos o contexto e objetivos do trabalho, esta dissertação é composta por mais quatro capítulos.

No Capítulo 2, apresentamos as características fundamentais das redes ATM que interessam ao nosso estudo. Tal apresentação é feita tomando por base de comparação as redes Ethernet. Pontos fundamentais sobre o ambiente de rede ATM que utilizamos são colocados, tais como: o funcionamento de uma rede ATM *Classical IP* e especificações técnicas sobre todo o equipamento ATM utilizado, bem como a topologia da rede montada no laboratório.

No Capítulo 3 apresentamos detalhes da implementação de *sockets* BSD como base para o entendimento dos *sockets* ATM disponibilizados pela plataforma HARP. Outras implementações existentes para ATM nativo em sistemas BSD são discutidas. Um tópico de fundamental importância é discutido: a análise dos dados coletados em nossos experimentos. É feita uma previsão teórica da largura de banda de transmissão que é efetivamente disponibilizada a uma aplicação depois de se descontar os *overheads* impostos por todas as camadas de protocolo inferiores. Por fim, neste capítulo, encontram-se números resultantes de experimentos de terceiros com ATM nativo e TCP(UDP)/IP sobre ATM.

O Capítulo 4 descreve, detalhadamente, a implementação da ferramenta utilizada em nossas coletas de dados, isto é, o programa para avaliação de desempenho de redes de computadores, Netperf. Seu código fonte, nos trechos relevantes ao nosso estudo, é dissecado até o nível das chamadas de *socket*. As modificações realizadas para a utilização dos *sockets* ATM no lugar dos *sockets* BSD são discutidas. Por fim, são apresentados os resultados de testes de transmissão em *loopback* e testes para avaliar o tempo de resposta na rede.

O Capítulo 5 contém os gráficos resultantes das medidas de taxas de transmissão de acordo com a variação dos parâmetros adotados, isto é, tamanho dos *buffers* de transmissão e recepção e tamanho das mensagens. Os gráficos são analisados e interpretados.

O Capítulo 6 condensa, de maneira conclusiva, a análise dos gráficos apresentados no capítulo anterior e aponta direções para a continuidade do trabalho, além de propor investigações que ainda poderão ser consideradas em trabalhos futuros.

Capítulo 2

Hardware, Topologia e Protocolos

Como forma de discutir as principais características das redes ATM, faremos um paralelo destas com redes Ethernet. Escolhemos Ethernet como base de comparação por ser um padrão de interconexão de computadores em rede amplamente utilizado ainda hoje, inclusive em nosso laboratório, interligando os computadores entre si e estes à rede departamental.

Trataremos das arquiteturas das redes, da largura de banda de transmissão disponível no meio físico, das pilhas de protocolos, encapsulamento dos dados e qualidades de serviço oferecidas. Por fim, apresentamos as especificações técnicas do equipamento disponível no laboratório bem como a topologia de nossa subrede.

2.1 Ethernet *versus* ATM

Ethernet [3] é uma tecnologia de rede local (LAN) muito comum ainda nos dias de hoje. O padrão Ethernet foi publicado em 1982 pela Xerox. O grupo IEEE (Institute of Electrical and Electronics Engineers), mais tarde, desenvolveu outro padrão para Ethernet que ficou conhecido como IEEE 802.3. Este é o padrão utilizado hoje nas redes Ethernet.

O ATM é uma tecnologia de rede comparativamente nova, projetada para B-ISDN (Broadband Integrated Services Digital Network) para transmissões de voz, vídeo e dados em redes de computadores a altas velocidades.

Em uma comparação das tecnologias Ethernet e ATM, seria insuficiente simplesmente considerar suas larguras de banda, já que outras diferenças importantes devem ser consideradas.

O padrão Ethernet permite transmitir informações entre computadores a uma taxa máxima de 10 Mbps através de um canal de comunicação compartilhado por todas as estações. Já o padrão Fast Ethernet, de 1994, permite a troca de dados a uma taxa máxima de 100 Mbps também sobre um meio compartilhado. O padrão Gigabit Ethernet (padrão finalizado em Junho de 1998) oferece largura de banda máxima de 1 Gbps.

Por outro lado, a maioria das redes ATM atuais trabalha com largura de banda de 100 Mbps, 155 Mbps ou 622 Mbps [3].

A Figura 2.1 mostra as arquiteturas de redes Ethernet e ATM típicas. Uma rede Ethernet é formada por máquinas (estações) conectadas a um meio físico compartilhado. O acesso a esse meio físico compartilhado é determinado pelo

mecanismo MAC (Medium Access Control) que faz parte da interface de rede em cada estação. Um pacote Ethernet só pode ser enviado quando o canal (meio físico) estiver desocupado. Se duas estações tentam transmitir no mesmo instante, seus sinais colidem e a transmissão de ambas é reescalonada. Por possuir um canal compartilhado, quanto mais estações estiverem ligadas ao meio e quanto mais aplicações estiverem transmitindo dados, mais a rede se torna lenta devido ao maior número de colisões.

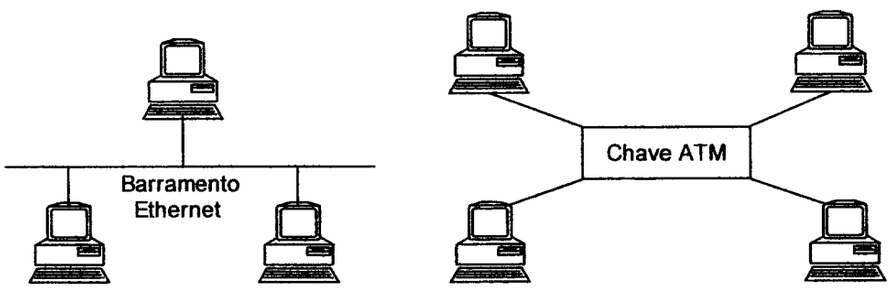

Figura 2.1 Topologias Ethernet e ATM

Switched Ethernet (Ethernet 'Chaveada') [17] é outra tecnologia que tenta adicionar as vantagens de redes 'chaveadas' às redes Ethernet tradicionais. Isto é feito introduzindo *switches* na rede para reduzir o compartilhamento do meio físico. Este modelo é geralmente mais barato que uma rede ATM, mas ainda é precário em termos de outros benefícios que são inerentes às redes ATM, como por exemplo, qualidade de serviço. Em alguns casos, a introdução de chaves em redes Ethernet é considerada uma etapa intermediária na migração para redes ATM [17].

A chave (*switch*), é um equipamento fundamental em ATM, projetada para transferir dados em velocidades extremamente altas. Uma chave ATM recebe dados das máquinas conectadas a ela e transfere estes dados para seu destino. A máquina de destino pode ser um outro computador ou pode ser uma outra chave ATM

intermediária no caminho da fonte dos dados até o destino. Em uma rede ATM, os únicos meios de comunicação compartilhados são internos à chave, capazes de suportar altas taxas e múltiplas transmissões simultâneas de dados sem degradação da taxa de transmissão.

Os dados são transmitidos em células ATM. Uma célula é uma estrutura de dados de 53 bytes, dos quais, 5 bytes são reservados para um cabeçalho que armazena informações de controle da célula restando 48 bytes para dados. Cada cabeçalho de célula inclui um número de circuito virtual (VC). Este número é utilizado pelas chaves ATM para determinar por onde a célula deve ser enviada e é utilizado pela máquina de destino para determinar qual processo deverá receber os dados (isto permite que os dados transmitidos pela rede sejam *demultiplexados* em hardware).

A especificação de circuito virtual (VC) é composta por dois números: o identificador de canal virtual (VCI) e o identificador de caminho virtual (VPI). Observe a Figura 2.2 a seguir. Ela ilustra dois VPC's (Virtual Path Connection) e três VCC's (Virtual Channel Connection) compartilhando o mesmo meio físico. O circuito com VPI = 2 é *chaveado* como um caminho virtual (VP) entre as portas 2 e 5 enquanto que o circuito de VPI = 1 é desmembrado em seus VCC's individuais. Apesar de ambos se originarem na porta 2, um VCC é direcionado para a porta 4 enquanto o outro vai pela porta 3. Algumas chaves mais antigas, conhecidas como *cross-connects* eram capazes de *chavear* apenas VPs [2].

Todos os dados enviados por uma rede ATM são associados com um circuito virtual. Há dois tipos de circuitos virtuais: circuitos virtuais permanentes (PVC's) e circuitos virtuais *chaveados* (SVC's). Circuitos virtuais permanentes são usualmente criados na chave ATM pelo administrador da rede. Circuitos virtuais *chaveados* são

conexões estabelecidas de acordo com a demanda através do uso de complexos protocolos de sinalização [2].

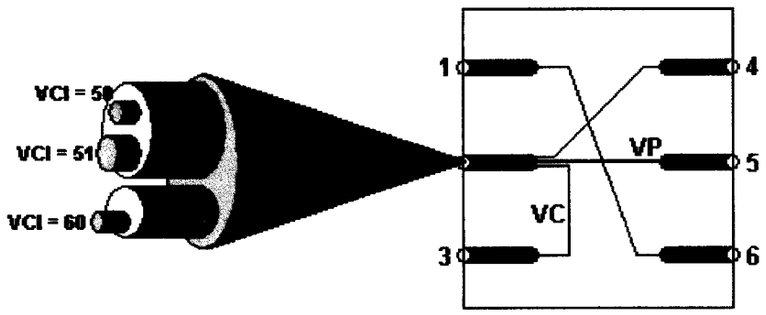

Figura 2.2 *Chaveamento* de caminhos virtuais (VP) e canais virtuais (VC)

A área de 48 bytes de dados da célula ATM é bem pequena quando comparada à área de dados de pacotes como os de redes Ethernet ou FDDI. Para resolver discrepâncias como esta, a pilha de protocolos ATM inclui vários tipos de serviços de adaptação organizados em uma camada chamada de AAL (ATM Adaptation Layer). Além disso, a camada de adaptação define como as informações como voz, dados e vídeo provenientes de camadas superiores são inseridas em células ATM. A camada AAL também pode adicionar cabeçalhos aos dados permitindo a emulação de classes de serviços que não sejam baseados em conexão.

Quatro classes de serviços foram definidas.

Tabela 2.1 Classificação de Serviços para a camada AAL

	Classe A	Classe B	Classe C	Classe D
Temporização	Síncrona		Assíncrona	
Taxa de Bits	Constante	Variável		
Modo	Orientado a Conexão			Sem Conexão
Protocolo AAL	Tipo 1	Tipo 2	Tipo 3/4, Tipo 5	Tipo 3/4

A classificação é feita de acordo com três parâmetros relativos ao tipo da transmissão: relação de tempo entre fonte e destino, taxa de bits constante ou variável e modo de conexão.

Classe A: existe uma relação de tempo entre fonte e destino, a taxa de bits é constante e o serviço é orientado a conexão (exemplo: um canal de voz).

Classe B: existe uma relação de tempo entre fonte e destino, a taxa de bits é variável e o serviço é orientado a conexão (exemplo: um canal de voz ou vídeo)

Classe C: nenhuma relação de tempo entre fonte e destino, a taxa de bits é variável e o serviço é orientado a conexão (exemplo: uma conexão para transferência de arquivos).

Classe D: nenhuma relação de tempo entre fonte e destino, a taxa de bits é variável e o serviço não necessita de conexão (exemplos: interconexão de LANs).

Tais classes de serviços também são conhecidas por AAL1, AAL2, AAL4/3 e AAL5, respectivamente. [1,2].

2.2 Pilhas de Protocolos

Os protocolos de rede convencionais seguem o modelo de referência ISO/OSI (International Standards Organization / Open Systems Interconnection) [5]. O protocolo Ethernet se encaixa nas camadas física e de enlace de dados (camadas 1 e 2) deste modelo.

Sobre a camada Ethernet geralmente é usado o protocolo IP (Internet Protocol) [4,18]. O protocolo IP é um protocolo da camada de rede (camada 3 OSI)

que provê transmissões sem conexão e não confiáveis, sendo também responsável pelo roteamento. Esta camada envia dados para um endereço de destino de 32 bits, único para cada máquina, chamado de endereço IP.

Dois protocolos de camada de transporte (camada 4 OSI) são geralmente utilizados diretamente sobre o protocolo IP. Um deles é o protocolo UDP (User Datagram Protocol) [4,19] que provê um serviço sem conexão e nenhuma garantia de que os dados alcancem o seu destino. O outro protocolo é o TCP (Transmission Control Protocol) [4,20] que provê um serviço orientado a conexão e confiável. Em nossos experimentos não utilizamos os protocolos TCP(UDP)/IP sobre Ethernet mas sim sobre a pilha de protocolos ATM.

Existem três camadas distintas na pilha de protocolos ATM: a camada física, a camada ATM e a camada de adaptação ATM (AAL). No nível mais baixo, a camada física controla como os bits são transmitidos pelo meio físico, cuida dos limites das células ATM e coloca as células no tipo apropriado de pacote que o meio físico estiver utilizando. A camada ATM é responsável por estabelecer conexões e passar as células pela rede ATM. A camada AAL recebe pacotes dos protocolos das camadas superiores a os divide em segmentos de 48 bytes que formam o conteúdo (*payload*) de uma célula ATM. Como já mostramos na Tabela 2.1 há atualmente quatro tipos de serviços oferecidos pela camada AAL: AAL1 suporta serviços orientados a conexão que requerem taxa de bits constante e têm necessidades específicas de temporização e atraso; AAL2 suporta serviços orientados à conexão que não exigem taxa de bits constante; na camada AAL3/4 são oferecidos tanto serviços orientados a conexão quanto sem conexão com taxa de bits variável e, por fim, a camada AAL5 suporta serviços orientados a conexão com taxa de bits variável.

A pilha de protocolos ATM não segue o modelo OSI. A rede ATM roteia as células a nível de hardware, o que se aproxima mais de uma camada de enlace de dados, porém, ela também administra os contratos das conexões e o controle de fluxo. Nenhuma dessas tarefas faz parte da camada de enlace no modelo OSI.

A plataforma de software que integra ATM ao sistema operacional FreeBSD (HARP), utilizada em nossos experimentos, coloca as pilhas de protocolos TCP/UDP e IP sobre a pilha AAL/ATM. Esta disposição de camadas de protocolos é um dos maiores defeitos deste modelo como poderemos constatar nos resultados de nossos experimentos.

2.3 Encapsulamento, Segmentação e Remontagem

Ethernet e ATM usam esquemas diferentes para segmentar, encapsular dados em pacotes para transmissão e remontagem dos dados no destino. Assim, diferentes quantidades de *overhead* são adicionadas pelos dois esquemas.

O protocolo TCP adiciona um cabeçalho de 20 bytes aos dados provenientes de uma aplicação empacotando-os em um "segmento TCP". O tamanho máximo do bloco de dados que pode ser armazenado em um segmento TCP é conhecido como MSS (*Maximum Segment Size*). Este tamanho pode variar de uma máquina para outra e é negociado quando a conexão é criada. O protocolo IP adiciona, em geral, um cabeçalho de 20 bytes aos dados provenientes da camada superior (TCP ou UDP, no nosso caso) para compor então um "*datagrama* IP". Haverá necessidade de fragmentação se este *datagrama* for maior que a unidade de transmissão máxima

(MTU - *Maximum Transmission Unit*), um limite que é imposto pela camada de enlace de dados. Normalmente, uma rede Ethernet com protocolos TCP/IP impõe um limite no MTU de 1500 bytes, considerando os cabeçalhos do TCP e do IP, sobra 1460 bytes para dados.

O MSS do TCP é sempre dimensionado de acordo com o MTU e assim, não há fragmentação na camada IP para TCP sobre IP.

Na seqüência do encapsulamento, um cabeçalho de 14 bytes é adicionado ao início do pacote IP e ainda 4 bytes de checagem de erros (*checksum*) são adicionados ao final do pacote para compor um quadro Ethernet (*frame*), que é passado para a interface de rede para ser transmitido. Os dados são recebidos no destino seguindo o caminho inverso na pilha de protocolos. A Figura 2.3 mostra o encapsulamento de um segmento TCP usando um MSS de 1460 bytes e um MTU de 1500 bytes. Na seção 3.2 serão apresentados maiores detalhes da implementação do TCP/IP no sistema operacional FreeBSD.

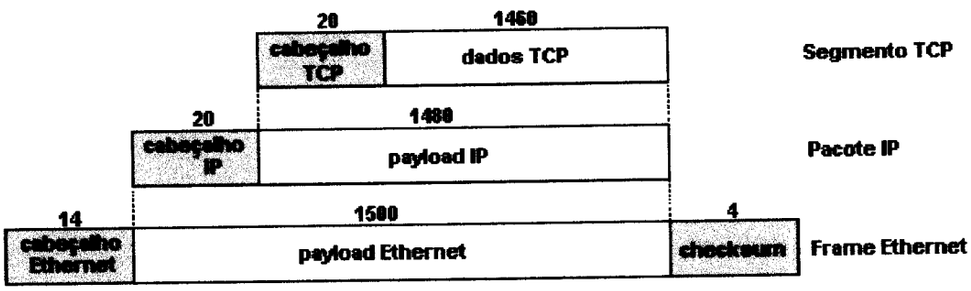

Figura 2.3 Encapsulamento TCP, IP e Ethernet

Ao contrário dos quadros Ethernet e dos pacotes IP, as células ATM são de tamanho fixo. Como já mencionamos anteriormente, esse tamanho é de 53 bytes. O uso de células de tamanho fixo permite que possam ser, de modo fácil e rápido,

manipuladas pelo hardware, o que torna possível uma certa previsão de atraso na rede, fazendo da tecnologia ATM a mais adequada para transmitir tanto informações de tempo real (como por exemplo voz e vídeo) assim como dados.

A plataforma HARP que utilizamos para os testes de desempenho da rede ATM suporta apenas o serviço tipo 5 da camada AAL, isto é, o AAL5. Descreveremos então, a camada AAL5 que é dividida em duas sub-camadas: a sub-camada de convergência (CS) e a sub-camada de segmentação e remontagem (SAR). Como mostrado na Figura 2.4, os dados da camada de aplicação que são passados para a camada AAL5 não podem exceder o máximo de 65536 bytes (64 KB) [1,21].

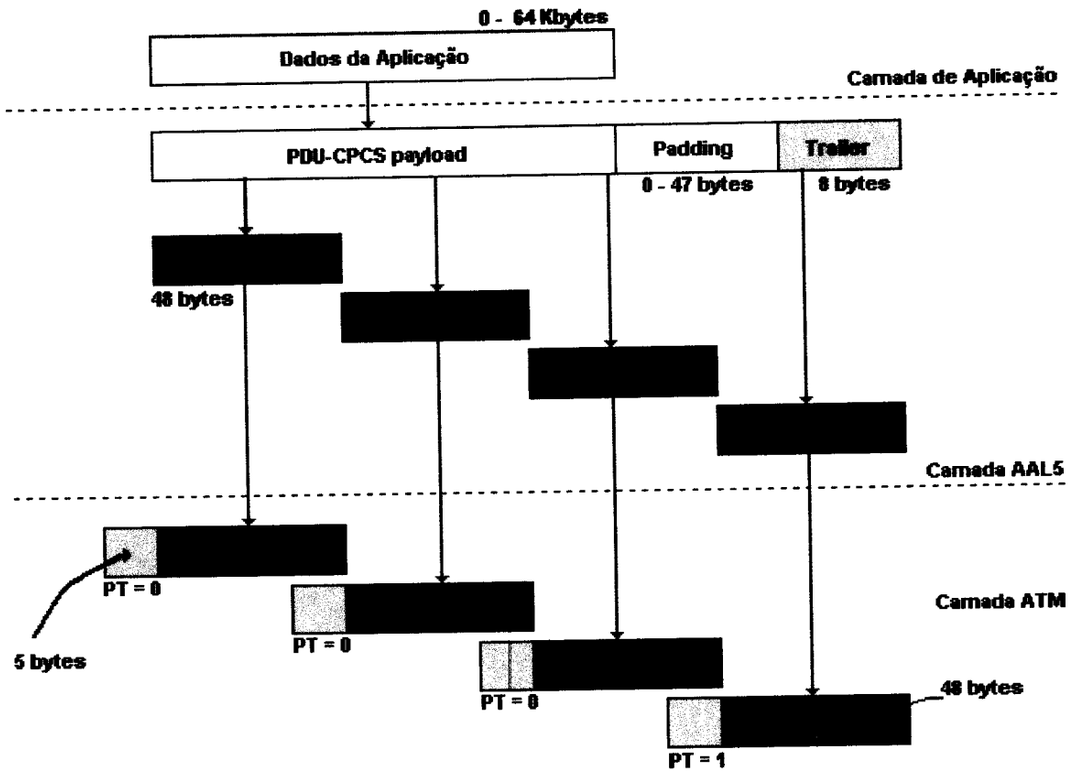

Figura 2.4 Preparação das células para transmissão

A sub-camada de convergência (CS) da camada AAL5, depois de uma checagem de erros (*checksum*), adiciona um preenchimento (*padding*) de

comprimento variável e mais 8 bytes ao final do bloco de dados (*trailer*) para compor um CPCS-PDU (*Common Part Convergence Sublayer Protocol Data Uunit*). O preenchimento garante que o CPCS-PDU formado seja dividido perfeitamente em blocos de 48 bytes que formarão o *payload* das células. A sub-camada de segmentação e remontagem (SAR) divide o CPCS-PDU em blocos de 48 bytes e a camada ATM coloca cada bloco em uma célula atribuindo ao campo PT (*Payload Type*), do cabeçalho da célula, o valor "1" para a última célula e "0" para todas as outras. Quando as células chegam ao destino, a camada ATM extrai o *payload* da célula. A sub-camada SAR reagrupa os SAR-PDU's em um CPCS-PDU. O último segmento SAR-PDU de um CPCS-PDU é reconhecido pelo valor "1" no tipo de *payload* do cabeçalho da célula. Finalmente, a sub-camada CS examina o *checksum* do CPCS-PDU (no *trailer*) para verificar se os dados foram transmitidos e reagrupados corretamente e então passa-os para a camada superior.

2.4 Qualidade de Serviço

A eficiência de uma rede em particular pode ser afetada pela maneira como seus recursos são utilizados. Em uma rede Ethernet, a largura de banda de transmissão é disponibilizada a uma aplicação de acordo com os recursos disponíveis no momento em que tal aplicação tenta utilizar a rede. Assim, em uma rede congestionada, a taxa de transmissão para uma aplicação pode ser muito afetada.

Uma vantagem das redes ATM está no uso de circuitos virtuais. Isto torna fácil prover garantias de desempenho da rede para as aplicações. Redes ATM

oferecem garantias estatísticas da qualidade de serviço (QoS), permitindo que os recursos da rede sejam utilizados de maneira flexível e que larguras de banda sejam reservadas e garantidas para aplicações onde a temporização é um fator primordial.

Um forte competidor para a qualidade de serviço oferecida pelas redes ATM, é o RSVP (Resource Reservation Protocol) [11], projetado para redes IP. No entanto, a principal desvantagem deste protocolo é que ele implementa apenas uma qualidade de serviço denominada "*best-effort*". Isto é, embora uma requisição por uma certa qualidade de serviço seja aceita, a rede pode responder por ela, ou não.

O ATM permite especificar a qualidade de serviço com as seguintes classes: Taxa de Bits Constante (CBR), Taxa de Bits Variável (VBR), Taxa de Bits Disponível (ABR) ou Taxa de Bits não Especificada (UBR).

A plataforma HARP não implementa qualidade de serviço. Assim, em nossos experimentos esta característica das redes ATM não pôde ser explorada. Na verdade, isso fugiria dos limites propostos para este trabalho.

2.5 IP em Redes ATM

2.5.1 *Classical IP*

Como já mencionamos, consideramos neste trabalho a transmissão de dados utilizando os protocolos TCP(UDP)/IP sobre as camadas de protocolos ATM. Tal ambiente de transmissão é chamado de *Classical IP*.

O termo "*classical*" indica que a rede ATM tem as mesmas propriedades das LANs comuns. Isto é, apesar da tecnologia ATM permitir redes de longa distância, também pode ser usada em ambientes restritos como uma substituição direta da

tecnologia de rede local existente. O modelo clássico de LANs conectadas através de roteadores IP é mantido em redes ATM com *Classical IP*. O documento RFC-1577 [9,10] estabelece o padrão para *Classical IP* sobre ATM.

O *Classical IP* sobre ATM difere do IP em LANs usuais pelo fato de que a rede ATM provê um ambiente de conexão virtual através do uso de PVCs e SVCs. O gerenciamento de SVCs é executado com base na especificação UNI 3.0/3.1 (ATM Forum), que especifica o padrão de sinalização Q.2931. Este último é um protocolo de sinalização projetado para estabelecer conexões dinamicamente na interface usuário-rede (UNI). O pradrão Q.2931 usa o protocolo SSCOP (Service Specific Connection Oriented Protocol) como um protocolo de transporte confiável e toda sinalização acontece através da conexão virtual VPI:0,VCI:5. Conexões Q.2931 são bidirecionais, sendo que o mesmo par VPI/VCI pode ser utilizado para transmitir e receber.

Uma vez que uma conexão *Classical IP* tenha sido estabelecida, os pacotes IP são encapsulados utilizando-se o padrão IEEE 802.2 LLC/SNAP e são segmentados em células ATM pela camada AAL5. A unidade máxima de transmissão (MTU) é de 9180 bytes (o cabeçalho LLC/SNAP tem 8 bytes) sendo que o tamanho limite de um pacote é de 65.535 bytes. Atualmente não há suporte para *broadcast* ou *multicast* de pacotes IP no ambiente *Classical IP*.

Um conceito importante de redes *Classical IP* é o de Sub-redes IP Lógicas (LIS). Uma LIS é formada por um grupo de estações configuradas como membros da mesma sub-rede IP (isto é, têm os mesmos números de rede e sub-rede). Neste aspecto, uma LIS pode se comportar como uma LAN comum. É possível manter várias sub-redes lógicas sobrepostas em uma mesma rede ATM física. Assim, em uma rede ATM *Classical IP*, colocar uma estação em uma sub-rede específica é uma

escolha lógica e não física. Neste tipo de ambiente, a comunicação entre estações localizadas em diferentes sub-redes lógicas só é permitida através de um roteador IP que deve ser membro de ambas as sub-redes.

A fim de suportar roteamento entre várias sub-redes, o *firmware* do adaptador de rede permite que uma estação seja configurada como membro de até quatro sub-redes lógicas distintas. As mesmas ferramentas usadas pelo sistema operacional para configuração de rede (como por exemplo, `ifconfig`) podem ser usadas para controlar o roteamento através de cada uma das interfaces *Classical IP* da mesma forma que é feito o roteamento entre interfaces físicas comuns. Note que, apesar de cada interface *Classical IP* associada a uma certa interface física usar o mesmo hardware, cada uma delas é configurada separadamente com seu próprio MTU, endereço IP e endereço ATM.

2.5.2 NSAP e ILMI

Para que uma estação esteja apta a estabelecer conexões por uma interface física, ela precisa conhecer o endereço NSAP (Network Service Access Point) para tal interface. O propósito principal do protocolo ILMI (Interim Local Management Interface) é descobrir e registrar estes endereços NSAP dinamicamente.

Para uma rede privada ATM, os endereços identificam de maneira única os pontos finais, ou seja, as estações. O formato dos endereços, estabelecido pelo protocolo UNI 3.0, foi modelado de acordo com o padrão OSI NSAP. Três formatos de endereço foram especificados: DCC, ICD e E.164 [2]. O equipamento da empresa americana Fore, que utilizamos em nosso trabalho, adota o formato ICD [7,14]. Pela

especificação UNI 3.0, toda rede privada deve aceitar mensagens iniciais de configuração contendo endereços ATM em um dos formatos aprovados e responder tais mensagens se necessário. Um endereço NSAP consiste do seguinte:

- uma parte dedicada à rede (13 bytes) - Este é o prefixo NSAP do SCP (Switch Control Processor) ao qual a estação está conectada;
- uma parte dedicada à estação (7 bytes) - formada por:
  - Identificador de sistema final (ESI - End System Identifier): ESI é o endereço MAC IEEE único da interface (6 bytes);
  - um byte (seletor) que identifica uma interface lógica *Classical IP* específica.

Como exemplo concreto, tomemos duas máquinas da rede ATM em nosso laboratório. Tais máquinas, denominadas de atm2 e atm7, possuem a plataforma HARP instalada. Através do comando "`atm show interface`", podemos saber o endereço NSAP de uma estação. Assim, executando tal comando em atm2 temos:

```
#atm2> atm show interface
Interface  Sigmgr   State
hfa0       UNI 3.1  ACTIVE
ATM address = 0x47.0005.80.ffe100.0000.f21a.639f.002048083f75.00
Network interfaces: ni0
```

Da mesma forma, executando tal comando em atm7 temos:

```
#atm7> atm show interface
Interface  Sigmgr   State
hfa0       UNI 3.1  ACTIVE
ATM address = 0x47.0005.80.ffe100.0000.f21a.639f.0020484039e6.00
Network interfaces: ni0
```

É importante notar que o endereço ATM é igual para as duas máquinas nos 13 primeiros bytes, diferindo apenas na parte reservada ao endereço do sistema final. Isso significa que ambas as máquinas estão ligadas à mesma chave ATM. Note ainda que o byte seletor dos dois endereços é zero. Isso acontece porque ambas as

IFSC-USP SERVIÇO DE BIBLIOTECA INFORMAÇÃO

máquinas estão configuradas com apenas uma interface lógica do tipo *Classical IP* chamada `ni0`.

Chaves ATM Fore Sytems com o *firmware* ForeThought 3.0 ou versões posteriores[1] têm suporte para ILMI. Se existe suporte a ILMI em todas as estações e chaves de uma rede, quando o SCP (Switch Control Processor) da chave é iniciado (*boot up*), o protocolo ILMI possibilita ao SCP identificar dinamicamente todas as estações ligadas a ele e enviar a elas seus prefixos NSAP. Como resposta, as estações anexam àquele prefixo seus ESI (End System Identifier) e bytes seletores, formando um endereço NSAP completo. Então, as estações notificam o SCP de seus NSAPs completos. Estas mensagens SNMP são enviadas e recebidas sobre os serviços AAL5 utilizando VPI:0 e VCI:16. Uma vez que o processo de registro ILMI tenha terminado, pode-se, então, configurar as conexões.

2.5.3 ARP e Servidores ARP

Em ambiente *Classical IP*, para que uma estação estabeleça uma conexão com outra estação, ela primeiro precisa determinar o endereço NSAP da outra estação. ATM ARP (ATM Address Resolution Protocol) é o procedimento usado para resolver um endereço IP em um endereço ATM.

Como o padrão ATM não suporta *broadcast* em uma rede local, a resolução de endereços é feita através da comunicação direta com um servidor ARP especial, ao invés de um *broadcast* de requisições ARP como é feito nas LANs usuais.

---

[1] A chave Fore ASX-200BX que utilizamos possui atualmente a versão ForeThought 5.3.0.

Cada sub-rede lógica (LIS) deve possuir apenas um servidor ARP configurado. No entanto, um único servidor ARP pode servir a várias sub-redes lógicas. Cada estação em uma LIS deve ser configurada com o endereço NSAP da estação que for a servidora ARP para tal LIS. O endereço do servidor ARP é normalmente configurado em cada estação durante a instalação (os comandos são colocados na seqüência de carregamento do *firmware*) ou então, a qualquer momento, se o administrador da rede precisar alterá-lo.

Uma vez que uma estação já conheça seu próprio endereço ATM e o endereço ATM do seu servidor ARP, ela irá tentar estabelecer uma conexão com o servidor ARP que será usada para enviar requisições ARP e receber respostas ARP. Quando a conexão com o servidor ARP for estabelecida, o servidor ARP envia uma requisição ARP inversa (InARP) no novo VC para aprender o endereço IP da estação. Quando uma resposta InARP é recebida, o servidor ARP coloca este mapeamento de endereço IP para endereço ATM em seu *cache*. Assim, com o passar do tempo, o servidor ARP aprende dinamicamente os mapeamentos de endereços IP-ATM de todas as estações em sua LIS.

Quando uma estação quer se comunicar com outra estação na mesma LIS, ela primeiro envia uma requisição ARP para o servidor ARP contendo seu endereço IP para ser resolvido. Quando uma resposta ARP é recebida do servidor ARP, a estação cria uma entrada em seu *cache* e armazena este mapeamento de endereços IP-ATM. Para garantir que todos os mapeamentos de endereços IP-ATM conhecidos por uma certa estação permaneçam atuais, é exigido das estações que façam uma validação de seus mapeamentos a cada 15 minutos [7]. Se uma conexão *Classical IP* fica ociosa por 15 minutos, ela é automaticamente desfeita.

2.5.4 Sinalização ATM

Em qualquer ambiente ATM, um usuário precisa ser capaz de sinalizar o estabelecimento de conexões através da rede. Tais conexões, criadas dinamicamente através de um mecanismo de sinalização, são conhecidas como SVCs (*Switched Virtual Circuit*), em contraste com os PVCs (*Permanet Virtual Circuits*) que são criados pelo administrador da rede. Os esforços para definir as formas de sinalização de SVCs começaram com o órgão ITU (CCITT naquela época) enquanto que, mais recentemente, o grupo ATM Forum tem utilizado três recomendações para gerar o padrão UNI. Três versões deste padrão suportam SVCs – 3.0, 3.1 e a mais recente 4.0. A partir da versão 3.1 o ATM Forum segue o protocolo ITU Q.2931.

A sinalização Q.2931 pode ser considerada uma aplicação executando no topo das camadas inferiores (física, ATM e AAL). No caso do ATM, a sinalização ocorre através da camada SAAL (*Signaling ATM Adaptation Layer*) residente entre a camada ATM e o protocolo de sinalização Q.2931. A camada SAAL provê um meio confiável para o transporte de mensagens de sinalização entre sistemas ATM, incluindo o restabelecimento de múltiplas falhas no fluxo de dados. Esta característica contrasta com o restabelecimento simples (único) disponível nos protocolos LAPB e TCP e é desta forma bastante adequado ao transporte de informação de sinalização através da rede ATM devido à sua robustez. A camada SAAL é composta de duas sub-camadas: uma parte comum e uma parte de serviços específicos. A parte de serviços específicos é subdividida em SSCF (*Service-Specific Coordination Function*) e SSCOP (*Service-Specific Connection Oriented Protocol*). A sub-camada SSCF faz o mapeamento das requisições Q.2931 para SSCOP enquanto que a sub-camada SSCOP inclui mecanismos para estabelecimento e

liberação de conexões e, ainda, para monitorar informações de sinalização trocadas entre entidades de sinalização. É nessa sub-camada que é feita a recuperação de SDUs perdidos ou corrompidos. As recomendações ITU Q.2130 e Q.2110 descrevem as sub-camadas SSCF e SSCOP. A parte comum, baseada nas sub-camadas da camada AAL, SAR e CPCS, garante transferências de informações e detecção de SDUs corrompidos. Entre chaves ATM, uma camada de convergência NNI (Network-Network Interface), através da sub-camada SAAL, é utilizada para sinalização PNNI (Private Network-Network Interface). A inter-relação entre as várias sub-camadas de sinalização é descrita na Figura 2.5.

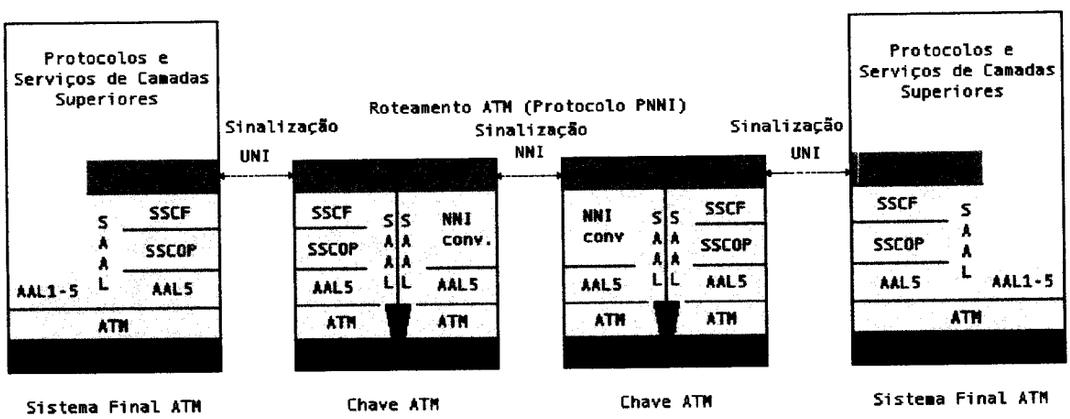

Figura 2.5 Sinalização ATM.

A sinalização entre sistemas finais e a chave ATM acontece, usualmente, sobre o canal VPI=0/VCI=5, embora não seja sempre esse o caso.

2.6 Especificação do Equipamento Utilizado

A Figura 2.5 a seguir ilustra a topologia da rede local utilizada nesse trabalho. As máquinas são todas conectadas à rede Ethernet do departamento a 10 Mbps e a maioria delas está conectada também a uma das duas chaves ATM presentes no laboratório.

As chaves ATM são duas *Fore Runner* ASX-200 BX com capacidade máxima de 2,5 Gbps. O desenho das chaves na figura abaixo não é meramente ilustrativo. Ele mostra com precisão os módulos pertencentes a cada uma com relação ao número e tipo das portas. As chaves são identificadas através dos nomes IFSC-USP e LIE-ATM e estão interligadas através de par trançado (UTP 5) a 155 Mbps (porta B1 em LIE-ATM e porta D1 em IFSC-ATM)[2].

Todas as conexões físicas entre as chaves e as máquinas (traço mais espesso) utilizam o protocolo de meio físico SONET STS-3c sobre cabos UTP 5.

A chave IFSC-USP é o servidor ATM ARP para nossa sub-rede lógica *Classical IP*.

---

[2] A nomenclatura das portas nas chaves segue o padrão Fore Systems, onde a letra identifica o módulo e o número identifica a porta, de 1 a 4, da esquerda para a direita.

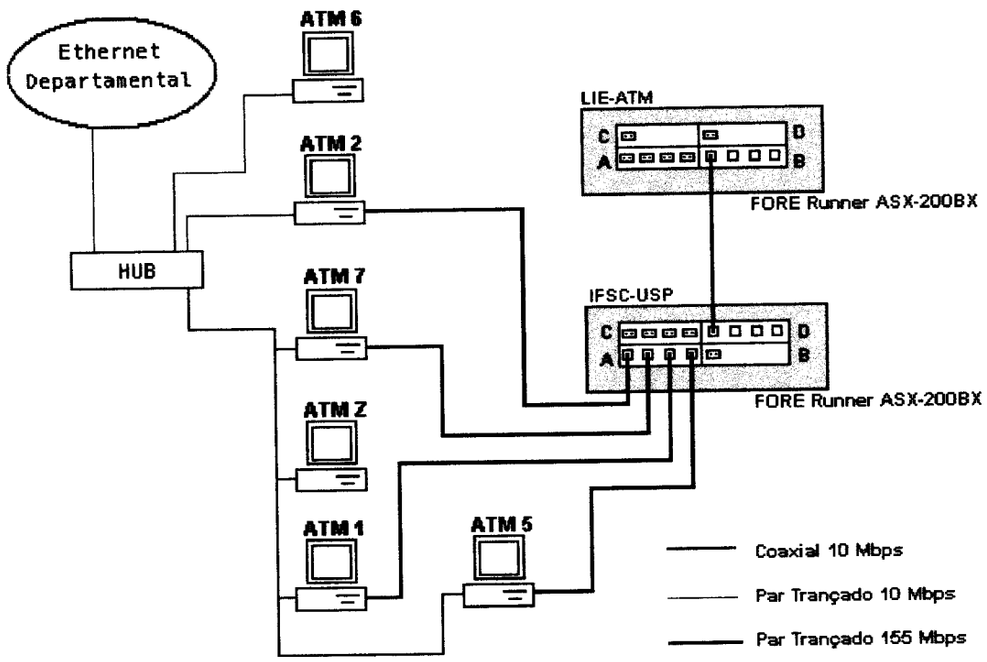

Figura 2.6 Estrutura de rede do laboratório.

Cada chave possui quatro módulos denominados de A, B, C e D com as seguintes especificações:

Tabela 2.2 Configuração da Chave LIE-ATM.

CHAVE LIE-ATM	
Módulo A: NM-4/155MMSCC	Quatro portas SONET/SDH, 155.52 Mbps, *Multimode fiber*
Módulo B: NM-4/155UTP5C	Quatro portas SONET/SDH, 155.52 Mbps, UTP categoria 5
Módulo C: NM-1/622MMSCC	Uma porta SONET/SDH, 622.08 Mbps, *Multimode fiber*
Módulo D	idêntico ao módulo C

Tabela 2.3 Configuração da Chave IFSC-USP.

CHAVE IFSC-USP	
Módulo A: NM-4/155UTP5LC	Quatro portas SONET/SDH, 155.52 Mbps, UTP categoria 5
Módulo B: NM-1/622SMIRD	Uma porta SONET/SDH, 622.08 Mbps, *Singlemode fiber*
Módulo C: NM-4/155MMSCD	Quatro portas SONET/SDH, 155.52 Mbps, *Multimode fiber*
Módulo D	Idêntico ao módulo A

Maiores detalhes técnicos podem ser encontrados no Manual do Usuário da chave ATM [7,14].

As máquinas utilizadas nos experimentos são aquelas que têm conexão ATM, ou seja: ATM1, ATM2, ATM5 e ATM7. Todas as máquinas acima utilizam o sistema operacional FreeBSD 3.1, Fore PCA-200EPC com processador RISC i960 de 25 MHz, em barramento PCI e a plataforma HARP instalada em sua versão mais recente, qual seja, a 3.0 [13].

Através do comando HARP "`atm show config`" extraímos mais detalhes sobre o hardware e *firmware* da placa ATM. Tais informações são mostradas a seguir. Para cada máquina são mostradas configurações básicas e o resultado do comando mencionado acima.

Tabela 2.4 Configuração das máquinas utilizadas nos experimentos.

ATM1	Pentium 100 MHz, 24 MB de memória RAM
	Intf Vendor Model Media Bus Serial No hfa0 Fore PCA-200E OC-3c PCI 14923 MAC address = 00:20:48:08:3a:4b Hardware version = 1.0.5 Firmware version = 4.0.0
ATM2	Pentium II 233 MHz, 64 MB de memória RAM
	Intf Vendor Model Media Bus Serial No hfa0 Fore PCA-200E OC-3c PCI 16245 MAC address = 00:20:48:08:3f:75 Hardware version = 1.0.5 Firmware version = 4.0.0
ATM5	Pentium Pro, 64 MB de memória RAM
	Intf Vendor Model Media Bus Serial No hfa0 Fore PCA-200E OC-3c PCI 4209005 MAC address = 00:20:48:40:39:6d Hardware version = 2.0.1 Firmware version = 4.0.0
ATM7	Pentium II 233 MHz, 64 MB de memória RAM
	Intf Vendor Model Media Bus Serial No hfa0 Fore PCA-200E OC-3c PCI 4209126 MAC address = 00:20:48:40:39:e6 Hardware version = 2.0.1 Firmware version = 4.0.0

Capítulo 3

ATM: Implementações, Experiências e Desempenho

FreeBSD é um sistema operacional de código aberto e utilizado em computadores compatíveis com a arquitetura Intel (x86), DEC Alpha e PC-98. É baseado no sistema operacional 4.4BSD desenvolvido na Universidade da Califórnia, Berkeley. Assim como o sistema Linux, e outros de domínio público, o sistema FreeBSD foi, e ainda é, desenvolvido por um vasto time de programadores voluntários espalhados pelo mundo todo. Mais informações podem ser encontradas no *site* do projeto FreeBSD na Internet [16]. Detalhes de implementação podem ser encontrados no livro que descreve o sistema operacional 4.4BSD [22,23].

Neste capítulo, analisamos aspectos da organização de uma parte do núcleo do sistema que cuida das funções de rede. Apresentamos também a plataforma HARP (Host ATM Research Plataform) e mencionamos outras implementações de ATM nativo e seus desempenhos. Por fim, calculamos o *overhead* devido às pilhas de protocolos nos ambientes TCP/IP/ATM e ATM nativo.

3.1 Detalhes de Implementação dos *sockets* BSD

Nesta seção, faremos uma explanação sobre a implementação de *sockets* ao nível do sistema operacional. Cremos que é importante analisar tais características pois estudos anteriores [24] mostram que o sistema operacional e os custos de gerenciamento de memória são as duas principais fontes de *overhead* em redes TCP/IP. Uma terceira fonte seria o protocolo TCP.

Várias operações dispendiosas são executadas pelo sistema operacional. Operações tais como manipulação de interrupções, alocação e liberação de *buffer* e inicializações de entrada/saída requerem um tempo de processamento considerável. O tempo para mover dados na memória também acrescenta *overhead*. Durante cada operação de entrada/saída sobre a rede, dados são copiados do espaço de endereçamento do usuário para o do *kernel* e vice-versa. Os ciclos de memória necessários para cada cópia, a quantidade de dados copiados e o tempo de acesso à memória são todos fatores que limitam a velocidade da transmissão dos dados.

O sub-sistema de rede do *kernel* dos sistemas BSD é composto, basicamente, por três camadas: a camada de *socket*, a camada de protocolo e a camada de interface de rede [22]. Os dados a serem transmitidos transitam desde a aplicação passando pelas três camadas citadas, na mesma ordem em que foram apresentadas acima. A Figura 3.1 a seguir ilustra tais camadas.

As necessidades impostas pelos protocolos de rede ao esquema de gerenciamento de memória são bem diferentes das exigências de outras partes do sistema operacional. Embora todas necessitem de eficiência na alocação e liberação de memória, protocolos de comunicação, em particular, precisam de espaços de

memória de tamanhos bastante variáveis [22]. Protocolos de comunicação devem acrescentar ou remover cabeçalhos aos dados "empacotados".

A medida que os dados são enviados, os dados no *buffer* devem ser divididos em pacotes e pacotes recebidos devem ser combinados em um único objeto. Além disso, pacotes e outras informações de gerenciamento devem ser enfileirados enquanto estiverem esperando para serem transmitidos ou recebidos.

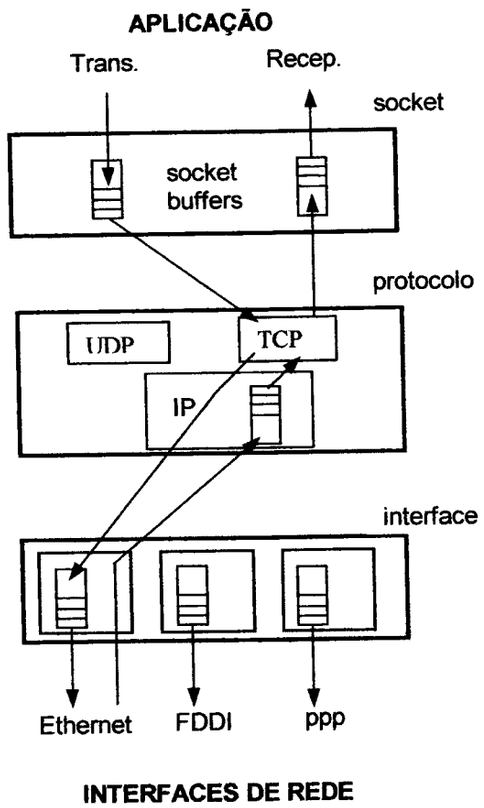

Figura 3.1 Sub-sistema de rede do *kernel* BSD.

Os mecanismos de gerenciamento de memória atendem essas necessidades através de uma estrutura de dados especial chamada de *mbuf (memory buffer)*. *mbufs* são estruturas de dados de tamanho fixo localizadas no espaço de endereçamento do

kernel e podem ser compartilhadas por vários protocolos. O número de *mbufs* disponíveis ao sistema é dinamicamente configurado pelo *kernel* com base na quantidade de memória física. A estrutura *mbuf* é formada por um cabeçalho e um espaço para dados. Para transmitir uma quantidade dados maior que o espaço de dados de um *mbuf*, várias dessas estruturas podem ser ligadas umas às outras para formar uma "cadeia de *mbufs*". Cadeias de *mbufs* são usadas para armazenar pacotes. Dados podem ser adicionados a qualquer extremidade da cadeia ligando a ela mais *mbufs*. Cadeias de *mbufs* podem ser ligadas umas às outras para formar uma fila de pacotes. Um *mbuf* tem 128 bytes dos quais 20 bytes formam o cabeçalho e os 108 bytes restantes são reservados para dados (*payload*). Em uma cadeia de *mbufs*, que armazena um pacote de dados, o primeiro *mbuf* tem seu cabeçalho acrescido de mais 8 bytes de informações sobre o pacote. Assim, teremos uma cadeia de *mbufs* onde o primeiro poderá armazenar até 100 bytes (128 - 20 - 8) e os demais 108 bytes. Se o pacote de dados for maior que 208 bytes, ao invés de armazenar os dados no *mbuf*, a estrutura pode referenciar um bloco de memória externo que recebe o nome de *mbuf cluster*. Este tem um tamanho máximo de 2048 bytes nos sistemas BSD[3]. Isto reduz o número de *mbufs*, e como tais *clusters* são copiados por referência, apenas ponteiros para os blocos de memória são copiados. Esta estratégia melhora muito a eficiência na transferência de dados do espaço das aplicações para o espaço do *kernel*. A Figura 3.2 mostra com mais detalhes os quatro tipos de *mbuf* implementados pelo sistema 4.4BSD [22]. Os dados são armazenados, ou na área de dados do *mbuf*, ou em um espaço externo (*cluster*), mas nunca em ambos ao mesmo tempo.

---

[3] O sistema operacional FreeBSD, que utilizamos em nossos experimentos, é absolutamente compatível com o sistema 4.4BSD por ser uma versão baseada neste.

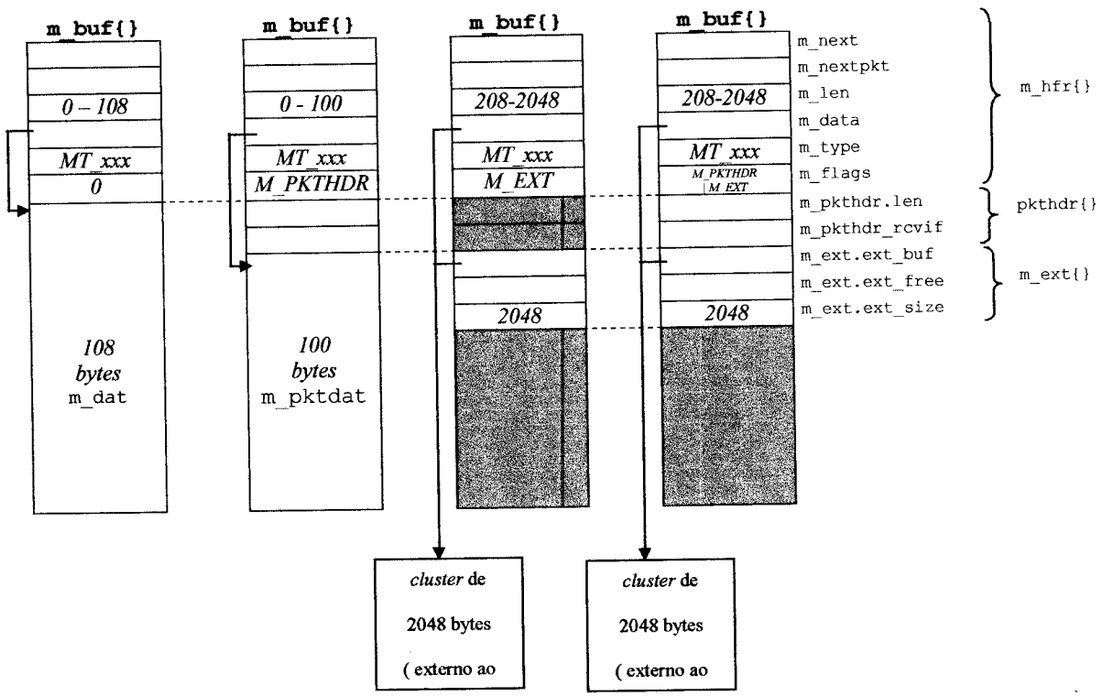

Figura 3.2 Tipos de *mbuf* do sistema BSD[4].

A camada de *socket* tem duas funções principais. Ela é responsável por transferir dados do espaço de endereçamento de usuário para o espaço de dados dos *mbufs*. Sua segunda função é enfileirar os dados entre o usuário e o *kernel*. Se um processo tenta transmitir muitos dados e o *buffer* do *socket* fica cheio, então a camada de *socket* suspende a execução do processo temporariamente (*sleep*) até que mais espaço esteja disponível. Da mesma forma, se um processo tenta ler dados da fila e a fila está vazia, então a camada de *socket* pode suspender o processo até que mais dados estejam disponíveis.

[4] As áreas escuras não são utilizadas.

Todo processamento dos protocolos é realizado na camada de protocolo. Por exemplo, os protocolos TCP, UDP e IP são todos implementados na camada de protocolo no sub-sistema de rede do BSD [22]. Quando está transmitindo, a camada de protocolo recebe dados da camada de *socket*, adiciona os cabeçalhos de protocolo necessários, e passa o pacote para ser transmitido pela camada de interface de rede. Quando está recebendo dados, a camada de protocolo tira os pacotes de sua da fila de entrada e determina o caminho que cada pacote deve seguir. Se o pacote é destinado a outra máquina, então a camada de protocolo passa-o de volta à camada de interface de rede. Se o pacote é destinado a um processo local, então a camada de protocolo enfileira o pacote no *buffer* do *socket* relacionado àquele processo e avisa a camada de *socket* que novos dados estão disponíveis.

A camada de interface de rede transfere os pacotes entre o hardware da interface e a camada de protocolo. Quando está transmitindo dados, a camada de interface de rede recebe pacotes através de sua fila e os transmite pelo meio físico. Quando está recebendo dados, a camada de interface de rede determina a qual protocolo passar o pacote, enfileira o pacote para tal protocolo e então prepara uma interrupção de software para atender o protocolo.

Um *socket* mantém duas estruturas conhecidas como *buffer* de transmissão (*send buffer*) e *buffer* de recepção (*receive buffer*). Cada uma delas contém uma fila de *mbufs*. Cada *buffer* possui um limite superior de tamanho conhecido como *socket buffer size* para limitar o tamanho da fila. A Figura 3.3 a seguir mostra a declaração da estrutura de dados usada para declarar os *buffers* de transmissão e recepção [23].

```
struct sockbuf {
    u_long    sb_cc;
    u_long    sb_hiwat;
    u_long    sb_mbcnt;
    u_long    sb_mbmax;
    long        sb_lowat;
    struct  mbuf    *sb_mb;
    struct  selinfo    sb_sel;
    short        sb_flags;
    short        sb_timeo;
} so_recv, so_snd;
```

Figura 3.3 Estrutura dos *buffers* de *socket* nos sistemas BSD.

Como podemos ver, cada *buffer* contém informações de controle como, por exemplo, ponteiros para os dados armazenados nas cadeias de *mbufs*. `sb_mb` aponta para o primeiro *mbuf* na cadeia, `sb_cc` é o total, em bytes, de dados nos *mbufs*. `sb_hiwat` e `sb_lowat` regulam os algoritmos de controle de fluxo dos *sockets*. `sb_mbcnt` é a quantidade total de memória reservada para *mbufs* no *buffer*. `sb_mbmax` é um limite superior para a quantidade de memória a ser reservada para *mbufs* para cada *buffer* de *socket*. Os limites padrões são especificados por cada protocolo quando uma aplicação executa uma chamada da função `socket()`. A Tabela 3.1 mostra os valores padrões para os protocolos de Internet, isto é, protocolos do domínio de *sockets* AF_NET [23,25]. Quando uma aplicação cria um *socket*, os *buffers* de transmissão e recepção são criados e configurados mas seus tamanhos podem ser alterados usando a função de sistema `setsockopt()` desde que não ultrapasse o limite imposto pelo *kernel* de 262.144 bytes para cada *buffer*. A alteração desses tamanhos é local à aplicação e portanto, não afeta outras aplicações.

Tabela 3.1 Limites dos *buffers* de *socket* para protocolos do domínio AF_NET.

Protocolo	so_snd			so_rcv		
	sb_hiwat	sb_lowat	sb_mbmax	sb_hiwat	sb_lowat	sb_mbmax
UDP	9 x 1024	2048 (ignorado)	2 x sb_hiwat	40 x (1024+16)	1	2 x sb_hiwat
TCP	8 x 1024	2048	2 x sb_hiwat	8 x 1024	1	2 x sb_hiwat
IP (raw) ICMP IGMP	8 x 1024	2048 (ignorado)	2 x sb_hiwat	8 x 1024	1	2 x sb_hiwat

A tabela mostra, por exemplo, que *sockets* TCP utilizam um tamanho padrão de 16 KB para seus *buffers* de transmissão e recepção (`so_snd` e `so_rcv`).

3.2 A Plataforma HARP

HARP (*Host ATM Research Plataform*) [15] é uma plataforma de código aberto e de alta qualidade para pesquisadores que utilizam o ambiente IP/ATM. Foi desenvolvida pelo grupo denominado Advanced Networking Group (ANG) pertencente ao Networking Computing Services Inc. como um de seus vários projetos de pesquisa patrocinados pelo DARPA (do Departamento de Defesa americano).

A plataforma HARP provê um meio para que estações utilizando o protocolo IP se conectem a redes ATM. Ela suporta os métodos padrões de comunicação, usados pelo protocolo IP, sobre ATM. Com a plataforma HARP, uma aplicação que utiliza o protocolo IP pode enviar e receber *datagramas* através de uma interface ATM. A plataforma HARP provê funcionalidades similares às que são oferecidas pelo controlador de dispositivo do fabricante. A fim de alcançar alto desempenho, a

plataforma HARP foi implementada para fazer parte do *kernel*. Para minimizar o tamanho do *kernel* em tempo de execução, apenas um pequeno componente chamado de "ATM Core Services" fica constantemente residente no *kernel*. Todos os outros módulos serão carregados quando o sistema for configurado, tal que residam no *kernel* apenas se seu uso for requerido.

A plataforma HARP tem seu código fonte inteiramente disponível para que pesquisadores possam realizar experiências na medida do necessário com o protocolo IP sobre ATM. No entanto, neste trabalho, nós apenas utilizamos o software sobre o sistema operacional FreeBSD[5]. Assim, não foi necessária nenhuma alteração em seu código fonte. Modificações foram feitas apenas no código do aplicativo Netperf que foi adaptado para utilizar os sockets ATM disponíveis também em HARP. Mais detalhes sobre Netperf e as alterações em seu código serão detalhadamente apresentadas no Capítulo 4.

Apresentaremos agora, de forma sucinta, as principais características da plataforma HARP, as funcionalidades suportadas e o que ainda não é suportado.

3.2.1 O que a plataforma HARP suporta

- Sistemas Operacionais
  - SunOS 4.1.x para *workstation's* SPARC (sun4c, sun4m)
  - FreeBSD 2.1.5, FreeBSD 2.2.x, FreeBSD 3.0 e versões posteriores
- Interfaces de rede
  - FORE Systems, Inc. SBA-200 e SBA-200E (adaptadores SBus)

---

[5] A plataforma HARP 3 foi incorporada ao sistema operacional FreeBSD a partir de sua versão 3.0. A versão do FreeBSD instalada em nossas máquinas é a 3.1.

IFSC-USP SERVIÇO DE BIBLIOTECA E INFORMAÇÃO

- FORE Systems, Inc. PCA-200E (adaptadores PCI)
- Efficient Networks, Inc. ENI-155p (adaptadores PCI)

- Protocolos de sinalização ATM
  - ATM Forum UNI 3.0 e 3.1
  - ATM Forum ILMI (apenas registro de endereços)
  - FORE Systems SPANS (protocolo proprietário)
  - Circuitos Virtuais Permanentes (PVCs)
- Modelo IETF "Classical IP and ARP over ATM"
  - RFC 1483, "Multiprotocol Encapsulation over ATM Adaptation Layer 5"
  - RFC 1577, "Classical IP and ARP over ATM"
  - RFC 1626, "Default IP MTU for use over ATM AAL5"
  - RFC 1755, "ATM Signaling Support for IP over ATM"
  - RFC 2225, "Classical IP and ARP over ATM"
  - RFC 2334, "Server Cache Synchronization Protocol (SCSP)
  - Internet Draft, "A Distrubuted ATMARP Service Using SCSP"
- API para *sockets* ATM

3.2.2 O que a plataforma HARP não suporta

- Sinalização UNI para comunicação ponto-multiponto
- Controle de Tráfego e Qualidade de Serviço
- IP *Multicast* sobre ATM
- LANE (LAN Emulation) [6]
- MPOA (Multi-Protocol over ATM) [8]

3.2.3 Visão Funcional da Plataforma HARP

A Figura 3.4 abaixo apresenta uma visão funcional da plataforma HARP. Em seguida, apresentamos com mais detalhes cada um dos módulos que compõem a plataforma.

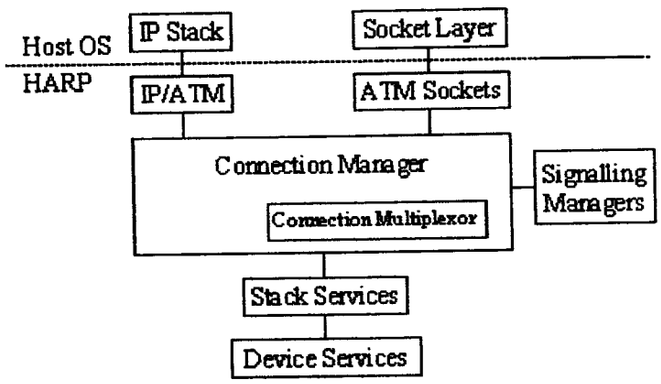

Figura 3.4 Visão funcional da plataforma HARP

Aqui, optamos por não traduzir os nomes dos componentes da figura anterior porque acreditamos que, com a tradução, estaríamos perdendo a clareza e objetividade de tal visão funcional. Assim, segue abaixo uma breve descrição das principais características de cada módulo.

Connection Manager

- Cuida do gerenciamento de todo fluxo de dados e mensagens de controle entre:
  - Pontos finais (IP/ATM, ATM Sockets)
  - Controladores de sinalização
  - Pilhas de protocolos
- Realiza processamento genérico de chamadas
- Inclui um Multiplexador de Conexões (RFC 1483 - LLC - Logical Link Control)

IP/ATM

- Aparece como uma interface de rede padrão do sistema
- Permite até 256 interfaces de rede por interface física
- Suporta os protocolos e especificações:
  - Classical IP over ATM [9,10]
  - Default IP MTU for use over ATM AAL5 [29]
  - ATM Signaling Support for IP over ATM [30]
  - cliente e servidor ATMARP
  - SCSP (Server Cache Synchonization Protocol) [26]
  - cliente NHRP (Next-hop Resolution Protocol)

ATM Sockets

- Os *sockets* ATM são uma extensão da API *sockets* BSD para permitir que aplicações do usuário acessem os serviços nativos do protocolo ATM. Para criar esses novos tipos de *sockets* foi implementada uma nova família de protocolos, isto é, um novo domínio chamado `AF_ATM` e uma nova estrutura para endereçamento chamada `struct sockaddr_atm`, em analogia à outras famílias de sockets como, por exemplo, a família de protocolos Internet, chamada `AF_NET`, que possui a estrutura de endereçamento `struct sockaddr_in`.
- A implementação dos *sockets* ATM, presente na plataforma HARP, está em conformidade com as especificações do grupo intitulado "The Open Group" [27] e com o documento ATM Forum "Native ATM Services" [28].
- Além da conformidade com relação aos documentos acima, a plataforma HARP ainda implementa:

- suporte a PVCs (*Permanent Virtual Connection*)
- LLC com multiplexação/compartilhamento
- Seleção de interface de rede
- SPANS (*Simple Protocol for ATM Network Signalling*) - protocolo de sinalização proprietário da Fore Systems Inc. [14].

Signalling Managers

- Implementa funções de sinalização entre usuário e rede
  - ATM Forum UNI (3.0 e 3.1)
    - suporta tanto SVCs quanto PVCs
    - inclui o protocolo ILMI (*Interim Local Management Interface*)
    - nenhum suporte para conexões multiponto
  - SPANS
    - Suporta tanto SVCs quanto PVCs
    - inclui suporte para CLS (Connectionless Datagram Service)
    - nenhum suporte para multiponto
- Cuidam da criação e liberação de todos os canais virtuais
- Cada interface física possui um administrador de sinalização

Stack Services

- Especifica uma estrutura para ligar dinamicamente módulos de protocolo
- Os provedores de "serviços de pilha" implementam as funções dos protocolos:
  - AAL5, AAL3/4, SSCOP
- Implementa uma interface de passagem de mensagens de baixo *overhead*

- Uma instância de pilha é criada para cada VC como parte do procedimento de estabelecimento do VC.

Device Services

- Implementa todas as funcionalidade para controladores dos seguintes dispositivos:
  - FORE Systems, Inc. (PCA-200E, SBA-200E)
  - Efficient Network, Inc. (ENI-155p)
- Nenhum suporte para qualidade de serviço

3.3 Outras Implementações para ATM em FreeBSD

Nesta seção pretendemos mostrar algumas implementações de serviços nativos para o protocolo ATM. Vale salientar que, apesar das várias implementações e propostas de implementações encontradas na literatura especializada, nem todas se aplicam a sistemas BSD. Com relação ao sistema FreeBSD especificamente, existem no momento duas formas de suporte ao protocolo ATM que já fazem parte do sistema. A primeira delas é a própria plataforma HARP, que acabamos de descrever na seção anterior. A outra, que é apresentada a seguir, intitula-se `natm`, de *native ATM*. Esta última, apesar de bastante interessante, não suporta as placas Fore PCA200EPC que dispomos em nossas máquinas de teste.

Uma outra implementação, testada em sistema FreeBSD, voltada para máquinas de pequeno custo, é discutida no seção 3.3.2 a seguir. Tal implementação poderá ser alvo de testes em trabalhos futuros.

Procuramos mostrar nessa seção as implementações de ATM nativo apenas para o sistema FreeBSD e que já tivessem sido testadas. Propostas não testadas ou idealizadas para outros sistemas operacionais não são mencionadas. Uma outra proposta de implementação de serviços ATM para sistemas BSD pode ser encontrada em [33].

3.3.1 `natm` - Integrando ATM a sistemas BSD

Este trabalho de Charles D. Cranor [53] apresenta o projeto e implementação de uma camada de software ATM para sistemas operacionais baseados no sistema BSD. Foram feitas mudanças mínimas no sistema operacional. O `natm` permite o tráfego de pacotes IP sobre ATM assim como tráfego em modo ATM nativo.

A integração do software ATM ao sistema BSD foi realizada através da implementação de uma camada de rede ATM, independente de dispositivo, além de um controlador de dispositivo para placas ATM. As únicas placas suportadas são aquelas baseadas no *chipset* Efficient Networks Midway ATM.

A implementação já faz parte dos sistemas FreeBSD, NetBSD e OpenBSD.

A camada de rede ATM desenvolvida suporta IP sobre PVCs e também operações de entrada/saída, através de *sockets*, em circuitos virtuais ATM, ou seja, operações sobre ATM nativo. Podem ser utilizados os serviços AAL0 ou AAL5. Não há nenhum outro protocolo entre a camada de *sockets* e a rede. As chamadas de *sockets* são as mesmas dos sistemas BSD. A diferença é uma nova família de

endereçamento, `AF_NATM`, e de uma nova estrutura de endereçamento ATM denominada `sockaddr_natm`, de modo bastante similar ao que é usado pela plataforma HARP.

Testes de transmissão unidirecional utilizando UDP/IP sobre ATM entre duas máquinas (FreeBSD 2.1.5, 200 MHz, 64MB de memória) conectadas por canais de 155 Mbps em uma chave ATM resultam em taxas de até 133 Mbps [53].

3.3.2 Uma Camada de Transporte para ATM Nativo

Esse trabalho descreve os detalhes de projeto, implementação e testes de desempenho de uma camada de transporte ATM [31].

A. Jain e S. Keshav discutem as questões relativas ao transporte do modelo de camada ATM proposto em [32] para o sistema FreeBSD.

A camada foi idealizada para minimizar o *overhead* de transmissões sobre a rede ATM, prover garantias de qualidade serviço por circuito virtual e tirar proveito das funcionalidades da camada AAL5. Permite transmitir dados através de um serviço confiável ou não-confiável com controle de fluxo pelo algoritmo *leaky-bucket*. Estes serviços podem ser combinados para criar um serviço de transporte personalizado para um circuito virtual.

O trabalho inova por explorar tais serviços garantindo alto desempenho e confiabilidade utilizando computadores de arquitetura de baixo custo.

Na publicação do trabalho são descritos os mecanismos e o suporte do sistema operacional necessários para prover os serviços nativos do protocolo ATM.

A implementação tem grande portabilidade e foi testada nos sistemas operacionais DOS, IRIX, Solaris, FreeBSD e Brazil[6]; nas plataformas PC, SPARC e Silicon Graphics. A implementação para FreeBSD e resultados dos testes de desempenho são descritos em [31]. Tais testes foram realizados em máquinas bastante simples: dois PCs equipados com processadores Intel 80486 de 66 MHz conectados a uma chave ATM Fore ASX-200. Na época em que os testes foram realizados, tais máquinas representavam tecnologia com dois anos de atraso. Apesar disso, eles conseguem bons resultados, comparáveis até a resultados obtidos em sofisticadas estações de trabalho.

Nos testes, uma das máquinas agia como transmissor e a outra como receptor. O transmissor enviava uma pequena mensagem e o receptor, ao receber a mensagem, respondia ao transmissor. O atraso médio imposto pela rede, isto é, tempo de ida da mensagem do transmissor até o receptor, foi de 650 μs.

Medidas de taxa de transmissão utilizando serviço confiável, resultaram em um pico de 54 Mbps com mensagens de 80 Kbytes. Utilizando-se serviço não confiável (*best-effort*) obteve-se picos de velocidade de 56 Mbps com mensagens de 40 Kbytes. Os autores identificam que o processador do receptor é o 'gargalo' do sistema e estimam que, utilizando CPUs da classe Pentium, alcançariam um pico de 100 Mbps.

---

[6] Brazil é uma versão para pesquisa do sistema operacional Plan 9 dos Laboratórios Bell [32].

3.4 Previsão de Desempenho

A Figura 3.5 ilustra de forma simples o nosso ambiente de rede a ser analisado. A plataforma HARP permite-nos utilizar qualquer aplicação baseada nos protocolos TCP/UDP e IP sobre ATM através da interface *sockets* BSD tradicional de forma transparente.

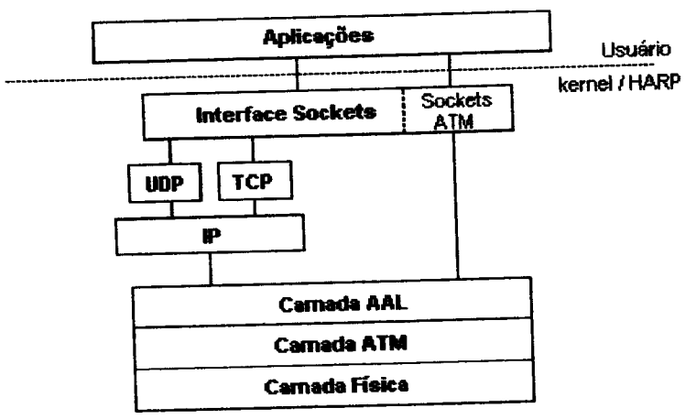

Figura 3.5 Pilha de protocolos TCP/ATM e ATM nativo

Por outro lado, este caminho, TCP/IP/ATM, é carregado de redundâncias pois a pilha de protocolos TCP/IP foi colocada no topo de outra pilha de protocolos, ou seja, a pilha de protocolos ATM. Além disso, as aplicações não são capazes de utilizar as funções nativas do protocolo ATM tais como controle de fluxo e qualidade de serviço.

A plataforma HARP também fornece uma forma de ter acesso às camadas ATM sem passar pelo processamento dos protocolos de transporte e rede usuais. Através de uma extensão dos *sockets* BSD tradicionais, permite acessar os serviços ATM (com as restrições já comentadas na seção 3.2). É o que denominamos de ATM nativo, implementado através dos chamados *sockets* ATM.

Afim de que possamos interpretar melhor os resultados experimentais, a largura de banda máxima disponível para cada camada de protocolo será calculada aqui. Apenas o *overhead* imposto pelos cabeçalhos e *trailers* de cada protocolo foram considerados. Outras fontes de *overhead* tais como o gerenciamento de memória e comunicação entre processos não foram tratadas, pois ultrapassam os limites do presente trabalho.

A largura de banda de transmissão disponível a uma camada é deduzida a partir da relação entre quantidade de dados no PDU (*payload*) e tamanho total do PDU, em bytes, de acordo com o formato de cada pacote.

Como já mencionado na seção 2.5, na especificação do equipamento utilizado, todas as interfaces de rede instaladas em nossas máquinas se comunicam com a chave ATM através do protocolo de meio físico SONET STS-3c/OC-3.

O protocolo SONET [40, 46] é baseado na transmissão de quadros (*frames*) de dados. Um quadro é enviado a cada 125 μs, resultando em um total de 8.000 quadros por segundo. O tamanho do quadro depende da taxa de dados. Um quadro OC-1 tem 810 bytes, resultando em uma taxa de dados de 51,84 Mbps. Quadros para taxas de dados maiores são múltiplos desse tamanho básico. Quadros OC-3 ou OC-3c têm 2430 (810*3) bytes, distribuídos em 9 linhas de 270 bytes cada, resultando em 155,52 Mbps. Quadros OC-12 ou OC-12c têm 9720 (810*12) bytes (9 linhas de 1080 bytes cada), resultando em 622,08 Mbps. O protocolo SONET tem três tipos de *overhead*:

- Section *overhead* - SOH (27 bytes)
- Line *overhead* - LOH (54 bytes)
- Path *overhead* - POH (9 bytes)

Isto consome 90 dos 2430 bytes do quadro OC-3c. Assim, a largura de banda disponível para a camada ATM, localizada logo acima da camada SONET, é:

$$BW_{ATM} = \frac{payload\_quadro}{tamanho\_quadro} * taxa\_bits = \frac{2430-90}{2430} * 155{,}52 = 149{,}76 \quad Mbps \quad (1)$$

A camada ATM também introduz *overhead* já que cada célula contém 53 bytes dos quais 5 são para o cabeçalho. Assim a largura de banda que sobra para a camada AAL, que vem logo acima da camada ATM, é:

$$BW_{AAL} = \frac{payload\_celula}{tamanho\_celula} * BW_{ATM} = \frac{48}{53} * 149{,}76 = 135{,}63 \quad Mbps \quad (2)$$

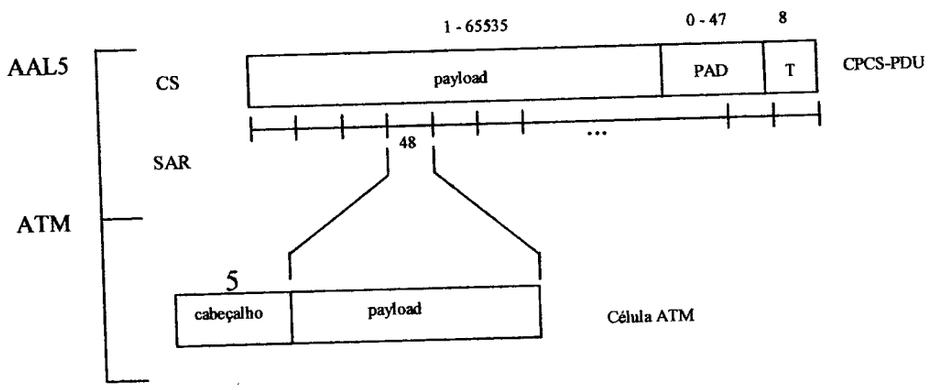

Figura 3.6 Relação entre as camadas ATM e AAL5

Analisando a Figura 3.6, vemos que há duas fontes de *overhead* na camada AAL5: o *trailer* que é acrescentado ao final do quadro e o preenchimento para que o PDU da sub-camada de convergência (CS) tenha um tamanho múltiplo de 48, tal que possa ser convertido em células, já que o tamanho do payload é variável (máximo de

64 Kbytes). O *trailer* tem 8 bytes, indica o tamanho do PDU e contém um *checksum* que verifica o PDU como um todo.

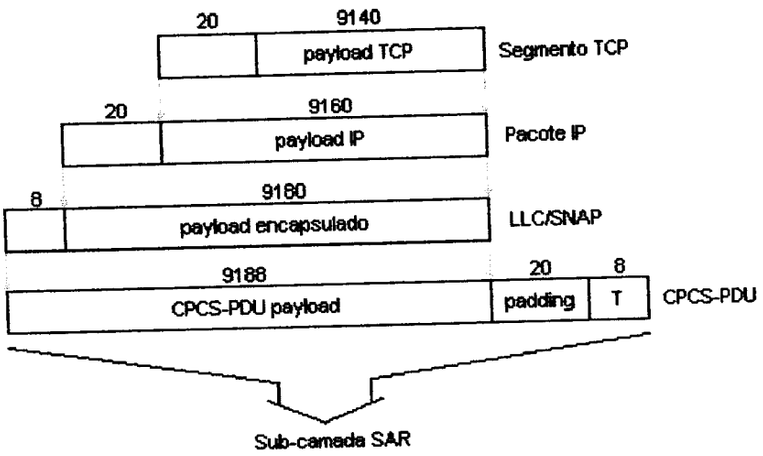

Figura 3.7 Encapsulamento LLC/SNAP para pacotes IP sobre AAL5

Temos que calcular ainda o *overhead* imposto pelos protocolos TCP sobre IP e UDP sobre IP. Tais protocolos estão logo acima da sub-camada de convergência (CS), como mostra a Figura 3.7 acima.

A implementação do protocolo IP sobre ATM na plataforma HARP, segue o padrão *Classical IP* (sinalização Q.2931) sobre AAL5 [9,10]. O padrão *Classical IP* usa o encapsulamento IEEE 802.2 LLC/SNAP e o MTU padrão de 9180 bytes [29].

Logo, a largura de banda de transmissão disponível ao nível de rede, ou seja, à camada IP, é dada por:

$$BW_{IP} = \frac{IP\_MTU}{CPCS\_PDU} * BW_{AAL} = \frac{9180}{9216} * 135{,}63 = 135{,}10 \quad Mbps \quad (3)$$

Assim, uma aplicação que utiliza IP via TCP, pode contar com uma largura de banda máxima de:

$$BW_{AP\_TCP} = \frac{IP\_MTU - IP\_header - TCP\_header}{IP\_MTU} * BW_{IP} = 134{,}51 \quad Mbps \quad (4)$$

Se a aplicação estiver utilizando UDP no lugar de TCP, então, no lugar do cabeçalho de 20 bytes, teremos o cabeçalho UDP de 8 bytes. Portanto, uma aplicação que utiliza IP via UDP, pode contar com uma largura de banda máxima de:

$$BW_{AP\_UDP} = \frac{IP\_MTU - IP\_header - UDP\_header}{IP\_MTU} * BW_{IP} = 134{,}69 \quad Mbps \quad (5)$$

Observe que os valores máximos para TCP e UDP são bastante próximos. No entanto, não podemos esperar o mesmo desempenho real de ambos os protocolos devido ao fluxo maior de informações de controle que o protocolo TCP manipula.

Podemos concluir com estes simples cálculos que qualquer aplicação que esteja transmitindo dados utilizando os protocolos TCP/UDP/IP via rede ATM, pode esperar uma largura de banda de aproximadamente 134 Mbps ou menor. Isto corresponde a 86% da taxa de transmissão real do meio físico.

O fator de *overhead* de protocolo deixa as aplicações que utilizam ATM nativo em vantagem desde que, teoricamente, menos *overhead* é introduzido, comparado com aplicações utilizando TCP/IP sobre ATM. No entanto, a presença de outras fontes de *overhead* tais como o sistema operacional, disco rígido, movimentação de dados na memória como descritas em [24,34 e 35] também têm impacto significativo na taxa de transmissão final, fazendo com que as aplicações alcancem taxas menores que o máximo teórico disponível de 134 Mbps.

3.5 Testes de Desempenho em Chaves ATM Fore

Nesta seção pretendemos relacionar alguns resultados encontrados em trabalhos semelhantes que também utilizaram equipamentos ATM do mesmo fabricante que os disponíveis em nosso laboratório, ou seja, Fore Systems Inc.

No levantamento bibliográfico foram encontramos vários estudos que realizaram medições de taxa de transmissão considerando os ambientes "ATM nativo" e "TCP/IP sobre ATM". As plataformas de hardware utilizadas nesses estudos são bastante diferentes das utilizadas em nossos experimentos. Na maioria dos casos, as plataformas de testes são formadas por estações Sun ou Silicon Graphics (SGI) com sistemas operacionais Sun OS ou Solaris. Vem daí uma das motivações de nosso estudo, já que nossas máquinas utilizam hardware de baixo custo e um sistema operacional de código aberto. Pretendemos verificar o desempenho de nossas máquinas executando testes semelhantes.

Todos esses estudos utilizam a API Fore para *sockets* ATM nos testes de transmissão em ambiente ATM nativo. Trata-se de uma API baseada na camada AAL, permitindo a criação de aplicações sem a pilha de protocolos TCP/IP e sem os mecanismos de controle de *buffer* da camada de *socket*. Assim, por estar mais próxima da interface de rede, esta API possibilita o acesso a parâmetros e funções específicos do protocolo ATM que não podem ser acessados usando a implementação de *sockets* BSD padrão. Ela fornece funções para estabelecimento e liberação de conexões, envio e recebimento de dados.

No estabelecimento de uma conexão, a aplicação deve especificar o tipo da conexão (simplex, duplex ou multicast), o tipo de AAL a ser utilizado na transmissão

dos dados (AAL 3/4, AAL5 ou AAL0) e a qualidade de serviço requisitada (peak and mean bandwidth). Mensagens de 1 a 4096 bytes podem ser transmitidas em CPCS-PDUs que são segmentados pela sub-camada SAR e passados para a camada ATM. No lado receptor, as células ATM são reagrupadas e direcionadas à aplicação que espera a mensagem.

É importante deixar claro que esta API Fore para *sockets* ATM não está disponível para sistemas FreeBSD. Por isso não a utilizamos. Mesmo assim, apresentamos alguns experimentos que a utilizam para que tenhamos uma base de comparação para os resultados obtidos em nossos testes.

As Tabelas 3.2 e 3.3, a seguir, mostram um resumo das taxas máximas obtidas em quatro estudos que consideram os ambientes TCP/ATM e ATM nativo. Nas tabelas, a primeira linha contém o nome dos autores do trabalho, juntamente com as respectivas referências bibliográficas. A primeira coluna aponta as características mais importantes dos sistemas utilizados. As linhas que indicam o tamanho do *buffer* de *socket* e o tamanho do *buffer* de dados (oitava e nona linhas), mostram os valores com os quais se obteve taxa de transmissão máxima.

Luckenbach *et al.* [36] realizam testes em uma rede constituída de uma chave ATM Fore ASX-100 e várias estações SPARCstation10 conectadas à chave por interfaces de 100 Mbps utilizando sinalização TAXI. O objetivo principal do trabalho foi comparar placas adaptadoras SBus SBA-100 e SBA-200. Seus experimentos sobre ATM Nativo mostram o fenômeno denominado *saw tooth* (dente de serra) - uma seqüência periódica de subidas e descidas no gráfico da taxa de transmissão com relação ao tamanho da mensagem sendo transmitida - observado quando se aumenta (em pequenos acréscimos) o tamanho do PDU da sub-camada CPCS. Observa-se que os picos se distanciam de 48 bytes um do outro. Isto se deve à

---

[7] O encapsulamento utilizado pelo *Fore IP* não utiliza o cabeçalho LLC/SNAP, de oito bytes, como acontece no encapsulamento *Classical IP*. Assim, sobram mais 8 bytes para dados. Enquanto *Classical IP* utiliza sinalização Q.2931, *Fore IP* utiliza a sinalização SPANS [14].

IFSC-USP SERVIÇO DE BIBLIOTECA INFORMAÇÃO

influência do preenchimento (*padding*) do CPCS-PDU (Figura 2.4). Os picos no gráfico são observados quando nenhum preenchimento é realizado, isto é, quando o tamanho do CPCS-PDU é 8 bytes menor que um múltiplo de 48 bytes (8 bytes são usados para o *trailer*, veja Figura 3.7). Com placas SBA-200, conexão unidirecional utilizando AAL5 e um MTU de 4096 bytes (o máximo permitido pelo driver utilizado), foi obtida uma taxa de transmissão de 83 Mbps, Tabela 3.3 a seguir. Não há nenhuma indicação no trabalho sobre qual ferramenta foi utilizada para as medições.

Tabela 3.2 Taxas de transmissão máximas no ambiente TCP/ATM

	Luckenbach *et al.* [36]	Fouquet *et al.* [37]	Matijasevic *et al.* [39]	Andrikopoulos *et al.* [21]
Máquinas	Sparcstation10	Sparcstation10	Sparcstation10	Sparcstation20
Sistema Operacional	SunOS 4.1.3 (Solaris 1.0)	SunOS 4.1.3 (Solaris 1.0)	SunOS 4.1.3 (Solaris 1.0)	Solaris 2.4
Chave ATM	ASX-100	ASX-100	ASX-100	ASX-200
Placa ATM	SBA-200	SBA-200	SBA-200	SBA-200
Interface Física	TAXI (100 Mbps)	TAXI (100 Mbps)	TAXI (100 Mbps)	OC-3 (155 Mbps)
Benchmark	ttcp	Netperf	Netperf	Netperf
Buffer de Socket	52428 bytes	8192 bytes	52428 bytes	18296 bytes
Buffer de Dados	?	4096 bytes	65536 bytes	64036 bytes
Throughput	**59 Mbps**	**21 Mbps**	**49 Mbps**	**94 Mbps**

As medidas no ambiente TCP/ATM (Fore IP[7]) foram feitas com o *benchmark* de domínio público 'ttcp' (*Test TCP(UDP) Performance*), uma ferramenta para testar o desempenho dos protocolos TCP ou UDP sobre IP.

Os testes mostraram que a taxa de transmissão aumenta conforme aumenta o tamanho dos *buffers* de *socket*. A taxa máxima foi alcançada quando ambos os lados,

[7] O encapsulamento utilizado pelo *Fore IP* não utiliza o cabeçalho LLC/SNAP, de oito bytes, como acontece no encapsulamento *Classical IP*. Assim, sobram mais 8 bytes para dados. Enquanto *Classical IP* utiliza sinalização Q.2931, *Fore IP* utiliza a sinalização SPANS [14].

IFSC-USP SERVIÇO DE BIBLIOTECA E INFORMAÇÃO

receptor e transmissor, foram configurados ao máximo permitido para os *buffers* de *socket*, 52.428 bytes, Tabela 3.2.

Fouquet *et al.* [37], em uma plataforma de teste idêntica, utilizam o *benchmarck*, também de domínio público, Netperf [38]. Eles executaram testes em *loopback* - tanto na máquina transmissora quanto na receptora - e definiram os resultados como sendo "limites de arquitetura", que são livres dos gargalos da rede.

Tabela 3.3 Taxas de transmissão máximas no ambiente ATM nativo

	Luckenbach *et al.* [36]	Fouquet *et al.* [37]	Matijasevic *et al.* [39]
Máquinas	Sparcstation10	Sparcstation10	Sparcstation10
Sistema Operacional	SunOS 4.1.3 (Solaris 1.0)	SunOS 4.1.3 (Solaris 1.0)	SunOS 4.1.3 (Solaris 1.0)
Chave ATM	ASX-100	ASX-100	ASX-100
Placa ATM	SBA-200	SBA-200	SBA-200
Interface Física	TAXI (100 Mbps)	TAXI (100 Mbps)	TAXI (100 Mbps)
AAL	AAL5	AAL5	AAL5
Benchmark	?	Netperf	Netperf
Buffer de Dados	4096	4096 bytes	4092 bytes
Throughput	**83 Mbps**	**62 Mbps**	**84 Mbps**

Em seus experimentos para ATM nativo, Fouquet *et al.*, utilizam mensagens de 1, 5, 10 e 15 MB e *buffer* de dados de até 4096 bytes (o máximo permitido pelo *driver*). Nos experimentos com TCP sobre ATM foi utilizado um *buffer* de *socket* de 8 KB e *buffer* de dados de até 4096 bytes. Comparando os resultados obtidos de máquinas conectadas sem passar pela chave ATM, eles deduzem que a chave não introduz nenhum atraso observável. Ocorreram perdas de células em transmissões memória-memória mas não em transmissões disco-disco. Eles acreditam que os atrasos decorrentes dos acessos a disco podem ter evitado que o *buffer* da chave ATM ficasse lotado. Seus resultados também mostram que a taxa de transmissão aumenta com o aumento do tamanho do *buffer* de dados (quantidade de dados que é passada, por chamada, da aplicação para a camada de *sockets*) , com nenhuma

diferença observável para mensagens de 1 MB, 5 MB e 10 MB. No entanto, mensagens de 15 MB introduziram atrasos adicionais devido ao *swapping* de memória virtual. Os resultados dos dois ambientes, TCP/ATM e ATM nativo, confirmam o fenômeno "*saw tooth*" observado por Luckenback *et al.* durante a alteração do tamanho do *buffer* de dados. Para TCP/ATM, transmissões de memória a memória alcançaram uma taxa máxima de 21 Mbps (28% do limite de arquitetura que era de 75 Mbps), e 15 Mbps (52% do limite de arquitetura de 29 Mbps) para transmissões de disco a disco. Tais resultados, tão baixos, quando comparados com os obtidos por Luckenbach *et al.*, podem ser conseqüência da escolha de *buffers* de *socket* muito pequenos. Para ATM nativo, transferências memória-memória alcançaram uma taxa máxima de 62 Mbps (83% do limite de 75 Mbps) e transferências disco-disco, 25 Mbps (86% do limite de 29 Mbps).

Matijasevic *et al.* [39] também utilizam o *benchmark* Netperf para realizar testes nos ambientes TCP/IP/ATM, UDP/IP/ATM e ATM nativo entre duas Sparcstations10 com placas Sbus SBA-200 ligadas a uma chave ATM Fore ASX-100. A interface de meio físico utiliza sinalização TAXI a 100 Mbps. Os experimentos sobre TCP/ATM (TCP/IP/ATM) foram realizados com vários tamanhos de *buffer* de dados e *buffer* de *sockets*. Uma taxa máxima de 49 Mbps foi observada quando os *buffers* de *socket* tinham seu tamanho máximo permitido (52428 bytes). O *throughput* aumenta com o tamanho do *buffer* de dados mas alcança um máximo rapidamente em 4 KB e fica nesse nível. Os experimentos com ATM nativo também mostram que a taxa de transmissão aumenta com o tamanho do *buffer* de dados. Uma taxa máxima de 84,71 é observada utilizando-se o tamanho máximo para o *buffer* de dados, isto é, 4092 bytes.

Andrikopoulos *et al.* [21], também utiliza Netperf para examinar as taxas de transmissão do ambiente de rede TCP/ATM sobre várias plataformas usando placas SBA-200 e uma chave ATM Fore ASX-200 com conexões OC-3 a 155 Mbps. Ele também realizou testes de *loopback* e obteve os chamados limites de arquitetura. Utilizando *Fore IP* entre duas estações Sparcstation20, ele obtém uma taxa máxima de 93,91 Mbps, o que significa 70% do limite teórico do sistema que era de 134,6 Mbps. O pico em 93,91 Mbps ocorre quando o tamanho dos *buffers* de *socket* é duas vezes o tamanho do MSS (o tamanho do *buffer* de *socket* sofreu incrementos de 1 MSS nas medições).

Capítulo 4

Substituição das APIs e Testes Iniciais

Antes de partirmos diretamente aos resultados propriamente ditos, discutimos neste capítulo as funcionalidades e detalhes de implementação do *benchmark* utilizado nas medições: o Netperf. Além disso, esclarecemos as adaptações feitas em partes de seu código para substituir a *API Fore* pela API de *sockets* ATM da plataforma HARP. Por uma questão de simplicidade, esta última será chamada de API HARP.

Testes iniciais foram realizados no intuito de obter uma estimativa sobre os efeitos provenientes dos limites de arquitetura, hardware e sistema operacional, em cada máquina. Medidas de taxa de transmissão foram realizadas em *loopback* para cada uma das máquinas envolvidas nos testes.Além disso, também foram realizados testes simples para se estimar o tempo de resposta na rede ATM.

4.1 Netperf

4.1.1 Funcionalidade

Netperf é um *benchmark* que pode ser usado para medir vários aspectos do desempenho de uma rede. Seu foco principal está na medição do desempenho de redes TCP(UDP)/IP utilizando aplicações implementadas com a interface *sockets* BSD. Netperf permite medir o desempenho de transmissões na forma de grandes volumes de dados, que denominaremos transmissão unidirecional (STREAM) ou transmissões de dados com confirmação (*request/response*).

Além disso, há testes opcionais para DLPI, Unix Domain Sockets, ATM e HiPPI-LLA da HP. Com exceção dos testes ATM, nenhum dos outros testes nos interessam, já que não foram utilizados no trabalho.

Em nossos testes, realizamos medições unidirecionais de TCP e UDP sobre ATM e transmissões em modo ATM nativo. Como as rotinas originais do aplicativo Netperf, que implementam os testes em ATM nativo, foram desenvolvidas com base na API ATM da Fore, a qual não foi encontrada para o sistema operacional FreeBSD, fizemos a substituição pela API HARP.

O sistema operacional FreeBSD suporta a implementação de aplicações para ATM nativo de duas formas. A primeira é através da plataforma HARP que provê uma API para sockets ATM bastante semelhante à API de *sockets* BSD tradicional. A segunda é através de uma camada de protocolo de transporte para ATM nativo denominada `natm`, comentada na seção 3.3.1, que também acompanha o sistema FreeBSD atualmente. Porém, `natm` não suporta as interfaces de rede ATM Fore PCA-200EPC que utilizamos em nossas máquinas.

Assim, adotamos a API de *sockets* ATM da plataforma HARP e fomos obrigados a modificar as rotinas de Netperf que implementam os testes ATM. O Anexo A mostra o código, já adaptado, de tais rotinas.

O aplicativo Netperf é mantido, e informalmente apoiado, pelo grupo Networking Performance Team da HP, Information Networks Division, e tem seu código integralmente liberado para modificações e melhoramentos.

4.1.2 Implementação e Funcionamento

O aplicativo Netperf é implementado sobre o conceito básico de aplicações cliente-servidor. Há dois programas executáveis, o `netperf` e o `netserver`. Normalmente, o usuário ativa manualmente o cliente `netperf` com as opções desejadas em linha de comando. Há também *scripts* prontos que acompanham os arquivos de código fonte. O servidor `netserver` pode ser ativado de duas formas: como um processo filho (*child*) do servidor mestre para serviços de rede do sistema operacional (`inetd` no FreeBSD) ou como um processo servidor (*daemon*) independente.

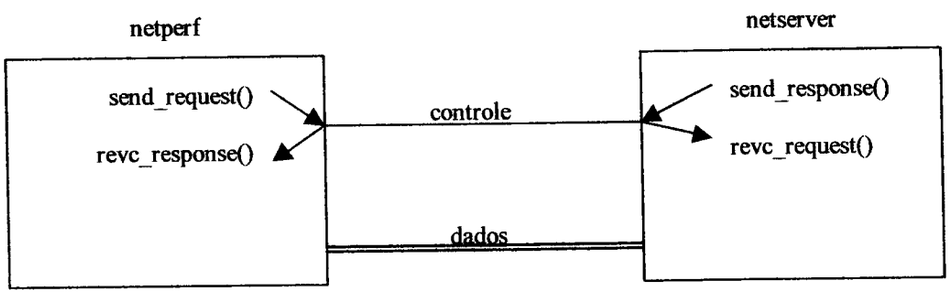

Figura 4.1 Modelo cliente-servidor de Netperf

A Figura 4.1 ilustra o modelo sobre o qual Netperf é implementado. Quando `netperf` é executado, uma conexão de controle é estabelecida com o servidor `netserver` na máquina remota. Esta conexão será utilizada para transmitir informações de configuração relativas ao teste requisitado, além dos resultados de tal teste, que retornam da máquina remota para a máquina que executou o cliente `netperf`. Independente do tipo de teste requisitado pelo usuário, esta conexão de controle é sempre uma conexão TCP utilizando os *sockets* BSD tradicionais.

Uma vez que a conexão de controle tenha sido estabelecida e as informações de configuração do teste tenham sido trocadas (tamanho de *buffers*, tamanho da mensagem, medir ou não medir tempo de CPU, etc), uma conexão separada é criada para a transmissão de dados utilizando-se a API e os protocolos específicos para o teste requisitado. Quando o teste termina, os resultados são enviados ao cliente, pela conexão de controle, que os imprime. O console é o padrão para a impressão dos resultados na máquina cliente. Se a opção -d (*debug*) fizer parte da linha de comandos no cliente, a saída será mais detalhada, mostrando informações sobre tudo o que acontece desde o estabelecimento da conexão de controle, passando pela transmissão dos dados até a liberação das conexões após o término do teste. Além disso, na máquina remota, que recebe os dados, informações semelhantes a respeito do desenrolar do teste são gravadas em /tmp/netpef.debug.

As funções send_request(), recv_response(), send_response() e recv_request(), mostradas na Figura 4.1, são utilizadas para envio e recebimento de estruturas de dados que contêm informações de controle e configuração para um dado teste. A Figura 4.2, abaixo, mostra um esboço das estruturas de dados principais: netperf_request{} e netperf_response{}.

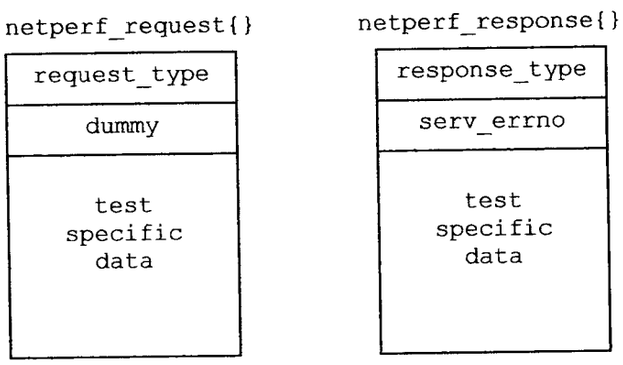

Figura 4.2 Estruturas de dados que carregam informações pela conexão de controle.

Tais estruturas são utilizadas por todos os testes pois abrigam outras estruturas de dados específicas para o controle de um tipo de teste. O campo designado "`test specific data`" acomoda as estruturas específicas para cada teste. O formato e significado destas estruturas depende do tipo do teste. Como exemplo, veja no Anexo B, o arquivo nettest_atm.h que define as estruturas específicas para os testes FORE_STREAM e FORE_RR, isto é, teste unidirecional de dados em ATM nativo e teste com confirmação em ATM nativo, respectivamente.

A implementação do aplicativo Netperf, como um todo, é bastante organizada e modularizada. Para cada tipo de teste há um arquivo onde são implementadas as rotinas de transferência de dados no modo *stream* (unidirecional) e *request-response* (bidirecional). A Figura 4.3 a seguir mostra uma lista com todos os arquivos que fazem parte do pacote `netperf2.1pl3.tgz` [38].

```
ACKNWLDGMNTS
COPYRIGHT
README
Release_Notes
arr_script
hist.h
makefile
makefile.win32
netlib.c
netlib.h
netperf.c
netperf.man
netperf.ps
netserver.c
netserver.man
netsh.c
netsh.h
nettest_bsd.c
nettest_bsd.h
nettest_dlpi.c
nettest_dlpi.h
nettest_fore.c
nettest_fore.h
nettest_hippi.c
nettest_hippi.h
nettest_ipv6.c
nettest_ipv6.h
nettest_unix.c
nettest_unix.h
nettest_xti.c
nettest_xti.h
snapshot_script
tcp_range_script
tcp_rr_script
tcp_stream_script
udp_rr_script
udp_stream_script
```

Figura 4.3 Lista dos arquivos do pacote netperf-2.1pl3.tgz.

Por exemplo, o arquivo `nettest_bsd.c`, implementa os testes TCP/UDP. Suas principais funções são: `send_tcp_stream()`, `recv_tcp_stream()`, `send_tcp_rr()`, `recv_tcp_rr()`, `send_udp_stream()`, `recv_udp_stream()`, `send_udp_rr()` e `recv_udp_rr()`.

As funções que têm o nome começando por 'send' (send_tcp_stream, send_fore_stream, send_udp_stream, etc) são executadas pelo cliente (`netperf`) e as que têm o nome começando por 'recv' são executadas pelo servidor (`netserver`).

IFSC-USP SERVIÇO DE BIBLIOTECA INFORMAÇÃO

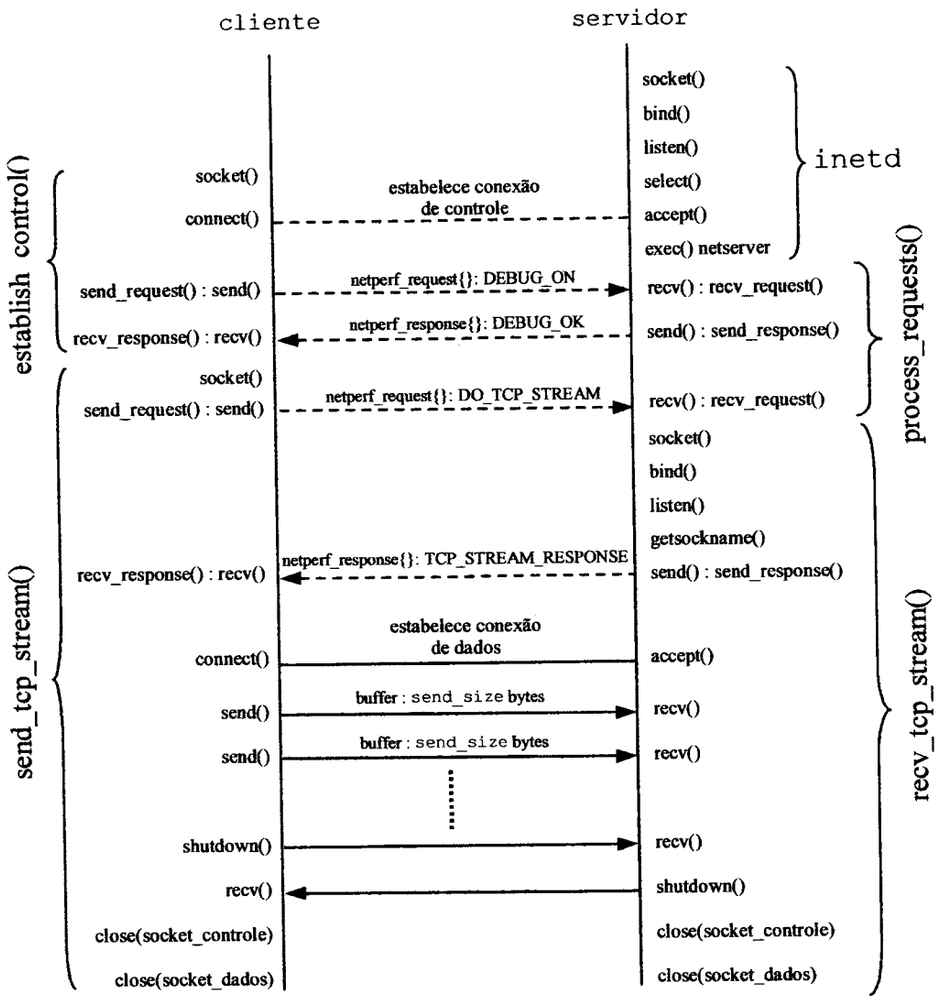

Figura 4.4 Funcionamento da transmissão de dados do teste TCP_STREAM de Neperf.

A Figura 4.4 mostra, detalhadamente, o funcionamento de um teste TCP_STREAM até o nível das chamadas de sistema. Esta figura considera o caso em que o programa servidor (`netserver`) é disparado por `inetd`. As chamadas de sistema, tais como socket(), bind(), connect(), accept(), send(), recv(), etc, foram colocadas próximas às linhas denominadas `cliente` (netperf) e `servidor` (netserver) e as funções do programa, que as chamam, logo ao lado. Assim, por exemplo, quando no lado cliente, Netperf executa a função establish_control(), esta

utiliza as chamadas de sistema socket() e connect() para criar uma conexão de controle. Em seguida são chamadas as funções send_request() e recv_response() que utilizam as chamadas de sistema send() e recv(), respectivamente, para testar a conexão de controle recém estabelecida. As estruturas de dados trocadas pela conexão de controle entre cliente e servidor têm sempre o formato mostrado na Figura 4.2, sendo que o campo `test_especific_data` tem um tamanho máximo de 64 bytes para carregar estruturas específicas de cada tipo de teste.

Note que este tipo de teste, TCP_STREAM, realiza uma transmissão de dados unidirecional. O tamanho da mensagem (`send_size`) que é transmitida de cada vez é determinado pelo usuário em linha de comando no momento em que executa o programa `netperf`. Este tamanho da mensagem, algumas vezes, é chamado de *buffer* de dados da aplicação.

A função `allocate_buffer_ring()`, do aplicativo Netperf, cria uma lista encadeada circular onde cada nó aponta para um bloco de dados de tamanho `send_size`. A quantidade de nós na lista também pode ser determinada em linha de comando. Este anel de *buffers* é criado tanto do lado cliente quanto do lado servidor.

O teste TCP_RR, realiza transmissões de mensagens em ambas as direções, chamadas de requisição e resposta. O tamanho das mensagens de requisição e de resposta pode ser determinado em linha de comando. A diferença deste teste é que existem agora dois anéis de *buffers* de cada lado da transmissão, um para recepção das mensagens e outro para mensagens sendo enviadas.

4.1.3 Substituição das APIs

O arquivo `nettest_fore.c`, original da implementação de Netperf, define as funções para a transmissão de dados utilizando-se da API Fore para *sockets* ATM. As funções são: `send_fore_stream()`, `recv_fore_stream()`, `send_fore_rr()` e `recv_fore_rr()`. O Apêncide A contém o código fonte das duas primeiras. Modificamos estas funções de forma que as chamadas de funções pertencentes à *API Fore* foram substituídas por chamadas de funções correspondentes pertencentes à API para sockets ATM da plataforma HARP. A Tabela 4.1, abaixo, mostra alguns exemplos de correspondência entre funções das duas APIs.

Tabela 4.1 Diferenças entre API Fore e API HARP.

API Fore	API HARP
atm_open()	socket()
atm_gethostbyname()	*nenhuma equivalente*
atm_bind()	bind()
atm_connect()	connect()
atm_send()	write(), send()
atm_recv()	read(), recv()
atm_listem()	listem()
atm_accept()	accept()
atm_close()	close()

Note que a API HARP mantém os nomes das funções da API de *sockets* BSD tradicional. Na verdade, são as mesmas funções utilizando uma família de protocolos diferente (`AF_ATM`), conforme já discutimos na seção 3.2.3.

Mais detalhes sobre tais estruturas podem ser encontrados na documentação que acompanha o código fonte da plataforma HARP e, principalmente, no arquivo `atm.h` presente em qualquer sistema FreeBSD que possua a plataforma HARP instalada. Outras informações também podem ser encontradas em [27,28].

Todas as variáveis e funções relacionadas à qualidade de serviço que eram implementadas na API Fore foram removidas do código do arquivo `nettest_fore.c`. Algumas opções de linha de comando, também relativas a qualidade de serviço, deixaram de fazer sentido e foram retiradas da implementação. Isto se fez necessário pois a API para sockets ATM da plataforma HARP não tem suporte para qualidade de serviço.

Finalmente, os arquivos que eram denominados `nettest_fore.c` e `nettest_fore.h` passaram a se chamar `nettest_atmn.c` e `nettest_atmn.h`, respectivamente, depois de adaptados.

4.2 Testes em *loopback*

Com a finalidade de estimar o poder de processamento de cada máquina em particular, realizamos testes em *loopback*. Esses testes são os resultados da execução de `netperf` sem nenhuma opção na linha de comando. Assim, como a máquina destino não é especificada na linha de comando, o programa realiza o teste padrão: TCP_STREAM, sobre a interface `lo0` durante 10 segundos. Como exemplo, tomemos um teste realizado na máquina ATM5. Abaixo, vemos a linha de comando e os resultados do teste.

```
atm5# /usr/local/netperf/netperf
TCP STREAM TEST to localhost
Recv   Send    Send
Socket Socket  Message  Elapsed
Size   Size    Size     Time     Throughput
bytes  bytes   bytes    secs.    10^6bits/sec

16384  16384   16384    10.01     273.53
```

Realizamos esse teste 20 vezes em cada máquina. A Tabela 4.2, a seguir, mostra os resultados finais, ou seja, a taxa de transmissão média, em Mbps, e seu desvio padrão.

Tabela 4.2 Resultados dos testes em loopback.

	Média (Mbps)	Desvio
ATM1	101	1
ATM2	304.2	0.9
ATM5	265	6
ATM7	303.8	0.7

Analisando a Tabela 4.2, notamos, de imediato, que o desempenho da máquina ATM1 é muito ruim quando comparado ao desempenho das outras. De fato, isso era o esperado, já que esta máquina é a que tem a configuração de hardware mais precária (veja Tabela 2.4). Além disso, nota-se que os desempenhos das máquinas ATM2 e ATM7 são praticamente iguais. Este também era um resultado esperado, pois estas máquinas possuem configurações idênticas de hardware. O que não era de fato esperado é o desempenho da máquina ATM5. Apesar de não ser um desempenho ruim, é muito flutuante, com um desvio padrão de 6 Mbps.

4.3 Tempos de Resposta

Para determinar os tempos de resposta de uma máquina a outra na rede ATM, utilizamos a ferramenta `ping` do sistema operacional. `ping` é um programa que envia uma requisição de resposta chamada de `ECHO_REQUEST`, para uma máquina específica e espera pela resposta, conhecida como `ECHO_RESPONSE`. Estas mensagens, requisição e resposta, fazem parte do protocolo ICMP (*Internet Control Message Protocol*) [4,25]. ICMP é um protocolo que se localiza na mesma camada que o protocolo IP. Mensagens ICMP são encapsuladas em pacotes IP. Uma característica interessante de se enviar requisições de resposta através do protocolo ICMP é que a operação deste protocolo não depende de protocolos de camadas superiores, tais como TCP ou UDP. A Figura 4.5, a seguir, ilustra o formato de uma mensagem ICMP.

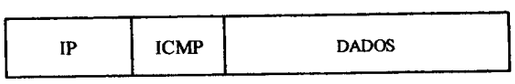

Figura 4.5 Mensagem ICMP

A quantidade de dados na mensagem poder ser controlada por uma opção de linha de comando de `ping` e tem um limite de 8184 bytes no sistema operacional FreeBSD. O cabeçalho IP possui 20 bytes e o cabeçalho ICMP tem sua estrutura implementada em *unions* e, por isso, seu tamanho depende do tipo da mensagem. No entanto, em todos os testes utilizamos o tamanho padrão de 56 bytes para as mensagens ICMP.

Vejamos, então, um exemplo da execução de um comando `ping` entre as máquinas ATM2 e ATM7. Executando, em ATM2, o comando:

```
atm2# ping -c 20 172.30.1.71
```

obtivemos a seguinte saída:

```
PING 172.30.1.71 (172.30.1.71): 56 data bytes
64 bytes from 172.30.1.71: icmp_seq=0 ttl=255 time=12.171 ms
64 bytes from 172.30.1.71: icmp_seq=1 ttl=255 time=0.280 ms
64 bytes from 172.30.1.71: icmp_seq=2 ttl=255 time=0.279 ms
64 bytes from 172.30.1.71: icmp_seq=3 ttl=255 time=0.290 ms
64 bytes from 172.30.1.71: icmp_seq=4 ttl=255 time=0.274 ms
64 bytes from 172.30.1.71: icmp_seq=5 ttl=255 time=0.285 ms
64 bytes from 172.30.1.71: icmp_seq=6 ttl=255 time=0.278 ms
64 bytes from 172.30.1.71: icmp_seq=7 ttl=255 time=0.278 ms
64 bytes from 172.30.1.71: icmp_seq=8 ttl=255 time=0.296 ms
64 bytes from 172.30.1.71: icmp_seq=9 ttl=255 time=0.285 ms
64 bytes from 172.30.1.71: icmp_seq=10 ttl=255 time=0.288 ms
64 bytes from 172.30.1.71: icmp_seq=11 ttl=255 time=0.287 ms
64 bytes from 172.30.1.71: icmp_seq=12 ttl=255 time=0.282 ms
64 bytes from 172.30.1.71: icmp_seq=13 ttl=255 time=0.289 ms
64 bytes from 172.30.1.71: icmp_seq=14 ttl=255 time=0.296 ms
64 bytes from 172.30.1.71: icmp_seq=15 ttl=255 time=0.295 ms
64 bytes from 172.30.1.71: icmp_seq=16 ttl=255 time=0.290 ms
64 bytes from 172.30.1.71: icmp_seq=17 ttl=255 time=0.357 ms
64 bytes from 172.30.1.71: icmp_seq=18 ttl=255 time=0.293 ms
64 bytes from 172.30.1.71: icmp_seq=19 ttl=255 time=0.286 ms

--- 172.30.1.71 ping statistics ---
20 packets transmitted, 20 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.274/0.884/12.171/2.589 ms
```

Perceba que, depois de enviadas as 20 mensagens de requisição de resposta, são apresentadas estatísticas que contêm os tempos de resposta (*round-trip-time*) máximo (*max*), mínimo (*min*) e médio (*avg*), juntamente com o desvio padrão (*stddev*). Além disso, note que o tempo de resposta para a primeira mensagem é muito maior que os tempos de resposta para as mensagens seguintes. Isso acontece porque, antes de começar a transmitir as mensagens ICMP, a chave ATM deve estabelecer um circuito virtual (SVC) do tipo Classical IP ligando as duas máquinas. Assim, as mensagens seguintes serão enviadas todas pelo mesmo SVC. Este fica ativo enquanto estiver sendo utilizado. Porém, depois de ficar ocioso por mais de 15 minutos, o SVC é desfeito.

A opção -c determina quantas mensagens ICMP deverão ser enviadas. Em nossos testes adotamos uma quantidade de 20 mensagens. Julgamos esse número

suficiente para se estimar um tempo médio de resposta em uma rede sem tráfego. Para minimizar o desvio padrão da média dos tempos de resposta, executamos o `ping` sobre um SVC pré estabelecido. Assim, eliminamos o tempo gasto no estabelecimento da conexão.

A Tabela 4.3 abaixo, contém os tempos de resposta médios para 20 mensagens ICMP enviadas. A primeira coluna indica a direção das requisições de resposta. Por exemplo, atm1→atm2, significa que o comando ping foi executado em ATM1. Isto é, ATM1 enviava `ECHO_REQUESTs` e ATM2 respondia com `ECHO_RESPONSEs`.

Tabela 4.3 Tempos de resposta para requisições ICMP na rede ATM.

Direção	Tempo (ms)	Desvio
atm1→atm2	0.385	0.02
atm1→atm5	0.423	0.02
atm1→atm7	0.393	0.02
atm2→atm1	0.349	0.02
atm2→atm5	0.323	0.02
atm2→atm7	0.294	0.02
atm5→atm1	0.398	0.04
atm5→atm2	0.337	0.04
atm5→atm7	0.358	0.07
atm7→atm1	0.362	0.03
atm7→atm2	0.298	0.02
atm7→atm5	0.329	0.02

É importante lembrar que, até então, todas as máquinas estavam conectadas à chave ATM IFSC-USP exatamente como mostrado na Figura 2.6, isto é, todas ligadas às portas do módulo A dessa chave.

Com a finalidade de investigar o tempo gasto no estabelecimento dos SVC's, realizamos testes adicionais entre as máquinas ATM2 e ATM7. Escolhemos estas duas máquinas por apresentarem a mesma configuração de hardware.

Ainda com ATM2 e ATM7 conectadas ao módulo A da chave IFSC-USP, executamos o comando `ping` 10 vezes de ATM2 para ATM7. Porém, antes de cada execução do comando `ping`, o SVC entre as duas máquinas era apagado através do comando:

```
atm delete svc hfa0 <vpi> <vci>,
```

onde hfa0 é o nome da interface física de rede ATM.

Assim, forçávamos que um novo SVC fosse estabelecido a cada vez que executávamos o comando `ping`. Na verdade, estávamos interessados apenas no tempo de resposta para a primeira mensagem ICMP.

O mesmo procedimento foi adotado com a máquina ATM2 conectada à porta B2 da outra chave ATM, ou seja, à chave LIE-ATM. Os tempos de resposta, em milisegundos , para a primeira mensagem de cada um dos 10 comandos `ping` executados, são apresentados na Tabela 4.4, a seguir.

Tabela 4.4 Tempos (em ms) para a primeira mensagem ICMP do comando ping.

Comando	atm2 → atm7 (atm2 na chave IFSC-USP)	atm2 → atm7 (atm2 na chave LIE-ATM)
1	12.171	28.018
2	16.269	21.105
3	19.205	25.328
4	17.332	22.473
5	10.675	19.5
6	14.646	23.881
7	14.75	23.662
8	16.516	22.765
9	10.016	28.551
10	18.593	23.05
Média	15 ms	24 ms
Desvio Padrão	3 ms	3 ms

Utilizando ferramentas do software de controle da chave ATM - *ForeThought* - pudemos facilmente verificar que no primeiro caso, isto é, ATM2 e ATM7 conectadas às portas A1 e A2, respectivamente, na chave IFSC-USP, é criado um canal virtual entre estas portas para a transmissão das mensagens ICMP. Já no segundo caso, isto é, ATM2 conectada à porta B2 da chave LIE-ATM, são necessários dois canais virtuais para estabelecer uma conexão entre as duas máquinas. Um canal entre as portas B1 e B2 na chave LIE-ATM e outro canal entre as portas A2 e D1 na chave IFSC-USP. Isso explica os tempos maiores para a primeira mensagem ICMP no segundo caso. O procedimento adotado foi o mesmo, ou seja, antes de cada comando `ping`, o SVC existente entre as máquinas era desfeito manualmente.

Capítulo 5

Medidas de Taxas de Transmissão

As medidas de taxas de transmissão (*throughput*) foram realizadas tomando-se como parâmetros o tamanho da mensagem sendo transmitida e os tamanhos dos *buffers* dos *sockets*. Quando mencionamos "tamanho da mensagem", estamos nos referindo à quantidade de dados da aplicação que é passada para a camada de *socket* através das chamadas de sistema send(), sendto() ou write(). Isso vale também para o caminho inverso dos dados no lado receptor.

Consideramos transmissões de dados nos seguintes ambientes de rede:

- TCP/IP sobre ATM, utilizando os testes TCP de Netperf direcionados à interface lógica de rede ATM *Classical IP (ni0)*;
- UDP/IP sobre ATM, utilizando os testes UDP de Netperf direcionados à interface lógica de rede ATM *Classical IP (ni0)*;
- ATM nativo, utilizando os testes ATM de Netperf com as funções adaptadas.

Todos os testes consistem de transmissões em uma única direção. Esse tipo de teste é denominado, pelo manual de Netperf [38], de teste de transmissão de dados em 'rajadas' ou simplesmente STREAM.

Durante um determinado teste, as máquinas e a chave ATM, não transmitiam nada além dos dados relativos àquele teste, eliminando assim, tanto quanto possível, qualquer tipo de interferência sobre o resultado da taxa de transmissão.

Todas as medidas de taxas de transmissão foram realizadas da máquina ATM2 para a máquina ATM7 por possuírem as mesmas configurações de software e hardware.

5.1 TCP/IP sobre ATM

De acordo com vários estudos [29,35,41,42,45], o que influencia, de maneira significativa, o desempenho de transmissão do protocolo TCP sobre ATM, podendo gerar até resultados surpreendentemente inesperados, é a combinação dos seguintes itens:

1. o tamanho do *buffer* de saída do *socket* do transmissor;
2. o tamanho do *buffer* de entrada do *socket* do receptor;
3. o tamanho da mensagem a ser transmitida;
4. o modo como os dados da aplicação (mensagem) são colocados no *buffer* de transmissão do protocolo TCP;
5. o MTU da interface ATM;
6. a prevenção de SWS (*Silly Window Syndrome*) pelo algoritmo de Nagle [35];
7. o algoritmo, no lado receptor, para adiamento de reconhecimentos de pacotes (*delayed ACKs)* [35].

Os dois primeiros podem ser configurados pela aplicação. Nos sistemas BSD, tais parâmetros podem ser modificados utilizando-se a chamada de sistema

setsockopt() [25]. O terceiro é a quantidade de dados que a aplicação passa para a camada de *socket*, e vice-versa, a cada chamada de sistema para transmitir/receber dados. O MTU da interface de rede ATM, adotado na implementação da plataforma HARP, é de 9180 bytes para *Classical IP* sobre ATM, como ilustrado na Figura 3.7.

Silly Window Syndrome (SWS) [47,35], é um fenômeno caracterizado por uma situação em que um padrão uniforme de incrementos na janela de transmissão do protocolo TCP resulta em pequenos segmentos de dados sendo transmitidos. O envio de pequenos segmentos de dados diminui a taxa de transmissão do TCP porque os cabeçalhos do TCP e do IP consomem largura de banda. Para evitar SWS, ambos, transmissor e receptor, devem implementar o algoritmo de prevenção [47]. No lado receptor, deve-se evitar avançar a borda direita da janela de transmissão em incrementos pequenos quando se recebe pequenos blocos de dados. No lado transmissor, deve-se evitar o envio de pequenos segmentos de dados ainda que o receptor tenha espaço disponível para aceitá-los.

Para utilizar melhor os recursos da rede, o protocolo TCP tenta guardar no *buffer*, segmentos pequenos quando comparados com os cabeçalhos TCP e IP. No entanto, para evitar *deadlock*, o protocolo TCP não deve guardar no *buffer* um segmento que precisa ser liberado imediatamente. O algoritmo de Nagle [49] fornece uma solução simples para o dilema: se há dados ainda não reconhecidos (*unacknowledged*), isto é, a conexão não está ociosa, o protocolo TCP guarda no *buffer* todos os dados até que possa enviar um segmento do tamanho de um MSS ou até que todos os blocos de dados em consideração sejam reconhecidos.

O protocolo TCP, no lado receptor, pode aumentar a taxa de transmissão, reduzir o processamento do protocolo de ambos os lados e gerar menos tráfego utilizando reconhecimentos atrasados (*delayed ACKs*). O protocolo TCP receptor

implementa tais reconhecimentos atrasados gerando menos de um reconhecimento (ACK) por segmento. Por exemplo, enviando um reconhecimento somente depois de ter recebido dois segmentos inteiros.

No entanto, certas combinações dos tamanhos do *buffer* de transmissão e do *buffer* de recepção, podem causar uma situação cíclica de espera onde o transmissor espera que o receptor envie um reconhecimento antes de enviar mais dados e o receptor, por sua vez, espera que o transmissor envie mais dados antes de enviar o próximo reconhecimento. Esta situação permanece até que o temporizador usado para enviar os reconhecimentos atrasados (*delayed ACK's*) expira e o receptor pode, então, enviar um reconhecimento, quebrando assim a situação de espera mútua, mas por pouco tempo. A situação se repete constantemente de tal forma que o transmissor espera por um tempo desnecessário para enviar dados, causando uma queda dramática na taxa de transmissão.

O MTU utilizado pela rede ATM e a má escolha dos tamanhos dos *buffers* de transmissão e de recepção são as principais causas desse comportamento anômalo.

Com a finalidade de verificar o comportamento da rede ATM *Classical IP*, em nosso laboratório, em tais situações, executamos os testes seguintes variando o tamanho da mensagem e os tamanhos dos *sockets* de transmissão e recepção.

Em todos os gráficos, por medida de simplicidade, utilizamos os nomes abreviados `send_sock` e `recv_sock` para indicar, respectivamente, o tamanho do *buffer* de transmissão e o tamanho do *buffer* de recepção.

A Figura 5.1, a seguir, mostra quatro testes de medida da taxa de transmissão de dados em função do tamanho da mensagem. No gráfico 'a', variamos o tamanho da mensagem de 1024 até 65536, isto é, de 1K até 64K, com incrementos de 1K. Os

buffers de *socket* foram mantidos em 16K, que é o padrão dos sistemas BSD em transmissões TCP.

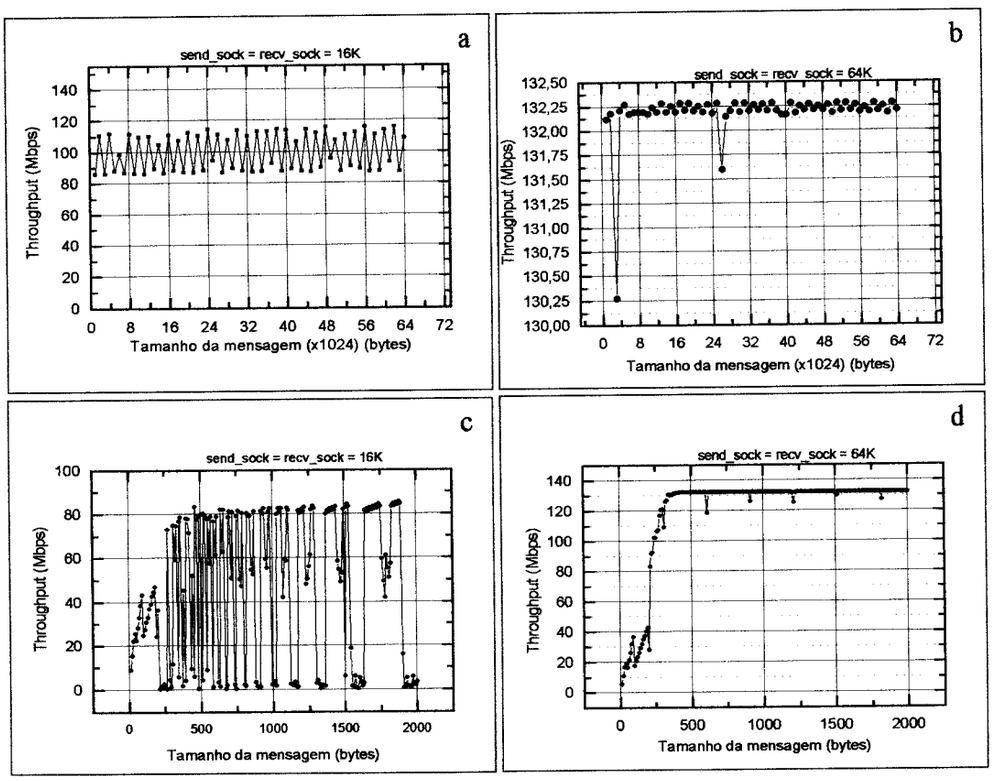

Figura 5.1 *Thrhoughput* em função do tamanho da mensagem para testes TCP/ATM.

Cada ponto do gráfico é resultado de um teste de 10 segundos utilizando-se o programa Netperf. Isso vale para todos os gráficos a seguir. O gráfico 'b', é o resultado do mesmo teste, porém, com os *buffers* de *socket* ajustados para 64K, que é o máximo permitido para *sockets* TCP nos sistemas BSD. Pode-se notar que em 'b', atinge-se uma taxa em torno de 132,25 Mbps sem variações muito grandes. O que já não acontece em 'a', onde a taxa máxima alcançada é de aproximadamente 112 Mbps, com grandes variações. Nos testes 'c' e 'd', o tamanho das mensagem variava de 10 bytes a 2000 bytes com incrementos de 10 bytes. Para o tamanho padrão dos *buffers* de *socket*, 16K, obtivemos um comportamento bastante estranho. A taxa de

transmissão vai praticamente a zero em muitos pontos e a máxima não passa de 85 Mbps. Ainda assim, para mensagens maiores que 250 bytes parece haver um padrão que se repete no gráfico. Apesar do comportamento estranho, não foi constatada nenhuma mensagem de erro nas máquinas que participavam da transmissão, ATM2 e ATM7, nem perda de células ou PDUs na chave. Perceba que o gráfico 'd', resultado do mesmo teste, porém, com *sockets* de 64K, mostra um comportamento bem mais lógico e estável para mensagens maiores que 250 bytes. Tanto em 'c' quanto em 'd', podemos perceber a contribuição do gerenciamento de memória do sistema, através dos *mbufs*, para o *overhead* na transmissão de mensagens pequenas. Repare que existe uma descontinuidade para mensagens de 100 bytes e depois para mensagens de 200 bytes sendo que, acima deste valor, a taxa de transmissão aumenta muito devido ao fato de que para blocos de dados maiores que 208 bytes é usado um *cluster* externo ao *mbuf*. Isto faz com que a troca de dados entre aplicação e *kernel* se restrinja à passagem de ponteiros para tais *clusters*, diminuindo muito o atraso devido ao gerenciamento de memória. Vale lembrar que os *mbufs* também armazenam os cabeçalhos TCP/IP.

Quedas inesperadas na taxa de transmissão são observadas em alguns pontos dos gráficos 'b' e 'd'. Depois de várias hipóteses levantadas na tentativa de explicar tais pontos, acabamos por descobrir algo um tanto inusitado. As quedas são resultado de uma atualização na imagem de fundo da área de trabalho, do ambiente gráfico do sistema, de onde os testes foram disparados, ou seja, uma operação de vídeo que requer recursos do processador e da memória, o que acaba prejudicando o resultado do teste executando no momento em que acontece a atualização. Mais especificamente, a imagem de fundo é gerada e controlada pelo aplicativo *xearth* que desenha a face do globo terrestre iluminada pelo sol de acordo com o relógio da

máquina. A imagem gira de tempos em tempos e o padrão para tais intervalos é de 300 segundos. Este é, aproximadamente, o tempo que separa os pontos de queda nos gráficos 'b' e 'd'.

Tomemos, como exemplo, o gráfico 'd' da Figura 5.1. O gráfico possui 200 pontos sendo, cada um, o resultado de uma execução do programa Netperf durante 10 segundos para um certo tamanho de mensagem, como já foi dito. Por uma questão de praticidade, os dados foram colhidos com a ajuda de um programa que montava a linha de comando de Netperf variando o tamanho das mensagens. Para este gráfico, especificamente, foi utilizado o seguinte programa:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define ti 10
#define tf 2000
#define incr 10
main() {

   long t;

   char lcomando[200];

   char *cmd="/usr/local/netperf/netperf ";
   char *opt="-t TCP_STREAM -H 172.30.1.71 -P 0 -- -s 65536 -S 65536 -m ";
   char tmsg[10];

   for(t=ti; t<=tf; t+=incr) {
      strcpy(lcomando,cmd);
      strcat(lcomando,opt);
      sprintf(tmsg,"%ld",t);
      strcat(lcomando,tmsg);
      system(lcomando);
   }
}
```

Como temos 200 pontos, isto resulta em, aproximadamente, 35 minutos de execução do programa, já que existe um tempo de preparação entre os testes.

O fato é que, quando repetimos a execução do programa acima, fora do ambiente gráfico do sistema, as quedas inesperadas desaparecem por completo. Veja os gráficos a seguir, resultantes da repetição dos testes 'b' e 'd', fora do ambiente gráfico, ou seja, sem que o processo *xearth* estivesse executando.

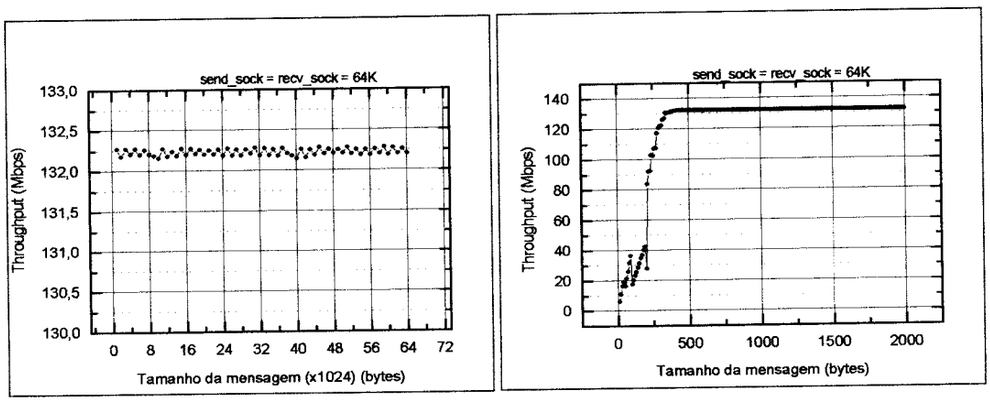

Figura 5.2 Eliminação dos pontos de queda de *throughput* nos testes TCP/ATM.

Perceba que o patamar de 132 Mbps permanece inalterado, porém as quedas periódicas desaparecem.

A maioria dos gráficos mostrados daqui em diante, independente do ambiente de transmissão (TCP/ATM, UDP/ATM ou ATM Nativo) possuem tais pontos de queda. Optamos por não retirar tais pontos dos resultados para ilustrar como outros processos executando na máquina podem afetar o *throughput* de uma transmissão de dados.

Os gráficos das Figuras 5.3, 5.4 e 5.5, a seguir, mostram os resultados dos testes de taxa de transmissão em função dos tamanhos dos *buffers* de *socket*. O tamanho dos *buffers* vão de 16K até 64K, com incrementos de 4K, em cada gráfico. O que muda de um gráfico para outro é o tamanho da mensagem. Repetimos o mesmo teste de variação dos tamanhos dos *buffers* de *socket* para mensagens de 16K até 64K, também incrementando de 4K.

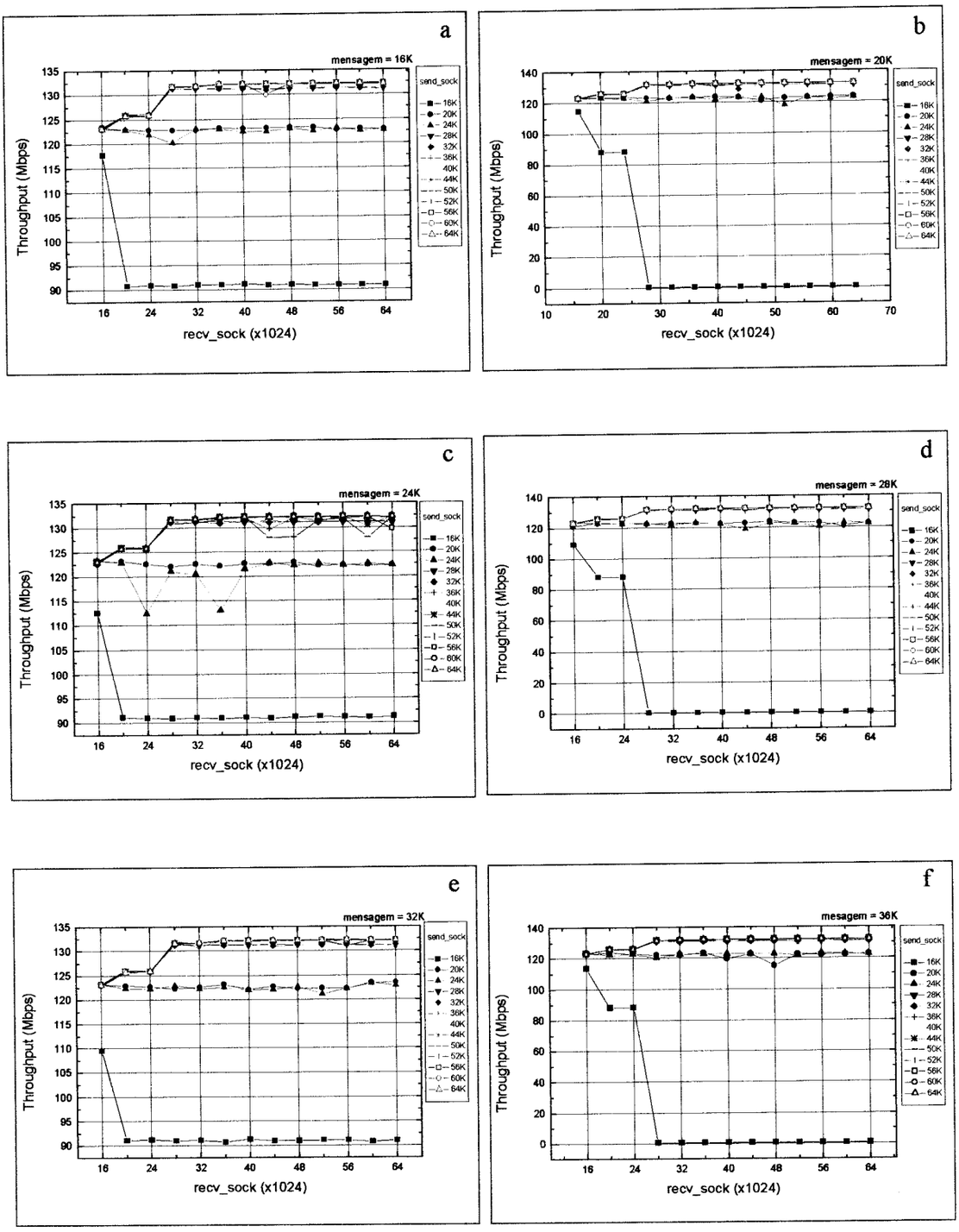

Figura 5.3 *Throughput* em função do tamanho dos *sockets* nos testes TCP/ATM.

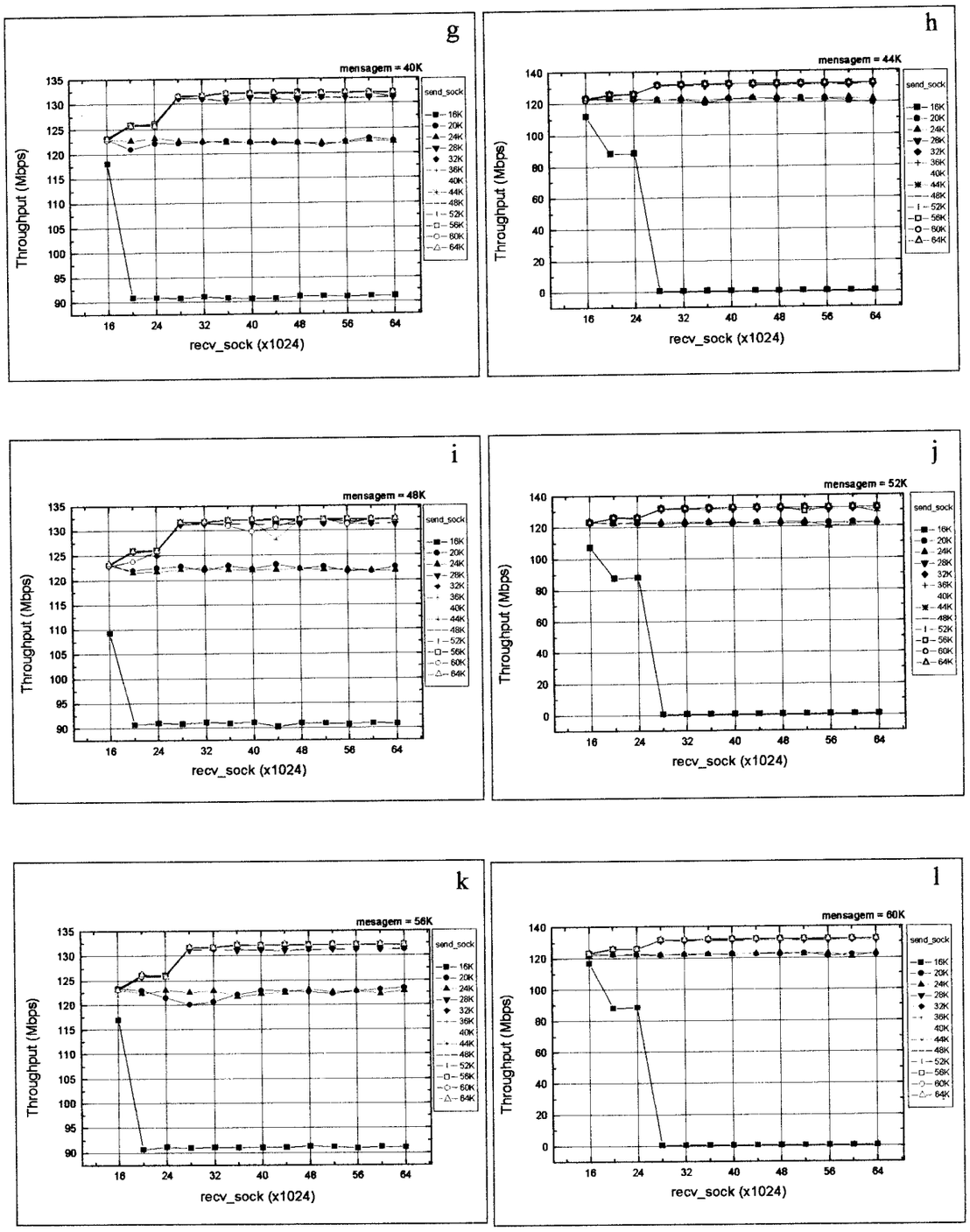

Figura 5.4 *Throughput* em função do tamanho dos *sockets* nos testes TCP/ATM.

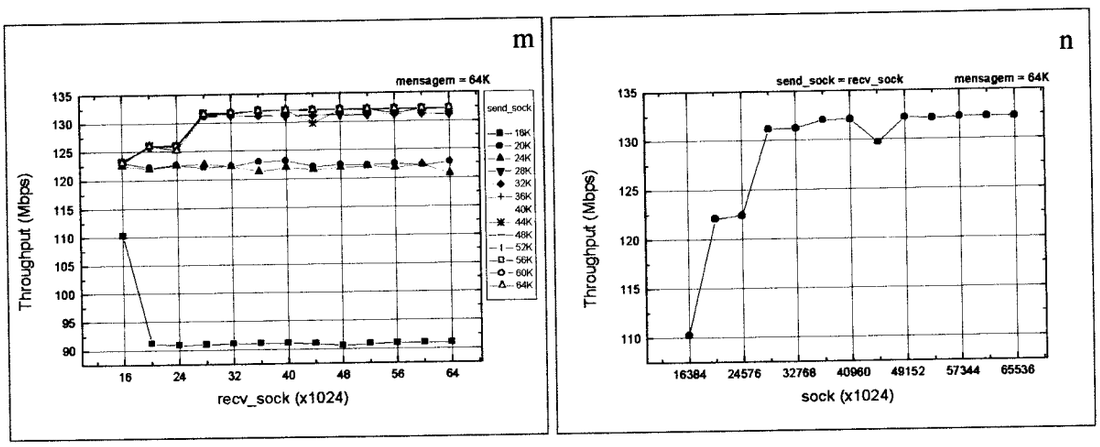

Figura 5.5 *Throughput* em função do tamanho dos *sockets* nos testes TCP/ATM.

Observando os gráficos podemos perceber um padrão que se repete onde o tamanho da mensagem é tamanho múltiplo de 8K (16, 24, 32, 40, 48, 56 e 64) e um outro padrão que se repete para mensagens com tamanho não múltiplo de 8K (20, 28, 36, 44, 52, e 60). Mas em todos eles prevalece uma característica: para *buffer* de transmissão de 16K (`send_sock`) e *buffer* de recepção (`recv_sock`) diferente de 16K, a taxa de transmissão é sempre bem menor comparada às outras combinações de *buffers*, sendo que, quando o tamanho da mensagem não é múltiplo de 8K, a taxa vai praticamente a zero. Mas porque 8K? Cremos que este valor tem relação com o controle dos *buffers* de *sockets* feito pelo sistema operacional. Observe novamente a Tabela 3.1 e note que para o protocolo TCP, as variáveis que regulam o fluxo de dados dos *sockets* são `sb_hiwat=8K` e `sb_lowat=2K`.

Outra característica, presente em todos os gráficos, diz respeito à importância do tamanho do *buffer* de transmissão. Comer [35], em seus estudos sobre o desempenho do protocolo TCP sobre ATM, com relação aos tamanhos dos *buffers* de *socket*, conclui que, para um MSS de 9148 bytes, basta que o tamanho do *buffer* de

transmissão seja 3*MSS para que não haja SWS (*Silly Window Syndrome*), independente do tamanho do *buffer* de recepção.

Em nossa rede *Classical IP*, que utiliza o encapsulamento LLC/SNAP (Figura 3.7), o MSS é de 9140 bytes, pois tal encapsulamento consome mais 8 bytes de cabeçalho. Ainda com esta diferença, vemos em nossos gráficos o efeito de tal relação. Observe que em todos eles, independente do tamanho da mensagem e do tamanho do *buffer* de recepção, a taxa de transmissão chega a um patamar de 133 Mbps sempre que o tamanho do *buffer* de transmissão (`send_sock`) é maior ou igual a 28K. O que confirma os resultados de Comer, já que 28K é maior que 27420 (3*9140).

5.2 UDP/IP sobre ATM

O protocolo UDP não pode tratar segmentação de pacotes, o que significa que o tamanho da mensagem fica limitado pelo menor dos *buffers* de transmissão e recepção. Como UDP é um protocolo não confiável, isto é, não garante que o *datagrama* irá alcançar o seu destino, os gráficos abaixo mostram tanto a taxa do transmissor quanto a taxa do receptor.

No gráfico da Figura 5.6, mostramos a taxa de transmissão alcançada no transmissor, em vermelho, e a taxa alcançada no receptor, geralmente um pouco menor devido à perda de mensagens, em preto. O tamanho do *buffer* de transmissão varia acompanhando o tamanho da mensagem, isto é, de 1024 até 64512 bytes com incrementos de 1024 bytes. O tamanho do *buffer* do receptor foi ajustado ao máximo permitido, 64K.

IFSC-USP SERVIÇO DE BIBLIOTECA INFORMAÇÃO

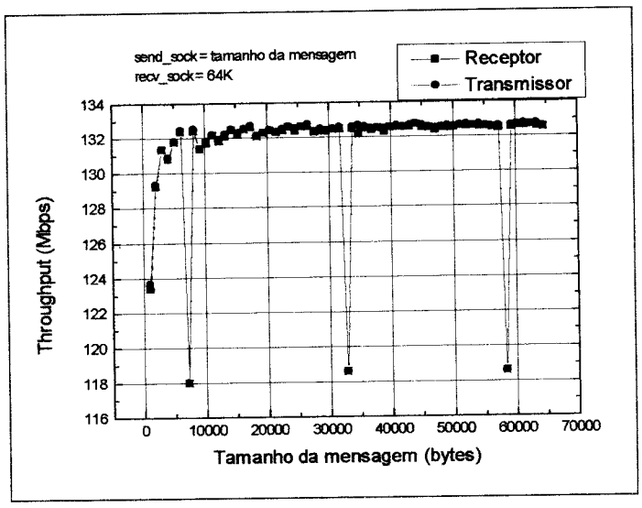

Figura 5.6 *Throughput* em função do tamanho da mensagem nos testes UDP/ATM.

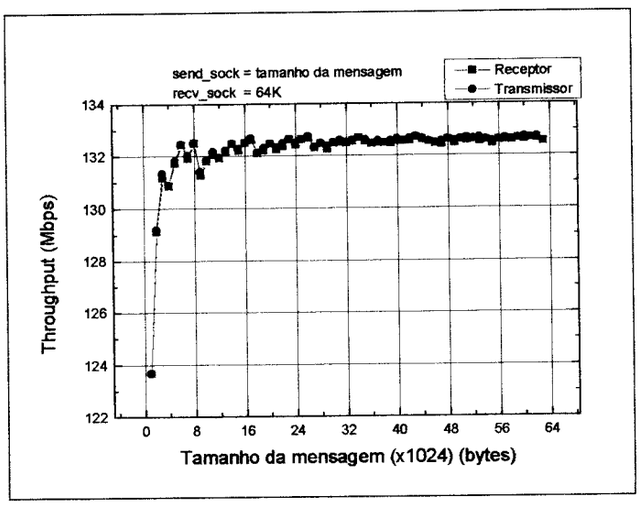

Figura 5.7 Eliminação dos pontos de queda nos testes UDP/ATM.

Podemos notar que, apesar do protocolo UDP não implementar garantia de recebimento dos pacotes, a taxa de transmissão no receptor não difere muito da taxa do transmissor. Notamos também que o desempenho é melhor e mais estável conforme aumenta-se o tamanho da mensagem.

Vale notar, novamente, como o aplicativo *xearth* influencia na taxa de transmissão. Conforme já explicamos anteriormente, esses pontos de quedas 'inesperadas' não dizem respeito ao caráter do nosso estudo.

Observe na Figura 5.7 o mesmo teste anterior realizado fora do ambiente gráfico e perceba como os pontos 'estranhos' desaparecem.

O gráfico 'a' da Figura 5.8, a seguir, mostra o resultado de um teste parecido com aquele mostrado na Figura 5.6. Porém, aqui, o tamanho das mensagens variou de 1024 até 65024 bytes com incrementos de 512. O tamanho dos *buffers* de transmissão e de recepção foram ajustados para 64K. Continuam válidos todos os comentários feitos acima para o gráfico da Figura 5.6. No entanto, é possível enxergar, com uma obervação atenta, a formação do chamado "dente de serra". Analisamos melhor estas formas nos gráficos 'b' e 'c' da Figura 5.8.

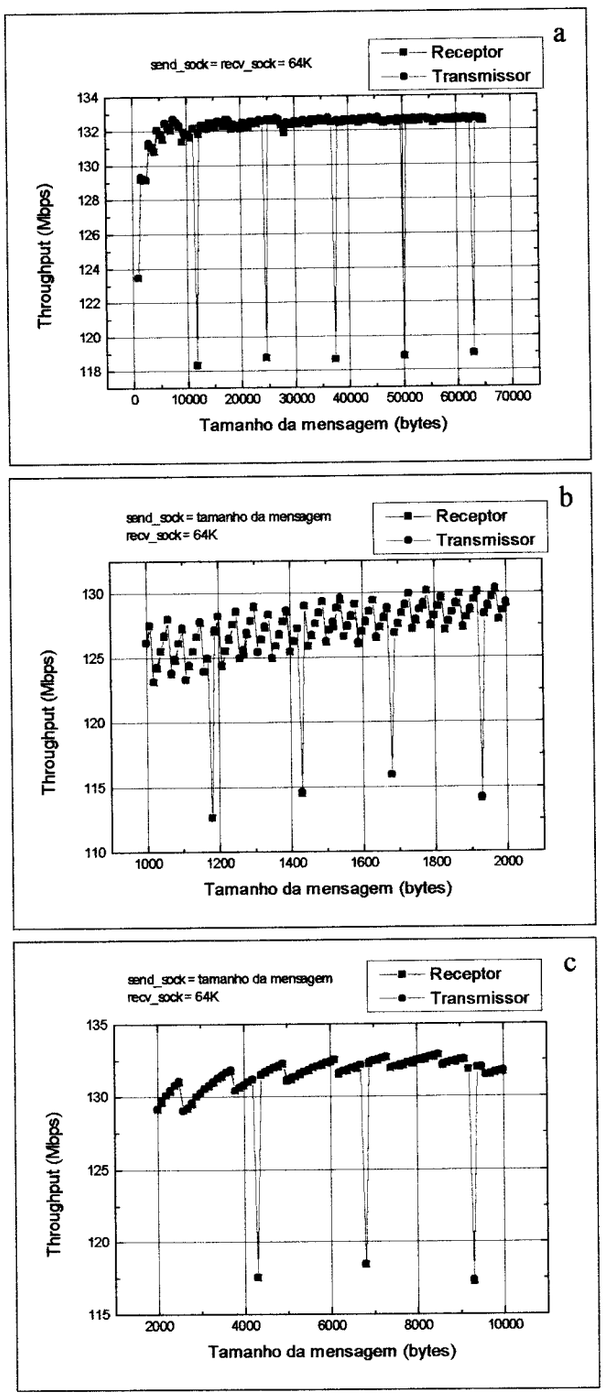

Figura 5.8 *Throughput* em função do tamanho da mensagem nos testes UDP/ATM.

Nos gráficos 'b' e 'c', no intuito de observar melhor as descontinuidades que começam a aparecer no gráfico 'a', focalizamos um intervalo menor de amostragem para os tamanhos das mensagens e aumentamos o número de pontos nesse intervalo. Assim, no gráfico 'b', variamos o tamanho das mensagens de 1000 até 2000 bytes com incrementos de 10 bytes e, no gráfico 'c', variamos o tamanho das mensagens de 2000 até 10000 com incrementos de 100 bytes. Podemos observar, claramente, no gráfico 'c' o fenômeno de dente de serra, ou *saw tooth*, proveniente do processo de preenchimento (*padding*) dos PDUs da subcamada de convergência da camada AAL5, como mostrado na Figura 2.4. A maior taxa ocorre quando não há necessidade de preenchimento, isto é, o CPCS-PDU, já com o datagrama UDP, resulta em um tamanho múltiplo de 48 bytes.

5.3 ATM Nativo

Assim como o protocolo UDP, a API HARP também oferece um serviço não garantido, ou seja, sem confirmações de recebimento. Portanto, também nesta seção, mostramos nos gráficos, tanto as taxas de transmissão alcançadas pelo transmissor quanto as taxas resultantes do lado receptor que, devido às perdas de mensagens, é sempre um pouco menor. Porém, constatamos que uma fração ínfima das mensagens não chega corretamente ao receptor.

Para os testes em ATM nativo, utilizamos também o aplicativo Netperf, porém, com as rotinas específicas modificadas, como descrito na seção 4.1.3. O gráfico da Figura 5.9, a seguir, mostra a taxa de transmissão em função do tamanho da mensagem, variando este último de 1024 até 8192 bytes com incrementos de 1024

bytes. Vale lembrar que o MTU para a interface de rede ATM *Classical IP* é de 9180 bytes não sendo permitido mensagens maiores no modo ATM nativo. Os *buffers* de *socket* foram mantidos em 64K. Podemos notar que a taxa aumenta com o tamanho da mensagem e que quase não há perda de mensagens, já que os pontos se mostram praticamente sobrepostos.

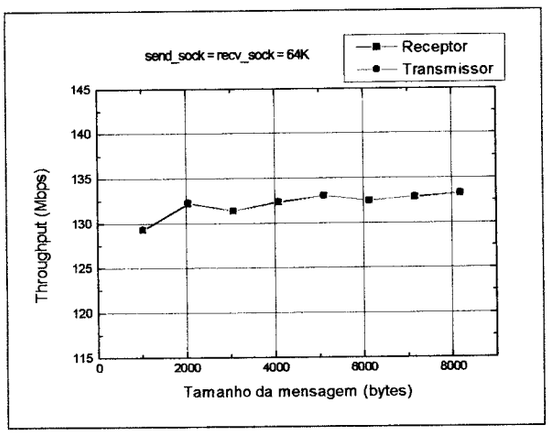

Figura 5.9 *Throughput* em função do tamanho da mensagem nos testes ATM.

Podemos observar melhor o comportamento da taxa de transmissão com relação ao aumento no tamanho das mensagens através dos gráficos da Figura 5.10. Em 'a', variamos o tamanho da mensagem de 1000 até 2500 bytes com incrementos de 10 bytes. Em 'b', variamos o tamanho da mensagem de 1000 até 9100 bytes com incrementos de 100 bytes. Em ambas, os *buffers* de *socket* foram mantidos em 64K.

Mais uma vez, notamos o efeito do preenchimento (*padding*) causando a aparência de 'dente de serra' no gráfico. Além disso, aparecem também os pontos estranhos de queda de desempenho devido ao processo *xearth*. O gráfico 'c', ainda da Figura 5.10, mostra o resultado da repetição do teste 'b', porém, fora do ambiente gráfico do sistema. Novamente, os pontos de queda desaparecem.

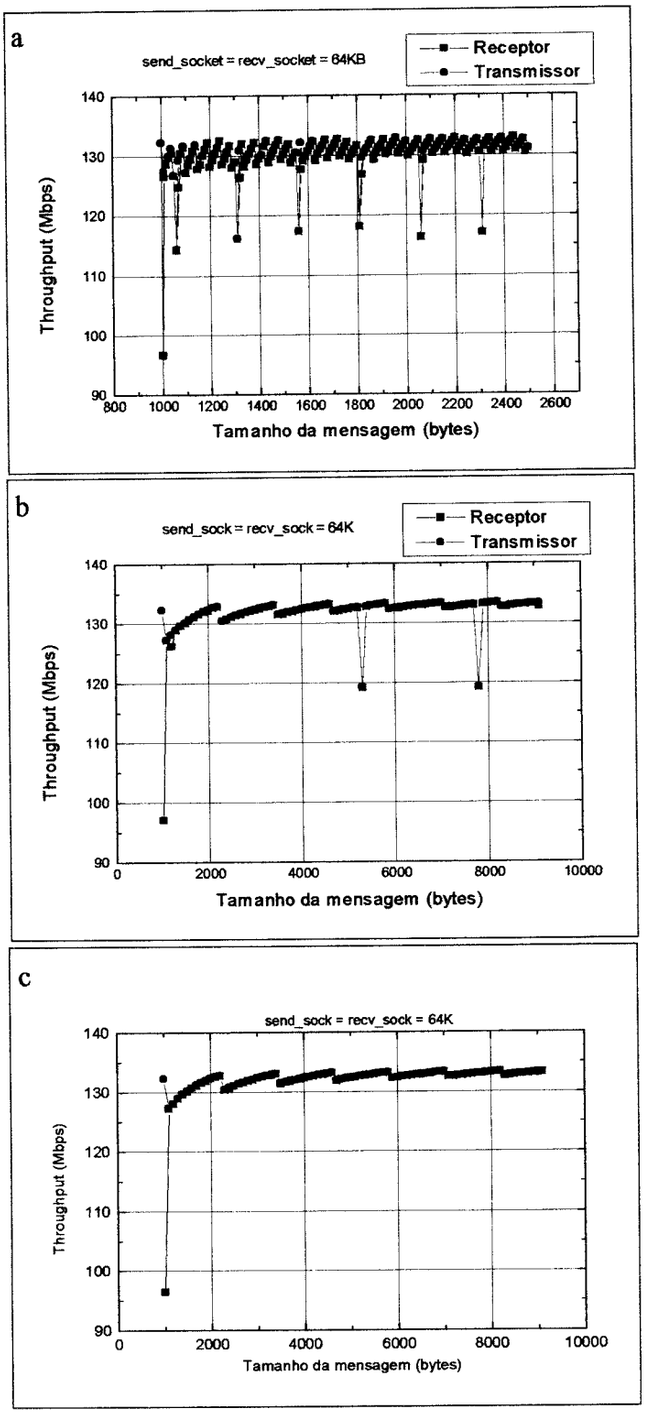

Figura 5.10 *Throughput* em função do tamanho da mensagem nos testes ATM.

Nos testes de variação do tamanho da mensagem em ATM nativo, descobrimos que mensagens pequenas, isto é, mensagens menores que 1KB, geram mensagens de erro no receptor e, não raro, a queda da conexão e conseqüente interrupção do teste. Por exemplo, quando executamos o programa de teste para medir a taxa de transmissão considerando mensagens de 10 até 2000 bytes, com incrementos de 10 bytes, logo no início do teste, ou seja, para mensagens de 10 bytes, o receptor já começa a exibir mensagens de erro em seu console. Observe o programa de testes:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define ti 10
#define tf 2000
#define incr 10
main() {

   long t;

   char lcomando[200];

   char *cmd="/usr/local/netperf/netperf ";
   char *opt="-t FORE_STREAM -H atm7 -P 0 -- -s 65536 -S 65536 -m ";
   char tmsg[10];

   for(t=ti; t<=tf; t+=incr) {
      strcpy(lcomando,cmd);
      strcat(lcomando,opt);
      sprintf(tmsg,"%ld",t);
      strcat(lcomando,tmsg);
      system(lcomando);
   }
}
```

Quando o programa chega a mensagens de 70 bytes, ele é interrompido porque todas as conexões ATM já caíram, inclusive as de controle, ou seja, canais VPI:0/VCI:5 e VPI:0/VCI:16 ! Veja, na Figura 5.11, as mensagens de erro exibidas no console do receptor, desde o início do teste até o momento em que é interrompido. Convém dizer que, para as mensagens de 10 bytes até 70 bytes, a taxa de transmissão

ficou em torno de 30 Mbps no transmissor e em torno de 15 Mbps no receptor. Mais da metade das mensagens se perdiam.

```
Mar  8 19:03:14 atm7 /kernel: unisig: signalling channel SSCF session lost
Mar  8 19:03:14 atm7 /kernel: unisig: signalling channel active
Mar  8 19:03:26 atm7 /kernel: unisig: signalling channel SSCF session lost
Mar  8 19:03:26 atm7 /kernel: unisig: signalling channel active
Mar  8 19:03:38 atm7 /kernel: unisig: signalling channel SSCF session lost
Mar  8 19:03:39 atm7 /kernel: unisig: signalling channel active
Mar  8 19:03:50 atm7 /kernel: unisig: signalling channel SSCF session lost
Mar  8 19:03:51 atm7 /kernel: unisig: signalling channel active
Mar  8 19:03:52 atm7 /kernel: unisig: signalling channel SSCF session lost
sscop badpdu received: vcc=(0,5)
: bfr=0xf0638800 data=0xf0638824 len=4:
Mar  8 19:03:52 atm7 /kernel: sscop_maa_error: intf=hfa0 vpi=0 vci=5 code=U
state=2
sscop badpdu received: vcc=(0,5)
: bfr=0xf0649480 data=0xf06494a4 len=4:
sscop badpdu received: vcc=(0,5)
: bfr=0xf0638800 data=0xf0638824 len=4:
Mar  8 19:04:00 atm7 last message repeated 2 times
Mar  8 19:04:08 atm7 /kernel: unisig: signalling channel active
```

Figura 5.16 Mensagens de erro no console da máquina receptora.

Até mesmo no teste ilustrado pelo gráfico 'a' da Figura 5.10 foram exibidas algumas mensagens como as primeiras linhas da Figura 5.11. Porém o teste não parou e as conexões se mantiveram até o fim. Após o término do teste, executamos alguns comandos de gerenciamento ATM, oferecidos pela plataforma HARP, no receptor e foi possível notar que, para mensagens pequenas há problemas de alocação de *buffers* pequenos. O motivo pode ser a incapacidade de processamento da máquina receptora, já que não descobrimos nenhum limite documentado para transmissão de dados, nem nos manuais da placa ATM, nem na documentação, escassa, da plataforma HARP. Mostramos abaixo, na Figura 5.12, os comandos executados na máquina receptora que nos permitiram chegar essa conclusão. Os comandos mostram estatísticas em vários níveis, isto é, desde a camada física - *phy* -

até o nível do controlador de dispositivo. As linhas de comando encontram-se em negrito.

```
> atm sh stats in hfa0 phy
Fore PCA-200E OC-3c Statistics
Section     Path    Line      Line     Path     Corr   Uncorr
BIP8        BIP8    BIP24     FEBE     FEBE     HCS    HCS
Errs        Errs    Errs      Errs     Errs     Errs   Errs
     0         0       0         0        0        0        0

> atm sh stats in hfa0 dev
Fore PCA-200E Device Statistics
Type 1      Type 1      Type 2      Type 2
Small Buff  Large Buff  Small Buff  Large Buff  Receive     Receive
Alloc Fail  Alloc Fail  Alloc Fail  Alloc Fail  Queue Full  Carrier
       503           0           0           0           0  On

> atm sh stats in hfa0 atm
Fore PCA-200E ATM Layer Statistics
  Cells In   Cells Out   VPI Range  VPI NoConn   VCI Range  VCI NoConn
         0           0           0           0           0       58286

> atm sh stats in hfa0 aal5
Fore PCA-200E AAL 5 Statistics
                        CRC/Len                                   CRC    Proto  PDU
  Cells In   Cells Out  Errs     Drops     PDUs In    PDUs Out   Errs   Errs   Drops
 546797504       34432  849879       0    16647285       17226   39627      0  39629

> atm sh stats in hfa0 driver
Fore PCA-200E Device Driver Statistics
  No  Xmit   Max   Seg           No     No           No    IQ     No    Cmd     No
 VCC Queue   Seg   Not    Seg   DMA    VCC    No   Mbuf  Full    DMA  Queue   DMA
 Out  Full  Size Align    Pad   Out     In  Buff     In    In    Sup   Full   Cmd
   0    16     0     0      0     0      0     0      0  4709      0      0     0
```

Figura 5.12 Estatísticas de uma transmissão mostradas para várias camadas.

Há uma observação interessante a ser feita sobre a formação dos SVCs nos testes de TCP/ATM e ATM nativo. Quando nos referimos a 'teste', vale salientar que estamos nos referindo a uma seqüência de chamadas de Netperf, uma após a outra, com parâmetros de valores diferentes. Para testes no ambiente TCP(UDP)/ATM, é estabelecido um SVC logo na primeira chamada de Netperf. Este SVC permanece até o final do teste, isto é, para todas as transmissões de 10 segundos, é utilizado o mesmo canal virtual. Já nos testes em ambiente ATM nativo, é criado um SVC para cada teste de 10 segundos de Netperf. Tomemos como exemplo o testes da Figura 5.9 Nesse teste, o programa Netperf é executado 8 vezes, para 8 tamanhos de

mensagem diferentes. Se para a primeira chamada de Netperf foi estabelecido o SVC VPI:0/VCI:150, por exemplo, ao final dos 10 segundos de transmissão este canal será desfeito. Na segunda chamada de Netperf o canal VPI:0/VCI:151 será criado e assim por diante.

Também fizemos testes para investigar a dependência da taxa de transmissão com relação ao tamanho dos *buffers* de *socket* no ambiente ATM nativo. As tabelas a seguir mostram os resultados para mensagens de 1KB e para mensagens de 9180 bytes (MTU da interface ATM). O extremo superior esquerdo mostra o tamanho da mensagem em questão em cada tabela.

Observando as tabelas, rapidamente notamos que o *throughput* é o mesmo para todas as combinações de *buffers* de transmissão/recepção que testamos. No entanto, os valores são diferentes para tamanhos de mensagens diferentes, como já havíamos detectado nos gráficos anteriores nesta seção. Para mensagens de 1024 bytes obtém-se uma taxa aproximada de 129 Mbps e para mensagens de 9180 bytes, 133 Mbps. As perdas são mínimas. Os maiores valores encontrados foram de 0,3% das mensagens transmitidas para mensagens de 1024 bytes e de 0,5% para mensagens de 9180 bytes.

Tabela 5.1 *Throughput* em função do tamanho dos *sockets* nos testes ATM.

1K Transmissor	send_sock												
recv_sock	16384	20480	24576	28672	32768	36864	40960	45056	49152	53248	57344	61440	65536
16384	129,3	129,22	129,29	129,29	129,23	129,3	129,3	129,31	129,24	129,3	129,3	129,29	129,21
20480	129,23	129,3	129,3	129,29	129,28	129,24	129,24	129,24	129,3	129,24	129,22	129,23	129,29
24576	129,29	129,24	129,25	129,24	129,22	129,3	129,29	129,22	129,22	129,3	129,3	129,14	129,24
28672	129,23	129,22	129,3	129,3	129,3	129,22	129,3	129,31	129,3	129,25	129,25	129,19	129,3
32768	129,3	129,3	129,24	129,26	129,24	129,31	129,23	129,22	129,22	129,29	129,29	129,22	129,22
36864	129,23	129,22	129,3	129,29	129,29	129,23	129,3	129,31	129,3	129,18	129,29	129,29	129,3
40960	129,3	129,31	129,24	129,3	129,29	129,13	129,25	129,17	129,26	129,29	129,23	129,22	129,22
45056	129,25	129,25	129,3	129,07	129,22	129,15	129,29	129,3	129,23	129,21	129,31	129,3	129,3
49152	129,29	129,13	129,22	129,2	129,3	129,3	129,23	129,3	129,3	129,29	129,25	129,24	129,23
53248	129,23	129,3	129,3	129,29	129,25	129,24	129,3	129,23	129,2	129,25	129,29	129,29	129,23
57344	129,3	129,21	129,24	129,22	129,29	129,23	129,22	129,3	129,3	129,3	129,25	129,29	129,3
61440	129,23	129,31	129,3	129,29	129,29	129,29	129,22	129,24	129,25	129,24	129,3	129,2	129,23
65536	129,3	129,25	129,25	129,25	129,23	129,24	129,19	129,29	129,3	129,22	129,23	129,3	129,3

Tabela 5.2 *Throughput* em função do tamanho dos *sockets* nos testes ATM.

1K Receptor	send_sock												
recv_sock	16384	20480	24576	28672	32768	36864	40960	45056	49152	53248	57344	61440	65536
16384	128,98	129,2	129,29	129,29	129	128,95	129,2	129,2	128,98	128,94	129,16	129,28	129,21
20480	129,23	129,2	128,97	129,17	129,15	129,24	129,23	129,12	128,96	129,23	129,22	129,08	129,29
24576	129,13	129,14	129,25	129,24	129,22	129,18	129,29	129,04	129,22	129,21	128,99	128,92	128,93
28672	129,23	129,1	129,22	128,99	129,11	129,07	128,98	129,2	129,3	129,25	129,25	129,05	128,92
32768	129,18	129	129,24	129,26	129,14	129,2	129,23	129,22	129,14	129,2	129,29	129,02	129,16
36864	129,23	129,22	129,02	129,29	129,29	129,23	129,09	129,23	129,2	129,03	129,2	129,17	129,07
40960	129,23	129,25	128,99	129,23	129,19	129,04	129,25	129,04	129,15	129,29	129,23	129,22	129,13
45056	129,25	129,15	129,19	129,07	129,04	129,15	129,18	129,12	129,15	129,14	129	129,24	129,03
49152	129,29	129,13	129,22	129,11	128,9	129,12	129,07	129,21	129,21	129,29	129,23	129,24	129,13
53248	129,07	129,17	129,03	129,17	129,25	129,04	129,22	129,23	129,2	129,16	129,29	129,29	129,09
57344	129,19	129,21	129,24	129,11	129,29	129,12	129,22	129,07	129,25	129,12	128,98	129,04	129,08
61440	128,9	129,25	129,24	129,13	129,22	129,17	129,14	129,24	129,25	129,17	129,14	129,2	129,23
65536	129,1	129,24	129,17	129,18	129,23	129,24	129,19	129,21	129,3	129,13	129,18	129,25	129,26

Tabela 5.3 *Throughput* em função do tamanho dos *sockets* nos testes ATM.

9180 bytes	send_sock													
Transmissor / recv_sock	16384	20480	24576	28672	32768	36864	40960	45056	49152	53248	57344	61440	65536	
16384	133,03	133,03	133,02	133,03	133,03	133,03	133,03	133,02	133,02	133,03	133,02	133,02	133,03	
20480	133,02	132,94	133,03	133,03	133,02	133,02	133,03	133,02	133,03	133,03	133,03	133,02	133,02	
24576	133,02	133,02	133,03	133,03	133,03	133,03	133,02	133,02	133,03	133,03	133,02	133,02	133,03	
28672	133,02	133,03	133,02	133,03	133,02	133,03	133,03	133,03	133,03	133,03	133,02	133,02	133,03	
32768	133,03	133,02	133,02	133,03	133,03	133,02	133,03	133,03	133,03	133,03	133,02	133,02	133,03	
36864	133,03	133,03	133,02	133,03	133,02	133,02	133,02	133,02	133,03	133,03	133,02	133,02	133,02	
40960	133,02	133,02	133,03	133,02	133,03	133,03	133,02	133,02	133,02	133,03	133,02	133,02	133,03	
45056	133,02	133,03	133,02	133,03	133,03	133,03	133,02	133,03	133,03	133,03	133,02	133,02	133,03	
49152	133,03	133,03	133,02	133,03	133,02	133,02	133,03	133,02	133,03	133,02	133,02	133,03	133,02	
53248	133,02	133,02	133,03	133,02	133,02	133,03	133,02	133,02	133,02	133,03	133,02	133,02	133,02	
57344	133,03	133,02	133,03	133,03	133,02	133,02	133,03	133,03	133,03	133,02	133,02	133,02	133,02	
61440	133,02	133,03	133,03	133,02	133,02	133,03	133,03	133,02	133,03	133,03	133,02	133,02	133,02	
65536	133,03	133,03	133,03	133,03	133,02	133,03	133,02	133,02	133,02	133,03	133,03	133,02	133,02	

Tabela 5.4 *Throughput* em função do tamanho dos *sockets* nos testes ATM.

9180 bytes	send_sock													
Receptor / recv_sock	16384	20480	24576	28672	32768	36864	40960	45056	49152	53248	57344	61440	65536	
16384	132,97	133	133	132,84	132,98	132,99	133,02	133,02	133	133,03	133	132,93	132,89	
20480	132,92	132,44	133,03	133,03	132,94	132,93	132,94	132,94	132,92	132,92	132,92	133,02	133,02	
24576	133,02	132,31	132,92	132,89	133,03	133,02	133,02	132,92	133,03	133,01	133,02	132,91	133,03	
28672	132,91	133,03	133,02	133,03	132,96	132,92	133,03	133,03	132,96	132,96	132,96	133,02	132,96	
32768	133,03	132,96	132,96	132,98	133,03	133,02	132,95	132,94	133,03	133,03	132,94	132,95	133,03	
36864	132,95	133,03	133,02	133,03	132,93	132,94	133,02	133,02	132,95	133,03	133,02	133,02	132,97	
40960	133,02	132,95	132,97	133,02	133,03	133,03	132,97	132,97	133,02	132,95	132,95	132,96	133,03	
45056	132,97	133,03	133,02	132,96	132,95	132,98	133,02	133,03	132,93	133,03	133,02	133,02	132,95	
49152	133,03	132,99	132,96	133,03	133,02	133,02	132,99	133,02	133,03	132,98	132,97	132,98	133,02	
53248	133,02	133,02	133,03	132,98	132,98	132,97	133,02	132,98	132,96	133,03	133,02	132,95	133,02	
57344	132,97	132,97	132,99	133,03	133,02	132,97	132,97	133,03	133,03	132,96	133,02	133,02	132,99	
61440	133,02	133,03	133,03	132,96	133,02	133,03	133,03	132,98	132,99	133,03	132,94	132,96	133,02	
65536	132,99	132,99	132,99	133,03	132,95	132,97	132,92	133,02	133,02	132,97	133,03	133,02	132,97	

Capítulo 6

Conclusões, Contribuições e Trabalhos Futuros

Pela Tabela 6.1, podemos considerar excelentes os resultados obtidos e concluir que a plataforma HARP é capaz de explorar muito bem a largura de banda disponível em uma rede ATM, tanto em modo nativo como empregando outros protocolos como TCP e UDP sobre a camada AAL5.

	TCP/ATM	UDP/ATM	ATM Nativo
Máximo obtido	132,2	132,6	133,0
Máximo teórico	134,5	134,7	135,6

Tabela 6.1 Resumo dos resultados das taxas de transmissão.

Devemos considerar também que as máquinas de pequeno porte utilizadas nos testes desempenharam com eficiência o processamento dos protocolos na transmissão dos dados. Uma ressalva deve ser feita quanto ao poder de processamento da máquina receptora. Nos testes realizados em modo ATM nativo, constatamos que a incapacidade do receptor de processar pequenos blocos de dados

pode gerar muitas perdas de mensagens devido a problemas de alocação de *buffers* pequenos. Nos testes do protocolo TCP sobre ATM, também nota-se que mensagens pequenas podem gerar taxas de transmissão inesperadamente baixas quando os *buffers* de *socket* são pequenos, por exemplo, 16K.

Em todos os testes, a taxa de transmissão se mostrou tanto mais alta quanto maior se configurava o tamanho das mensagens e o tamanho dos *buffers* de *socket*. No entanto, deve-se tomar cuidado na escolha da combinação dos *buffers* de *socket*, de transmissão e de recepção, para evitar-se comportamentos inesperados provenientes do fenômeno SWS (*Silly Window Syndrome*). Uma solução é proposta por Comer em [35]: o tamanho do *buffer* de transmissão não deve ser menor que 3*MSS. Esta solução é efetiva em ambientes de rede heterogêneos já que o protocolo TCP do lado transmissor não tem controle sobre o tamanho do *buffer* de *socket* do receptor.

A respeito da perda de células, constatamos que não houve nenhuma perda na chave ATM. Exatamente o esperado, já que apenas duas das máquinas conectadas à chave transmitiam dados em cada teste, o que não é capaz de gerar tráfego acima de 2.5 Gbps, a capacidade máxima da chave.

Vale mencionar que a API HARP fornece uma solução relativamente fácil para *sockets* ATM, já que as mudanças que devem ser feitas no código de um programa que utilize a API BSD tradicional são poucas. Os resultados mostram eficiência no aproveitamento da largura de banda pela aplicação. Porém, muito trabalho ainda pode ser feito no sentido de dar à plataforma HARP mais condições de utilizar os serviços ATM, como, por exemplo, qualidade de serviço. Esta, inclusive, é uma das propostas para trabalhos futuros.

Por fim, o trabalho de investigação realizado revelou valiosas informações e contribuiu muito para o crescimento do nosso grupo de trabalho. Mais uma vez queremos deixar claro que todas as medidas foram realizadas em ambiente de rede ideal, isto é, sem tráfego proveniente de outras máquinas ou aplicações. Ainda assim, são importantes para o entendimento do funcionamento e desempenho dos protocolos TCP(UDP)/IP sobre ATM.

Como possíveis trabalhos futuros, podemos colocar as seguintes propostas:

1. Investigar o desempenho da rede ATM em transmissões bidirecionais;
2. Aplicar a API HARP a aplicativos de rede já consagrados, como `ftp`, e investigar o desempenho da rede, e das máquinas envolvidas, na transmissão de arquivos;
3. Aplicar a API HARP ao aplicativo de transmissão de vídeo `vic`;
4. Agregar à plataforma HARP suporte a alguns serviços ainda não implementados como: controle de tráfego, qualidade de serviço, IP *multicast* e emulação de LAN;
5. Investigar o desempenho da rede ATM em transmissões de áudio e vídeo com relação à qualidade de serviço.

Anexo A – Código de nettest_atmn.c

Este anexo contém uma parte do código fonte do arquivo `nettest_atmn.c` que implementa as rotinas que realizam os testes de transmissão no ambiente ATM nativo. Mostramos apenas as funções de transmissão unidirecional, STREAM, já que as funções para transmissões bidirecionais não foram utilizadas em nossos testes. As funções utilizam a API HARP para *sockets* ATM. Na estrutura de código do aplicativo Netperf, este arquivo substitui o arquivo `nettest_fore.c`.

```
#ifdef lint
#define DO_FORE
#define DIRTY
#define INTERVALS
#endif /* lint */
#ifdef DO_FORE
char   nettest_fore[]="\
@(#)nettest_fore.c (c) Copyright 1994,1995 Hewlett-Packard Co. Version 2.1";

/****************************************************************/
/*                                                              */
/*      nettest_atmn.c                                          */
/*                                                              */
/*       the FORE sockets parsing routine...                    */
/*                                                              */
/*       scan_fore_args()                                       */
/*                                                              */
/*      the actual test routines...                             */
/*                                                              */
/*      send_fore_stream()    perform a fore stream test        */
/*      recv_fore_stream()                                      */
/*      send_fore_rr()        perform a fore request/response   */
/*      recv_fore_rr()                                          */
/*                                                              */
/****************************************************************/

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/file.h>
#include <fcntl.h>
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <netdb.h>
#include <string.h>
#include <time.h>
#include <stdlib.h>

#include <sys/param.h>
#include <sys/socket.h>
#include <net/if.h>
#include <netatm/atm.h>

#include "netlib.h"
#include "netsh.h"
#include "nettest_atmn.h"

#define MAX_LEN       9180    /*Maximum PDU length*/
#define MY_ID         11      /*BLLI Layer 2 protocol*/
#define CLI_APP       "netperf"
#define SER_APP       "netserver"


Atm_addr_nsap  atm7_nsap = {
0x47,
{0x00, 0x05, 0x80, 0xff, 0xe1, 0x00, 0x00, 0x00, 0xf2, 0x1a, 0x63, 0x9f},
{0x00, 0x20, 0x48, 0x40, 0x39, 0xe6},
0x00
};  /* endereço ATM da maquina ATM7 */


Atm_addr_nsap  atm5_nsap = {
0x47,
{0x00, 0x05, 0x80, 0xff, 0xe1, 0x00, 0x00, 0x00, 0xf2, 0x1a, 0x63, 0x9f},
{0x00, 0x20, 0x48, 0x40, 0x39, 0x6d},
0x00
};  /* endereço ATM da maquina ATM5 */



/* these variables are specific to the FORE sockets tests. declare  */
/* them static to make them global only to this file. some of these */
/* might not actually have meaning in a FORE API test. */

static int
```

```
  req_size,           /* request size */
  rsp_size,           /* response size */
  send_size,          /* how big are individual sends      */
  recv_size,          /* how big are individual receives   */
  rss_size,           /* remote socket send buffer size    */
  rsr_size,           /* remote socket recv buffer size    */
  lss_size,           /* local  socket send buffer size    */
  lsr_size,           /* local  socket recv buffer size    */
  linha;

static int
  init_done = 0;

char fore_usage[] = "\n\
Usage: netperf [global options] -- [test options] \n\
\n\
FORE Sockets API Test Options:\n\
    -h                Display this text\n\
    -s send[,recv]    Set local socket send/recv buffer sizes\n\
    -S send[,recv]    Set remote socket send/recv buffer sizes\n\
    -m bytes          Set the send size (FORE_STREAM)\n\
    -M bytes          Set the recv size (FORE_STREAM)\n\
    -r bytes,bytes    Set request,response size (FORE_RR)\n\
\n\
For those options taking two parms, at least one must be specified;\n\
specifying one value without a comma will set both parms to that\n\
value, specifying a value with a leading comma will set just the second\n\
parm, a value with a trailing comma will set just the first. To set\n\
each parm to unique values, specify both and separate them with a\n\
comma.\n";

/*-------------------------------------------------------------------------*/

void
print_cause(int s)
{
        struct  t_atm_cause    cause;
        int     optlen;

        optlen = sizeof(cause);
        if (getsockopt(s, T_ATM_SIGNALING, T_ATM_CAUSE, &cause, &optlen) < 0) {
                perror("getsockopt(cause)");
                return;
        }

        fprintf(stderr, "Cause: coding=%d loc=%d cause=%d diag=(%d,%d,%d,%d)\n",
                cause.coding_standard, cause.location, cause.cause_value,
                cause.diagnostics[0], cause.diagnostics[1],
                cause.diagnostics[2], cause.diagnostics[3]);
}

/*-------------------------------------------------------------------------*/

/* this routine will set the default values for all the test specific */
/* variables. it is declared static so that it is not exported */
/* outside of the module */

static void
init_test_vars()
{

  if (init_done) {
    return;
  }
  else {

    req_size = 1;
    rsp_size = 1;
    send_size = 0;
    recv_size = 0;

    init_done = 1;
  }
}
```

```
/*---------------------------------------------------------------------*/

/* This routine will create a data (listen) socket with the apropriate */
/* options set and return it to the caller. this replaces all the */
/* duplicate code in each of the test routines and should help make */
/* things a little easier to understand. since this routine can be */
/* called by either the netperf or netserver programs, all output */
/* should be directed towards "where." when being called from */
/* netserver, care should be taken to insure that the globals */
/* referenced by this routine are set. initially, this routine is */
/* very simple because it would seem that most of the "sockops" are */
/* actually set at connect time and not at socket creation time. */

int
create_fore_socket()
{

  int temp_socket;
  int sock_opt_len;

  /*set up the data socket                                */
  temp_socket = socket(AF_ATM,SOCK_SEQPACKET,ATM_PROTO_AAL5);
  if (temp_socket < 0){
    perror("netperf:create_fore_socket: socket");
    fprintf(where,
            "netperf: create_fore_socket: could not open device : %d\n",
            errno);
    fflush(where);
    exit(1);
  }

  if (debug) {
    fprintf(where,"create_fore_socket: socket %d obtained...\n",temp_socket);
    /* include salient points from the atm_device_info structure here */
    fflush(where);
  }


  /***** Altera o tamanho dos buffers locais do socket aberto ******/

  if (lss_size > 0) {
    if(setsockopt(temp_socket, SOL_SOCKET, SO_SNDBUF,
                  (char *)&lss_size, sizeof(int)) < 0) {
      fprintf(where,
              "netperf: create_data_socket: SO_SNDBUF option: errno %d\n",
              errno);
      fflush(where);
      exit(1);
    }
    if (debug > 1) {
      fprintf(where,
              "netperf: create_data_socket: SO_SNDBUF of %d requested.\n",
              lss_size);
      fflush(where);
    }
  }
  if (lsr_size > 0) {
    if(setsockopt(temp_socket, SOL_SOCKET, SO_RCVBUF,
                  (char *)&lsr_size, sizeof(int)) < 0) {
      fprintf(where,
              "netperf: create_data_socket: SO_RCVBUF option: errno %d\n",
              errno);
      fflush(where);
      exit(1);
    }
    if (debug > 1) {
      fprintf(where,
              "netperf: create_data_socket: SO_SNDBUF of %d requested.\n",
              lss_size);
      fflush(where);
    }
  }

  /* Now, we will find-out what the size actually became, and report */
  /* that back to the user. If the call fails, we will just report a -1 */
```

```
/* back to the initiator for the recv buffer size. */

sock_opt_len = sizeof(int);
if (getsockopt(temp_socket,
               SOL_SOCKET,
               SO_SNDBUF,
               (char *)&lss_size,
               &sock_opt_len) < 0) {
  fprintf(where,
          "netperf: create_data_socket: getsockopt SO_SNDBUF: errno %d\n",
          errno);
  fflush(where);
  lss_size = -1;
}
if (getsockopt(temp_socket,
               SOL_SOCKET,
               SO_RCVBUF,
               (char *)&lsr_size,
               &sock_opt_len) < 0) {
  fprintf(where,
          "netperf: create_data_socket: getsockopt SO_SNDBUF: errno %d\n",
          errno);
  fflush(where);
  lsr_size = -1;
}

if (debug) {
  fprintf(where,
          "netperf: create_data_socket: socket sizes determined...\n");
  fprintf(where,
          "                             lss_size: %d lsr_size: %d\n",
          lss_size,lsr_size);
  fflush(where);
}


 return(temp_socket);

}


/*-----------------------------------------------------------------------*/

void
send_fore_stream(remote_host)
char *remote_host;
{
  /****************************************************************/
  /*                                                              */
  /*                  FORE Unidirectional Send Test               */
  /*                                                              */
  /****************************************************************/

  char *tput_title =
    "Socket  Message  Elapsed      Messages                   \n\
Size     Size     Time         Okay Errors   Throughput\n\
bytes    bytes    secs            #      #   %s/sec\n\n";

  char *tput_fmt_0 =
    "%7.2f\n";

  char *tput_fmt_1 =
    "%5d   %5d    %-7.2f   %7d %6d    %7.2f\n";

  char *tput_fmt_2 =
    "%5d   %5d    %-7.2f   %7d %6d    %7.2f\n\
%5d            %-7.2f   %7d           %7.2f\n\n";

  char *cpu_title =
    "Socket  Message  Elapsed      Messages                     CPU     Service\n\
Size     Size     Time         Okay Errors   Throughput   Util    Demand\n\
bytes    bytes    secs            #      #   %s/sec   %%       us/KB\n\n";

  char *cpu_fmt_0 =
    "%6.2f\n";
```

```
  char *cpu_fmt_1 =
    "%5d    %5d     %-7.2f    %7d %6d      %7.1f       %-6.2f  %-6.3f\n\
%5d             %-7.2f    %7d              %7.1f       %-6.2f  %-6.3f\n\n";

  int   messages_recvd;
  float elapsed_time,
  local_cpu_utilization,
  remote_cpu_utilization;

  float local_service_demand, remote_service_demand;
  double       local_thruput, remote_thruput;
  double       bytes_sent;
  double       bytes_recvd;


  int   len, optlen;
  int   *message_int_ptr;
  struct ring_elt *send_ring;
  int   failed_sends;
  int   messages_sent;
  int   data_socket;


#ifdef INTERVALS
  int   interval_count;
#endif /* INTERVALS */
#ifdef DIRTY
  int   i;
#endif /* DIRTY */

   struct sockaddr_atm satm;
   struct t_atm_aal5   aal5;
   struct t_atm_traffictraffic;
   struct t_atm_bearer bearer;
   struct t_atm_qos    qos;
   struct t_atm_net_intf       netintf;
   struct t_atm_app_name       appname;

   struct        sigaction       action;

   struct       fore_stream_request_struct    *fore_stream_request;
   struct       fore_stream_response_struct   *fore_stream_response;
   struct       fore_stream_results_struct    *fore_stream_results;

   init_test_vars();

   fore_stream_request =  (struct fore_stream_request_struct *)
 netperf_request.content.test_specific_data;

   fore_stream_response= (struct fore_stream_response_struct *)
 netperf_response.content.test_specific_data;

   fore_stream_results = (struct fore_stream_results_struct *)
 netperf_response.content.test_specific_data;

   if ( print_headers ) {
     printf("FORE UNIDIRECTIONAL SEND TEST\n");
     if (local_cpu_usage || remote_cpu_usage)
       printf(cpu_title,format_units());
     else
       printf(tput_title,format_units());
   }

   failed_sends = 0;
   messages_sent       = 0;
   times_up     = 0;

   /*set up the data socket */
   data_socket = create_fore_socket();


   if (data_socket < 0){
     perror("netperf: send_fore_stream: data socket");
     exit(1);
```

```
}

        /****************************/
        /* Set up destination SAP   */
        /****************************/

        bzero((caddr_t) &satm, sizeof(satm));
        satm.satm_family = AF_ATM;
#if (defined(BSD) && (BSD >= 199103))
        satm.satm_len = sizeof(satm);
#endif
        /* Destination ATM address */
        satm.satm_addr.t_atm_sap_addr.SVE_tag_addr = T_ATM_PRESENT;
        satm.satm_addr.t_atm_sap_addr.SVE_tag_selector = T_ATM_PRESENT;
        satm.satm_addr.t_atm_sap_addr.address_format = T_ATM_ENDSYS_ADDR;
        satm.satm_addr.t_atm_sap_addr.address_length = sizeof(Atm_addr_nsap);

if ( strcmp(remote_host, "atm5") == 0 )
        bcopy((caddr_t)&atm5_nsap,
                (caddr_t)satm.satm_addr.t_atm_sap_addr.address,
                sizeof(atm5_nsap));

if ( strcmp(remote_host, "atm7") == 0 )
        bcopy((caddr_t)&atm7_nsap,
                (caddr_t)satm.satm_addr.t_atm_sap_addr.address,
                sizeof(atm7_nsap));


        /* BLLI Layer-2 protocol */
        satm.satm_addr.t_atm_sap_layer2.SVE_tag = T_ATM_PRESENT;
        satm.satm_addr.t_atm_sap_layer2.ID_type = T_ATM_USER_ID;
        satm.satm_addr.t_atm_sap_layer2.ID.user_defined_ID = MY_ID;

        /* BLLI Layer-3 protocol */
        satm.satm_addr.t_atm_sap_layer3.SVE_tag = T_ATM_ABSENT;

        /* BHLI protocol */
        satm.satm_addr.t_atm_sap_appl.SVE_tag = T_ATM_ABSENT;



 if(debug)
        fprintf(where, "\nsend_fore_stream: set up destination SAP done.\n");


        /*********************************/
        /* Set up connection parameters  */
        /*********************************/

        aal5.forward_max_SDU_size = MAX_LEN;
        aal5.backward_max_SDU_size = MAX_LEN;
        aal5.SSCS_type = T_ATM_NULL;
        optlen = sizeof(aal5);
        if (setsockopt(data_socket, T_ATM_SIGNALING, T_ATM_AAL5, (caddr_t)&aal5,
                        optlen) < 0) {
                perror("setsockopt(aal5)");
                exit(1);
        }

        traffic.forward.PCR_high_priority = T_ATM_ABSENT;
        traffic.forward.PCR_all_traffic = 100000;
        traffic.forward.SCR_high_priority = T_ATM_ABSENT;
        traffic.forward.SCR_all_traffic = T_ATM_ABSENT;
        traffic.forward.MBS_high_priority = T_ATM_ABSENT;
        traffic.forward.MBS_all_traffic = T_ATM_ABSENT;
        traffic.forward.tagging = T_NO;
        traffic.backward.PCR_high_priority = T_ATM_ABSENT;
        traffic.backward.PCR_all_traffic = 100000;
        traffic.backward.SCR_high_priority = T_ATM_ABSENT;
        traffic.backward.SCR_all_traffic = T_ATM_ABSENT;
        traffic.backward.MBS_high_priority = T_ATM_ABSENT;
        traffic.backward.MBS_all_traffic = T_ATM_ABSENT;
        traffic.backward.tagging = T_NO;
        traffic.best_effort = T_YES;
        optlen = sizeof(traffic);
```

```
        if(setsockopt(data_socket,T_ATM_SIGNALING,T_ATM_TRAFFIC, (caddr_t)&traffic,
                      optlen) < 0) {
               perror("setsockopt(traffic)");
               exit(1);
        }

        bearer.bearer_class = T_ATM_CLASS_X;
        bearer.traffic_type = T_ATM_NULL;
        bearer.timing_requirements = T_ATM_NULL;
        bearer.clipping_susceptibility = T_NO;
        bearer.connection_configuration = T_ATM_1_TO_1;
        optlen = sizeof(bearer);
        if(setsockopt(data_socket,T_ATM_SIGNALING,T_ATM_BEARER_CAP, (caddr_t)&bearer,
                      optlen) < 0) {
               perror("setsockopt(bearer)");
               exit(1);
        }

        qos.coding_standard = T_ATM_NETWORK_CODING;
        qos.forward.qos_class = T_ATM_QOS_CLASS_0;
        qos.backward.qos_class = T_ATM_QOS_CLASS_0;
        optlen = sizeof(qos);
        if (setsockopt(data_socket, T_ATM_SIGNALING, T_ATM_QOS, (caddr_t)&qos,
                       optlen) < 0) {
               perror("setsockopt(qos)");
               exit(1);
        }

        strncpy(netintf.net_intf, "ni0", IFNAMSIZ);
        optlen = sizeof(netintf);
        if(setsockopt(data_socket,T_ATM_SIGNALING,T_ATM_NET_INTF, (caddr_t)&netintf,
                      optlen) < 0) {
               perror("setsockopt(net_intf)");
               exit(1);
        }

        strncpy(appname.app_name, CLI_APP, T_ATM_APP_NAME_LEN);
        optlen = sizeof(appname);
        if(setsockopt(data_socket,T_ATM_SIGNALING,T_ATM_APP_NAME, (caddr_t)&appname,
                      optlen) < 0) {
               perror("setsockopt(app_name)");
               exit(1);
        }

        /************************/

if(debug)
        fprintf(where, "\nsend_fore_stream: set up connection parameters done.\n");

  /* now, we want to see if we need to set the send_size. if the user */
  /* did not specify a send_size on the command line, we will use the */
  /* mtu from the atm_device_info structure that was returned when the */
  /* socket was opened */
  if (send_size == 0) {
      send_size = MAX_LEN;
  }


  /* set-up the data buffer with the requested alignment and offset, */
  /* most of the numbers here are just a hack to pick something nice */
  /* and big in an attempt to never try to send a buffer a second time */
  /* before it leaves the node...unless the user set the width */
  /* explicitly. */
  if (send_width == 0) send_width = 32;

  send_ring = allocate_buffer_ring(send_width,
                                   send_size,
                                   local_send_align,
                                   local_send_offset);

  /* if the user supplied a cpu rate, this call will complete rather */
  /* quickly, otherwise, the cpu rate will be retured to us for */
  /* possible display. The Library will keep it's own copy of this data */
  /* for use elsewhere. We will only display it. (Does that make it */
  /* "opaque" to us?) */
```

IFSC-USP SERVIÇO DE BIBLIOTECA INFORMAÇÃO

```
  if (local_cpu_usage)
    local_cpu_rate = calibrate_local_cpu(local_cpu_rate);

  /* Tell the remote end to set up the data connection. The server */
  /* sends back the port number and alters the socket parameters there. */
  /* Of course this is a datagram service so no connection is actually */
  /* set up, the server just sets up the socket and binds it. */

if (debug)
       fprintf(where,"send_fore_stream: rss_size = %d  rsr_size = %d\n",
    rss_size, rsr_size);

  netperf_request.content.request_type = DO_FORE_STREAM;
  fore_stream_request->send_buf_size    = rss_size;
  fore_stream_request->recv_buf_size    = rsr_size;
  fore_stream_request->message_size  = send_size;
  fore_stream_request->recv_alignment       = remote_recv_align;
  fore_stream_request->recv_offset   = remote_recv_offset;
  fore_stream_request->measure_cpu   = remote_cpu_usage;
  fore_stream_request->cpu_rate             = remote_cpu_rate;
  fore_stream_request->test_length   = test_time;

  send_request();

  recv_response();

  /* at some point here I need to be able to distinguish between an */
  /* atm_errno and a unix errno. perhaps I'll make one negative or */
  /* something raj 8/94 in the meantime, I'll just live with bogus */
  /* error messages */
  if (!netperf_response.content.serv_errno) {
    if (debug)
      fprintf(where,"send_fore_stream: remote data connection done.\n");
  }
  else {
    errno = netperf_response.content.serv_errno;
    perror("send_fore_stream: error on remote");
    exit(1);
  }

  /* Place the port number returned by the remote into the sockaddr */
  /* structure so our sends can be sent to the correct place. Also get */
  /* some of the returned socket buffer information for user display. */

/*  server.asap = fore_stream_response->server_asap; */

  rss_size = fore_stream_response->send_buf_size;
  rsr_size = fore_stream_response->recv_buf_size;

if(debug)
       fprintf(where, "\nsend_fore_stream: rss_size = %d  rsr_size = %d.\n",
              rss_size, rsr_size);

  remote_cpu_rate     = fore_stream_response->cpu_rate;

  /* We "connect" up to the remote. since this is a stream test, we */
  /* can specify that the connection is suppsed to be "simplex" */

  if ( connect(data_socket, (struct sockaddr *) &satm, sizeof(satm)) < 0){
    perror("send_fore_stream: data socket connect failed");
    print_cause(data_socket);
    exit(1);
  }
  else
    if(debug)
      fprintf(where, "\nsend_fore_stream: connect done.\n");


  /* set up the timer to call us after test_time    */
  start_timer(test_time);

  /* Get the start count for the idle counter and the start time */

  cpu_start(local_cpu_usage);
```

```
#ifdef INTERVALS
  interval_count = interval_burst;
#endif

  /* Send datagrams like there was no tomorrow. at somepoint it might */
  /* be nice to set this up so that a quantity of bytes could be sent, */
  /* but we still need some sort of end of test trigger on the receive */
  /* side. that could be a select with a one second timeout, but then */
  /* if there is a test where none of the data arrives for awile and */
  /* then starts again, we would end the test too soon. something to */
  /* think about... */

if(debug)
       fprintf(where,"\nsend_fore_stream: ready to enter while(!times_up)\n");

  while (!times_up) {

#ifdef DIRTY
    /* we want to dirty some number of consecutive integers in the buffer */
    /* we are about to send. we may also want to bring some number of */
    /* them cleanly into the cache. The clean ones will follow any dirty */
    /* ones into the cache. */
    message_int_ptr = (int *)(send_ring->buffer_ptr);
    for (i = 0; i < loc_dirty_count; i++) {
      *message_int_ptr = 4;
      message_int_ptr++;
    }
    for (i = 0; i < loc_clean_count; i++) {
      loc_dirty_count = *message_int_ptr;
      message_int_ptr++;
    }
#endif /* DIRTY */

    if (( len = write(data_socket,
                      send_ring->buffer_ptr,
                      send_size))  != send_size) {
      /* need to check what is returned when the atm_send called is */
      /* interrupted by a signal */
      if ((len >= 0) || (errno == EINTR))
       break;
      if (errno == ENOBUFS) {
       /* what is the error message when we are sending too fast? is */
       /* there one? will I be flow controlled based on the qos? */
       failed_sends++;
       continue;
      }
      perror("fore_send: data send error");
      exit(1);
    }
    messages_sent++;

    /* now we want to move our pointer to the next position in the */
    /* data buffer... */

    send_ring = send_ring->next;


  #ifdef INTERVALS
    /* in this case, the interval count is the count-down couter */
    /* to decide to sleep for a little bit */
    if ((interval_burst) && (--interval_count == 0)) {
      /* call the sleep routine for some milliseconds, if our */
      /* timer popped while we were in there, we want to */
      /* break out of the loop. */
      if (msec_sleep(interval_wate)) {
       break;
      }
      interval_count = interval_burst;
    }

  #endif

  } /* while(!times_up) */
```

```
if(debug)
       fprintf(where, "\nsend_fore_stream: out the while(!times_up)\n");


  /* This is a timed test, so the remote will be returning to us after */
  /* a time. We should not need to send any "strange" messages to tell */
  /* the remote that the test is completed, unless we decide to add a */
  /* number of messages to the test. */

  /* the test is over, so get stats and stuff */
  cpu_stop(local_cpu_usage,
           &elapsed_time);

  /* Get the statistics from the remote end */
  recv_response();
  if (!netperf_response.content.serv_errno) {
    if (debug)
      fprintf(where,"send_fore_stream: remote results obtained\n");
  }
  else {
    errno = netperf_response.content.serv_errno;
    perror("send_fore_stream: error on remote");
    exit(1);
  }

  /* The test is over. */

  if ( close(data_socket) < 0) {
    /* we will not consider this a fatal error. just display a message */
    /* and move on */
    perror("netperf: cannot shutdown fore socket");
  }
  else
    if(debug)
      fprintf(where, "\nsend_fore_stream: close data_socket done\n");


  bytes_sent   = send_size * messages_sent;
  local_thruput        = calc_thruput(bytes_sent);

  messages_recvd       = fore_stream_results->messages_recvd;
  /*bytes_recvd        = send_size * messages_recvd;*/
  bytes_recvd = fore_stream_results->bytes_received;

  /* we asume that the remote ran for as long as we did */

  remote_thruput       = calc_thruput(bytes_recvd);


  if(debug) {
    fprintf(where,"\nbytes sent: %ld", bytes_sent);
    fprintf(where,"\nlocal throuput: %f", local_thruput);
    fprintf(where,"\nremote throuput: %f", remote_thruput);
    fprintf(where,"\nbytes received: %ld\n", bytes_recvd);
  }


  /* print the results for this socket and message size */

  if (local_cpu_usage || remote_cpu_usage) {
    /* We must now do a little math for service demand and cpu */
    /* utilization for the system(s) We pass zeros for the local */
    /* cpu utilization and elapsed time to tell the routine to use */
    /* the libraries own values for those. */
    if (local_cpu_usage) {
      if (local_cpu_rate == 0.0) {
        fprintf(where,"WARNING WARNING WARNING  WARNING WARNING WARNING  WARNING!\n");
        fprintf(where,"Local CPU usage numbers based on process information only!\n");
        fflush(where);
      }

      local_cpu_utilization   = calc_cpu_util(0.0);
      local_service_demand    = calc_service_demand(bytes_sent,
                                                    0.0,
                                                    0.0,
```

```
                                                    0);
    }
    else {
      local_cpu_utilization   = -1.0;
      local_service_demand    = -1.0;
    }

    /* The local calculations could use variables being kept by */
    /* the local netlib routines. The remote calcuations need to */
    /* have a few things passed to them. */
    if (remote_cpu_usage) {
      if (remote_cpu_rate == 0.0) {
fprintf(where,"DANGER   DANGER  DANGER   DANGER  DANGER   DANGER   DANGER!\n");
fprintf(where,"REMOTE CPU usage numbers based on process information only!\n");
fflush(where);
      }

      remote_cpu_utilization  = fore_stream_results->cpu_util;


       remote_service_demand = calc_service_demand(bytes_recvd,
                                                   0.0,
                                                   remote_cpu_utilization,
                                                   fore_stream_results->num_cpus);
    }
    else {
      remote_cpu_utilization  = -1.0;
      remote_service_demand   = -1.0;
    }

    /* We are now ready to print all the information. If the user */
    /* has specified zero-level verbosity, we will just print the */
    /* local service demand, or the remote service demand. If the */
    /* user has requested verbosity level 1, he will get the basic */
    /* "streamperf" numbers. If the user has specified a verbosity */
    /* of greater than 1, we will display a veritable plethora of */
    /* background information from outside of this block as it it */
    /* not cpu_measurement specific...  */

    switch (verbosity) {
    case 0:
      if (local_cpu_usage) {
       fprintf(where,
               cpu_fmt_0,
               local_service_demand);
      }
      else {
       fprintf(where,
               cpu_fmt_0,
               remote_service_demand);
      }
      break;
    case 1:
      fprintf(where,
              cpu_fmt_1,              /* the format string */
              lss_size,               /* local sendbuf size */
              send_size,              /* how large were the sends */
              elapsed_time,           /* how long was the test */
              messages_sent,
              failed_sends,
              local_thruput,          /* what was the xfer rate */
              local_cpu_utilization,  /* local cpu */
              local_service_demand,   /* local service demand */
              rsr_size,
              elapsed_time,
              messages_recvd,
              remote_thruput,
              remote_cpu_utilization, /* remote cpu */
              remote_service_demand); /* remote service demand */
      break;
    }
  }
  else {
    /* The tester did not wish to measure service demand. */
    switch (verbosity) {
```

```
  case 0:
    fprintf(where,
            tput_fmt_0,
            local_thruput);
    break;
  case 1:
    if(!linha) {
      fprintf(where,
            tput_fmt_2,             /* the format string */
            lss_size,               /* local sendbuf size */
            send_size,              /* how large were the sends */
            elapsed_time,           /* how long did it take */
            messages_sent,
            failed_sends,
            local_thruput,
            rsr_size,               /* remote recvbuf size */
            elapsed_time,
            messages_recvd,
            remote_thruput
            );
      /*fprintf(where, "\nBytes recebidos: %lf\n\n", bytes_recvd);*/
     }
     else
       fprintf(where,
            tput_fmt_1,             /* the format string */
            lss_size,               /* local sendbuf size */
            send_size,              /* how large were the sends */
            elapsed_time,           /* how long did it take */
            messages_sent,
            failed_sends,
             local_thruput
            );
     break;
    }
  }

if(debug)
   fprintf(where, "\nsend_fore_stream: end of function\n");
}

/*----------------------------------------------------------------------------*/
/* this routine implements the receive side (netserver) of the                */
/* FORE_STREAM performance test.                                              */
/*----------------------------------------------------------------------------*/

int
recv_fore_stream()
{
  struct ring_elt *recv_ring;

        struct sockaddr_atm   satm, claddr;
        struct t_atm_aal5     aal5;
        struct t_atm_traffic  traffic;
        struct t_atm_bearer   bearer;
        struct t_atm_qos      qos;
        struct t_atm_net_intf netintf;
        struct t_atm_app_name appname;
        struct t_atm_addr     addr;


   int connection_id;

   int  s_data, clsock, cllen;
   int  len, optlen, satmlen, addrlen, k;
   double       bytes_received = 0;
   float elapsed_time;

   int  message_size;
   int  messages_recvd = 0;
   int  measure_cpu;

   struct        sigaction     action;

   struct       fore_stream_request_struct    *fore_stream_request;
```

```
struct      fore_stream_response_struct   *fore_stream_response;
struct      fore_stream_results_struct    *fore_stream_results;

init_test_vars();
debug = 2;

fore_stream_request  = (struct fore_stream_request_struct *)
netperf_request.content.test_specific_data;

fore_stream_response = (struct fore_stream_response_struct *)
netperf_response.content.test_specific_data;

fore_stream_results  = (struct fore_stream_results_struct *)
netperf_response.content.test_specific_data;

if (debug) {
  fprintf(where,"netserver: recv_fore_stream: entered...\n");
  fflush(where);
}

/* We want to set-up the listen socket with all the desired */
/* parameters and then let the initiator know that all is ready. If */
/* socket size defaults are to be used, then the initiator will have */
/* sent us 0's. If the socket sizes cannot be changed, then we will */
/* send-back what they are. If that information cannot be determined, */
/* then we send-back -1's for the sizes. If things go wrong for any */
/* reason, we will drop back ten yards and punt. */

/* If anything goes wrong, we want the remote to know about it. It */
/* would be best if the error that the remote reports to the user is */
/* the actual error we encountered, rather than some bogus unexpected */
/* response type message. */

if (debug > 1) {
  fprintf(where,"recv_fore_stream: setting the response type...\n");
  fflush(where);
}

netperf_response.content.response_type = FORE_STREAM_RESPONSE;

if (debug > 2) {
  fprintf(where,"recv_fore_stream: the response type is set...\n");
  fflush(where);
}

/* We now alter the message_ptr variable to be at the desired */
/* alignment with the desired offset. */

if (debug > 1) {
  fprintf(where,"recv_fore_stream: requested alignment of %d\n",
          fore_stream_request->recv_alignment);
  fflush(where);
}

if (recv_width == 0) recv_width = 2;

recv_ring = allocate_buffer_ring(recv_width,
                                 fore_stream_request->message_size,
                                 fore_stream_request->recv_alignment,
                                 fore_stream_request->recv_offset);

if (debug > 1) {
  fprintf(where,"recv_fore_stream: receive alignment and offset set...\n");
  fflush(where);
}

/* create_fore_socket expects to find some things in the global */
/* variables, so set the globals based on the values in the request. */
/* once the socket has been created, we will set the response values */
/* based on the updated value of those globals. raj 7/94 */

/*  selected_atm_qos.mean_burst.target = fore_stream_request->recv_buf_size; */

  lss_size = fore_stream_request->send_buf_size;
  lsr_size = fore_stream_request->recv_buf_size;
```

```
if(debug)
        fprintf(where, "\nsend_fore_stream: lss_size = %d  lsr_size = %d.\n",
                lss_size, lsr_size);


  s_data = create_fore_socket();

  if (s_data < 0) {
    netperf_response.content.serv_errno = errno;
    send_response();
    exit(1);
  }


        /*****************************/
        /* Set up destination SAP  */
        /*****************************/

        bzero((caddr_t) &satm, sizeof(satm));
        satm.satm_family = AF_ATM;
#if (defined(BSD) && (BSD >= 199103))
        satm.satm_len = sizeof(satm);
#endif
        /* Destination ATM address */
        satm.satm_addr.t_atm_sap_addr.SVE_tag_addr = T_ATM_ANY;
        satm.satm_addr.t_atm_sap_addr.SVE_tag_selector = T_ATM_ANY;

        /* BLLI Layer-2 protocol */
        satm.satm_addr.t_atm_sap_layer2.SVE_tag = T_ATM_PRESENT;
        satm.satm_addr.t_atm_sap_layer2.ID_type = T_ATM_USER_ID;
        satm.satm_addr.t_atm_sap_layer2.ID.user_defined_ID = MY_ID;

        /* BLLI Layer-3 protocol */
        satm.satm_addr.t_atm_sap_layer3.SVE_tag = T_ATM_ABSENT;

        /* BHLI protocol */
        satm.satm_addr.t_atm_sap_appl.SVE_tag = T_ATM_ABSENT;


        /*********************************/
        /* Set up connection parameters */
        /*********************************/

        aal5.forward_max_SDU_size = MAX_LEN;
        aal5.backward_max_SDU_size = MAX_LEN;
        aal5.SSCS_type = T_ATM_NULL;
        optlen = sizeof(aal5);
        if (setsockopt(s_data, T_ATM_SIGNALING, T_ATM_AAL5, (caddr_t)&aal5,
                       optlen) < 0) {
                perror("setsockopt(aal5)");
                exit(1);
        }

        traffic.forward.PCR_high_priority = T_ATM_ABSENT;
        traffic.forward.PCR_all_traffic = 100000;
        traffic.forward.SCR_high_priority = T_ATM_ABSENT;
        traffic.forward.SCR_all_traffic = T_ATM_ABSENT;
        traffic.forward.MBS_high_priority = T_ATM_ABSENT;
        traffic.forward.MBS_all_traffic = T_ATM_ABSENT;
        traffic.forward.tagging = T_NO;
        traffic.backward.PCR_high_priority = T_ATM_ABSENT;
        traffic.backward.PCR_all_traffic = 100000;
        traffic.backward.SCR_high_priority = T_ATM_ABSENT;
        traffic.backward.SCR_all_traffic = T_ATM_ABSENT;
        traffic.backward.MBS_high_priority = T_ATM_ABSENT;
        traffic.backward.MBS_all_traffic = T_ATM_ABSENT;
        traffic.backward.tagging = T_NO;
        traffic.best_effort = T_YES;
        optlen = sizeof(traffic);
        if (setsockopt(s_data, T_ATM_SIGNALING, T_ATM_TRAFFIC, (caddr_t)&traffic,
                       optlen) < 0) {
                perror("setsockopt(traffic)");
                exit(1);
        }
```

```
    bearer.bearer_class = T_ATM_CLASS_X;
    bearer.traffic_type = T_ATM_NULL;
    bearer.timing_requirements = T_ATM_NULL;
    bearer.clipping_susceptibility = T_NO;
    bearer.connection_configuration = T_ATM_1_TO_1;
    optlen = sizeof(bearer);
    if (setsockopt(s_data, T_ATM_SIGNALING, T_ATM_BEARER_CAP, (caddr_t)&bearer,
                   optlen) < 0) {
           perror("setsockopt(bearer)");
           exit(1);
    }

    qos.coding_standard = T_ATM_NETWORK_CODING;
    qos.forward.qos_class = T_ATM_QOS_CLASS_0;
    qos.backward.qos_class = T_ATM_QOS_CLASS_0;
    optlen = sizeof(qos);
    if (setsockopt(s_data, T_ATM_SIGNALING, T_ATM_QOS, (caddr_t)&qos,
                   optlen) < 0) {
           perror("setsockopt(qos)");
           exit(1);
    }

    strncpy(appname.app_name, SER_APP, T_ATM_APP_NAME_LEN);
    optlen = sizeof(appname);
    if (setsockopt(s_data, T_ATM_SIGNALING, T_ATM_APP_NAME, (caddr_t)&appname,
                   optlen) < 0) {
           perror("setsockopt(app_name)");
           exit(1);
    }

    /*****************************************/

/* Let's get an address assigned to this socket so we can tell the */
/* initiator how to reach the data socket. There may be a desire to */
/* nail this socket to a specific address in a multi-homed, */
/* multi-connection situation, but for now, we'll ignore the issue */
/* and concentrate on single connection testing. we specify the qlen */
/* parm as 1 because this is the server side. raj 8/94 */

if ( bind(s_data, (struct sockaddr *) &satm, sizeof(satm)) < 0 ) {
  netperf_response.content.serv_errno = errno;
  send_response();
  print_cause(s_data);
  exit(1);
}


fore_stream_response->test_length = fore_stream_request->test_length;

netperf_response.content.serv_errno   = 0;

/* But wait, there's more. If the initiator wanted cpu measurements, */
/* then we must call the calibrate routine, which will return the max */
/* rate back to the initiator. If the CPU was not to be measured, or */
/* something went wrong with the calibration, we will return a -1 to */
/* the initiator. */

fore_stream_response->cpu_rate = 0.0;      /* assume no cpu */
if (fore_stream_request->measure_cpu) {
  /* We will pass the rate into the calibration routine. If the */
  /* user did not specify one, it will be 0.0, and we will do a */
  /* "real" calibration. Otherwise, all it will really do is */
  /* store it away... */
  fore_stream_response->measure_cpu = 1;
  fore_stream_response->cpu_rate =
    calibrate_local_cpu(fore_stream_request->cpu_rate);
}

message_size = fore_stream_request->message_size;
test_time    = fore_stream_request->test_length;

/* before we send the response back to the initiator, pull some of */
/* the socket parms from the globals */
```

```
if(debug)
     fprintf(where, "\nsend_fore_stream: lss_size = %d  lsr_size = %d.\n",
             lss_size, lsr_size);

fore_stream_response->send_buf_size = lss_size;
fore_stream_response->recv_buf_size = lsr_size;

send_response();

/* now we have to listen and accept on the connection. I have some */
/* question about where this should be. Should this be after the */
/* call to cpu_start? */
if ( listen(s_data, 4) < 0) {
  fprintf(where,
          "netperf: recv_fore_stream: listen: errno = %d\n",
          errno);
  fflush(where);
  netperf_response.content.serv_errno = errno;
  send_response();
  print_cause(s_data);
  exit(1);
 }


 /* we'll just ape the QOS that came to us on the atm_listen call. */
 /* there is no real reason to examine it or anything since we don't */
 /* have values that we might prefer. raj 8/94 */

 cllen = sizeof(claddr);
 clsock = accept( s_data, (struct sockaddr *) &claddr, &cllen);
 if ( clsock < 0) {
   fprintf(where,
           "netperf: recv_fore_stream: accept: errno = %d\n",
           errno);
   fflush(where);
   netperf_response.content.serv_errno = errno;
   send_response();
   print_cause(s_data);
   print_cause(clsock);  /* ?? */
   exit(1);
 }
 /* Now it's time to start receiving data on the connection. We will */
 /* first grab the apropriate counters and then start grabbing. */

 cpu_start(fore_stream_request->measure_cpu);

 /* The loop will exit when the timer pops, or if we happen to recv a */
 /* message of less than send_size bytes... */

 times_up = 0;
 start_timer(test_time + PAD_TIME);

 if (debug) {
   fprintf(where,"recv_fore_stream: about to enter inner sanctum.\n");
   fflush(where);
 }


 while (!times_up) {

   if ((len = read(clsock,
                   recv_ring->buffer_ptr,
                   message_size)) != message_size) {
     if ((len == -1) && (errno != EINTR)) {
       netperf_response.content.serv_errno = errno;
       send_response();
       print_cause(clsock);
       exit(1);
     }
     break;
   }
   messages_recvd++;
   bytes_received += len;
   recv_ring = recv_ring->next;
 }
```

```
if (debug) {
  fprintf(where,"recv_fore_stream: got %d messages.\n",messages_recvd);
  fflush(where);
}

/* The loop now exits due timer or < send_size bytes received. */

cpu_stop(fore_stream_request->measure_cpu,&elapsed_time);

if (times_up) {
  /* we ended on a timer, subtract the PAD_TIME */
  elapsed_time -= (float)PAD_TIME;
}
else {
  alarm(0);
}

if (debug) {
  fprintf(where,
          "recv_fore_stream: test ended in %f seconds.\n",elapsed_time);
  fflush(where);
}


/* We will count the "off" message that got us out of the loop */
bytes_received = (messages_recvd * message_size) + len;

/* send the results to the sender                              */

if (debug) {
  fprintf(where,
          "recv_fore_stream: got %d bytes\n",
          bytes_received);
  fflush(where);
}

netperf_response.content.response_type     = FORE_STREAM_RESULTS;
fore_stream_results->bytes_received        = bytes_received;
fore_stream_results->messages_recvd        = messages_recvd;
fore_stream_results->elapsed_time  = elapsed_time;
if (fore_stream_request->measure_cpu) {
  fore_stream_results->cpu_util    = calc_cpu_util(elapsed_time);
}
else {
  fore_stream_results->cpu_util    = -1.0;
}

if (debug > 1) {
  fprintf(where,
          "recv_fore_stream: test complete, sending results.\n");
  fflush(where);
}

send_response();

if( close(clsock) < 0) {
  perror("close clsock");
  exit(1);
}
close(s_data);

}

/*----------------------------------------------------------------------*/
/* this routine implements the send side (netperf) of a FORE_RR test    */

int send_fore_rr(remote_host)
char    remote_host[];
{
        /* esta rotina nao foi utilizada nos experimentos */
}

/*----------------------------------------------------------------------*/
/* this routine implements the receive side (netserver) of a FORE_RR test */
```

```
int
recv_fore_rr()
{
       /* esta rotina nao foi utilizada nos experimentos */
}

/*--------------------------------------------------------------------------*/

void
print_fore_usage()
{
  printf("%s",fore_usage);
  exit(1);
}

/*--------------------------------------------------------------------------*/

void
scan_fore_args(argc, argv)
     int       argc;
     char      *argv[];

{
#define FORE_ARGS "hm:M:r:s:S:x"
  extern int   optind, opterrs;  /* index of first unused arg      */
  extern char  *optarg;          /* pointer to option string*/

  int          c;

  char
    arg1[BUFSIZ],  /* argument holders              */
    arg2[BUFSIZ];

  /* the first thing that we want to do is set all the defaults for */
  /* the test-specific parms. */

  init_test_vars();
  init_done = 1;

  /* Go through all the command line arguments and break them */
  /* out. For those options that take two parms, specifying only */
  /* the first will set both to that value. Specifying only the */
  /* second will leave the first untouched. To change only the */
  /* first, use the form "first," (see the routine break_args.. */

  while ((c= getopt(argc, argv, FORE_ARGS)) != EOF) {
    switch (c) {
    case '?':
    case 'h':
      print_fore_usage();
      exit(1);
    case 'r':
      /* set the request/response sizes */
      break_args(optarg,arg1,arg2);
      if (arg1[0])
       req_size = atoi(arg1);
      if (arg2[0])
       rsp_size = atoi(arg2);
      break;
    case 'm':
      /* set the send size */
      send_size = atoi(optarg);
      break;
    case 'M':
      /* set the recv size */
      recv_size = atoi(optarg);
      break;

    case 's':
      break_args(optarg,arg1,arg2);
      if (arg1[0])
       lss_size = convert(arg1);
      if (arg2[0])
       lsr_size = convert(arg2);
```

```
      break;
    case 'S':
      break_args(optarg,arg1,arg2);
      if (arg1[0])
        rss_size = convert(arg1);
      if (arg2[0])
        rsr_size = convert(arg2);
      break;

    case 'x':
      /* mostra apenas a primeira linha de resultados */
      linha = 1;
      break;

    };
  }
}
#endif /* DO_FORE */
```

Anexo B – Código de nettest_atmn.h

Este anexo contém o arquivo `nettest_atmn.h`, que é o arquivo de cabeçalho para o código `nettest_atmn.c`. Ele contém as estruturas de dados que transportam informações específicas para os testes em ambiente ATM nativo. Estas estruturas são embutidas nas estruturas `netperf_request{}` e `netperf_response{}`. Estas últimas são transmitidas através da conexão de controle do aplicativo Netperf, como foi discutido no Capítulo 4, seção 4.2.1.

```
/*        Copyright (C) 1993, Hewlett-Packard Company     */

 /* This file contains the test-specific definitions for netperf's BSD */
 /* sockets tests */

#define PAD_TIME 2

struct  fore_stream_request_struct {
  int    send_buf_size;
  int   recv_buf_size;
  int   message_size;
  int   recv_alignment;
  int   recv_offset;
  int   checksum_off;
  int   measure_cpu;
  float cpu_rate;
  int   test_length;
 };

 struct  fore_stream_response_struct {
   int   recv_buf_size;
   int   send_buf_size;
   int   measure_cpu;
   int   test_length;
   int   server_asap;
   float cpu_rate;
   int   so_rcvavoid;   /* could the remote avoid receive copies? */
   int   so_sndavoid;   /* could the remote avoid send copies? */
 };

 struct  fore_stream_results_struct {
   int   messages_recvd;
   double       bytes_received;
   float elapsed_time;
   float cpu_util;
   int   num_cpus;      /* how many CPUs had the remote? */
 };
```

```
struct  fore_rr_request_struct {
  int   recv_buf_size; /* how big does the client want it   */
  int   send_buf_size;
  int   recv_alignment;
  int   recv_offset;
  int   send_alignment;
  int   send_offset;
  int   request_size;
  int   response_size;
  int   measure_cpu;   /* does the client want server cpu   */
  float cpu_rate;      /* do we know how fast the cpu is?   */
  int   test_length;   /* how long is the test?             */
};

struct  fore_rr_response_struct {
  int   recv_buf_size; /* how big does the client want it   */
  int   no_delay;
  int   measure_cpu;   /* does the client want server cpu   */
  int   test_length;   /* how long is the test?             */
  int   send_buf_size;
  int   server_asap;   /* connect to me here */
  float cpu_rate;              /* could we measure    */
  int   so_rcvavoid;   /* could the remote avoid receive copies? */
  int   so_sndavoid;   /* could the remote avoid send copies? */
};

struct fore_rr_results_struct {
  int   bytes_received;       /* ignored initially */
  int   recv_calls;    /* ignored initially */
  int   trans_received;       /* not ignored  */
  float elapsed_time;  /* how long the test ran */
  float cpu_util;      /* -1 if not measured */
  float serv_dem;      /* -1 if not measured */
  int   num_cpus;      /* how many CPUs had the remote? */
};
```

Referências Bibliográficas

[1] Black, U. *ATM - Foundation for Broadcast Networks.* New Jersey, Prentice-Hall, 1995.

[2] Ginsburg, D. *ATM Solutions for Enterprise Internetworking.* England, Addison-Wesley Longman Limited, 1996.

[3] Stallings, W. *Local and Metropolitan Area Networks.* 5.ed. New Jersey, Prentice-Hall, 1997.

[4] Comer, Douglas E. *Internetworking With TCP/IP Volume 1.* 3.ed. New Jersey, Prentice-Hall, 1995.

[5] Soares, Luiz Fernando G.; Lemos, G.; Colcher, S. *Redes de Computadores - Das LANs , MANs e WANs às Redes ATM.* 2.ed. Rio de Janeiro, Campus, 1995.

[6] Comitê Técnico ATM Forum, *LAN Emulation over ATM Version 1.0.* Especificação af-lane-0021.000, jan. 1995. Disponível em ftp://ftp.atmforum.com/pub/approved-specs/

[7] Fore Systems Inc. *ForeRunner ATM Switch.* Configuration Manual, Warrendale,PA, 1996.

[8] Comitê Técnico ATM Forum, *Multi-Protocol Over ATM Specification V1.0.* Especificação af-mpoa-0087.000, jul. 1997. Disponível em ftp://ftp.atmforum.com/pub/approved-specs/

[9] Laubach, M. *Classical IP and ARP over ATM.* RFC 1577, jan. 1994.

[10] Laubach, M.; Halpern, J. *Classical IP and ARP over ATM.* RFC 2225, abr. 1998.

[11] Braden, R.; Zhang, L.; Berson, S. Herzog, S.; Jamin, S. *Resource Reservation Protocol (RSVP) - Version 1, Functional Specification.* RFC 2205, set.1997.

[12] Schulzrinne, H.; Casner, S.; Fredrick, R.; Jacobson, V. *RTP: A Transport Protocol for Real-Time Applications.* RFC 1889, jan. 1996.

[13] Fore Systems Inc. *ForeRunner HE/200E/LE ATM Adapters for the PC.* User's Manual, Warrendale,PA, fev. 1999.

[14] Fore Systems Inc. *ForeRunner ATM Switch.* User's Manual, Warrendale,PA, 1996.

[15] Network Computing Services Inc. - Advanced Networking Group, *HARP - Host ATM Research Plataform*, Software e Documentação atualmente disponíveis em: http://www.msci.magic.net/harp/

[16] Lehey, G. *Complete FreeBSD Manual.* 2.ed. Walnut Creek Cdrom Books, 1996.

[17] Anixter Inc., *Ethernet Switching.* White Paper, 1997. Disponível em http://www.anixter.com/techlib/whiteppr/network/anixeswp.htm.

[18] Postel. J. *Internet Protocol.* STD5, RFC 791, set. 1981.

[19] Postel, J. *User Datagram Protocol.* STD6, RFC 768, ago. 1980.

[20] Postel, J. *Transmission Control Protocol.* STD7, RFC 793, set. 1981.

[21] Andrikopoulos, I.; Örs, T.; Matijasevic, M.; Leitold, H.; Posch, R. A Study of TCP/IP Throughput Performance in Heterogeneous ATM Local Area Networks. In: Kouvatsos, D., ed. *ATM Networks: Performance Modeling and Analysis.* London, Chapman & Hall, 1997. p.447-465.

[22] McKusick, M.; Bostic, K.; Karels, M. *The Design and Implementation of the 4.4BSD Operating System.* Addison-Wesley, 1996.

[23] Wright, G.R.; Stevens, W.R. *TCP/IP Illustrated Volume 2*. Massachusetts, Addison-Wesley Publishing Company, 1995.

[24] Clark, D.D.; Jacobson, V.; Romkey, J.; Salwen, H. An Analysis of TCP Processing Overhead. *IEEE Communications Magazine*, n.6, p.23-29, jun. 1989.

[25] Stevens, W.R. *UNIX Networking Programming*. New Jersey, Prentice-Hall, 1990.

[26] Luciani, J.; Armitage, G.; Halpern, J.; Doraswamy, N. *Server Cache Synchronization Protocol (SCSP)*. RFC 2334, abr. 1998.

[27] The Open Group, Techical Standard: Networking Services (XNS) Issues 5.2 Draft 2.0. 1999.

[28] Comitê Técnico ATM Forum, *Native ATM Services: Semantic Description Version1.0*. Especificação af-saa-0048.000, fev. 1996.

[29] Atkinson, R. *Default IP MTU for use over ATM AAL5*. RFC 1626, mai. 1994.

[30] Peres, M. *et. al. ATM Signaling Support for IP over ATM*. RFC 1755, fev. 1995.

[31] Jain, A.; Keshav, S. Native-Mode ATM in FreeBSD: Experience and Performance. AT&T Bell Laboratories, New Jersey, abr. 1996. Disponível em: http://www.cs.cornell.edu/Info/People/skechav/papers.html.

[32] Ahuja, R.; Keshav, S.; Saran, H. Design, Implementation and Performance Measurement of a Native-Mode ATM Transport Layer (Extended Version). *IEEE/ACM Transactions of Networking*, v.4, n.4, ago. 1996.

[33] Afifi, H.; Bonjour, D.; Elloumi, O. TCP over Non-existent IP for ATM Networks. In: *Proceedings of JENC'7*, mai. 1996.

[34] Almesberger, W. *High-speed ATM networking on low-end computer.* Laboratoire de Réseaux de Communication, Lausanne, Suíça, ago. 1995. Disponível em : ftp://ftp.lirmm.fr/pub/odbi/almesberger1.ps.gz

[35] Comer, D.E.; Lin, J.C. TCP Buffering And Performance Over An ATM Network. *Internetworking: Research and Experience*, v.6, n.1, p.23-29, Junho de 1989.

[36] Luckencbach, T.; Rupplet, R.; Schulz, F. Performance Experiments within Local ATM Networks. In: *Proceedings of the 12th European Fiber Optic Communication and Networks Conference*, Heidelberg, Germany, Junho de 1994. Disponível em: ftp://ftp.fokus.gdm.de/pub/docs/nthp/tlrrfs-PERFEX-EFOC94.ps.Z

[37] Fouquet, Y.A.; Schneeman, R.D.; Mink, A. ATM Performance Measurement. In: Proceedings of International Conferece on Telecommunication, Distribution, Parallelism, Londe Les Maures, France, Junho de 1996. p.63-75.

[38] Information Networks Division, *Netperf: A Network Performance Benchmark, Revision 2.1.* Hewlett-Packard, 1996. Atualmente disponível em : ftp://ftp.cup.hp.com/dist/networking/benchmarks/netperf/archive/netperf-2.1pl3.tar.gz.

[39] Matijasevic, M.; Posch, R.; Pucher, F. ATM LAN Throughput Measurements and Evaluation for BSD socket and FSI Application Programming Interface. In: Proceedings Of The 3rd Workshop On Performance Modeling And Evaluation Of Atm Networks, Ilkley, West Yorkshire, UK, 1995. p.64/1-64/10.

[40] SILLER Jn., C.A.; SHAFI, M., ed. *SONET/SDH: A Sourcebook of Synchronous Networking*. New York, IEEE PRESS, 1996.

[41] Dharanikota, S.; Maly, K.; Overstreet, C.M. *Performance evaluation of TCP(UDP)/IP over ATM networks*. Norfolk, 1994. Computer Science Department, Old Dominion University. Relatório Técnico disponível em: http://www.cs.odu.edu/~techrep/techreports/TR_94_23.ps.Z

[42] Moldekev, K.; Gunningberg, P. How a Large ATM MTU Causes Deadlocks in TCP Data Transfers. *IEEE/ACM Transactions on Networking*, v.3, n.4, ago. 1995.

[43] Armitage, G.J.; Adams, K.M. How Inefficient is IP over ATM anyway? /Apresentado em Australian Broadband Switching Services Symposium, Wollongong, jul. 1993/ Atualmente disponível em http://www.ee.mu.oz.au/papers/kma/html.stuff/gja.html

[44] Wolman. A.; Voelker, G.; Thekkath, C.A. *Latency Analysis of TCP on an ATM Network*. Department of Computer Science and Engineering, University of Washington, Seattle,WA, 1993. Atualmente disponível em: ftp://ftp.lirmm.fr/pub/odbi/wolman.ps.gz

[45] Morris, R.; Kung, H.T. Detailed Behavior of TCP over ATM. Apresentado ao GLOBECOM'95 (IEEE Global Telecommunications Conference), Singapore, nov. 1995. Disponível em: ftp://ftp.lirmm.fr/pub/odbi/rtm.ps.gz

[46] Cavanaugh, J.D. *Protocol Overhead in IP/ATM Networks*. Minnesota Supercomputer Center Inc., 1994. Disponível em ftp://ftp.lirmm.fr/pub/odbi/cavanaugh2.ps.gz

[47] Clark, D. *Window and Acknowledgment Strategy in TCP*. RFC 813, jul. 1982.

[48] Heinanen, J. *Multiprotocol Encapsulation over ATM Adaptation Layer 5*. RFC 1483, jul. 1993.

[49] Nagle, J. *Congestion Control in IP/TCP Internetworkings*. RFC 896, jan. 1984.

[50] Braden, R.; Clark, D.; Shenker, S. *Integrated Services in the Internet*. RFC 1633 IETF; jun. 1994.

[51] Nichols, K.; Blake, St. *Differentiated Services Operational Model and Definitions* (draft-nicols-dsopdef-00.txt), fev. 1998.

[52] Blake, St.; Black, D.; Carlson, M.; Davis, E.; Wang, Z.; Weiss, W. *An Architecture for Differentiated Services*;; (draft-ietf-diffserv-arch-02.txt), out. 1998.

[53] Cranor, Charles D. Integrating ATM Networking into BSD. Departmente of Computer Science, Washington University, St. Louis, ago. 1998.