Universidade de São Paulo
Instituto de Física de São Carlos
Departamento de Física e Informática

TRÁFEGO MULTIMÍDIA COM QoS EM
REDES DEFLEXIVAS E REDES HÍBRIDAS:
SIMULAÇÃO E IMPLEMENTAÇÃO

André Muezerie

Tese apresentada ao Instituto de Física de São Carlos, da Universidade de São Paulo, para obtenção do título de Doutor em Ciências: Física Aplicada.

Orientador: *Prof. Dr. Jan Frans Willem Slaets*

SÃO CARLOS
2005

Dedico este trabalho à Márcia,
que sempre tem me confortado
nos momentos difíceis.

Agradecimentos

Ao prof. Dr. Jan F. W. Slaets, por todo apoio e compreensão dispensados durante a orientação deste trabalho.

Ao prof. Dr. Pawel Gburzynski e ao prof. Dr. Ioanis Nikolaidis, sem os quais meu doutorado sanduíche na Universidade de Alberta não teria sido possível.

Ao prof. Dr. Gonzalo Travieso por sua constante disposição em bem atender a ocasionais pedidos de ajuda.

Aos professores Dr. José Nelson Amaral e Dr. Mário A. Nascimento, pela ajuda recebida na minha chegada ao Canadá.

Aos demais professores do grupo que tornaram minha convivência no Instituto mais agradável: Guilherme, Totó, Atílio e Teresa.

Aos colegas do grupo: Elô, Thatyana, Francisco, Paulino, Zem e André de Angelis.

A todas as outras pessoas que de uma forma ou outra também contribuíram.

Agradeço à CAPES pelo suporte financeiro
concedido tanto durante meu doutorado quanto
durante meu estágio de doutorado no exterior.

Sumário

1	Introdução	1
1.1	Motivações	1
1.2	Internet e QoS	2
1.2.1	Características de QoS	3
1.2.2	Técnicas para obtenção de QoS	4
1.2.3	QoS com ATM	5
1.3	Objetivos	6
1.4	Organização	6
2	Conceitos básicos	7
2.1	Ethernet	7
2.2	ATM	9
2.3	Camada física da arquitetura ATM	13
2.4	Camada ATM	15
2.5	Camada de adaptação ATM (AAL)	16
2.6	Camada de adaptação AAL5	17
2.7	Protocolo Internet (IP) versão 4	19
2.8	User Datagram Protocol (UDP)	21
2.9	Real-Time Transport Protocol (RTP)	21
3	Trabalhos e padrões relacionados	25
3.1	CIF	25
3.2	ASHMEN	26
3.3	CAFE	28
3.4	PACE	29
3.5	Padrão IEEE 802.1Q	30
3.6	Padrão IEEE 802.1p	31
3.7	Type of Service (ToS) no IPv4	32
3.8	Serviços Integrados	33
3.9	Serviços Diferenciados	34
3.10	MPLS	35
3.11	IPv6	36
4	Gerenciador de tráfego ATM e Ethernet	39
4.1	Modelo proposto	39
4.2	Implementação	40
4.3	Resultados experimentais	46
4.3.1	Perda de pacotes	47
4.3.2	Latências de comunicação	49

4.3.2.1	Programa latency	49
4.3.2.2	Latências com canal UBR entre gerenciadores	52
4.3.2.3	Latências com canal CBR entre gerenciadores	58
4.3.3	Comportamento do tráfego do VLC	63
4.3.3.1	Arquivo de vídeo em disco	63
4.3.3.2	Vídeo em tempo real	65
4.4	Conclusões	65
5	Simulador de redes deflexivas	67
5.1	Redes deflexivas	67
5.2	Modelo empregado	70
5.3	Implementação	72
5.4	Resultados	72
5.4.1	Topologia retangular	72
5.4.2	Topologia triangular	79
5.4.3	Topologias irregulares	81
5.5	Conclusões	83
6	Trabalhos futuros	85
7	Conclusões gerais	87
A	Hardware utilizado	99
A.1	A Interface MultiKron II	99
A.1.1	Introdução	99
A.1.2	Registro de Eventos	100
A.2	Interface ATM ForeRunner LE 155	101
A.3	Chaves ATM	102
A.3.1	Chave ATM ForeRunner LE 155	103
A.3.2	Chave ATM ForeRunner ASX-200BX	103
A.4	Chave Fast Ethernet	104
A.5	Placa de aquisição WinCast TV	104
B	Ferramentas utilizadas	107
B.1	GCC	107
B.2	STL	107
B.3	Threads	108
B.4	GNU Common C++	109
B.5	AutoTools	110
B.5.1	Automake	110
B.5.2	Autoconf	111
B.6	Glade2	111
B.7	Glade--	112
B.8	gtkmm	112
B.9	libsigc++	113
B.10	KDevelop	113
B.11	Doxygen	113
B.12	VLC	115

B.13	Netperf	118
B.14	Kernel do Linux	118
B.14.1	Suporte ao ATM no Linux	118
B.14.2	Habilitando o suporte ao ATM no <i>kernel</i>	119
B.14.3	Pseudodiretório /proc	120
B.15	ATM Tools	121
C	Modelo de comunicação e protocolos de rede utilizados	123
C.1	Fases de uma conexão	123
C.1.1	Fases para <i>sockets</i> PVC	124
C.1.2	Fases para <i>sockets</i> SVC	125
C.2	Descritores de endereço	126
C.3	Endereçamento PVC	127
C.3.1	Endereços PVC especiais	127
C.3.2	Representação textual	128
C.4	Endereçamento SVC	128
C.4.1	Endereçamento do sistema final	129
C.5	Descritores de QoS	130
C.5.1	Parâmetros de tráfego	130
C.5.2	Classes de tráfego	131
C.5.3	Taxa de células de pico	132
C.5.4	Variação da latência das células	133
C.5.5	Tamanho do SDU	133
C.6	Rotinas de biblioteca	133
C.6.1	text2atm	133
C.6.2	atm2text	134
C.6.3	atm_equal	135
C.6.4	sdu2cell	135
C.7	Preparação da conexão	136
C.7.1	Criação do <i>socket</i>	136
C.7.2	Alocação e inicialização dos descritores de endereço	137
C.7.3	Alocação e inicialização dos descritores de QoS	138
C.7.4	Atribuição dos parâmetros de QoS	138
C.7.5	Exemplo	138
C.8	Criação da conexão	139
C.8.1	Chamadas de sistema para criação de um PVC	139
C.8.2	Chamadas de sistema para criação de um SVC ativo	140
C.8.3	Chamadas de sistema para criação de um SVC passivo	140
C.8.4	Exemplo	141
C.9	Estado da conexão	141
C.10	Término da conexão	142
C.10.1	Desligamento de uma conexão	142
C.10.2	Fechamento	142
C.11	Resumo do controle de conexão	142
D	Firewire	145
E	USB	147

F	Mecanismo CSMA/CD	149
G	Condicionador de tráfego	151

Lista de Figuras

2.1	Formato do datagrama original do <i>Ethernet</i>	7
2.2	Formato de uma célula ATM	9
2.3	Formato do cabeçalho das células ATM UNI e NNI	10
2.4	Estrutura do campo PT do cabeçalho da célula ATM	11
2.5	Hierarquia de canais em uma rede ATM	13
2.6	Comutação de caminhos e circuitos realizada no interior de uma chave ATM	14
2.7	Estrutura da camada AAL5	18
2.8	Formato do CPCS-PDU da camada AAL5	19
2.9	Formato do pacote IPv4	20
2.10	Formato do campo <i>Flags</i> do IPv4	20
2.11	Formato do datagrama UDP	22
2.12	Posição do RTP na pilha de protocolos	22
2.13	Cabeçalho do protocolo RTP	22
3.1	Topologia e pilha de protocolos na arquitetura CIF	26
3.2	Topologia e pilha de protocolos na arquitetura ASHMEN	27
3.3	Arquitetura baseada no CAFE	29
3.4	Formato do datagrama <i>Ethernet</i> compatível com o padrão IEEE 802.1Q	31
3.5	Campo ToS do cabeçalho do protocolo IP	33
3.6	Campo DS usado nos serviços diferenciados. Substitui o campo ToS do IPv4	34
3.7	Formato do datagrama MPLS	36
3.8	Formato do cabeçalho do IPv6	37
4.1	Topologia de rede com gerenciadores	40
4.2	Menu principal do programa controlador	41
4.3	Interface gráfica de programa capaz de criar novos canais	42
4.4	Janela que permite a seleção do gerenciador a receber a solicitação de criação de novo canal	42
4.5	Diagrama ilustrando o esquema de funcionamento dos gerenciadores	44
4.6	Listagem dos canais ativos gerada pela opção " <i>Print list of tunnels</i> " do programa controlador	45
4.7	Visualização das estatísticas de um canal em dado instante através da opção " <i>Print Statistics of tunnel</i> " do programa controlador	45
4.8	Programa que permite o monitoramento contínuo das estatísticas de um canal	46
4.9	Janela que permite a seleção do gerenciador a ser contatado, do canal a ser monitorado e do intervalo de tempo entre uma amostragem e outra	47

4.10	Topologia de rede utilizada na avaliação da perda de pacotes	47
4.11	Montagem experimental para medida da latência no envio direto e indireto	50
4.12	Opções do programa <i>latency</i> , usado para medir a latência na comunicação	51
4.13	Representação gráfica das opções do programa <i>latency</i>	51
4.14	Posição no programa <i>latency</i> nos modos cliente e servidor onde é realizado o registro de tempo através de uma escrita em memória . . .	51
4.15	Latência média para mensagens com comprimento na faixa de 32 a 1024 <i>bytes</i> , transmitidas de modo direto em rajadas de 5 mensagens a cada 20 <i>ms</i> via <i>Fast Ethernet</i>	52
4.16	Latência média para mensagens com comprimento na faixa de 32 a 1024 <i>bytes</i> , transmitidas de modo indireto em rajadas de 5 mensagens a cada 20 <i>ms</i> via canal UBR	53
4.17	Latência para várias mensagens de 32 <i>bytes</i> enviadas em modo direto e indireto via canal UBR	54
4.18	Tempos obtidos com o aplicativo <i>ping</i>	54
4.19	Histograma das latências de 25000 mensagens de 32 <i>bytes</i> enviadas em rajadas de 5 mensagens em modo direto via <i>Fast Ethernet</i>	55
4.20	Histograma do atraso do recebimento das mensagens em relação ao instante do recebimento da primeira mensagem de cada rajada	56
4.21	Histograma do atraso do envio das mensagens em relação ao instante do envio da primeira mensagem de cada rajada	56
4.22	Histograma das latências de 25000 mensagens de 32 <i>bytes</i> enviadas em rajadas de 5 mensagens em modo indireto via canal UBR	57
4.23	Latência para várias mensagens de 1024 <i>bytes</i> enviadas em rajadas de 5 mensagens em modo direto e indireto via canal UBR	58
4.24	Histograma das latências de 25000 mensagens de 1024 <i>bytes</i> enviadas em rajadas de 5 mensagens em modo direto via <i>Fast Ethernet</i>	59
4.25	Histograma das latências de 25000 mensagens de 1024 <i>bytes</i> enviadas em rajadas de 5 mensagens em modo indireto via canal UBR	59
4.26	Histograma das latências de 25000 mensagens de 32 <i>bytes</i> enviadas em rajadas de 5 mensagens em modo indireto via canal CBR com capacidade para 5000 células por segundo com <i>link</i> ocioso	60
4.27	Histograma das latências de 25000 mensagens de 32 <i>bytes</i> enviadas em rajadas de 5 mensagens em modo indireto via canal CBR com capacidade para 5000 células por segundo com <i>link</i> totalmente ocupado	61
4.28	Histograma das latências de 25000 mensagens de 1024 <i>bytes</i> enviadas em rajadas de 5 mensagens em modo indireto via canal CBR com capacidade para 5000 células por segundo com <i>link</i> ocioso	62
4.29	Histograma das latências de 25000 mensagens de 1024 <i>bytes</i> enviadas em rajadas de 5 mensagens em modo indireto via canal CBR com capacidade para 5000 células por segundo com <i>link</i> totalmente ocupado	63
4.30	Tamanho das mensagens transmitidas pelo VLC em função do instante de chegada da mensagem usando como fonte de vídeo um arquivo MPEG-I armazenado em disco local	64

4.31	Zoom da Figura 4.30 ilustrando a separação temporal das mensagens dentro de uma mesma rajada	64
4.32	Tamanho das mensagens transmitidas pelo VLC em função do instante de chegada da mensagem com vídeo capturado em "tempo real" com uma filmadora Canon	65
4.33	Zoom da Figura 4.32 ilustrando a separação temporal das mensagens dentro de uma mesma rajada	66
5.1	Topologia da rede MSN	68
5.2	Topologia retangular	73
5.3	Taxa de perda e atraso em topologia retangular com <i>threshold</i> igual a metade do tamanho do <i>reassembly buffer</i>	74
5.4	Fração de sessões VoIP com QoS aceitável e atraso fim-a-fim em topologia retangular com 200 sessões VoIP simultâneas e <i>threshold</i> igual a 4 <i>slots</i>	75
5.5	Fração de sessões VoIP com QoS aceitável e latência fim-a-fim em topologia retangular com 350 sessões VoIP simultâneas e <i>threshold</i> igual a 4 <i>slots</i>	76
5.6	Fração de sessões VoIP com QoS aceitável e latência fim-a-fim em topologia retangular com <i>threshold</i> igual a 4 <i>slots</i>	77
5.7	Fração de sessões VoIP com QoS aceitável e atraso fim-a-fim em topologia retangular com <i>threshold</i> igual a 20 <i>slots</i>	78
5.8	Fração de sessões VoIP com QoS aceitável em função do período entre atualizações (LSUP) para diferentes coeficientes de frequência (FC) na topologia retangular com 350 sessões simultâneas e roteamento QoS	79
5.9	Topologia triangular	80
5.10	Fração de canais VoIP com QoS aceitável e atraso fim-a-fim em topologia triangular	80
5.11	Fração das sessões VoIP com QoS aceitável em função do LSUP para diferentes coeficientes de frequência (FC) na topologia triangular com 100 sessões simultâneas e roteamento QoS	81
5.12	Fração de sessões VoIP com QoS aceitável e latência fim-a-fim em topologia irregular com 35 nós	82
A.1	Interface MultiKron II para barramento PCI	100
A.2	Relógio central utilizado pelas placas MultiKron II com botão de reset simultâneo	101
A.3	Interface de rede utilizando o <i>chip</i> NICStAR	103
A.4	Interface WinCast TV da Hauppauge baseada no <i>chip</i> BT848	105
B.1	Rotina da API POSIX responsável pela criação de uma nova <i>thread</i>	109
B.2	Janela principal do Glade e janela de aplicação sendo desenvolvida	111
B.3	Janela com <i>widgets</i> e janela de edição de propriedades do Glade	112
B.4	<i>Screenshot</i> do KDevelop	114
B.5	Painel de controle do aplicativo VLC	115
B.6	Janela de seleção de origem do conteúdo multimídia a ser exibido	116
B.7	Janela do aplicativo VLC para configuração da forma de transmissão de conteúdo multimídia	117

B.8	Opções de <i>drivers</i> para placas ATM no <i>kernel</i> 2.4.24	120
B.9	Resumo das opções habilitadas no <i>kernel</i> 2.4.24 para obter o suporte ao ATM em Linux	121
C.1	Diagrama de estados para a abertura de um <i>socket</i> PVC	124
C.2	Ciclo de vida de um <i>Active open</i> de um <i>socket</i> SVC	125
C.3	Diagrama de estados para um <i>Passive open</i> de um <i>socket</i> SVC	126
C.4	Estrutura geral do descritor de endereço ATM	127
C.5	Estrutura de um endereço PVC	127
C.6	Endereçamento SVC	129
C.7	Estrutura de endereço SVC	130
C.8	Estrutura do descritor de QoS	130
C.9	Estrutura usada para armazenar parâmetros de tráfego	131
C.10	Parâmetros utilizados em cada classe de tráfego	132
C.11	Opções de processamento na conversão da representação textual de um endereço ATM para sua codificação binária correspondente	134
C.12	Opções de processamento na conversão de um endereço ATM binário	135
C.13	Opções de processamento na comparação de dois endereços ATM	135
C.14	Estrutura de dados usada como descritor de conexão PVC	137
C.15	Estrutura de dados usada como descritor de conexão SVC	137
C.16	Trecho de código que ilustra o uso do descritor de QoS	139
C.17	Trecho de código que ilustra o <i>two-phase connect</i>	141
G.1	Mecanismo de condicionamento de tráfego usado para certificar que o tráfego segue as regras estabelecidas pelo TCA	151

Lista de Tabelas

2.1	Tabela de comutação de caminhos e circuitos virtuais de uma chave ATM	13
2.2	Divisão de camadas da arquitetura ATM	14
2.3	Classes de Serviço originais mantidas pela AAL	16
3.1	Mapeamento das prioridades do usuário nas classes de tráfego sugerido pelo padrão 802.1D	32
4.1	Tabela que mostra a perda percentual de pacotes através de um canal UBR e CBR entre gerenciadores em situações de tráfego diversas . . .	48

Lista de Acrônimos

AAL	<i>ATM Adaptation Layer</i>
ABR	<i>Available Bit Rate</i>
ACM	<i>Association for Computer Machinery</i>
AD	<i>Attachment Device</i>
API	<i>Application Program Interface</i>
ASCII	<i>American Standard Code for Information Interchange</i>
ASHMEN	<i>Atm over SHared MEdia Networks</i>
ATM	<i>Asynchronous Transfer Mode</i>
B	<i>Bytes</i>
bps	<i>bits por segundo</i>
BSD	<i>Berkeley Software Distribution</i>
BSDI	<i>Berkeley Software Design Inc.</i>
CAFE	<i>Cut-through ATM based Forwarding Engine</i>
CBR	<i>Constant Bit Rate</i>
CDV	<i>Cell Delay Variation</i>
CFI	<i>Canonical Format Indicator</i>
CIF	<i>Cells In Frames</i>
CLIP	<i>Classical IP</i>
CLP	<i>Cell Loss Priority</i>
CoS	<i>Class of Service</i>
CPCS	<i>Common Part Convergence Sublayer</i>
CPI	<i>Common Part Indicator</i>
CPU	<i>Central Processing Unit</i>
CRC	<i>Cyclic Redundancy Check</i>

CS	<i>Convergence Sublayer</i>
CSMA/CD	<i>Carrier Sense Multiple Access with Collision Detection</i>
DEC	<i>Digital Equipment Corporation</i>
DLL	<i>Data Link Layer</i>
DMA	<i>Direct Memory Access</i>
DS	<i>Differentiated Services</i>
DSCP	<i>Differentiated Services Code Point</i>
DTV	<i>Digital Television</i>
DWDM	<i>Dense Wavelength-Division Multiplexing</i>
EAT	<i>Ethernet ATM Transformation</i>
ELAN	<i>Emulated LAN</i>
EPFL	<i>École polytechnique Fédérale de Lausanne</i>
E/S	Entrada / Saída
FC	<i>Frequency Coefficient</i>
FE	<i>Fast Ethernet</i>
FEC	<i>Forwarding Equivalence Class</i>
FIFO	<i>First In, First Out</i>
fps	<i>frames por segundo</i>
Gbps	10^9 (Giga) bits por segundo
GFC	<i>Generic Flow Control</i>
HEC	<i>Header Error Check</i>
HLEN	<i>Header LENgth</i>
HTML	<i>Hypertext Markup Language</i>
ICMP	<i>Internet Control Message Protocol</i>
IDL	<i>Interface Definition Language</i>
IEE	<i>Institution of Electrical Engineers</i>
IEEE	<i>Institute of Electrical and Electronics Engineers</i>
IETF	<i>Internet Engineering Task Force</i>
ILMI	<i>Interim Local Management Interface</i>

I/O	<i>Input / Output</i>
IP	<i>Internet Protocol</i>
IPG	<i>Inter-Packet Gap</i>
ISA	<i>Industry Standard Architecture</i>
ISDN	<i>Integrated Services Digital Network</i>
ISO	<i>International Organization for Standardization</i>
ISP	<i>Internet Service Provider</i>
ITU	<i>International Telecommunication Union</i>
ITU-T	<i>ITU - Telecommunication Standardization Sector</i>
KB	10^3 (Kilo) Bytes
Kbps	10^3 (Kilo) bits por segundo
LAN	<i>Local Area Network</i>
LANE	<i>LAN Emulation</i>
LRC	<i>Laboratory for computer Communications and Applications</i>
LSP	<i>Label Switched Path</i>
LSUP	<i>Link State Update Period</i>
mA	<i>mili Ampères</i>
MAC	<i>Medium Access Control</i>
MAN	<i>Metropolitan Area Networks</i>
MB	10^6 (Mega) Bytes
Mbps	10^6 (Mega) bits por segundo
MDR	<i>Minimum Data Rate</i>
MIMD	<i>Multiple Instruction/Multiple Data</i>
MPEG	<i>Moving Picture Experts Group</i>
MPEG-TS	<i>Moving Picture Experts Group - Transport Stream</i>
MPLS	<i>Multi-Protocol Label Switching</i>
MPOA	<i>Multi-Protocol over ATM</i>
MSN	<i>Manhattan-Street Networks</i>
MTBF	<i>Mean Time Between Failures</i>

MTTR	<i>Mean Time To Repair</i>
MTU	<i>Maximum Transmission Unit</i>
nem	<i>Network Manipulator</i>
NIST	<i>National Institute of Standards and Technology</i>
NNI	<i>Network to Network Interface</i>
NSAP	<i>Network Service Access Point</i>
OAM	<i>Operation, Administration and Maintenance</i>
OHCI	<i>Open Host Controller Interface</i>
ONU	<i>Optical Network Unit</i>
OSI	<i>Open Systems Interconnection</i>
PC	<i>Personal Computer</i>
PCI	<i>Peripheral Component Interconnect</i>
PCM	<i>Pulse Code Modulation</i>
PCR	<i>Peak Cell Rate</i>
PDA	<i>Personal Digital Assistant</i>
PDF	<i>Portable Document Format</i>
PDR	<i>Peak Data Rate</i>
PDU	<i>Protocol Data Unit</i>
PE-MAC	MAC com tecnologia PACE
PFR	<i>Probabilistic Frequency Reduction</i>
PHB	<i>Per Hop Behavior</i>
PHP	<i>Hypertext Preprocessor</i>
PHY	<i>Physical Layer</i>
POSIX	<i>Portable Operating System Interface</i>
PPP	<i>Point to Point Protocol</i>
PT	<i>Payload Type</i>
PTI	<i>Payload Type Indicator</i>
PVC	<i>Permanent Virtual Circuit</i>
PVR	<i>Personal Video Recorder</i>

QoS	<i>Quality of Service</i>
R&R	<i>Routing and Relaying</i>
RFC	<i>Request For Comments</i>
RSVP	<i>Resource reSerVation Protocol</i>
RTCP	<i>Real-Time Transport Control Protocol</i>
RTF	<i>Rich Text Format</i>
RTP	<i>Real-Time Transport Protocol</i>
RTT	<i>Round Trip Time</i>
s	segundos
SAP	<i>Service Access Point</i>
SAR	<i>Segmentation And Reassembly</i>
SCR	<i>Sustainable Cell Rate</i>
SCSI	<i>Small Computer System Interface</i>
SDR	<i>Sustained Data Rate</i>
SDU	<i>Service Data Unit</i>
SEAL	<i>Simple Efficient Adaptation Layer</i>
SIG	<i>Special Interest Group</i>
SLA	<i>Service Level Agreement</i>
SLT	<i>Subscriber Line Terminator</i>
STL	<i>Standard Template Library</i>
SVC	<i>Switched Virtual Circuit</i>
TC	<i>Transmission Convergence</i>
TCA	<i>Traffic Conditioning Agreement</i>
TCP	<i>Transmission Control Protocol</i>
ToS	<i>Type of Service</i>
TTL	<i>Time To Live</i>
UBR	<i>Unspecified Bit Rate</i>
UDP	<i>User Datagram Protocol</i>
UHCI	<i>Universal Host Controller Interface</i>

UNI	<i>User to Network Interface</i>
USB	<i>Universal Serial Bus</i>
USBIF	<i>USB Implementers Forum</i>
UTOPIA	<i>Universal Test and Operations Physical Interface for ATM</i>
UTP	<i>Unshielded Twisted Pair</i>
UU	<i>User to User</i>
V	<i>Volts</i>
V4L	<i>Video for Linux</i>
VBR	<i>Variable Bit Rate</i>
VBR-NRT	<i>Variable Bit Rate - Non-Real Time</i>
VBR-RT	<i>Variable Bit Rate - Real Time</i>
VC	<i>Virtual Circuit, Virtual Channel, Virtual Connection</i>
VCC	<i>Virtual Channel Connection</i>
VCI	<i>Virtual Channel Identifier</i>
VLAN	<i>Virtual LAN</i>
VLAN ID	<i>Virtual LAN Identifier</i>
VLC	<i>VideoLan Client</i>
VME	<i>Versa Module Eurocard</i>
VoIP	<i>Voice over IP</i>
VP	<i>Virtual Path</i>
VPC	<i>Virtual Path Connection</i>
VPI	<i>Virtual Path Identifier</i>
VPN	<i>Virtual Private Network</i>
WAN	<i>Wide Area Network</i>
WDM	<i>Wavelength-Division Multiplexing</i>
WKS	<i>Widest-K-Shortest</i>

Resumo

Este trabalho tem por objetivo estudar aspectos de qualidade de serviço (QoS) na transmissão de áudio e vídeo por redes de grande abrangência geográfica (WANs). Para tal dois cenários bastante distintos foram estudados. O primeiro foi inspirado em uma situação comum em WANs: foi assumido que um *backbone* ATM é usado para interligar LANs baseadas em tecnologia *Ethernet*. Para este cenário foi feita uma proposta de um método de gerenciamento que permite que computadores de uma rede local (LAN) equipados com interfaces *Ethernet* possam criar canais dinamicamente através de um *backbone* ATM para tráfego de voz e vídeo. O modelo proposto define máquinas gerenciadoras responsáveis por receber tráfego de uma LAN, transmiti-lo através do *backbone* usando a capacidade da rede para prover QoS e repassar o tráfego para outra LAN baseada em uma tecnologia incapaz de prover tais parâmetros de QoS. Uma implementação do modelo foi avaliada para ilustrar a viabilidade do método proposto. No segundo cenário, como maneira alternativa de obter QoS em WANs baseadas em IP, foi mostrado através de extensas simulações computacionais que o princípio da deflexão pode ser usado com êxito para oferecer QoS a aplicações em tempo real. Nestas simulações foi usado como exemplo prático uma aplicação de voz sobre IP (VoIP).

Abstract

The aim of this project is to study quality of service (QoS) aspects in audio and video transmissions through wide area networks (WANs). In order to do so two distinct scenarios were studied. The first one was inspired by a common situation in WANs: it was assumed that an ATM backbone is used for interconnecting local area networks (LANs) based on Ethernet technology. For this scenario a management method was proposed. This method allows the Ethernet interfaced LAN computers to dynamically create channels through an ATM backbone for audio and video traffic. The proposed model describes gateways responsible for receiving traffic from a LAN, transmit it through the backbone using the network's capacity to provide QoS and pass the traffic again to another LAN based on a technology unable to provide QoS. An implementation of the model was evaluated to illustrate the viability of the proposed method. In the second scenario, as an alternative way to obtain QoS in IP based WANs, it was shown through extensive computer simulations that the principle of deflection can successfully be used to offer QoS to real-time applications. Synthetic traffic with typical voice over IP (VoIP) characteristics was generated and used in these simulations.

CAPÍTULO

1

Introdução

1.1 Motivações

A área de tecnologia vem rapidamente passando por grandes transformações. Novos produtos, ferramentas e serviços são criados da noite para o dia, fazendo com que, não raramente, a demanda por recursos de rede aumente em proporções antes inimagináveis, tanto no ambiente doméstico quanto empresarial e acadêmico. Usuários domésticos estão adquirindo, a preços cada vez menores, equipamentos digitais como câmeras de vídeo [122, 118], tocadores de mp3 portáteis, televisores de alta definição [109] e *Personal Video Recorders* (PVRs) [107, 70], entre outros. Com estes aparelhos surge de forma quase natural a necessidade de interligá-los em rede. Tal interligação, quando feita por distâncias maiores que poucos metros, deixa de ser trivial, requerendo esquemas para mapear a qualidade de serviço (QoS) [128] do IEEE 1394 (padrão de comunicação adotado na maioria destes dispositivos¹) no *Ethernet* (tecnologia de uso mais comum em LANs), hardware especial para fazer a conversão de uma tecnologia para outra [91], e em alguns casos, adaptadores físicos para que o mesmo cabeamento possa ser utilizado pelas distintas tecnologias [82]. Outra aplicação recente que tem recebido cada vez maior aceitação e que requer parâmetros específicos de QoS da rede é *Voice over IP* (VoIP)². Ainda no mercado doméstico, existe também uma tendência cada vez maior de conectar os aparelhos de videogame (*video game*) de última geração em rede para que participantes geograficamente afastados possam competir entre si em um mesmo jogo simultaneamente [18, 103, 21]. Dentre os aspectos de QoS que estes jogos requerem, um dos que mais se destaca é a baixa latência necessária para que as ações dos diversos participantes sejam rapidamente atualizadas nos ambientes virtuais dos demais jogadores [66, 57]. Já no mercado empresarial, é cada vez maior a depen-

¹A tecnologia Firewire é brevemente descrita na seção D.

²Dentre as principais razões da adoção de VoIP em detrimento do sistema de telefonia tradicional destacam-se o fato da tarifa para ligações VoIP de longa distância ser potencialmente mais barata e o fato das tarifas para ligações VoIP locais/regionais corresponderem a uma taxa mensal fixa, independentemente da quantidade e da duração das ligações.

dência da Internet para a realização de vendas (*eCommerce*) e de redes digitais para oferecer serviços de entretenimento [40]. Além disso, instituições de ensino e pesquisa têm demonstrado crescente interesse em prover conteúdo cada vez mais atrativo a clientes e alunos, com recursos multimídia e interação com o usuário, seja com objetivo de entreter ou ensinar (a exemplo do ensino à distância) [100].

Atualmente, a maior parte das redes locais é construída com chaves *Ethernet* que oferecem 10 ou 100 Mbps de largura de banda a cada link ponto-a-ponto, não apresentando gargalos representativos ao tráfego que por elas passa, permitindo inclusive o tráfego de voz e vídeo local sem grandes dificuldades. Já os *backbones*, por representarem os trechos de rede com maior índice de compartilhamento e por geralmente empregarem equipamentos bem mais caros que os utilizados nas LANs, são tipicamente mais explorados com relação à sua capacidade máxima.

Apesar dos *backbones* possuírem capacidade nominal suficiente para a transmissão de vídeo e áudio em tempo real, além dos dados que são normalmente transmitidos, torna-se necessário um gerenciamento eficiente que viabilize a reserva de recursos para a obtenção de garantias de QoS. Várias propostas já foram feitas em outros estudos (discutidos no capítulo 3) para integrar tecnologias de rede que oferecem QoS (como a ATM) com outras incapazes de oferecê-la (como a *Ethernet*), mas nenhuma reúne todas as boas características presentes na proposta descrita neste trabalho, que além de preocupar-se com os aspectos práticos da QoS, minimizou ineficiências na transmissão pelo *backbone* ATM, eliminando o *overhead* imposto pelos protocolos IP e superiores tipicamente empregados em redes *Ethernet*.

1.2 Internet e QoS

Quando a Internet foi criada, não havia a percepção da necessidade de oferecer garantias de QoS às aplicações. De fato, a operação do protocolo adotado como padrão na Internet (IP) baseia-se no princípio do "melhor esforço". É verdade que alguns mecanismos de priorização de tráfego foram definidos no IP, mas seu uso ainda não se popularizou.

A convergência de diversas mídias sobre uma mesma rede de transmissão levou ao reconhecimento que diferentes aplicações têm necessidades distintas. Por exemplo, aplicações multimídia suportam perdas relativamente pequenas nos dados, enquanto que transmissões de dados não. Já transmissões de dados são tolerantes a atrasos relativamente grandes, sendo que o mesmo não ocorre com transmissões de vídeo ou voz em tempo real. As características do tráfego de um aplicativo determinam o tipo de serviço a ser requisitado da rede. Estas características podem ser agrupadas para formar um determinado padrão de QoS.

Até hoje, mesmo com ampla disseminação de tecnologias que exigem resposta em tempo hábil como **VoIP**, vídeo-conferência, aprendizado à distância, ambientes virtuais distribuídos, etc., a questão da necessidade de oferecer garantias de QoS ainda é polêmica [137]. Uma opinião freqüente é que fibras e *Wavelength-Division Multiplexing* (**WDM**) irão tornar a largura de banda tão abundante e barata que automaticamente a QoS oferecida será boa. Outra opinião comum é que não importa quanta largura de banda se ofereça, novas aplicações serão criadas para consumí-la, e portanto ainda assim mecanismos serão necessários para prover QoS. Também existem dúvidas com relação ao desenvolvimento e adoção de redes totalmente ópticas baseadas em **WDM** (ou **DWDM**), pois a grande maioria das redes baseadas em fibras ainda hoje emprega equipamentos eletrônicos para fazer o roteamento e chaveamento dos datagramas, de forma que apesar da banda teórica oferecida pelas fibras ser enorme, ela ainda não pode ser plenamente usada pela limitação na velocidade dos conversores óptico-eletrônicos e demais circuitos eletrônicos das chaves e roteadores.

1.2.1 Características de QoS

Há várias características de QoS que podem ser consideradas, mas de maneira geral temos entre os aspectos importantes[124, 37, 135, 5]:

- *Largura de banda: Peak Data Rate (PDR), Sustained Data Rate (SDR) e Minimum Data Rate (MDR);*
- *Latência: latência fim-a-fim e latência de ida e volta (RTT);*
- *Jitter: variação da latência;*
- *Disponibilidade: uma porcentagem do tempo que se espera que o recurso esteja disponível (por exemplo 99%), tempo médio entre falhas (MTBF) ou tempo médio até reparo (MTTR);*
- *Taxa de erros: probabilidade de ocorrer um erro (por exemplo, 1 bit errado a cada 10^{12} bits transmitidos);*
- *Perda de pacotes;*
- *Entrega em ordem ou fora de ordem.*

Estas e várias outras características de QoS podem ser usadas para definir serviços na rede.

1.2.2 Técnicas para obtenção de QoS

Diversas técnicas têm sido desenvolvidas, e muitas vezes empregadas em conjunto, para se obter QoS em redes. As principais são:

Provisionamento em excesso: baseia-se em dimensionar a capacidade dos roteadores ou chaves, dos *buffers* e da largura de banda de tal forma que sempre haverá recursos disponíveis, de maneira que a demanda será sempre atendida. O maior problema com esta solução costuma ser o custo.

Bufferização: fluxos podem ser armazenados temporariamente em *buffers* nos equipamentos de rede e na máquina destino. A bufferização pode levar a um uso mais eficiente da largura de banda disponível na rede e a uma redução no *jitter*, mas o preço que se paga é expresso através do aumento na latência. Como já foi comentado, aplicações em tempo real não toleram atrasos muito grandes, o que significa que para estas aplicações pacotes atrasados demais não têm utilidade. Na prática, para estas aplicações latências altas demais resultam em perda de pacotes.

Suavização de tráfego (Traffic shaping): a emissão irregular de tráfego pode causar congestionamento na rede. A técnica de suavização de tráfego altera o padrão do tráfego, suavizando rajadas através de uma separação temporal mais uniforme entre os instantes de envio dos pacotes, para que a taxa instantânea não seja consideravelmente mais alta que a taxa média de transmissão. Os algoritmos *Leaky bucket* e *Token bucket* [5], e variações neles baseadas [77, 58, 38, 74, 59] são geralmente empregados para implementar a suavização de tráfego.

Reserva de recursos: o uso de uma rota específica para um fluxo (muitas vezes expresso através de um canal virtual) permite fazer uma reserva prévia dos recursos necessários para garantir que eles estejam disponíveis durante todo o tempo de vida deste fluxo.

Controle de admissão: antes de iniciar a transmissão de um novo fluxo, cada equipamento de rede na rota escolhida decide se admite ou rejeita o novo fluxo baseado na sua capacidade e nos compromissos já firmados com os demais fluxos. Mecanismos são usados [84, 78, 138] para converter (de forma geralmente não-trivial) as características desejadas pelas aplicações (largura de banda, latência, taxa aceitável de perdas) nos parâmetros realmente empregados pelos equipamentos (largura de banda, tamanho dos *buffers*, ciclos de CPU, etc.).

Policimento: monitoramento do tráfego é empregado para verificar se determinado tráfego obedece a um perfil previamente estabelecido em um *Service Level Agree-*

ment (SLA) [129, 24]. Caso contrário várias ações podem ser tomadas, entre elas:

- Suavização do tráfego;
- Marcação dos pacotes fora de conformância (para serem descartados adiante caso seja verificada situação de congestionamento);
- Descarte de pacotes.

Roteamento proporcional: ao invés de enviar todos os pacotes pelo melhor caminho até o destino, uma outra alternativa que tem sido usada para conseguir uma QoS mais alta é espalhar o tráfego para cada destino ao longo de vários caminhos diferentes. Um método simples é dividir o tráfego de forma proporcional à capacidade de cada *link* de saída. Este é também um princípio fundamental empregado em redes deflexivas, tema de estudo deste trabalho apresentado em detalhes no capítulo 5.

Escalonamento de pacotes: em um ambiente de múltiplos fluxos, sempre existe a possibilidade de alguns fluxos serem prejudicados por um fluxo mais intenso. Vários algoritmos de escalonamento foram criados para diminuir este efeito [46, 83, 139, 30].

1.2.3 QoS com ATM

Apesar de várias soluções terem sido propostas para tecnologias de rede sem conexão, em geral a QoS obtida em redes sem conexão apresenta granularidade grossa (aplica-se a um conjunto de fluxos), não sendo possível garantir a QoS para um fluxo específico. Por outro lado, a tecnologia *Asynchronous Transfer Mode* (ATM) permite garantir a QoS de cada fluxo de forma independente, possibilitando que voz, vídeo e dados trafeguem pelo mesmo meio físico sem problemas. Ela foi por isso vista como substituta a todas as outras tecnologias de redes [13]. Seu funcionamento se dá através da comutação de pequenos datagramas de tamanho fixo (chamados de células) por meio de canais virtuais (serviço orientado à conexão), sendo que tradicionalmente as redes de computadores realizam roteamento de datagramas de tamanho variável (serviço sem conexão), sem garantias de QoS. É justamente o pequeno tamanho fixo das células e a capacidade de reservar a banda necessária por meio das conexões virtuais que permite ao ATM garantir baixas latências às transmissões de voz e vídeo, o que é reconhecidamente um requisito fundamental em telefonia, área onde ele tem sido bastante aplicado.

A especificação do padrão ATM foi tão ampla que permitiria sua utilização nos três domínios de rede: LAN, MAN e WAN. De fato, no meio da década de 90 mui-

tos *backbones* foram sendo amplamente substituídos e expandidos com a utilização de equipamentos ATM [72], pois além desta ser vista como a tecnologia do futuro, também possuía grande largura de banda e escalabilidade. Infelizmente o ATM não teve o mesmo sucesso no *desktop*, apesar de haver soluções para emulação de redes locais baseadas em *Ethernet* ou *Token Ring* que mantinha compatibilidade com o protocolo IP.

1.3 Objetivos

Este trabalho tem por objetivo estudar aspectos de qualidade de serviço na transmissão de áudio e vídeo por WANs. Para tal dois cenários bastante distintos são estudados.

No primeiro propomos e implementamos um modelo baseado em *software* que permite que aplicações multimídia em máquinas conectadas a uma rede *Ethernet* usufruam das características de QoS de um *backbone* ATM. Um exemplo de aplicação deste modelo (que necessita de garantias de QoS) é a viabilização da operação remota do tomógrafo de ressonância magnética instalado na Santa Casa de Misericórdia de São Carlos [20] através de uma WAN ATM.

Nosso segundo estudo mostra através de simulações computacionais que, ao contrário do que se costuma acreditar, redes sem conexão empregando roteamento deflexivo podem também ser utilizadas na construção de redes capazes de transmitir conteúdo em tempo real sem perda significativa de qualidade.

De forma intencional ambas as soluções analisadas não exigem a criação de um equipamento novo nem exigem alterações no *hardware* tipicamente utilizado nas LANs e WANs. Isto faz com que ambas as soluções se tornem candidatas a serem utilizadas na prática (sem maiores dificuldades), se houver interesse.

1.4 Organização

O trabalho segue organizado da seguinte forma: no capítulo 2 são apresentados de forma resumida os conceitos necessários para o entendimento do trabalho. Quando conveniente, informações adicionais (de leitura opcional) são apresentadas no apêndice, como é o caso, por exemplo, da tecnologia *Firewire* (seção D) e do mecanismo *CSMA/CD* (seção F). No capítulo 3 são apresentados vários trabalhos relacionados, que por meio de diferentes maneiras tentam disponibilizar QoS (ou CoS) a máquinas conectadas em redes de tecnologias incapazes de oferecer garantias de QoS. Os capítulos 4 e 5 descrevem os trabalhos desenvolvidos em torno do gerenciador e do simulador de redes deflexivas, respectivamente. Por fim, o capítulo 7 apresenta as principais conclusões do trabalho.

CAPÍTULO 2 Conceitos básicos

Este capítulo apresenta de forma bastante resumida os conceitos necessários para o entendimento do trabalho. Quando conveniente, informações adicionais (de leitura opcional) são apresentadas no apêndice.

2.1 Ethernet

Em 1985 a [IEEE](#) definiu uma série de padrões para redes locais denominados de padrões IEEE 802 [10]. Eles foram muito bem aceitos e bastante usados na construção de LANs baseadas em datagramas, e o padrão IEEE 802.3 [1], popularmente chamado de "*Ethernet*", é a tecnologia predominante nas LANs hoje em dia.

O formato do datagrama *Ethernet* original é mostrado na Figura 2.1 e a função de cada campo está descrita a seguir:

Endereço destino: especifica um único destinatário em modo *unicast*, um grupo de destinatários no modo *multicast* ou o grupo formado por todas as máquinas do segmento em modo *broadcast*;

Endereço origem: informa o endereço da máquina transmissora do datagrama;

Comprimento: indica o tamanho do campo de dados;

CRC: provê detecção de erros causados pelo meio de transmissão ou por colisões.

Todo datagrama com um CRC inválido é descartado pelo *Medium Access Con-*

Figura 2.1: Formato do datagrama original do *Ethernet*.

trol (MAC) receptor sem processamento adicional e não há indicação de que o datagrama tenha sido descartado.

Originalmente, para realizar as comunicações o *Ethernet* usava um meio que era compartilhado entre todos os computadores da LAN, de forma que apenas um computador podia transmitir de cada vez. Para impedir que mais de uma máquina transmitisse simultaneamente foi usado um mecanismo de detecção chamado *Carrier Sense Multiple Access with Collision Detection* (CSMA/CD), descrito em maiores detalhes na seção F. Devido ao uso do mecanismo CSMA/CD era impossível usar toda a largura de banda teoricamente oferecida pela LAN. À medida que mais computadores eram adicionados à LAN, a probabilidade de ocorrência de colisões aumentava, diminuindo assim a eficiência no uso do meio de comunicação. Também devido ao mecanismo CSMA/CD, para uma taxa de transmissão fixa (em Mbps), a eficiência também era menor quando datagramas pequenos eram usados. Em condições adversas, eficiências menores que 30% podiam ser observadas [6].

Diversas técnicas foram empregadas para diminuir o problema das colisões, entre elas:

- uso de pontes, que propagam de um lado para outro apenas os datagramas necessários, diminuindo assim a probabilidade de ocorrência de colisões;
- quebra de grandes segmentos de rede em vários segmentos menores interligados por meio de roteadores;
- uso de chaves ao invés de *hubs*, pois elas quebram o conceito de um único meio compartilhado entre todas as máquinas, permitindo transmissões simultâneas entre máquinas distintas, muitas vezes inclusive em modo *full-duplex*¹.

Inicialmente o custo de uma chave era significativamente maior que o custo de um *hub*, mas com o tempo e a produção em larga escala os preços encolheram tanto que hoje a maior parte das LANs emprega chaves nas partes centrais, mas *hubs* ainda podem ser encontrados na periferia das redes.

Está claro que a tecnologia *Ethernet* não provê garantias de QoS, e diversas razões podem levar ao descarte de datagramas:

- o datagrama pode ter sido corrompido durante a transmissão pelo meio físico;
- uma ponte (ou chave) pode ter descartado o datagrama porque:
 - o *buffer* interno do equipamento encheu;

¹ *Full-duplex* corresponde à capacidade que um dispositivo tem de simultaneamente receber e transmitir dados.

- o tamanho do *payload* do datagrama é grande demais para a LAN para a qual ele se dirige;
- talvez o equipamento tenha sido forçado a descartar o datagrama para manter algum outro aspecto de QoS.

Maiores detalhes sobre *Ethernet* podem ser facilmente encontrados em livros sobre redes [130] e na Internet, como na enciclopédia virtual Wikipedia [11] por exemplo.

2.2 ATM

O *Asynchronous Transfer Mode* (ATM) é uma tecnologia de rede que se localiza nos níveis de enlace e físico do modelo *Open Systems Interconnection* (OSI) e baseia-se na transmissão de pequenas unidades de informação de tamanho fixo e formato padronizado denominadas células. Foi desenvolvida para ser usada com uma única rede de transporte para diversos serviços, tanto para redes locais como de longa distância, devendo ser muito escalável e apresentar baixos atrasos de comutação. Sua grande promessa é a integração eficiente de dados e tráfego multimídia com voz e vídeo através da garantia da QoS.

O tamanho das células é pequeno e fixo para facilitar a multiplexação de diferentes fluxos de informação através da rede. Isto permite que uma informação de maior importância possa ser priorizada em relação a uma informação de menor relevância sem a necessidade de remanejamentos complexos das unidades de informação.

Toda célula precisa possuir uma certa quantidade de informação para indicar onde ela deve ir, além de algumas informações adicionais descritas logo a seguir. Tais informações ficam armazenadas no cabeçalho, sendo que a informação de interesse a ser transmitida fica armazenada no restante da célula, chamado de *payload*. Entretanto, geralmente os dados do usuário são ainda encapsulados por outros protocolos, alguns dos quais serão descritos futuramente, fazendo com que nem sempre uma célula carregue 48 bytes de dados do usuário. O formato de uma célula ATM é mostrado na Figura 2.2 e o dos cabeçalhos das células para as interfaces *User to Network Interface* (UNI) e *Network to Network Interface* (NNI) são apresentados na Figura 2.3.

Figura 2.2: Formato de uma célula ATM.

Figura 2.3: Formato do cabeçalho das células ATM UNI e NNI.

O cabeçalho de 5 *bytes* impõe um *overhead* significativo se for considerado que o tamanho do *payload* de cada célula é de 48 *bytes*. Assumindo um tamanho fixo para o cabeçalho, para diminuir o *overhead* por ele causado seria necessário aumentar o tamanho da célula, mas isto tornaria mais difícil a multiplexação eficiente dos dados com conexões de tráfego em tempo real, pois com células maiores ficaria mais difícil garantir que o tráfego em tempo real não sofresse interrupções momentâneas durante a transmissão. Estes tamanhos para o cabeçalho e para os dados foram escolhidos como melhor compromisso entre *overhead* e QoS durante o processo de padronização do ATM.

As seguintes informações fazem parte dos cabeçalhos de uma célula ATM:

GFC (controle de fluxo genérico, do inglês *Generic Flow Control*): somente presente nas células UNI, sua função é controlar a quantidade de dados que entra na rede para evitar congestionamentos. Entretanto, essa funcionalidade está no campo **PT**, não sendo o campo GFC usado na prática.

VPI (identificador de caminho virtual, do inglês *Virtual Path Identifier*): identifica o caminho virtual que a célula deve seguir até o próximo elemento comutador da rede. Representa um mecanismo para direcionar um feixe de canais virtuais através da rede sem que seja necessário muito processamento. Este campo apresenta tamanhos diferentes na UNI e NNI.

VCI (identificador de canal virtual, do inglês *Virtual Channel Identifier*): identifica o canal virtual que a célula deve seguir até o próximo elemento comutador da rede.

PT (tipo de dados, do inglês *Payload Type*): também conhecido por **PTI** (identificador do tipo de dado, do inglês *Payload Type Indicator*). A estrutura de três bits deste campo está ilustrada na Figura 2.4. O bit A serve para indicar se a célula em questão é uma célula de usuário, transportando um *payload* que precisa ser chaveado adiante através da rede, ou uma célula de gerenciamento (célula **OAM**),

Figura 2.4: Estrutura do campo PT do cabeçalho da célula ATM.

que pode parar em uma chave ou outro tipo de dispositivo dependendo do tipo da célula de gerenciamento.

O bit de indicação de congestão é utilizado no mecanismo de controle de fluxo de conexões com qualidade de serviço do tipo ABR. O *User-user indication bit* é utilizado para indicar que a célula em questão corresponde à última unidade de informação de um CPCS-PDU. Chaves ATM com um algoritmo de descarte de células um pouco mais elaborado, ao descartar uma célula descartam também as células seguintes que vão pelo mesmo VPI:VCI (já que nas camadas superiores do receptor o PDU não poderá mesmo ser completamente remontado), até encontrar por este caminho uma célula que esteja com este bit setado, indicando que as próximas células pertencem a outro CPCS-PDU. Este algoritmo descarte seletivo de células é também chamado de *Packet Level Discard*).

CLP (prioridade de perda de célula, do inglês *Cell Loss Priority*): este bit, quando setado, indica que a célula tem prioridade para descarte em caso de congestionamento e significa que esta célula quebra o contrato estabelecido com a rede. Células com este bit setado são transmitidas pela rede apenas se houver disponibilidade de recursos. Mecanismos de policiamento de tráfego fazem uso deste bit.

HEC (verificação de erro no cabeçalho, do inglês *Header Error Check*): este campo armazena o resultado de uma geração polinomial com os dados do cabeçalho. Desta forma é possível detectar e corrigir um erro de um único bit e detectar um erro de múltiplos bits. Vale a pena enfatizar que ele não verifica os 48 bytes de *payload* da célula.

O envio de dados em uma rede ATM ocorre sobre circuitos virtuais estabelecidos através da estrutura física da rede, conforme mostrado na Figura 2.5. A cada canal virtual (VC) é atribuído um *Virtual Channel Identifier* (VCI) e uma determinada qualidade de serviço. Desta maneira, um único computador conectado diretamente a uma rede ATM através de um cabo do tipo *Unshielded Twisted Pair* (UTP), por exemplo,

pode estabelecer vários circuitos virtuais, direcionados a um mesmo destino ou não, cada um com sua respectiva QoS.

O ATM define diferentes tipos de serviço, cada qual com suas características de QoS. Os tipos básicos são:

CBR (Constant Bit Rate): voltado para aplicações em tempo real que geram tráfego a uma taxa constante. Oferece transmissão a uma taxa fixa, definida pelo *Peak Cell Rate* (**PCR**) estabelecido no momento da criação da conexão. Ele emula uma conexão com largura de banda fixa;

VBR-RT (Variable Bit Rate - Real Time): voltado para aplicações em tempo real que geram tráfego a uma taxa que varia no tempo. Este tipo de tráfego tipicamente apresenta rajadas cuja taxa de transmissão (**PCR**) é maior que a taxa de transmissão sustentada (**SCR**).

VBR-NRT (Variable Bit Rate - Non-Real Time): voltado para aplicações tolerantes a atrasos que geram tráfego a taxas variáveis.

ABR (Available Bit Rate): voltado para aplicações adaptáveis, capazes de reduzir ou aumentar a taxa de transmissão de acordo com as condições observadas na rede;

UBR (Unspecified Bit Rate): voltado para aplicações tolerantes a atrasos que geram tráfego a taxas variáveis. Não oferece garantias mínimas de largura de banda. Conexões UBR podem aproveitar a largura de banda reservada mas não usada de conexões CBR.

O desenvolvimento do projeto baseado nos gerenciadores (descrito no capítulo 4) depende do suporte que o Linux oferece à tecnologia ATM. Até o momento em que este projeto foi iniciado o Linux oferecia suporte apenas para canais CBR e UBR, sendo que para canais do tipo CBR o próprio *kernel* faz uma suavização do tráfego, fazendo com que a taxa de transmissão nunca ultrapasse o PCR previamente estabelecido. Isto garante que em caso de congestionamento na rede as células tenham sua entrega garantida, evitando que, por exemplo, a conversação entre dois interlocutores seja prejudicada por interrupções ou uma transmissão de vídeo em tempo real sofra paradas repentinas nos movimentos ou mesmo degradação na qualidade.

Percebe-se que em uso normal, facilmente um único equipamento ATM pode criar diversos canais simultaneamente. Para facilitar o gerenciamento nas chaves ATM foi criado o conceito de *Virtual Path* (**VP**), que agrupa diversos canais virtuais com mesmo ponto de terminação, como indicado na Figura 2.5. Cada caminho virtual possui um *Virtual Path Identifier* (**VPI**), que é atribuído assim que o caminho virtual é estabelecido. Todos os circuitos virtuais que dividem o mesmo caminho virtual possuem o mesmo VPI.

Figura 2.5: Hierarquia de canais em uma rede ATM.

A cada célula que alcança o porto de uma chave é verificada uma tabela de comutação, como a Tabela 2.1, que para todas as conexões de entrada (par VPI/VCI) identifica por qual porto a célula deve sair e com quais valores para os campos VPI/VCI, indicando a conexão virtual à qual a célula pertence. Na Figura 2.6 é ilustrada a comutação de caminhos e circuitos realizada no interior de uma chave.

Tabela 2.1: Tabela de comutação de caminhos e circuitos virtuais de uma chave ATM.

Entrada		Saída	
VPI	VCI	VPI	VCI
2	3	7	6
2	9	2	11
2	7	7	8
4	5	12	5
4	6	12	6

O **VPI** presente no cabeçalho das células pode ser modificado pelas chaves ATM, sem necessariamente alterar o **VCI** destas, processo chamado de comutação de caminhos ou *VP switching*. A comutação dos caminhos virtuais não implica na comutação dos circuitos virtuais. Assim, comutadores de circuito virtual também são comutadores de caminhos virtuais, mas o contrário não é sempre verdade.

2.3 Camada física da arquitetura ATM

A camada física compreende duas sub-camadas: a sub-camada do meio físico e a sub-camada de convergência de transmissão (**TC**), conforme ilustrado na Tabela 2.2.

A primeira delas diz respeito à transmissão de sinais sobre o meio, envolvendo alinhamento de bits e geração e recuperação de sincronismo, seja o meio composto de fios metálicos trançados (cabo **UTP**) ou fibra ótica. O tipo do meio físico determina o modo de operação desta sub-camada.

A subcamada **TC** é independente do meio físico empregado. Sua principal tarefa é

Figura 2.6: Comutação de caminhos e circuitos realizada no interior de uma chave ATM.

Tabela 2.2: Divisão de camadas da arquitetura ATM.

Modelo OSI	Camadas ATM	Subcamadas ATM	Funções das camadas
Camada de Enlace	Camada de Adaptação ATM	Subcamada de convergência	Convergência da mensagem
		Subcamada de Segmentação	Segmentação e remontagem
	Camada ATM	Camada ATM	Controle de fluxo Proc. cabeçalhos Tradução VPI/VCI Multiplexação de células
Camada Física	Camada Física	Subcamada de convergência de transmissão	Desacoplamento de taxa de células HEC geração/verificação Delineamento de células Adaptação de datagrama Geração e rec. de datagrama
		Subcamada do meio físico	Sincronização de bits Transmissão física

adaptar as células [ATM](#) à capacidade de transmissão do meio físico. Isso é realizado com o auxílio de cinco funções:

1. *Geração e recuperação do quadro de transmissão*: responsável pela geração dos quadros físicos.
2. *Adaptação do quadro de transmissão*: responsável pelas ações necessárias para combinar o fluxo de células ATM com o quadro utilizado pelo meio físico. No sentido contrário ela extrai as células do quadro recebido.
3. *Delineamento de células*: permite recuperar os limites de cada célula dentro da cadeia binária recebida.
4. *Geração do Header Error Check (HEC)*: mecanismo polinomial para detecção e correção de erros nos cabeçalhos das células.
5. *Desacoplamento da taxa de células*: permite à camada física inserir células vazias no fluxo de células enviadas pela camada superior. Isto pode ser importante para manter a operação da camada física sob uma taxa constante de quadros transmitidos. No receptor as células vazias são descartadas.

2.4 Camada ATM

A principal função da camada [ATM](#) é enviar e receber células, independentemente de como estas células são tratadas pela camada física. Ela implementa quatro funções:

1. *Multiplexação de células*: agrupa as células de todas as conexões virtuais em um único fluxo de células.
2. *Demultiplexação de células*: separa as células recebidas de acordo com as conexões virtuais às quais pertencem.
3. *Tradução de VPI/VCI*: realiza a comutação de células propriamente dita. Em comutadores [VP](#) apenas o [VPI](#) é modificado, enquanto que em comutadores [VC](#) são modificados tanto o [VPI](#) quanto o [VCI](#) do cabeçalho das células.
4. *Geração e extração de cabeçalho*: insere o cabeçalho da célula nos dados que são passados para a camada ATM pelo seu usuário (a camada AAL). O par [VPI/VCI](#) é obtido através de um mapeamento do *Service Access Point (SAP)* utilizado pela camada usuária para transmissão de células e, na recepção, o par [VPI/VCI](#) indica o [SAP](#) para o qual os dados devem ser entregues. Observa-se que as informações do cabeçalho ATM não são passadas para a camada AAL.

A camada ATM pode ainda realizar a função *Generic Flow Control* (GFC) para controle de fluxo. Ela é independente do circuito virtual e permite a uma estação marcar as células que excedem a taxa contratada com a rede.

2.5 Camada de adaptação ATM (AAL)

Para unificar os diversos tipos de serviços, o ATM utiliza a *ATM Adaptation Layer* (AAL), que se localiza na parte superior do nível de enlace e efetua a adaptação dos diversos tipos de tráfego que os serviços necessitam. Isto é necessário pois os serviços têm diferentes necessidades. Por exemplo, aplicações multimídia suportam perdas relativamente pequenas nos dados, enquanto que transmissões de dados não, porém, transmissões de dados são tolerantes a atrasos relativamente grandes, sendo que o mesmo não ocorre com transmissões de vídeo ou voz em tempo real.

Pode-se dividir as aplicações de acordo com suas necessidades, diferenciando-as pelas características mencionadas na Tabela 2.3, obtendo-se oito serviços distintos.

Tabela 2.3: Classes de Serviço originais mantidas pela AAL.

	A		B	C				D
Tempo- rização	tempo real	nenhu- ma	tempo real	nenhu- ma	tempo real	nenhu- ma	tempo real	nenhu- ma
Taxa de bits	constante		variável		constante		variável	
Modo	Orientado à conexão				Sem conexão			

A ITU-T considerou que somente quatro dos oito serviços eram realmente úteis, e os chamou de classes A, B, C e D. Para especificar cada um destes serviços, a ITU definiu os protocolos AAL1 a AAL4 na recomendação I.363 [64]. Posteriormente, o ITU decidiu que as diferenças entre as AALs 3 e 4 não justificavam a existência de dois protocolos independentes, fundindo-os em um único protocolo, que passou a ser chamado AAL 3/4. Passado algum tempo, o protocolo AAL 3/4 foi considerado complexo e inadequado à transmissão eficiente de dados por redes ATM pelos fabricantes de equipamentos. Foi então proposta pelo ATM Forum a criação de um protocolo mais simples, chamado inicialmente de *Simple Efficient Adaptation Layer* (SEAL), que foi adotado pelo ITU com o nome de AAL5. O protocolo AAL5 foi utilizado neste trabalho e portanto será descrito com maiores detalhes a seguir. Os demais tipos de AAL estão detalhados na literatura especializada.

A função básica da camada AAL é preparar os dados das camadas superiores para transmissão sobre a rede ATM. Os PDUs a serem transmitidos devem ser segmentados e arranjados nas células ATM de acordo com os requisitos de desempenho de cada

aplicação. Este processo de particionamento ocorre na subcamada de segmentação e remontagem (SAR). Outra subcamada da AAL é a subcamada de convergência (CS), que localiza-se logo acima da subcamada SAR, e sua função é adicionar mensagens de controle aos PDUs.

2.6 Camada de adaptação AAL5

O protocolo AAL5 é um protocolo simples idealizado para ser utilizado na transmissão de dados sem conexão. Ele não permite multiplexação de mensagens em uma mesma conexão virtual, nem o reordenamento de células, pois ele admite que elas são entregues sem erros e sem alteração de ordem. A perda de uma célula ATM pertencente a um *Protocol Data Unit* (PDU) AAL5 é detectada por um *Cyclic Redundancy Check* (CRC) de 32 bits e pode implicar no descarte de todo o PDU.

A estrutura da camada AAL5 é mostrada na Figura 2.7.

O *Common Part Convergence Sublayer* (CPCS)-PDU da camada AAL5 é ajustado de tal forma que o seu tamanho total (incluindo *payload* e *trailer*) seja múltiplo de 48 bytes, permitindo sua fácil segmentação em um número inteiro de SAR-PDUs. Seu formato é mostrado na Figura 2.8, e a descrição dos campos segue abaixo:

CPCS-PDU payload: é utilizado para transportar o CPCS-SDU e pode ter qualquer tamanho entre 1 e 65535 bytes, sendo que o *overhead* devido ao encapsulamento é relativamente maior para mensagens pequenas.

UU (User to User): é utilizado para transferir de maneira transparente informações de usuário para usuário.

CPI (Common Part Indicator): tem a única função de alinhar o *trailer* em 64 bits. Quando ele é utilizado exclusivamente como *padding* seu valor deve ser nulo.

Comprimento: armazena o comprimento útil da mensagem e com ele é possível a separação da mensagem do *padding*.

CRC (Cyclic Redundancy Check): é usado para verificar a integridade da mensagem, detectando eventuais perdas de células que compõem a mensagem. Seu algoritmo está bem descrito em [65]. Vale a pena lembrar que o CRC realizado na camada ATM é feito apenas sobre o cabeçalho das células ATM, e não sobre o seu *payload*. Certamente o cálculo do CRC representa um *overhead* no processamento, podendo influenciar na latência de uma transmissão em AAL5. A montagem da SAR-PDU é obtida dividindo-se o CPCS-PDU (que é sempre múltiplo de 48 bytes graças ao *padding*) em blocos de 48 bytes. O campo *payload type* é utilizado para indicar qual a última célula pertencente a um PDU

Figura 2.7: Estrutura da camada AAL5.

Figura 2.8: Formato do CPCS-PDU da camada AAL5.

AAL5, para que no receptor o PDU possa ser corretamente remontado e seu *trailer* identificado.

Há ineficiências significativas em mensagens pequenas (de aproximadamente 48 bytes) devido à introdução do *padding* para tornar a CPCS múltipla de 48 bytes.

A AAL5 é incapaz de multiplexar partes de mensagens em uma mesma conexão virtual, já que seria impossível separá-las na remontagem no lado receptor. As mensagens são enviadas seqüencialmente uma a uma, através de uma conexão virtual. Em conexões virtuais distintas não há restrições para o envio de mensagens em paralelo.

2.7 Protocolo Internet (IP) versão 4

O IP foi inicialmente projetado para permitir a interconexão de redes de computadores que utilizavam tecnologia de comutação de pacotes. O protocolo IP é dito sem conexão, o que significa que cada pacote IP é tratado como uma unidade independente, sem relação com qualquer outro pacote.

A versão do protocolo IP discutida aqui é a IPv4 (quarta versão), que foi a utilizada no trabalho.

Cada pacote IPv4 contém um cabeçalho de 20 bytes (quando não são usadas opções especiais) e um campo de dados (*payload*) de tamanho variável. O formato do cabeçalho pode ser visto na Figura 2.9 e a descrição de cada um dos campos segue abaixo:

Versão: contém a versão do IP usada na criação do pacote (no nosso caso 4);

Figura 2.9: Formato do pacote IPv4.

HLEN (Header LENgth): indica o tamanho do cabeçalho em palavras de 32 bits. Isto é feito pois o cabeçalho pode acomodar opções especiais e neste caso ele passa a ocupar mais que as usuais 5 palavras de 32 bits;

ToS (Type of Service): permite a uma máquina emissora especificar sua preferência sobre como o pacote deve ser encaminhado pela rede. A máquina emissora poderia considerar que no trajeto do pacote até seu destino, quando um roteador tiver mais de um caminho possível, deveria ser privilegiada a rota com menor latência, ou com maior confiabilidade, por exemplo. Na prática o campo ToS do IPv4 é simplesmente ignorado pela maioria dos roteadores. Maiores detalhes sobre este campo são fornecidos na seção 3.7;

Comprimento total: indica o comprimento total do pacote em *bytes*, incluindo cabeçalho e dados. Pode estar entre 20 e 65535, inclusive;

Identificação: é usado para identificar os fragmentos de um pacote IP;

Flags: este campo de 3 bits está ilustrado em maiores detalhes na Figura 2.10

Figura 2.10: Formato do campo *Flags* do IPv4.

Este campo *flags* é usado para controle. O terceiro bit é reservado e deve ser zero. Os demais têm as seguintes funções:

- **D** (Don't fragment): indica que este pacote IP não deve ser fragmentado;
- **M** (More fragments): é usado durante o processo de remontagem dos dados quando estes foram fragmentados em vários pacotes IP.

Fragment offset: este campo de 13 bits permite ao receptor determinar a posição (em *bytes*) do fragmento dentro do pacote IP original;

TTL (Time To Live): o valor deste campo é decrementado cada vez que o pacote atravessa um roteador. Quando ele atinge o valor zero o pacote é descartado;

Protocolo: indica a qual protocolo o campo de dados deve ser enviado (UDP, TCP, etc.);

Checksum do cabeçalho: contém o código de verificação do cabeçalho do pacote;

Endereço IP origem: indica o endereço IP da máquina que originalmente emitiu o pacote;

Endereço IP destino: indica o endereço IP da máquina final que deve receber o pacote;

Opções IP: campos adicionais (opcionais) podem aparecer após o endereço IP destino. Geralmente não são usados.

2.8 User Datagram Protocol (UDP)

O protocolo **UDP** provê um mecanismo que aplicativos utilizam para enviar mensagens a outros aplicativos. O UDP utiliza portas para distinguir entre os vários programas sendo executados em uma máquina. Além dos dados sendo transmitidos, cada pacote UDP contém o número dos portos de destino e origem, tornando possível ao aplicativo no destino enviar uma mensagem de resposta ao aplicativo transmissor correto.

Para o transporte de uma mensagem de uma máquina a outra o UDP faz uso do IP e oferece um envio de mensagens de maneira não orientada a conexão (não garantido). Ele não usa *acknowledgements* para certificar-se de que a mensagem foi recebida, não garante a ordem de entrega das mensagens e nem fornece *feedback* para controlar a taxa à qual a informação flui entre as máquinas, podendo levar à perda de pacotes no lado receptor.

Toda a responsabilidade pela entrega não garantida, incluindo perda, duplicação, atraso e entrega de mensagens fora de ordem fica a cargo da aplicação.

O formato do datagrama UDP é mostrado na Figura 2.11. Maiores detalhes sobre UDP podem ser encontrados em [42].

2.9 Real-Time Transport Protocol (RTP)

O *Real-Time Transport Protocol (RTP)*, descrito no RFC 1889 [121], é um protocolo de transporte genérico para tráfego em tempo real. Ele fica normalmente na camada

Figura 2.11: Formato do datagrama UDP.

de usuário, acima do protocolo *User Datagram Protocol* (UDP), conforme mostrado na Figura 2.12.

Figura 2.12: Posição do RTP na pilha de protocolos.

As aplicações multimídia podem mandar diversos fluxos de áudio e vídeo (e possivelmente outros) para a biblioteca que implementa o RTP. A biblioteca então multiplexa os fluxos e os coloca em pacotes RTP, que são posteriormente enviados através de um *socket*. Na outra extremidade do *socket* o *kernel* se encarrega de gerar os pacotes UDP e transmiti-los pela rede através de *unicast* ou *multicast*.

Os pacotes RTP de um mesmo fluxo são sequencialmente marcados para que o destino possa tomar as devidas providências na ocorrência da perda de um pacote. Muitas vezes isto consiste na interpolação do valor que falta. A retransmissão do pacote geralmente não é uma alternativa viável pois a maioria das aplicações de tempo real não pode esperar o tempo necessário para a chegada do pacote retransmitido.

O RTP não tem controle de fluxo, controle de erros, confirmação de chegada ou mecanismo de retransmissão. Seu cabeçalho é mostrado na Figura 2.13.

Figura 2.13: Cabeçalho do protocolo RTP.

Ver: contém a versão sendo usada;

P: indica se o pacote sofreu *padding* para torná-lo múltiplo de 4 bytes;

X: indica que um cabeçalho de extensão está sendo usado;

CC: indica quantas fontes estão presentes, de 0 a 15 (veja o campo *Identificação da origem da sincronização* abaixo);

M: bit marcador de uso definido pela aplicação. Pode ser usado para marcar o início de um datagrama, por exemplo;

Tipo de dado: informa qual algoritmo de codificação está sendo usado. Como todo pacote carrega este campo, o algoritmo de codificação pode ser trocado durante a transmissão;

Número de seqüência: contador que é incrementado a cada pacote **RTP** enviado;

Timestamp: indica quando a primeira amostra do pacote foi criada na origem;

Identificação da origem da sincronização: usado na demultiplexação, indica a qual fluxo o pacote pertence.

O **RTP** muitas vezes é usado em conjunto com o *Real-Time Transport Control Protocol* (**RTCP**), que fornece *feedback* sobre latência, *jitter*, largura de banda, congestionamento e outras propriedades da rede às fontes. Estas informações podem ser usadas pelos processos de codificação para aumentar a taxa de transmissão (e fornecer melhor qualidade) quando as condições na rede são boas e diminuir a taxa caso alguma dificuldade seja percebida. O RTCP também cuida da sincronização entre fluxos (problema criado pelo uso de relógios diferentes distribuídos pela rede).

Trabalhos e padrões relacionados

São apresentados nesta sessão vários trabalhos que por meio de diferentes maneiras tentam disponibilizar QoS (ou CoS) a máquinas conectadas em redes de tecnologias incapazes de oferecer garantias de qualidade de serviço (em geral *Ethernet*, por ser a tecnologia mais comumente empregada na construção de LANs). Sempre que possível são apresentadas as vantagens e desvantagens de cada uma das propostas.

3.1 CIF

Células em *frames* CIF [125] é uma tecnologia desenvolvida na *Cornell University* que provê um mecanismo para usar uma pilha de protocolos ATM completa, com garantias de QoS em redes baseadas em pacotes como *Ethernet* ou *Token Ring*. O objetivo principal das células em *frames* é conseguir prover serviços ATM a máquinas *Ethernet* diminuindo significativamente os custos iniciais de integração das redes de voz e dados, eliminando a necessidade da interface ATM nas estações de trabalho. Com esta proposta é possível fazer *upgrades* graduais sem que seja perdido o investimento feito previamente em tecnologias sem QoS.

CIF provê um mecanismo que permite rodar aplicações ATM nativas com provisionamento de QoS usando interfaces *Ethernet* convencionais. Uma das vantagens do CIF apontadas em [48] é que ele permite pacotes de tamanho variável, utilizando apenas um cabeçalho para um *payload* de até 31 células ATM de um mesmo VCI, reduzindo com isso o *overhead* imposto pelas tradicionais células ATM de tamanho fixo, mas implicando em um potencial aumento na latência de transmissão.

Um dos principais componentes da especificação CIF é o *Attachment Device* (AD), que é basicamente uma chave *Ethernet* com *software* adicional e um *uplink* ATM. Sua função é converter pacotes CIF em células ATM e vice-versa. Devido às semelhanças da chave CIF com uma chave *Ethernet* ela poderia custar não muito mais que uma chave *Ethernet* convencional.

Figura 3.1: Topologia e pilha de protocolos na arquitetura CIF.

A Figura 3.1 apresenta a topologia de rede e a pilha de protocolos usados na arquitetura CIF.

Conforme descrito em [125], Cornell contratou uma empresa para produzir um protótipo de AD em *hardware* que foi entregue juntamente com os *softwares* necessários para computadores equipados com Windows NT. No entanto, o protótipo ainda não era capaz de fazer sinalização ATM para requerer QoS.

3.2 ASHMEN

Uma organização, ao decidir migrar da tecnologia *Ethernet* para a ATM, dificilmente tem condições financeiras para fazer a migração de uma única vez. Imaginando que seria interessante que fosse possível interfacear ilhas de tecnologia antiga com a nova tecnologia ATM, o artigo [126] propõe uma arquitetura que chamou de ASHMEN (do inglês "ATM over SHared MEDIA Networks"). Trata-se de uma maneira alternativa para prover serviços ATM fim-a-fim sem a substituição completa dos equipamentos de rede. Serviços ATM são oferecidos sobre *Ethernet*, onde sistemas finais (que não necessitam de alterações) são conectados a dispositivos de conexão *Ethernet*-ATM, denominados AD (do inglês *Attachment Device*), conforme ilustrado na Figura 3.2.

A arquitetura de protocolos desenvolvida inclui um mapeamento de datagramas

Figura 3.2: Topologia e pilha de protocolos na arquitetura ASHMEN.

AAL 5 em datagramas *Ethernet* que foi implementado em máquinas UNIX. Garantias de QoS são oferecidas na rede ATM da maneira usual e nos segmentos *Ethernet* assume-se que a QoS oferecida é boa devido ao uso de chaves e ao módulo de gerenciamento de recursos implementado no AD.

Os sistemas finais usam um protocolo de sinalização comum para criar conexões ATM com seus destinatários através do *backbone* ATM. No segmento *Ethernet* os dados são transportados em datagramas *Ethernet* e o AD mapeia os datagramas *Ethernet* em células ATM e vice-versa. Em contraste com a proposta CIF, a proposta ASHMEN diminui as exigências sobre o processador das máquinas *Ethernet*, pois tarefas como segmentação e remontagem de células e o cálculo de *checksum* dos cabeçalhos são realizadas pelo AD usando *hardware* dedicado.

O artigo comenta ainda que nas redes locais, devido ao aumento crescente de largura de banda disponível, o gerenciamento de recursos com granularidade fina torna-se desnecessário para a maioria das aplicações. Porém, na WAN os recursos são custosos e a multiplexação das conexões é inevitável.

O AD recebe datagramas de máquinas na rede local e segmenta PDUs AAL 5 em células ATM. Na direção oposta, células ATM são recebidas para formar os PDUs AAL 5 que por sua vez são segmentados em uma seqüência de datagramas *Ethernet*.

O AD tem diversas responsabilidades, entre elas:

- Reenviar o tráfego entre o sistema final e o *backbone* ATM e vice-versa;

- Segmentar os PDUs AAL em células ATM e remontar os PDUs a partir das células;
- Segmentar os PDUs AAL em datagramas *Ethernet* e remontar os PDUs AAL a partir dos datagramas *Ethernet*;
- Cuidar da sinalização na rede ATM;
- Cuidar do mapeamento da sinalização entre as redes *Ethernet* e ATM;
- Traduzir endereços ATM, [SAPs](#) e endereços *Ethernet*;
- Rotear entre os sistemas finais e as conexões ATM;
- Manter informações sobre o estado das conexões;
- Calcular e verificar o *checksum* dos cabeçalhos;
- Tratar os erros;
- Gerenciar e alocar recursos (largura de banda) na rede local

A camada *Ethernet ATM Transformation* ([EAT](#)) transforma PDUs AAL 5 da camada de adaptação em datagramas *Ethernet* que são entregues ao *Ethernet Data Link Layer* (DLL). Além disso, a camada EAT inclui a sinalização ATM que é entregue ao AD para ser manipulado pelas funções de roteamento e envio [R&R](#).

3.3 CAFE

O artigo [[133](#)] apresenta uma proposta para construção de uma enorme WAN capaz de transmitir dados de redes IP através de um *backbone* ATM na busca para contornar algumas das limitações encontradas em WANs baseadas em roteadores [[133](#)]:

- Limitação da largura de banda causada pela limitada capacidade de roteamento de pacotes nos roteadores;
- Limitação da capacidade de processamento dos protocolos de roteamento (usada na construção das tabelas de rotas);
- *Overhead* imposto pelo tráfego relacionado aos protocolos de roteamento.

O elemento fundamental da proposta apresentada no artigo é o *Cut-through ATM based Forwarding Engine* ([CAFE](#)), que tem a função de reencaminhar datagramas *Ethernet* pelo *backbone* ATM. Diversos CAFEs são conectados uns aos outros em

Figura 3.3: Arquitetura baseada no CAFE.

malha completa (*full mesh*) através de **PVCs**, como mostrado na Figura 3.3. Qualquer CAFE pode enviar pacotes IP/*Ethernet* para qualquer outro CAFE sem processamento acima da camada ATM, ou seja, sem necessidade de reconstruir ou rotear pacotes.

As funções de roteamento e *forwarding* são separadas. O *forwarding* é realizado pelos CAFEs e o roteamento por uma entidade especialmente desenvolvida para este fim que processa o protocolo de roteamento em *software*.

Os endereços *Ethernet*/IP são extraídos da primeira célula de cada datagrama. Os autores ainda propõem um método de busca rápida baseada em *hardware* para obter os endereços de entrada em uma tabela adequada para processamento de cada célula.

3.4 PACE

A 3Com desenvolveu uma proposta de modificação do MAC *Ethernet* usado em chaves *Ethernet* para que os mesmos pudessem ser utilizados para tráfego em tempo real [23]. Ele consiste basicamente na modificação do algoritmo de *backoff* usado no *Ethernet* padrão, mantendo todas as demais características do padrão IEEE 802.3.

PACE é totalmente compatível com o padrão *Ethernet*, de modo que máquinas com placas *Ethernet* antigas conseguem operar normalmente com outras conectadas através de uma chave que implementa o novo MAC. Assim é possível fazer *upgrades* incrementais nas LANs baseadas em *Ethernet* sem necessidade de mudanças radicais.

O *Ethernet* "normal" apresenta uma latência de acesso à rede que pode ser da ordem de centenas de milisegundos (detalhes na seção F). Já o PACE foi projetado para oferecer latências típicas na faixa de 5 a 10 *ms*. O mecanismo é detalhadamente descrito em [23], mas essencialmente esta latência baixa é conseguida através da redução do tempo de *backoff* (para detalhes sobre o mecanismo de *backoff* ver seção F).

Para o correto funcionamento do PACE, cada MAC com tecnologia PACE (PE-

MAC) precisa saber se ele foi o último dispositivo a transmitir com sucesso. Também é necessário que o PE-MAC esteja em um dos dois estados possíveis:

- Tentando Transmitir
- Esperando para Receber

Quando um PE-MAC recebe um datagrama ele entra no estado "Tentando Transmitir". Neste estado ele faz diversas tentativas sucessivas de transmissão durante o intervalo entre datagramas (IPG, do inglês *Inter-Packet Gap*), com um tempo de *backoff* nulo. O número de tentativas é limitado e quando o limite é alcançado o datagrama é descartado. No estado "Esperando para Receber", o PE-MAC espera apenas o necessário para permitir ao nó na outra ponta do *link* transmitir um pacote, se ele tiver algum a transmitir. O tempo de *backoff* é calculado com base no número de colisões ocorridas na transmissão anterior. Se não houve colisão no estado "Tentando Transmitir" anterior, o tempo de *backoff* é nulo e neste caso, se houver mais um datagrama a enviar o PE-MAC irá tentar transmiti-lo assim que o IPG expirar. Entretanto, se houve colisões no estado "Tentando Transmitir" anterior será concedido um tempo maior para que o outro nó possa sair de seu *backoff* e assim transmitir seu datagrama.

Experimentos e simulações descritos em [23] mostram que PACE consegue oferecer uma latência e um *jitter* tipicamente menor que o MAC *Ethernet* padrão tanto nas versões de 10 Mbps como de 100 Mbps.

3.5 Padrão IEEE 802.1Q

O padrão IEEE 802.1Q [61] define uma arquitetura para a criação de redes locais virtuais (VLANs) nas redes baseadas no padrão IEEE 802. Assim torna-se possível criar topologias virtuais dentro da topologia física da rede instalada. Ele não define mecanismos de QoS, sendo que as maiores motivações para sua criação são diminuir a propagação de *broadcasts* e limitar de forma flexível e controlada a propagação do tráfego mais sensível à segurança.

VLANs são baseadas em chaves projetadas para esta finalidade, e para que as VLANs funcionem corretamente é necessário configurar tabelas nas chaves e pontes. Estas tabelas indicam quais VLANs são acessíveis por cada porto. Mesmo em redes baseadas em VLANs continua sendo possível conectar chaves e *hubs* antigos (que não entendem VLANs) na periferia da rede, de forma que investimentos prévios em equipamentos não são perdidos.

O padrão 802.1Q introduziu dois novos campos no datagrama *Ethernet* original, como pode ser visto comparando as Figuras 2.1 e 3.4. O primeiro campo (*Tipo*), com

Figura 3.4: Formato do datagrama *Ethernet* compatível com o padrão IEEE 802.1Q.

2 bytes de comprimento foi colocado exatamente no lugar do campo *comprimento* do datagrama *Ethernet* original. Ele identifica o datagrama como sendo compatível com o padrão 802.1Q e tem sempre o valor 0x8100. Uma vez que este número é maior que 1500, todas as interfaces *Ethernet* interpretam-no como um tipo ao invés de um comprimento.

O segundo campo (*Tag*), também com 2 bytes de comprimento contém três subcampos. O mais importante deles é o identificador de VLAN (**VLAN ID**), que ocupa os 12 bits menos significativos do campo *Tag*. Ele informa à qual VLAN o datagrama pertence. O campo *Prioridade* não tem relação alguma com VLANs, sendo usado pelo padrão 802.1p (descrito na seção 3.6). O último bit, *Canonical Format Indicator* (**CFI**), indica que o *payload* contém um datagrama 802.5 que espera encontrar outra LAN 802.5 no destino enquanto é transportado por *Ethernet* (também não tem relação alguma com VLANs).

3.6 Padrão IEEE 802.1p

O padrão IEEE 802.1p é parte da especificação IEEE 802.1D [62], e permite uma priorização de tráfego em uma rede *Ethernet* na camada 2 do modelo OSI através de 8 categorias diferentes de tráfego, representadas por um campo de 3 bits que é transmitido no datagrama *Ethernet*. Este campo que indica a prioridade do datagrama recebe o nome de *Prioridade* e é mostrado na Figura 3.4.

A especificação 802.1p permite a uma chave *Ethernet* possuir de uma a oito filas para cada porta, onde cada fila é associada a uma classe de tráfego. Os datagramas são colocados nestas filas de acordo com o seu nível de prioridade (definido pelo campo *Prioridade* do usuário), respeitando uma tabela de classes de tráfego que faz parte das informações de estado associadas a cada porta. *Frames* de prioridade mais baixa não são transmitidos quando existem datagramas em filas de prioridade mais alta.

Tabela 3.1: Mapeamento das prioridades do usuário nas classes de tráfego sugerido pelo padrão 802.1D.

		Número de classes de tráfego disponíveis							
		1	2	3	4	5	6	7	8
prioridade do usuário	0 (padrão)	0	0	0	1	1	1	1	2
	1	0	0	0	0	0	0	0	0
	2	0	0	0	0	0	0	0	1
	3	0	0	0	1	1	2	2	3
	4	0	1	1	2	2	3	3	4
	5	0	1	1	2	3	4	4	5
	6	0	1	2	3	4	5	5	6
	7	0	1	2	3	4	5	6	7

As classes de tráfego são numeradas de 0 a $N - 1$, onde N corresponde ao número de classes de tráfego para uma dada porta de saída. Convencionou-se que a classe de tráfego 0 corresponde ao tráfego de prioridade mais baixa. A tabela 3.1 mostra o mapeamento das prioridades do usuário nas classes de tráfego sugerido por este padrão. Nota-se que quando o equipamento *Ethernet* implementa poucos tipos de CoS a priorização dos datagramas pode ser bastante prejudicada pois deixa de ser feita uma diferenciação significativa entre as categorias de tráfego definidas pela prioridade do usuário.

Deve-se notar que no padrão 802.1p não é usado nenhum protocolo de controle de admissão e para que o tráfego usufrua da priorização os equipamentos *Ethernet* precisam obedecer à nova especificação (ser compatíveis com 802.1p). Equipamentos antigos simplesmente ignoram a prioridade do usuário e transmitem os datagramas sem nenhum tratamento especial. Além disso, já que o cabeçalho *Ethernet* só é lido pela camada 2, os roteadores não conseguem usufruir da priorização baseada no 802.1p. Isto significa que apesar da priorização ocorrer na rede chaveada, ela é perdida nos roteadores na borda da LAN com a WAN. Uma alternativa é mapear a prioridade da camada 2 para um esquema de priorização na camada 3, como o "*Type of Service*" (ToS), descrito a seguir.

3.7 Type of Service (ToS) no IPv4

No cabeçalho do protocolo IP versão 4 existe um campo de 8 bits chamado "*Type of Service*" [7]. Originalmente este campo continha um campo *Precedência* de 3 bits, 3 *flags* (D, T e R) e dois bits sem uso definido, conforme ilustrado na Figura 3.5. O campo *Precedência* era usado para indicar a prioridade de cada pacote, de 0 (normal)

Figura 3.5: Campo ToS do cabeçalho do protocolo IP.

a 7 (pacote de controle da rede). As três *flags* permitiam à máquina especificar a característica que ela considerava mais importante:

- Atraso (D - *Delay*)
- Taxa de transmissão (T - *Throughput*)
- Confiabilidade (R - *Reliability*)

Teoricamente o campo ToS permitiria ao roteador fazer escolha entre, por exemplo, um *link* de satélite com alta taxa de transmissão e latência ou uma linha dedicada com baixa taxa de transmissão e latência. Porém, sabe-se que a maioria dos roteadores nem se importavam com o conteúdo deste campo [4], e este fato serviu de inspiração para a IETF mudar levemente o significado deste campo para introduzir os Serviços Diferenciados, descritos na seção 3.9.

3.8 Serviços Integrados

Entre 1995 e 1997 a IETF publicou vários RFCs integrantes de um trabalho maior que chamou genericamente de "Serviços Integrados". Eles tinham por objetivo definir mais duas classes de serviço que pudessem ser usadas além da categoria tradicional de "melhor esforço". Elas são:

- *Serviço garantido*: para aplicações que exigem um limite máximo na latência;
- *Serviço de carga controlada*: para aplicações que requerem um serviço de melhor esforço confiável e melhor que o tradicional.

O protocolo principal usado nos serviços integrados é o *Resource reSerVation Protocol* (RSVP). Ele é descrito no RFC 2205 [35] e é usado para fazer a reserva dos recursos, enquanto outros protocolos são usados para a transmissão dos dados.

Uma das maiores críticas com relação aos serviços integrados é que apesar deles permitirem oferecer uma boa QoS a um ou mais fluxos através da reserva prévia dos recursos necessários, eles apresentam diversas dificuldades [8]:

- Os Serviços Integrados não escalam bem quando há milhares ou milhões de fluxos;
- Existe uma certa vulnerabilidade quanto a falhas dos roteadores pois esta solução requer que sejam mantidas informações a respeito de cada fluxo nos roteadores;
- As mudanças necessárias no roteadores são substanciais e envolvem complexas trocas de informações entre roteadores.

Por estas e outras razões a IETF decidiu procurar por uma solução mais simples, que pudesse ser implementada sem dificuldades nos roteadores. A esta nova solução chamou de "Serviços Diferenciados", que é brevemente descrita na seção a seguir.

3.9 Serviços Diferenciados

Os serviços diferenciados (DS, do inglês *Differentiated Services*) são descritos no RFC 2474 [102] e no RFC 2475 [31], entre outros.

Serviços diferenciados podem ser oferecidos por um conjunto de roteadores formando um domínio administrativo (como um *Internet Service Provider (ISP)*, por exemplo). A administração define um conjunto de classes de serviço com as regras de encaminhamento correspondentes. Um usuário que utiliza serviços diferenciados pode empregar um sistema de classificação para enviar ao domínio pacotes com determinados valores no campo DS (comentado logo a seguir) requerendo para algumas classes um serviço melhor que para outras. O tráfego dentro de uma classe pode ter que passar por uma suavização para passar a obedecer a alguns critérios específicos exigidos pela administração, mas em compensação não há necessidade de sinalização antes do envio dos dados, não há reserva de recursos antecedendo o envio dos dados e nem negociação para cada fluxo como nos serviços integrados. É assumido que a administração aloca recursos suficientes de acordo com as classes de serviço por ela estabelecidas.

A especificação dos serviços diferenciados substitui o campo ToS do IPv4 (Figura 2.9) pelo campo DS (Figura 3.6), mantendo compatibilidade com equipamentos que levam os três bits de precedência do antigo campo ToS em consideração.

Figura 3.6: Campo DS usado nos serviços diferenciados. Substitui o campo ToS do IPv4.

Seis bits do campo DS são usados como identificador **DSCP** (do inglês *Differentiated Services Code Point*) para selecionar o *Per Hop Behavior* (**PHB**) que o pacote deve apresentar em cada nó. Isto permite definir até 64 PHBs diferentes. Os dois últimos bits continuam não sendo usados e são ignorados.

3.10 MPLS

A tecnologia *Multi-Protocol Label Switching* (**MPLS**), descrita em vários **RFCs**, em especial no RFC 3031 [116], é o resultado da padronização de várias soluções proprietárias como *IP switching* [92, 79, 101] e *Tag switching* [112, 113] que surgiram na tentativa de juntar as melhores características do roteamento (nível 3 do modelo **OSI**) com o chaveamento de circuitos virtuais (nível 2).

A forma de encaminhamento adotada pelo MPLS proporciona algumas vantagens em relação à maneira tradicional:

- Melhor desempenho no encaminhamento dos pacotes;
- A criação dos caminhos virtuais, chamados *Label Switched Paths* (**LSPs**), facilita a engenharia de tráfego;
- Os requisitos de QoS de cada fluxo podem ser associados aos rótulos (*labels*) usados na transmissão dos pacotes.

Tradicionalmente, quando um roteador recebe um pacote ele faz uma consulta na sua tabela de roteamento para decidir para onde enviá-lo. Esta consulta é feita a cada pacote recebido e pode levar um tempo consideravelmente maior que o tempo gasto por uma chave no processo de encaminhamento de um datagrama, pois a chave usa um identificador presente no pacote como índice para fazer uma consulta em uma *lookup table* (processo que é extremamente rápido). *Multi-Protocol Label Switching* (**MPLS**) rompe este paradigma do roteamento usando um rótulo de tamanho fixo para decidir para onde os pacotes devem ser enviados. O rótulo aparece na Figura 3.7, que ilustra o formato do datagrama MPLS quando ele é usado em uma conexão ponto-a-ponto com o protocolo *Point to Point Protocol* (**PPP**). Costuma-se dizer que o MPLS fica na camada 2,5, pois tipicamente ele fica entre a camada de enlace (camada 2) e a camada de rede (camada 3).

Como já comentado, o campo *Rótulo* é usado para identificar cada fluxo. Ele só tem significado local à chave e é trocado por outro cada vez que o pacote atravessa uma chave. O campo *QoS* indica a classe de serviço e *S* é usado quando vários rótulos são empilhados em redes hierárquicas. *S* recebe valor 1 para o último rótulo, sendo que

Figura 3.7: Formato do datagrama MPLS.

para todos os outros ele tem valor 0. Este recurso é geralmente usado para implementar *Virtual Private Networks* (VPNs) e túneis recursivos.

No roteamento tradicional com circuitos virtuais não é possível agrupar diversos caminhos distintos com destinos diferentes sob um mesmo identificador de VC porque não há como distinguí-los no destino. Em MPLS isto é possível, pois os pacotes carregam, além do rótulo, o endereço de destino final - assim que o pacote chega ao fim do caminho chaveado o rótulo é retirado e o pacote pode seguir até o destino através de roteamento tradicional. Fluxos com rotas iguais e/ou requisitos de QoS semelhantes podem ser agrupados em uma mesma *Forwarding Equivalence Class* (FEC).

3.11 IPv6

A versão 6 do IP foi definida para suprir várias deficiências do IPv4. Entre seus principais objetivos destacam-se a necessidade de aumentar o espaço de endereçamento IP, simplificar o protocolo para que os roteadores possam processar os pacotes mais rapidamente, aumentar a segurança e permitir a coexistência do novo protocolo com o velho. Dentre as mudanças mais significativas no IPv6 em relação ao IPv4 destacam-se o fato dos endereços terem 16 *bytes* de comprimento, do cabeçalho ter significativamente menos campos (7 ao invés de 13, simplificando o processamento realizado nos roteadores) e melhor suporte a opções (campos antes obrigatórios passaram a ser opcionais). Também mais atenção foi dispensada à questão da QoS, como pode ser observado pelo formato do cabeçalho do IPv6 apresentado na Figura 3.8 e pela descrição dos seus campos logo a seguir.

Versão: contém a versão do IP usada na criação do pacote (6 no caso do IPv6);

Classe de tráfego: é usado para diferenciar os requisitos de QoS dos pacotes de forma semelhante ao campo ToS do IPv4;

Figura 3.8: Formato do cabeçalho do IPv6.

Rótulo de fluxo: pode ser usado para criar uma pseudoconexão com determinadas propriedades entre duas máquinas. Desta forma, todo pacote com rótulo diferente de zero receberá um tratamento semelhante ao proposto pelo MPLS (seção 3.10);

Comprimento dos dados: informa quantos *bytes* seguem após o cabeçalho de 40 *bytes*;

Próximo cabeçalho: informa qual cabeçalho de extensão segue este cabeçalho, se houver algum, ou a qual protocolo de transporte este pacote deve ser enviado (TCP, UDP, etc.);

Hop limit: tem a mesma função do campo **TTL** do IPv4. Ele é decrementado a cada roteador que o pacote transpõe e quando atinge o valor zero o pacote é descartado;

Endereço origem: este campo, que passou de 32 *bits* no IPv4 para 128 *bits* no IPv6, indica o endereço IP da máquina que originalmente emitiu o pacote;

Endereço destino: indica o endereço IP da máquina final que deve receber o pacote.

Gerenciador de tráfego ATM e Ethernet

Como exposto no capítulo 3, propostas anteriores já foram feitas com objetivo de disponibilizar a QoS de um *backbone* ATM a aplicações localizadas em redes sem QoS. Porém, a maioria destas propostas depende de novos desenvolvimentos em *hardware*, como é o caso das soluções CIF, ASHMEN e CAFE, ou alterações em equipamentos já existentes, como é o caso do PACE. Isto não apenas exige consideráveis investimentos na infra-estrutura necessária ao desenvolvimento de um projetos desses, como também dificulta a adoção futura da solução, a menos que a mesma seja adotada como um padrão pela indústria. Por estas e outras razões, o trabalho desenvolvido neste projeto faz uso de equipamentos padronizados facilmente encontrados no comércio (descritos na seção A) e ferramentas livres (descritas na seção B). O projeto foi planejado para de fato viabilizar sua aplicação em problemas reais, como é o caso da operação remota do tomógrafo de ressonância magnética desenvolvido pelo Instituto de Física de São Carlos e instalado na Santa Casa de Misericórdia de São Carlos [20].

4.1 Modelo proposto

O modelo aqui proposto permite a reserva de recursos a partir de redes que não oferecem garantias de QoS através de solicitações feitas a máquinas que denominamos de gerenciadores. Estes gerenciadores são conectados diretamente a uma WAN ATM e também a uma LAN desprovida de QoS. Neste estudo utilizamos *Fast Ethernet* para construção da LAN por ser uma das tecnologias mais comumente empregadas em redes locais.

Para funcionamento do modelo é necessária a presença de um gerenciador por LAN ou mesmo por MAN para que canais ATM sejam criados dinamicamente de domínio a domínio, conforme ilustra a Figura 4.1.

Uma aplicação multimídia pode fazer a solicitação de abertura de um canal de gerenciador a gerenciador, com as características de QoS que ela necessita, e ao invés de enviar o tráfego de vídeo ou áudio diretamente para a aplicação em outra LAN, ela

Figura 4.1: Topologia de rede com gerenciadores. G_1 e G_2 localizados na borda da WAN com as LANs representam os gerenciadores, H_1 e H_2 conectados aos gerenciadores através das LANs representam seus clientes e R_1 e R_2 representam os roteadores para uso tradicional do protocolo IP.

envia seu tráfego para o gerenciador do seu domínio, que se encarregará de reenviá-lo para o gerenciador do domínio da aplicação destino através do canal previamente estabelecido. Assim é possível a uma máquina equipada com uma interface *Ethernet* estabelecer um fluxo de dados com outra máquina similar com a QoS adequada às suas necessidades através de uma nuvem ATM sem o *overhead* causado pelo uso de protocolos adicionais como IP ou LANE (*LAN Emulation*). Desta forma, áudio e vídeo de alta qualidade podem ser transmitidos a longas distâncias, sem que haja a necessidade da adoção de uma solução proprietária fechada ou a implantação de uma segunda rede dedicada ao tráfego multimídia. Observe ainda que o modelo pode ser aplicado até mesmo em redes de produção (que não podem deixar de operar), pois não há necessidade de trocar nenhum equipamento necessário à operação normal da rede.

4.2 Implementação

Foi utilizado para implementação e testes o sistema operacional Linux com versão de *kernel* 2.4.22 compilado com suporte a interfaces ATM da Fore (atual Marconi) modelo ForeRunnerLE, baseadas no chip Nicstar 77211 (detalhes nas seções [B.14](#) e [A.2](#)).

O gerenciador foi implementado na linguagem C++ com o auxílio das bibliotecas **STL** para uso de estruturas e algoritmos comuns (seção [B.2](#)) e GNU Common C++ (seção [B.4](#)) que oferece um suporte eficiente a *threads*. GNU Common C++, apesar de ser bastante portátil, acaba por utilizar no ambiente Linux a implementação de *threads* **POSIX** do *kernel*, sendo bastante eficiente e impondo um mínimo de *overhead* no chaveamento dos processos.

Após a inicialização o gerenciador aguarda por comandos provenientes de aplica-

ções multimídia dentro de seu domínio. Os principais comandos são os que solicitam a criação de um novo canal, a destruição de um canal existente ou a coleta de estatísticas de uso do canal e dos *buffers* a este associados.

Foi construída uma biblioteca que pode ser usada para a modificação de uma aplicação multimídia para comunicação com o gerenciador. A biblioteca realiza a comunicação com o gerenciador através de *sockets*, de maneira que a aplicação que dela faz uso pode estar em qualquer máquina da rede.

Para utilização de aplicações multimídia não modificadas, como foi o caso do VLC (veja seção B.12), foi construído um programa em C++ para solicitar a abertura de canais, com o uso da biblioteca anteriormente mencionada. Este programa, denominado controlador, oferece várias opções, conforme mostrado na Figura 4.2.

```
Options
=====
1 - Create new tunnel
2 - Delete tunnel
3 - Print list of tunnels
4 - Print Statistics of tunnel
5 - Monitor Statistics of tunnel
6 - Save tunnels in file
7 - Load tunnels from file
8 - Zero statistics of tunnel

A - Create preconfigured tunnel 1
B - Create preconfigured tunnel 2
C - Create preconfigured tunnel 3
D - Create preconfigured tunnel 4
? - Prints this menu
q - Quit
Option:
```

Figura 4.2: Menu principal do programa controlador.

O programa controlador permite criar um novo canal, destruir um canal existente, mostrar uma lista com todos os canais ativos, mostrar as estatísticas de utilização de um canal e seus *buffers* em um dado instante, mostrar continuamente as estatísticas, salvar a lista de canais ativos em um arquivo, carregar um arquivo contendo uma lista de canais e solicitar a criação destes e zerar as estatísticas de um túnel. As opções A a D permitem solicitar a criação de canais com características previamente estabelecidas.

Como pode-se observar, o programa controlador é bastante rústico, mas foi fundamental para testar as funcionalidades dos gerenciadores. Foi também criada uma versão gráfica para a criação de um novo túnel usando Glade2 (veja seção B.6). A interface deste programa é mostrada na Figura 4.3.

Figura 4.3: Interface gráfica de programa capaz de criar novos canais.

O botão "Settings" abre uma janela que permite a seleção do gerenciador a receber a solicitação, conforme mostra a Figura 4.4. Neste exemplo, o programa controlador foi executado na mesma máquina que executava o programa gerenciador, por isso a máquina gerenciadora foi denominada por "localhost".

Figura 4.4: Janela que permite a seleção do gerenciador a receber a solicitação de criação de novo canal.

Na abertura de um novo canal é necessário informar quais são as quatro máquinas envolvidas, e suas respectivas portas:

ip0e é a máquina *Ethernet* de uma LAN (LAN0)

ip0a é o gerenciador responsável pelo domínio da LAN0

ip1a é o gerenciador responsável pelo domínio da LAN1

ip1e é a máquina *Ethernet* de outra LAN (LAN1)

Claro que em uma solução final boa parte destas informações poderia ser suprimida. Por exemplo, cada máquina *Ethernet* poderia saber de antemão quem é o gerenciador de seu domínio e esta informação poderia facilmente ser comunicada às máquinas envolvidas no outro domínio quando necessário. Porém, durante a fase de pesquisas considerou-se perfeitamente aceitável que todas estas informações sejam fornecidas pelo usuário.

Ainda na abertura de um canal, também é necessário informar quais são os parâmetros de QoS desejados pela aplicação. Destes parâmetros de QoS, o primeiro é o tipo de canal a ser criado: CBR ou UBR. Caso seja solicitado um canal CBR é necessário informar o PCR e o CDV (*Cell Delay Variation*) máximo.

Por fim pode-se determinar o tamanho dos *buffers* de ambos os gerenciadores. "0a Buffers" refere-se aos *buffers* do gerenciador da LAN0 e "1a Buffers" aos *buffers* da LAN1. "Line size" representa o tamanho de cada linha do buffer, que deve ser igual ou maior que a maior mensagem a ser transportada. "Lines" representa o número de linhas que o *buffer* deve possuir.

Ao clicar no botão Create a solicitação é enviada ao gerenciador previamente estabelecido.

O gerenciador atribui um SAP (*Service Access Point*) único a cada canal automaticamente e o transmite ao outro gerenciador envolvido para que ele possa solicitar a criação do SVC (*Switched Virtual Circuit*).

Quando ainda há recursos disponíveis a conexão ATM é estabelecida com sucesso e diversas *threads* são criadas. Uma delas fica permanentemente recebendo o tráfego enviado a uma porta específica do gerenciador pela rede *Ethernet* e colocando-o em um *buffer* circular capaz de absorver eventuais rajadas de tráfego das aplicações multimídia sendo executadas nos *endpoints*. Outra *thread* fica permanentemente retirando dados do *buffer* e enviando-os para outro gerenciador através da WAN ATM. Se em algum momento o *buffer* esvazia, a *thread* que retira dados dele é colocada para dormir (sem desperdício de CPU), sendo acordada assim que a outra *thread* colocar mais dados. O mesmo é feito na direção contrária, da WAN para a LAN.

São portanto criadas quatro *threads* em ambos os gerenciadores para cada canal ATM, conforme ilustra a Figura 4.5. Por uma razão ou outra uma das *threads* pode apresentar algum problema, como por exemplo, fazer `bind()` em uma porta que já está sendo utilizada. Este tipo de situação é detectado pelo gerenciador que então fecha o canal correspondente e destrói suas *threads* associadas. Uma mensagem é ainda enviada ao outro gerenciador para que ele também destrua suas *threads*. A estratégia de uso intensivo de *threads* foi escolhida pois ela facilita a implementação por eliminar a preocupação do programador com certos detalhes. Por exemplo, se houvesse uma rotina responsável simultaneamente pelo envio e pela recepção, ao enviar dados

Figura 4.5: Diagrama ilustrando o esquema de funcionamento dos gerenciadores.

por um canal CBR com um PCR pequeno ela poderia ficar bastante tempo bloqueada enquanto o `send()` fosse completado (quanto menor o PCR, mais demora para enviar uma mesma quantidade de dados pela rede), levando a possíveis perdas na recepção de dados. Com *threads* este problema desaparece pois cada uma pode trabalhar em ritmo próprio.

Sem as *threads* a coleta de estatísticas também não seria tão refinada já que o *kernel*, através do pseudo-diretório `/proc` (seção B.14.3), informa apenas a perda total de dados e não a perda por *socket* individualmente. Quando temos uma *thread* cuja única função é ficar recebendo dados da rede garantimos que estes dados serão lidos. Caso eles tenham que ser descartados pelo fato do *buffer* estar cheio podemos registrar não só a quantidade de *bytes* como também outras informações relevantes, a exemplo do instante do evento ou o canal de comunicação usado. Nota-se portanto que as possibilidades de monitoramento também aumentam bastante com *threads*.

É sabido que realizar cópia de dados na memória é custoso. Diversos trabalhos expondo técnicas que reduzem este *overhead* estão disponíveis na Internet [36, 136]. Levando isto em consideração, o *buffer* foi implementado de maneira bastante otimizada evitando ao máximo cópias de memória, minimizando a latência de repasse de pacotes de uma rede à outra. Para isto, no momento da leitura dos dados da rede com `read()` já é informado o endereço definitivo onde os dados serão guardados dentro do *buffer*, de modo que nenhuma cópia de memória é feita dentro da camada de usuário, além é claro da cópia que é feita do espaço de *kernel* para a camada de usuário. Para utilização de interfaces com taxas de transmissão da ordem de Gbps maiores otimizações podem ser necessárias.

Através da opção 3 do programa controlador (*Print list of tunnels*) é possível obter uma listagem de todos os canais ativos de um gerenciador, com as máquinas envolvidas, os parâmetros de QoS, os tamanhos dos *buffers* e os estados das *threads*. Como pode ser visto na Figura 4.6, as quatro *threads* do canal recém criado estão em estado

```

Option: 3
=====
TunID: 0
atm0: 47.0005.80FFE1000000F20F3C4C.0020480E2493.00:10
atm1: 47.0005.80FFE1000000F21C4E74.0020480E2DD8.00:10
ip0e: dogmatix:1234
ip0a: donald:1234
ip1a: abracurcix:1234
iple: asterix:1234
Transmit side >>>Traffic Class:CBR
max_pcr:5000    min_pcr:0        max_cdv:0        max_sdu:0
Threads: E2A_E:ready    E2A_A:ready    A2E_A:ready    A2E_E:ready
bufLineSize0:1600      bufLines0:5
bufLineSize1:1600      bufLines1:5
=====
End of Tunnels Addresses Listing =====

```

Figura 4.6: Listagem dos canais ativos gerada pela opção "*Print list of tunnels*" do programa controlador.

"*ready*", o que significa que conseguiram fazer todas as inicializações e conexões necessárias e estão bloqueadas em uma chamada de sistema para receber dados da rede ou dormindo esperando chegarem dados no *buffer*.

Através da opção 4 do programa controlador (*Print Statistics of tunnel*) é possível visualizar as estatísticas de um canal em um dado momento. Logo após selecionar esta opção é necessário informar o identificador do canal, como pode ser visualizado na Figura 4.7.

```

Option: 4
Enter tunnel number:0
Tstamp:1571474213 pkts:10044 pkts_dropped:0 B:13217904 B_dropped:0
sleeps:9769 out_pkts:10044 out_B:13217904 max_utiliz:80 utiliz:0
Tstamp:1571474219 pkts:0 pkts_dropped:0 B:0 B_dropped:0
sleeps:1 out_pkts:0 out_B:0 max_utiliz:0 utiliz:0

```

Figura 4.7: Visualização das estatísticas de um canal em dado instante através da opção "*Print Statistics of tunnel*" do programa controlador.

Como muitas informações são mostradas na visualização das estatísticas de um túnel, construiu-se um programa gráfico especialmente para monitoramento destas informações. A interface gráfica deste programa é mostrada na Figura 4.8.

O programa gráfico mostra de maneira mais clara a quantidade de pacotes recebidos da rede *Ethernet* (Eth2ATM in) e enviados para a rede ATM (Eth2ATM out) para o canal em questão, assim como no sentido contrário (ATM2Eth). Também é mostrada a quantidade de pacotes descartados devido a sobrecargas no *buffer*, a quantidade de

Figura 4.8: Programa que permite o monitoramento contínuo das estatísticas de um canal.

bytes recebidos, enviados e descartados, a utilização atual e máxima dos *buffers* e a quantidade de vezes que a *thread* responsável pela retirada de dados do *buffer* foi colocada para dormir. Na Figura 4.8 havia tráfego apenas no sentido *Ethernet* → *ATM* e portanto no sentido contrário esta *thread* foi colocada para dormir uma única vez, pois assim que ela foi criada encontrou o *buffer* vazio, imediatamente dormiu e não foi acordada mais pois nunca foi colocado nada neste *buffer*. No sentido do fluxo de dados o número de vezes que esta *thread* foi colocada para dormir é um pouco menor que o número de pacotes por ela enviados, indicando que freqüentemente, mas nem sempre, ela envia um único pacote presente no *buffer* e volta a dormir por encontrar o *buffer* vazio.

O botão "*Settings*" abre uma janela que permite a seleção do gerenciador a receber a solicitação, do canal a ser monitorado e do intervalo de tempo a ser esperado entre uma amostragem e outra, conforme mostrado na Figura 4.9.

4.3 Resultados experimentais

Diversas grandezas foram estudadas experimentalmente com a implementação descrita na seção 4.2. Nas próximas seções são apresentados alguns dos resultados obtidos, como:

Figura 4.9: Janela que permite a seleção do gerenciador a ser contactado, do canal a ser monitorado e do intervalo de tempo entre uma amostragem e outra.

- perda de pacotes sob condições de *link* ocioso e saturado;
- latência média de comunicação;
- *jitter* (variação da latência de transmissão);
- influência do número e canais ativos na latência e no *jitter*;
- caracterização do tráfego gerado pela aplicação multimídia VLC.

4.3.1 Perda de pacotes

Para avaliar a perda de pacotes na implementação do modelo proposto montamos uma rede conforme ilustrado na Figura 4.10. O núcleo da rede é composto por duas chaves

Figura 4.10: Topologia de rede utilizada na avaliação da perda de pacotes.

ATM conectadas por fibra a 155 Mbps. A cada chave é conectado um gerenciador (Donald e Abracurcix) e a cada gerenciador é ligada uma máquina via *Fast Ethernet*, simulando duas redes locais. Todas as máquinas com interfaces ATM foram configuradas para participar de uma mesma rede *Classical IP* (CLIP). As interfaces ATM utilizadas eram todas ForeRunner LE com capacidade para transmitir 155 Mbps.

Utilizamos a aplicação multimídia VLC para a transmissão de um vídeo MPEG com áudio de duração de 30 segundos entre a Arthur2 e a Automatix de forma indireta, com o tráfego sendo transmitido com o auxílio dos gerenciadores. Maiores detalhes sobre o VLC são fornecidos na seção B.12.

Para abertura do canal entre os gerenciadores optamos por fazer uso de um pequeno aplicativo que envia uma solicitação de criação de um novo canal a um dos gerenciadores. Outra possibilidade seria incluir a chamada de uma rotina de biblioteca dentro do próprio aplicativo VLC.

Para verificar o comportamento do sistema em situação de congestionamento no *link* entre as chaves utilizamos Netperf. Maiores detalhes sobre Netperf são fornecidos na seção B.13. O Netperf foi configurado de modo a gerar tráfego UDP das máquinas Obelix e Dogmatix para a Asterix sobre CLIP, sempre na taxa máxima possível. Como o ATM é *full-duplex* tomou-se o cuidado de direcionar este tráfego no mesmo sentido do vídeo.

Sempre que necessário o CLIP abre um canal do tipo UBR diretamente para a máquina destino. Com isto, o tráfego gerado pelo Netperf que satura o *link* ATM poderá ser parcialmente descartado pelas chaves se assim lhes convier.

Através do número de pacotes transmitidos e recebidos disponibilizados pelos gerenciadores envolvidos pode-se mensurar a perda de pacotes no *link* ATM.

Solicitamos inicialmente um canal do tipo UBR entre os gerenciadores para obter um comportamento similar ao da Internet atual, sem garantias de entrega. As perdas de pacotes usando este canal em várias situações de tráfego são mostradas na Tabela 4.1. Observa-se que na ausência de tráfego adicional passando através do *link* o vídeo é transmitido sem perda alguma de pacotes, apesar do canal ser do tipo UBR. Porém,

Tabela 4.1: Tabela que mostra a perda percentual de pacotes através de um canal UBR e CBR entre gerenciadores em situações de tráfego diversas.

Tráfego Netperf proveniente da		Perda de pacotes (%) com canal	
Obelix	Dogmatix	UBR	CBR
Não	Não	0	0
Sim	Não	12,8	0
Sim	Sim	62,3	0

com o *link* sendo saturado pelo tráfego do Netperf proveniente da máquina Obelix já ocorre uma perda de pacotes de aproximadamente 13% e a qualidade do vídeo e áudio exibidos degrada significativamente. À medida que a competição pelo *link* aumenta, com tráfego sendo gerado também pela Dogmatix, a perda de pacotes também cresce.

Usando um canal CBR com PCR de 5000 células por segundo entre os gerenciadores¹ o comportamento é bem diferente, pois o vídeo é transmitido sem perda alguma de pacotes, não importando o volume de tráfego direcionado ao *link* comum, pois a taxa de 5000 células por segundo é garantida.

4.3.2 Latências de comunicação

Um dos parâmetros mais importantes em uma transmissão multimídia é a latência. Torna-se então fundamental estudar a latência introduzida pela inserção de dois gerenciadores no percurso do tráfego. Para isto foi produzido o programa *latency*, apresentado a seguir.

4.3.2.1 Programa *latency*

Para permitir a determinação da latência de comunicação entre duas aplicações multimídia que se comunicam através de dois gerenciadores foi produzido o programa *latency*, que em modo transmissor gera tráfego sintético e registra o instante de tempo em que cada mensagem é enviada com o auxílio de uma placa MultiKron II (descrita na seção A.1). O mesmo programa executando em outra máquina em modo receptor registra os instantes de tempo de chegada de cada mensagem com o auxílio de uma segunda placa MultiKron II.

Nas experiências seguintes as duas placas utilizaram um relógio comum que permite a inicialização simultânea dos contadores de tempo das duas placas através de um único botão, conforme mostrado na Figura 4.11.

O programa *latency* pode ser executado como transmissor ou receptor através das chaves *-s* e *-r* respectivamente. No modo transmissor ele cria tráfego sintético que possui características semelhantes ao tráfego gerado por aplicações multimídia reais, que é caracterizado pela forte presença de rajadas, como é ilustrado na seção 4.3.3 para a aplicação multimídia VLC.

Dentre as opções do programa *latency*, listadas na Figura 4.12, nem todas aplicam-se ao modo de transmissão e recepção simultaneamente. No modo transmissor é possível especificar o tamanho das mensagens (opção *-l*), quantas mensagens devem ser enviadas em cada rajada (opção *-n*), o tempo que deve ser aguardado (em microssegundos)

¹5000 células por segundo equivalem a aproximadamente 1,92 Mbps, não levando em consideração o *overhead* causado pela camada AAL5 (e superiores, quando for o caso).

Figura 4.11: Montagem experimental para medida da latência no envio direto e indireto.

Usage: latency Mode [Options]

Mode:

- s Send packets with timestamps
- r Receive packets with timestamps

Options:

- a <addr> Address to send to
 - f Force to receive anything
 - g Show gettimeofday() timestamps on receiver
 - i <n> Interval to wait between packets in usec
 - l <n> Buffer length
 - n <n> Number of packets to transmit in a burst
 - m <n> Number of bursts to transmit
 - p <n> Port to send to or to bind on
 - t <n> Time to wait between packets in a burst in usec
 - v Be verbose
 - V Print version
-

Figura 4.12: Opções do programa latency, usado para medir a latência na comunicação.

gundos) entre cada mensagem de uma mesma rajada (opção -t), o tempo entre o fim de uma rajada e o início da seguinte (opção -i) e o número de rajadas a serem enviadas (opção -m). Estas opções estão graficamente representadas na Figura 4.13.

Figura 4.13: Representação gráfica das opções do programa latency.

A Figura 4.14 ilustra a posição no programa latency onde é realizado o registro de tempo através de uma escrita em memória, tanto no modo cliente como servidor.

Figura 4.14: Posição no programa latency nos modos cliente e servidor onde é realizado o registro de tempo através de uma escrita em memória.

No modo cliente o registro é feito logo antes da rotina `sendto()` e no modo servidor logo após a rotina `recvfrom()`.

4.3.2.2 Latências com canal UBR entre gerenciadores

Para mensurar a influência dos gerenciadores na latência da comunicação sem levar aspectos como a suavização de tráfego em consideração, solicitamos inicialmente aos gerenciadores para que criassem um canal UBR entre eles, de maneira que o tráfego no trecho ATM ficaria unicamente limitado pela velocidade física do *link* (155 Mbps) e pelo desempenho dos gerenciadores.

Realizando medidas de tempo com o programa `latency` para mensagens com comprimento na faixa de 32 a 1024 *bytes*, transmitidas de modo direto em rajadas de 5 mensagens seguidas (`-n 5`), sem intervalo de espera entre o envio das mensagens de uma mesma rajada (`-t 0`) e com 20 *ms* entre uma rajada e outra (`-i 20000`) obtivemos as latências médias apresentadas na Figura 4.15. Observa-se que a latência média aumenta linearmente em função do tamanho da mensagem, sendo que em uma mesma rajada cada mensagem apresentou uma latência um pouco maior que a apresentada pela mensagem anterior.

Figura 4.15: Latência média para mensagens com comprimento na faixa de 32 a 1024 *bytes*, transmitidas de modo direto em rajadas de 5 mensagens a cada 20 *ms* via *Fast Ethernet*.

Realizando o mesmo experimento com as transmissões de modo indireto através de um canal UBR obtivemos as latências médias apresentadas na Figura 4.16. Novamente observou-se um aumento linear da latência com o tamanho da mensagem, sendo que as

Figura 4.16: Latência média para mensagens com comprimento na faixa de 32 a 1024 *bytes*, transmitidas de modo indireto em rajadas de 5 mensagens a cada 20 *ms* via canal UBR.

latências no modo indireto foram ligeiramente maiores que no modo direto, como era esperado (pois as mensagens precisam percorrer um caminho mais longo, em *hops*).

Uma vez que a latência apresentou um comportamento linear em função do tamanho da mensagem tanto no caso direto quanto no indireto, resolvemos estudar seu comportamento em maior profundidade apenas para mensagens bem pequenas e para mensagens razoavelmente grandes, ou mais precisamente, mensagens de 32 e 1024 *bytes*. O comportamento de mensagens com tamanhos intermediários pode ser previsto a partir do comportamento destes tamanhos "extremos".

Medidas de tempo para o envio de 5 mensagens com 32 *bytes* de comprimento por rajada em modo direto e indireto para seqüências de 5000 rajadas resultaram nas latências apresentadas na Figura 4.17 para as primeiras 30 mensagens. Observa-se que o comportamento da latência das mensagens nas diversas rajadas é bastante semelhante. As latências ficaram na faixa de 250 a 450 μs para as mensagens enviadas diretamente e na faixa de 500 a 850 μs para as mensagens enviadas através dos gerenciadores, sendo que em uma rajada cada mensagem apresenta uma latência um pouco maior que a latência apresentada pela mensagem anterior. Uma exceção é a primeira mensagem da primeira rajada, que demorou um pouco mais que nas demais rajadas. Isto provavelmente foi influência do sistema operacional que deve ter despendido tempo para inicializar *buffers* ou estruturas de dados internas para coordenar este fluxo de dados.

Apenas para comparação mostramos os tempos obtidos com o aplicativo *ping* na Figura 4.18, onde solicitamos que o campo de dados da mensagem *ping* tivesse 32

Figura 4.17: Latência para várias mensagens de 32 *bytes* enviadas em modo direto e indireto via canal UBR.

bytes de comprimento. A este campo de dados é adicionado um cabeçalho **ICMP** de 8 *bytes* e este conjunto é entregue à camada IP [43]. Por este motivo aparece na Figura 4.18 que para cada *ping* é recebida uma mensagem de 40 *bytes* como resposta. Nossa aplicação usa o protocolo UDP, e portanto à nossa mensagem de 32 *bytes* é adicionado um cabeçalho UDP que ocupa 8 *bytes*, conforme mostrado na seção 2.8, resultando em pacotes IP com 40 *bytes* de dados, que são intencionalmente de mesmo tamanho que os pacotes obtidos com *ping*.

```
andremuz@jan4:~> ping -n -c 5 -s 32 automatix
PING automatix.if.sc.usp.br (143.107.229.59) 32(60) bytes of data.
40 bytes from 143.107.229.59: icmp_seq=1 ttl=255 time=0.608 ms
40 bytes from 143.107.229.59: icmp_seq=2 ttl=255 time=0.484 ms
40 bytes from 143.107.229.59: icmp_seq=3 ttl=255 time=0.490 ms
40 bytes from 143.107.229.59: icmp_seq=4 ttl=255 time=0.491 ms
40 bytes from 143.107.229.59: icmp_seq=5 ttl=255 time=0.492 ms

--- automatix.if.sc.usp.br ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4040ms
rtt min/avg/max/mdev = 0.484/0.513/0.608/0.047 ms
```

Figura 4.18: Tempos obtidos com o aplicativo *ping*.

Notamos que com o *ping* a primeira mensagem também demorou mais que as seguintes e como estes tempos representam o tempo gasto para a mensagem ir para o destinatário e voltar (**RTT**), seus valores são aproximadamente o dobro dos valores

encontrados com o aplicativo *latency* e as placas MultiKron II para as primeiras mensagens de cada rajada em modo de transmissão direto.

O histograma das latências para as 25000 mensagens de 32 *bytes* enviadas em rajadas de 5 mensagens em modo direto é apresentado na Figura 4.19. Observa-se que o *jitter*, que corresponde à dispersão das curvas apresentadas na Figura 4.19 é menor que 5 μs . Um fato curioso observado na mesma figura e que mereceu melhor estudo foi o fato da quarta mensagem, apesar de geralmente apresentar uma latência por volta de 418 μs , às vezes apresentar uma latência significativamente menor, por volta de 355 μs . A explicação para este fenômeno é apresentada nos parágrafos seguintes.

Figura 4.19: Histograma das latências de 25000 mensagens de 32 *bytes* enviadas em rajadas de 5 mensagens em modo direto via *Fast Ethernet*.

Sabe-se que para realizar uma transmissão o Linux copia a mensagem a ser enviada do espaço de memória do usuário para o espaço de memória do *kernel*, o que certamente acaba contribuindo para o aumento da latência de transmissão². Para realizar esta cópia, pode ser necessário ao *kernel* alocar memória ou inicializar ou modificar estruturas de dados para armazenamento da mensagem. O tempo gasto com estas tarefas pode ser significativamente maior que o tempo gasto na cópia das pequenas mensagens de 32 *bytes*.

No lado receptor, ao analisar o atraso do recebimento das mensagens em relação ao instante do recebimento da primeira mensagem de cada rajada, conforme ilustra a Figura 4.20, nota-se que o atraso entre a recepção de cada mensagem com relação

²Diferentes técnicas já foram propostas e implementadas para diminuir a latência de transmissão em aplicações paralelas de alto desempenho [115], onde a latência pode representar um dos fatores mais determinantes na performance da aplicação.

Figura 4.20: Histograma do atraso do recebimento das mensagens em relação ao instante do recebimento da primeira mensagem de cada rajada. Total de 25000 mensagens de 32 bytes enviadas em rajadas de 5 mensagens em modo direto via *Fast Ethernet*.

à anterior é uniforme e pouco inferior a 100 μs . Já no lado transmissor, conforme mostrado na Figura 4.21, a separação temporal entre o envio de cada mensagem é

Figura 4.21: Histograma do atraso do envio das mensagens em relação ao instante do envio da primeira mensagem de cada rajada. Total de 25000 mensagens de 32 bytes enviadas em rajadas de 5 mensagens em modo direto via *Fast Ethernet*.

Figura 4.22: Histograma das latências de 25000 mensagens de 32 *bytes* enviadas em rajadas de 5 mensagens em modo indireto via canal UBR.

menos uniforme e menor que os quase 100 μs observados no lado receptor. Isto indica que no lado transmissor, após serem copiados para o espaço de memória do *kernel*, os pacotes provavelmente tiveram que esperar em fila até o momento em que cada um deles pôde efetivamente ser transmitido pela rede. Ainda na Figura 4.21 verifica-se também que o registro do instante de envio da quarta mensagem às vezes atrasou mais que o usual. Como o registro de tempo no lado transmissor é feito logo antes do `sendto()` (Figura 4.14), conclui-se que o `sendto()` referente à terceira mensagem às vezes demorou mais que o usual para ser completado, atrasando o envio da quarta mensagem, fazendo com que esta quarta mensagem fosse inserida mais tarde na fila de pacotes a serem transmitidos e portanto causando uma aparente menor latência até a aplicação receptora, uma vez que esta quarta mensagem perdeu menos tempo na fila até ser enviada de fato pela rede³. Como queríamos realmente mensurar a latência percebida de aplicação para aplicação não aprofundamos mais as investigações das causas da variação no tempo gasto pelo *kernel* para completar as transmissões. Para isto técnicas como as empregadas em [96] seriam necessárias.

O histograma das latências para 25000 mensagens de 32 *bytes* em rajadas de 5 mensagens transmitidas em modo indireto por um canal UBR é apresentado na Figura 4.22. Nota-se que neste caso o *jitter* é menor que 50 μs .

Para mensagens com 1024 *bytes* de comprimento nas mesmas separações tempo-

³Na verdade, pelo fato do Linux padrão (distribuído em <http://www.kernel.org/>) não ser um sistema operacional de tempo real, não há *a priori*, garantias de um tempo máximo a ser gasto pelo *kernel* para completar uma determinada tarefa.

Figura 4.23: Latência para várias mensagens de 1024 *bytes* enviadas em rajadas de 5 mensagens em modo direto e indireto via canal UBR.

rais anteriores obtivemos as latências apresentadas na Figura 4.23 para as primeiras 30 mensagens. O comportamento observado foi bastante semelhante ao caso das mensagens de 32 *bytes* e também houve um aumento na latência da transmissão em modo indireto em relação ao modo direto. Porém, este aumento foi proporcionalmente menor que para mensagens com 32 *bytes* de comprimento.

O histograma das latências para as mensagens de 1024 *bytes* enviadas nos modos direto e indireto são apresentados nas Figuras 4.24 e 4.25, respectivamente. O *jitter* ficou abaixo de 5 μs no modo direto e abaixo de 10 μs em modo indireto.

4.3.2.3 Latências com canal CBR entre gerenciadores

Como a QoS na WAN ATM só pode ser conseguida por meio de canais CBR (só canais do tipo UBR e CBR estão disponíveis em Linux, conforme explicado na seção 2.2), solicitamos que fosse estabelecido um canal CBR com capacidade para 5000 células por segundo entre os gerenciadores. Com esta montagem realizamos medidas de latência no modo de transmissão indireto para mensagens pequenas e grandes em duas situações de tráfego bem distintas. Na primeira não há nenhum tráfego adicional competindo pelo *link* entre as chaves ATM. Na segunda é colocado tráfego adicional suficiente para saturar o *link* por completo. Estas duas situações opostas revelam as características extremas que podemos esperar tanto da latência quanto do *jitter*. Em ambas as situações é esperado que a latência com o uso do canal CBR seja maior que na utilização de um canal UBR devido à suavização de tráfego imposta pelos canais

Figura 4.24: Histograma das latências de 25000 mensagens de 1024 *bytes* enviadas em rajadas de 5 mensagens em modo direto via *Fast Ethernet*.

Figura 4.25: Histograma das latências de 25000 mensagens de 1024 *bytes* enviadas em rajadas de 5 mensagens em modo indireto via canal UBR.

CBR, especialmente para mensagens grandes, segmentadas em muitas células.

O histograma das latências para 25000 mensagens de 32 *bytes* em rajadas de 5 mensagens transmitidas em modo indireto pelo canal CBR sem tráfego adicional competindo pelo *link* ATM é apresentado na Figura 4.26. Novamente a latência de cada mensagem dentro de uma rajada é maior que a latência da mensagem anterior. Com-

Figura 4.26: Histograma das latências de 25000 mensagens de 32 bytes enviadas em rajadas de 5 mensagens em modo indireto via canal CBR com capacidade para 5000 células por segundo com *link* ocioso.

parando esta figura com a Figura 4.22 nota-se que de fato a suavização do tráfego é responsável por um aumento significativo da latência. Com UBR a primeira mensagem teve uma latência média de 485 μs , sendo que com a suavização de tráfego do CBR passou para 600 μs .

As mesmas mensagens, quando transmitidas em uma situação de extremo congestionamento, apesar de terem sua transmissão assegurada, têm sua latência e *jitter* aumentados, conforme pode ser constatado na Figura 4.27. Para a obtenção destes resultados foram criados mais 29 canais CBR com PCR equivalente a 5000 células por segundo entre os gerenciadores. Por cada um destes canais foi transmitido tráfego constituído de pacotes de 1316 bytes a uma taxa aproximada de 1,28 Mbps gerado por uma aplicação criada exclusivamente para este fim a partir da Dogmatix (Figura 4.11). Este tamanho de pacote e taxa de transmissão são iguais aos gerados pela aplicação multimídia VLC (maiores detalhes sobre o tráfego gerado por esta aplicação são apresentados na seção 4.3.3). O fluxo através destes canais teve também como objetivo testar o desempenho dos gerenciadores em regime de trabalho intenso. Vale a pena lembrar que para cada canal aberto são criadas quatro *threads* em cada gerenciador, e o escalonamento destas poderia afetar a latência e/ou o *jitter* das comunicações.

A escolha por uma aplicação dedicada para geração deste tráfego sintético ao invés do uso da própria aplicação VLC teve bons motivos. Em primeiro lugar, é mais fácil controlar o tráfego através da aplicação dedicada, pois diversas instâncias deste programa podem ser iniciadas facilmente por um único *script*. Em segundo lugar, mesmo

Figura 4.27: Histograma das latências de 25000 mensagens de 32 *bytes* enviadas em rajadas de 5 mensagens em modo indireto via canal CBR com capacidade para 5000 células por segundo com *link* totalmente ocupado por mais 29 canais transmitindo tráfego sintético e Netperf (UDP) sobre CLIP.

quando a aplicação VLC usa como origem de um vídeo um arquivo MPEG armazenado em disco, o VLC ainda codifica o vídeo para transformá-lo em um fluxo capaz de ser transmitido pela rede⁴. Isto certamente pode exigir ciclos de CPU e consideráveis movimentações de dados na memória. A aplicação geradora de tráfego sintético pode simplesmente transmitir um mesmo bloco de dados repetidas vezes, sem usar CPU ou banda de E/S desnecessária.

Para garantir que toda a banda disponível no *link* ATM fosse utilizada, colocamos o Netperf (seção B.13) na máquina Obelix (Figura 4.11) enviando tráfego na máxima taxa possível para a máquina Asterix através de UDP sobre CLIP. Como este tráfego segue a mesma direção do tráfego analisado ele compete por banda no *link* ATM entre as chaves. Como mostrado na seção 4.3.1, células do tráfego Netperf ao competirem com células do tráfego dos canais CBR são descartadas na chave ATM, mas o objetivo de saturar o *link* por completo é alcançado plenamente.

Para mensagens com 1024 *bytes* de comprimento nas mesmas separações temporais das medidas anteriores através do mesmo canal CBR com *link* ocioso obtivemos as latências apresentadas na Figura 4.28. Observa-se que para mensagens deste tamanho as latências são muito maiores, já que para transmitir 1024 *bytes* sobre AAL5 são necessárias 22 células, que são adequadamente espaçadas no tempo pelo suavizador de tráfego. Fica claro que no mundo ATM é interessante que as aplicações multimídia

⁴Tipicamente o formato MPEG-TS é usado para transmitir vídeo pela rede.

Figura 4.28: Histograma das latências de 25000 mensagens de 1024 *bytes* enviadas em rajadas de 5 mensagens em modo indireto via canal CBR com capacidade para 5000 células por segundo com *link* ocioso.

usem mensagens pequenas para reduzir a latência de transmissão. Porém, isto vai de encontro com o que é geralmente feito na Internet: para diminuir o *overhead* dos protocolos UDP/IP a maioria das aplicações em uso na Internet costuma usar mensagens relativamente grandes.

Por fim, a Figura 4.29 mostra o que ocorre quando paralelamente à transmissão sendo analisada saturamos o *link* com tráfego sintético e Netperf, como foi feito anteriormente com mensagens de 32 *bytes*. As latências médias aumentam um pouco e o *jitter* também aumenta, como pode ser observado pelo aumento na dispersão das distribuições.

Figura 4.29: Histograma das latências de 25000 mensagens de 1024 *bytes* enviadas em rajadas de 5 mensagens em modo indireto via canal CBR com capacidade para 5000 células por segundo com *link* totalmente ocupado por mais 29 canais transmitindo tráfego sintético e Netperf (UDP) sobre CLIP.

4.3.3 Comportamento do tráfego do VLC

Para ter uma idéia do comportamento do tráfego gerado pelo aplicativo VLC resolvemos executá-lo em uma máquina de tal forma que o tráfego dele fosse enviado diretamente a uma máquina executando o aplicativo *latency* com a opção *-f* para que qualquer mensagem chegando em uma determinada porta tivesse seu instante registrado com a placa MultiKron II.

4.3.3.1 Arquivo de vídeo em disco

Utilizamos inicialmente como fonte de vídeo um arquivo em formato MPEG-I com resolução de 320x240, 29,97 fps e taxa de 409,6 kbps armazenado em disco local. Selecionando mp4v como opção de transcodificação com limite máximo de 3072 bps registramos o instante de chegada de cada mensagem e também o tamanho da mesma. A Figura 4.30 ilustra o tamanho das mensagens em função do instante de chegada da mensagem em um intervalo de 0,6 segundos. Este tempo é suficiente para ter uma idéia do comportamento geral do tráfego. Observa-se que todas as mensagens têm exatamente 1316 *bytes* de comprimento e são transmitidas em grupos uniformemente distribuídos no tempo. A Figura 4.31 mostra as mesmas informações em um intervalo de tempo menor para melhor visualização do número de mensagens dentro dos grupos.

Figura 4.30: Tamanho das mensagens transmitidas pelo VLC em função do instante de chegada da mensagem. A fonte do vídeo é um arquivo MPEG-I com resolução de 320×240 e 29,97 fps armazenado em disco local. mp4v com limite máximo de 3072 bps.

Figura 4.31: Zoom da Figura 4.30 ilustrando a separação temporal das mensagens dentro de uma mesma rajada. A fonte do vídeo é um arquivo MPEG-I com resolução de 320×240 e 29,97 fps armazenado em disco local. mp4v com limite máximo de 3072 bps.

4.3.3.2 Vídeo em tempo real

Vídeo foi capturado com uma filmadora Canon conectada a uma placa WinCast TV (descrita na seção A.5) e transmitido diretamente para a máquina executando o aplicativo *latency* com a opção *-f*. Novamente selecionamos mp4v como opção de transcodificação com limite máximo de 3072 bps. A Figura 4.32 ilustra o tamanho das mensagens em função do instante de chegada de cada mensagem em um intervalo de 2,6 segundos. Novamente observa-se que todas as mensagens têm exatamente 1316 bytes de comprimento e são transmitidas em grupos uniformemente distribuídos no tempo. Porém, a separação temporal observada entre os grupos foi maior que no caso com o vídeo armazenado em disco visto anteriormente.

Figura 4.32: Tamanho das mensagens transmitidas pelo VLC em função do instante de chegada da mensagem. O vídeo é capturado em "tempo real" com uma filmadora Canon. mp4v com limite máximo de 3072 bps.

A Figura 4.33 mostra as mesmas informações em um intervalo de aproximadamente 0.05 segundos para melhor visualização do número de mensagens dentro dos grupos. Observa-se que há 43 mensagens neste grupo.

A constatação do agrupamento de pacotes em grupos durante a transmissão motivou a criação do aplicativo *latency* descrito na seção 4.3.2.1.

4.4 Conclusões

Concluimos que o modelo proposto de fato viabiliza a reserva de recursos de um *backbone* ATM a partir de máquinas pertencentes a LANs incapazes de oferecer QoS. Ao

Figura 4.33: Zoom da Figura 4.32 ilustrando a separação temporal das mensagens dentro de uma mesma rajada. O vídeo é capturado em "tempo real" com uma filmadora Canon. mp4v com limite máximo de 3072 bps.

contrário das soluções apontadas em outros trabalhos, nossa solução não exige a adoção de equipamentos difíceis de produzir ou alterar, permitindo sua adoção em cenários práticos reais. Com uma implementação do modelo proposto foi demonstrado através de verificações experimentais que o modelo permite transmitir áudio e vídeo através de WANs ATM sem perda nenhuma de qualidade mesmo em situações de intenso congestionamento. A latência introduzida pelo uso de canais do tipo CBR, apesar de bastante considerável para pacotes grandes, não impede que eles sejam usados mesmo para tráfego em tempo real. Latências menores podem ser obtidas através do uso de mensagens menores, o que nas WANs baseadas em IP tipicamente tem sido desencorajado devido ao alto *overhead* dos protocolos UDP/IP.

Um artigo baseado no trabalho desenvolvido com este modelo foi aceito no *IX Workshop de Gerência e Operação de Redes e Serviços (WGRS 2004)* [99], evento paralelo ao 22º *Simpósio Brasileiro de Redes de Computadores* promovido pela Sociedade Brasileira de Computação (SBC), realizado em Gramado em maio de 2004.

CAPÍTULO

5

Simulador de redes deflexivas

Este capítulo descreve parte do trabalho que foi desenvolvido com QoS em redes deflexivas durante o período do doutorado sanduíche (maio/2004 a março/2005) no Departamento de Ciência da Computação da Universidade de Alberta, sob supervisão do prof. Dr. Pawel Gburzyński.

5.1 Redes deflexivas

Considere um roteador no núcleo da rede que está na iminência de reencaminhar um pacote que ele acaba de receber da rede. Independentemente do algoritmo de roteamento empregado esta é a lista completa das opções disponíveis ao roteador:

1. Colocar o pacote na fila da porta de saída que vai enviá-lo ao destino em um futuro breve pela rota "mais interessante";
2. Descartar o pacote se não houver espaço suficiente para seu armazenamento (se o *buffer* estiver cheio);
3. Colocar o pacote na fila de uma porta de saída que seja considerada uma escolha secundária (através do algoritmo de escolha de rotas utilizado).

Com o paradigma de reencaminhamento convencional baseado em rotas pré-estabelecidas, a terceira possibilidade é normalmente excluída: o roteador não tem liberdade para escolher uma porta de saída. Os esforços na maximização dos recursos de rede são colocados na descrição do que seria a "melhor rota" até o destino em questão e no melhor algoritmo de policiamento no roteador.

A terceira opção só é permitida em redes deflexivas, onde os caminhos percorridos por diferentes pacotes da mesma sessão podem diferir de uma maneira *a priori* imprevisível. Tradicionalmente as redes deflexivas têm sido estudadas como sistemas síncronos, exemplificados pelas redes MSN (do inglês *Manhattan-Street Networks*) [85, 86, 88, 89, 90] e redes *Shuffle-exchange* [87, 75], onde as decisões de

Figura 5.1: Topologia da rede MSN.

roteamento afetam um conjunto de pacotes chegando às portas de entrada do roteador em um único ciclo. O problema do roteamento neste cenário [60] é definido como encontrar a melhor atribuição (permutação) destes pacotes às portas de saída, geralmente uma que minimiza a soma do comprimento dos caminhos mais curtos até os destinos. Devido à regularidade das topologias consideradas como *backbones* para redes deflexivas e ao grau dos nós relativamente baixo (o número de *links* de entrada e saída – veja a Figura 5.1), o problema de otimização do roteamento geralmente é equivalente a um conjunto de regras simples [88].

O estudo da deflexão na WAN recebeu pouca atenção no passado. Primeiro porque QoS em grandes redes irregulares foi geralmente associado ao provisionamento de recursos através de um caminho conhecido. Segundo porque a maioria dos estudos de performance em redes deflexivas foram feitos com roteamento síncrono em redes de formato regular operando com o tempo discretizado em *slots*, de forma que os resultados desses estudos não têm sido considerados interessantes (ou mesmo aplicáveis) para redes mais realísticas, a exemplo da Internet.

Alguns estudos anteriores [50, 49, 105, 106] tiveram por objetivo remover os pré-requisitos rígidos do roteamento por deflexão e trazer este conceito mais perto de uma implementação possível em redes irregulares reais. O roteamento por deflexão pode ser implementado de maneira assíncrona em redes irregulares onde um pacote é enviado de cada vez de modo que assim se evita a complexidade combinatorial do caso síncrono [50]. A colocação de pequenos *buffers* nos roteadores aumenta significativamente a qualidade das decisões de roteamento [41], levando a taxa máxima de transmissão possível da rede ao mesmo nível alcançado pela forma tradicional de roteamento usando *buffers* infinitos. O teorema de escalabilidade para redes deflexivas [33] diz que, por princípio, o roteamento por deflexão escala bem com o aumento do tamanho da rede. É possível implementar mecanismos de *feedback* e alocação de largura de banda que tornem redes deflexivas apropriadas para aplicações em tempo real [105]. Também há esquemas de deflexão que impõem limites rígidos no comprimento dos

caminhos ao mesmo tempo que mantêm a filosofia da deflexão que postula que não deva ser perdido nenhum pacote na rede [49].

Uma aparente falha do roteamento por deflexão, e a principal causa da falta de interesse neste conceito como um paradigma de roteamento com potencial para redes reais é a inerente imprevisibilidade dos caminhos. Em primeiro lugar, torna-se difícil limitar a latência e o *jitter*: um pacote pode acabar percorrendo um caminho muito mais longo que o menor caminho, e pode até mesmo ficar vagando pela rede, visitando os mesmos nós intermediários mais de uma vez. Em segundo lugar, os pacotes de uma mesma sessão podem chegar ao destino fora de ordem. Por isso, *reassembly buffers* são necessários para colocá-los novamente na ordem correta.

Desde que a topologia da rede não seja patológica, as redes deflexivas (mesmo as redes assíncronas) comportam-se de maneira muito estável, mesmo quando recebem uma carga tão alta que tende a saturá-las [50]. Apesar da possibilidade de um pacote ocasionalmente sofrer um grande atraso, a maioria dos pacotes tende a seguir um caminho de comprimento próximo do caminho mais curto. Formalmente, do ponto de vista dos requisitos de QoS de uma sessão isócrona, um pacote muito atrasado é considerado perdido. Em uma rede do tipo *store-and-forward* os pacotes também podem ser perdidos – por causa da falta de espaço nos *buffers* ou por causa de atrasos excessivos causados pela bufferização. Então não existe uma diferença fundamental entre os dois paradigmas de roteamento: defletir um pacote do caminho mais curto é equivalente a colocá-lo em um *buffer* na rede. Em ambos os casos armazenamento excessivo de pacotes na rede leva a perdas. Conceitualmente a escolha é entre armazenar explicitamente no espaço dos *buffers* nos roteadores ou armazenar no "*buffer virtual*" implícito no produto *largura de banda* $\times$ *atraso* dos *links* para os quais os pacotes são defletidos. Em ambos os casos o atraso causado pelo armazenamento do pacote pode contribuir à sua chegada tardia no destino. Além disso, no primeiro caso o *buffer* pode encher e o pacote pode ser imediatamente perdido.

A falta de circuitos virtuais explícitos na camada de rede no roteamento IP torna natural considerar deflexão como uma das opções se a saída preferida estiver congestionada. Quando esta opção é considerada pode-se esperar que o tamanho dos *buffers* nos roteadores pode ser reduzido, uma vez que a deflexão vai parcialmente fazer o papel da bufferização. Por outro lado, a necessidade de retirar os pacotes (desordenados) em ordem no destino pode impor a necessidade de um espaço de armazenamento para reordenamento dos pacotes. Desta forma a adoção da deflexão pode também ser vista como uma troca do espaço de armazenamento nos roteadores pelo espaço de armazenamento nos nós finais.

Em [29] é argumentado que deflexão pode ser um *approach* fundamentalmente melhor que bufferização excessiva ou descarte em caso de congestionamento. Em

primeiro lugar, gerenciar o *buffer* da cada sessão isoladamente no destino é consideravelmente mais fácil (e melhor definido como problema) que gerenciar o espaço de armazenamento compartilhado no roteador. Isto acontece porque o destino usa o *buffer* para uma única sessão de forma que as características de QoS obtidas podem ser monitoradas com máxima fidelidade. Em segundo lugar, se a sessão suportar pacotes chegando fora de ordem (quando pacotes puderem ser processados independentemente, por exemplo), o destino não precisa se incomodar com *reassembly buffers*, enquanto que o roteador tentaria "consertar o que não está quebrado" e armazenar os pacotes na tentativa de oferecer uma entrega ordenada que não é necessária. Com deflexão o espaço de armazenamento nos roteadores pode ser visto como um conjunto compartilhado de *buffers* globais e a rede inteira irá automaticamente distribuir o problema da bufferização entre todos os nós. Uma vez que o propósito das redes é prover compartilhamento de recursos entre as aplicações que delas fazem uso, faz sentido tentar empregar o mesmo princípio ao seu modo de operação interno.

5.2 Modelo empregado

No nosso modelo, uma máquina VoIP origem emprega codificação PCM (do inglês *Pulse Code Modulation*) com supressão de silêncio. O tráfego consiste de intervalos de fala (*talk spurts*) alternados com intervalos de silêncio, cujas distribuições seguem uma distribuição normal com médias $1,2s$ e $1,8s$ respectivamente. A duração da sessão também segue uma normal, com média de 1 minuto. Quando uma sessão chega ao fim, outra é imediatamente iniciada com um novo par de nós de origem e destino. O número total de sessões ativas é mantido constante e representa a carga oferecida à rede. O comportamento médio de todas as sessões VoIP é observado por um longo período (uma hora, por exemplo).

Assumimos que um sinal de voz codificado a 64 Kbps é coletado por aproximadamente $20ms$ e colocado em blocos de 160 bytes. Uma vez que a maioria dos sistemas VoIP usa a pilha de protocolos RTP/UDP/IP [56], adicionamos 40 bytes de *overhead* a cada pacote representando o *overhead* imposto pelos cabeçalhos dos protocolos RTP(12)/UDP(8)/IP(20).

Para fluxos de voz com supressão de silêncio é comum definir o instante inicial de *playout* no início de cada *talk spurt*, minimizando o impacto do *jitter* da sessão sobre a qualidade da voz. Nosso modelo permite ajustar o *playout threshold*, que é definido como o intervalo de tempo entre a recepção do primeiro pacote de um *talk spurt* e o início do *playout*. Pacotes subsequentes do mesmo *talk spurt* são retirados na mesma taxa (conhecida) na qual eles são gerados na origem, independentemente da taxa de chegada no destino.

Cada nó é capaz de atuar como um roteador e uma máquina de destino ao mesmo tempo, isto é, ele pode ser origem e/ou destino de uma sessão VoIP. Para expressar o *tradeoff* do espaço de armazenamento, representamos a quantidade total de memória a ser usada como *buffer* na rede por $B + b$, onde B denota a quantidade total de memória distribuída igualmente entre as máquinas destino (para ser usada como *reassembly buffers*) e b representa a quantidade de memória distribuída igualmente entre os *links* de saída de todos os roteadores. Apesar de cada nó implementar ambas as funcionalidades, suas operações de roteador e máquina destino são isoladas e os dois *buffers* são separados. Nas nossas simulações, a quantidade total de memória na rede, $B + b$ é mantida fixa durante toda a simulação, enquanto que a razão B/b , que determina a proporção entre as memórias usadas nas duas categorias de armazenamento, é variada durante a simulação.

Dois modelos de roteamento básico são considerados. No modelo tradicional de caminho único, os pacotes são sempre encaminhados ao longo de um único caminho mais curto que conecta a origem ao destino. É formalmente assumido que o caminho é selecionado na origem, isto é, os roteadores não têm flexibilidade para mudar o caminho. Como um variante do roteamento tradicional (e um representante dos esquemas de roteamento de caminho alternativo [67]), consideramos uma estratégia simples onde a origem conhece uma segunda melhor rota (disjunta da primeira rota pelo menos no primeiro *link*) e a utiliza sempre que a primeira aparenta estar congestionada.

Com roteamento deflexivo, cada roteador ao longo do caminho da origem ao destino tem permissão para reencaminhar qualquer pacote para qualquer um dos seus *links* de saída. Para esta operação seus *links* de saída são ordenados de acordo com o comprimento do menor caminho possível da origem para o destino seguindo o *link* em questão. O pacote é colocado na fila para encaminhamento no melhor *link* disponível (ou seja, no caminho mais curto) cujo *buffer* não estiver cheio.

Nas nossas simulações usamos redes de topologias regulares (candidatas tradicionais em redes deflexivas) assim como redes de topologia irregular, mais próximas de fragmentos típicos da Internet. As últimas foram obtidas com assistência do *software Network Manipulator (nem)* [81], que extrai grafos de tamanho especificado pelo usuário de mapas reais da Internet. Acredita-se que estes grafos apresentem as mesmas propriedades estatísticas encontradas em mapas extraídos da Internet [80].

Por simplicidade todos os *links* têm a mesma largura de banda de 1 Mbps. Em topologias regulares, os *links* têm um produto *largura de banda* $\times$ *capacidade* equivalente a 2 pacotes (por volta de 640 km) e nas topologias irregulares a capacidade do *link* é determinada pelas posições aleatórias dos nós geradas pelo *nem*.

5.3 Implementação

Construímos um simulador orientado a eventos na linguagem C++ fazendo uso de orientação a objetos e da biblioteca padrão STL (seção B.2) para implementar as diversas estruturas de dados empregadas no simulador. Esforço foi feito para tornar o simulador bastante flexível para que ele pudesse ser facilmente adaptado a novas situações que surgiam durante a pesquisa.

Uma das partes que mereceu maior atenção durante a construção do simulador foi a fila responsável pelo armazenamento dos eventos. Ela deve sempre estar ordenada de acordo com o *timestamp* dos eventos futuros armazenados, pois este *timestamp* determina o instante de início de cada um dos eventos. A fila de eventos tem um comportamento bastante dinâmico pois em alguns momentos atinge grandes tamanhos pela necessidade de armazenar muitos eventos simultaneamente, sendo que em outros fica bastante reduzida. A eficiência da implementação desta fila torna-se crítica pois ela é usada durante toda a simulação. Cada vez que um evento é criado ou destruído (removido da fila) os demais precisam continuar ordenados, para que se consiga localizar com eficiência o próximo evento a ocorrer. A STL provê a implementação de uma classe, chamada *priority_queue*, que faz exatamente isto. A implementação de *priority_queue* que foi usada é muito eficiente pois utiliza uma estratégia de alocação de memória sempre crescente, de maneira que mesmo quando a fila diminui de tamanho a memória previamente ocupada pela fila não é liberada novamente. Isto evita o desperdício de CPU e a possível fragmentação de memória causados por repetidas alocações e desalocações de memória. No caso da fila de eventos essa é uma característica interessante pois ela costuma crescer e diminuir de tamanho com bastante frequência no decorrer da simulação.

5.4 Resultados

5.4.1 Topologia retangular

A topologia retangular mostrada na Figura 5.2 é similar à topologia bidirecional (toroidal) da rede MSN [32], exceto pelas bordas, isto é, pela ausência dos *links* fechando o retângulo em um torus. Nos experimentos ilustrados na Figura 5.3(a), o *playout threshold* é colocado na metade do tamanho do *reassembly buffer*. A figura mostra o comportamento da perda de pacotes na rede regular à medida que a maior parte do espaço de armazenamento (B/b) desloca-se dos roteadores para os destinos. A curva $k = 1$ corresponde ao roteamento clássico de melhor esforço onde somente um caminho está disponível para cada destino. Se não há espaço suficiente no *buffer* de saída o

Figura 5.2: Topologia retangular.

pacote é descartado. $k = 2$ representa o caso onde a origem tem dois caminhos mais curtos distintos para os destinos, enquanto cada nó interno (roteador) atua exatamente da mesma maneira como no caso $k = 1$. A origem seleciona o segundo caminho se o *buffer* associado ao primeiro caminho (preferido) estiver cheio. Esta pode ser considerada uma forma rudimentar de roteamento QoS (*QoS routing*) [73].

Com deflexão nota-se claramente uma tendência de perda decrescente à medida que o espaço dedicado aos *buffers* é tirado dos roteadores e atribuído aos destinos. Isto não acontece com o roteamento tradicional pois na ausência de deflexão os roteadores estão mais desesperados em armazenar os pacotes que não podem ser imediatamente encaminhados ao único *link* de saída congestionado.

Um problema óbvio com *reassembly buffers* grandes nos destinos é o grande atraso do *playout* que, acima de um determinado valor, torna o atraso da sessão toda alto demais para ser considerado aceitável pela aplicação. Se insistirmos em fazer o *playout threshold* proporcional ao tamanho do *buffer* então o aumento da razão B/b (à medida que os *buffers* são deslocados dos roteadores para os destinos) irá inevitavelmente resultar em crescentes atrasos de *playout*. Como mostrado na Figura 5.3(b), praticamente todo o atraso percebido pela sessão vem dos *buffers* de *playout*. Manter um atraso razoavelmente baixo é um critério de QoS importante, e por isso pode fazer mais sentido manter o atraso de *playout* em um valor fixo – grande o bastante para manter a taxa de perdas aceitável e pequeno o bastante para evitar atrasos grandes demais.

Curiosamente, ocorre que o aumento no atraso do *playout* nos destinos acaba compensando a redução no espaço dos *buffers* nos roteadores. Considere a Figura 5.4(a), que mostra a porcentagem das sessões com QoS aceitável com um atraso de *playout* fixado em um tempo equivalente à chegada de 4 pacotes, independentemente do tamanho do *reassembly buffer* no destino. Consideramos "aceitável" uma taxa de perdas abaixo de 5%, que é tolerada pela maioria dos decodificadores de áudio sem perda

Figura 5.3: Topologia retangular com *threshold* igual a metade do tamanho do *reassembly buffer*: (a) taxa de perda e (b) atraso.

significativa na qualidade da voz¹.

A Figura 5.4 de fato demonstra que os benefícios (flexibilidade) da deflexão mais que compensam a redução no espaço dos *buffers* nos roteadores. A aparente imprevisibilidade da recepção devido à diversificação das rotas de pacotes individuais não exige um aumento significativo no tamanho dos *reassembly buffers*, certamente não na mesma proporção da redução dos *buffers* nos roteadores. Note que na região onde o roteamento tradicional falha completamente a deflexão ainda é capaz de prover um serviço razoável a um número grande de sessões. A Figura 5.4(b) mostra os respec-

¹Em geral existe um *tradeoff* entre o nível de compactação e a tolerância a perdas nos diversos algoritmos de compressão de áudio. Para uma dada qualidade sonora, algoritmos que conseguem uma melhor taxa de compressão costumam ser menos toleráveis a perdas que algoritmos com taxas de compactação menos eficientes e vice-versa. 5% de perda é um valor aceitável para a maioria dos decodificadores de áudio.

Figura 5.4: Topologia retangular com 200 sessões VoIP simultâneas e *threshold* igual a 4 slots: (a) fração de sessões VoIP com QoS aceitável e (b) atraso fim-a-fim.

tivos atrasos que, no caso da deflexão, ficam em uma faixa aceitável para aplicações VoIP típicas, com pequenos *buffers* nos roteadores. Pode-se especular que latências mais baixas podem ser obtidas se alguma forma de predição de atraso for empregada, como proposto em [47, 123].

A Figura 5.5 ilustra o que ocorre sob cargas mais altas (350 sessões concorrentes comparando com 200 na Figura 5.4). Em termos do número de sessões aceitáveis, o roteamento deflexivo consistentemente apresenta um desempenho melhor que o roteamento tradicional. Mesmo que o atraso fim-a-fim com deflexão tenda a ser levemente maior que para $k = 2$ (com *buffers* pequenos nos roteadores), o número total de sessões com uma taxa de perdas aceitável é consistentemente maior com deflexão. Também pode-se ver que a melhor distribuição de *buffers* para o roteamento clássico é por volta de $B/b = 384/16$, e para a deflexão é $398/2$. Confirmando observações feitas

Figura 5.5: Topologia retangular com 350 sessões VoIP simultâneas e *threshold* igual a 4 slots: (a) fração de sessões VoIP com QoS aceitável e (b) latência fim-a-fim.

em [41, 50, 49], a deflexão pode se beneficiar com alguma forma rudimentar de armazenamento nos roteadores. Em comparação com o caso completamente sem *buffers*, pequenos *buffers* trazem uma melhora considerável, enquanto que *buffers* grandes não trazem maiores ganhos.

Usando as melhores razões B/b encontradas nos experimentos anteriores, a Figura 5.6 mostra o impacto da carga oferecida na QoS das sessões VoIP. Uma vez que o número de sessões VoIP ultrapassa 250, o roteamento clássico por único caminho claramente oferece uma QoS pior que a deflexão. Deflexão por outro lado fornece QoS suficiente mesmo a cargas mais altas, alcançando aproximadamente 400 sessões antes da QoS rapidamente deteriorar. Nota-se que para $k = 1$ a maioria dos pacotes é perdida na rede devido à falta de espaço nos *buffers* antes deles alcançarem o destino. No caso da deflexão todos os pacotes chegam ao destino, mas uma boa parte chega

Figura 5.6: Topologia retangular com *threshold* igual a 4 *slots*: (a) fração de sessões VoIP com QoS aceitável e (b) latência fim-a-fim.

tarde demais. Note que esta fração de pacotes tardios está essencialmente sob controle da aplicação. De fato, como pode ser visto na Figura 5.7, se o *threshold* anterior for aumentado para 20 *slots*, roteamento deflexivo oferece QoS melhor que roteamento clássico mesmo sob cargas extremas. A maneira apropriada de interpretar isto é que no roteamento clássico, a origem da maior parte das perdas (devido à sobrecarga nos *buffers*) deixa pouco controle à aplicação para se ajustar às condições de congestionamento, enquanto que deflexão torna esta questão mais flexível transformando congestionamento em atraso ao invés de perdas.

Para confrontar a performance da deflexão com a melhor performance obténível através de QoS *routing* sob mecanismos mais sofisticados que aqueles baseados unicamente no critério de minimização do caminho, nós também consideramos o algoritmo WKS (do inglês *Widest-K-Shortest*) descrito em [67]. Este algoritmo seleciona os K

Figura 5.7: Topologia retangular com *threshold* igual a 20 slots: (a) fração de sessões VoIP com QoS aceitável e (b) atraso fim-a-fim.

Figura 5.8: Fração de sessões VoIP com QoS aceitável em função do período entre atualizações (LSUP) para diferentes coeficientes de frequência (FC) na topologia retangular com 350 sessões simultâneas e roteamento QoS.

caminhos mais curtos de cada origem a cada destino e reclassifica-os cada vez que novas informações sobre o estado dos *links* são recebidos dos vizinhos. A troca do estado dos *links* é o fator de custo dominante do roteamento QoS [28]. Por esta razão o algoritmo **PFR** (do inglês *Probabilistic Frequency Reduction*) [67] é usado para reduzir o custo das atualizações sem prejudicar significativamente a performance do roteamento. Usando PFR as informações dos *links* só são trocadas com o vizinho com uma certa probabilidade **FC** (do inglês *Frequency Coefficient*). A Figura 5.8 ilustrando a fração de sessões VoIP com QoS aceitável em função do período entre atualizações (LSUP) na topologia retangular com 350 sessões simultâneas usando roteamento QoS mostra que a performance do roteamento QoS pode ser melhorada consideravelmente através da troca das informações entre os nós; entretanto,

- o tempo entre as trocas precisa ser significativamente menor que a duração média de uma sessão VoIP (60 s) e
- o roteamento deflexivo é capaz de alcançar melhor performance sem ter que pagar o preço do *overhead* imposto pela troca das informações dos *links*.

5.4.2 Topologia triangular

Para uma exposição mais clara da performance da deflexão versus roteamento clássico usamos a topologia da Figura 5.9. Esta topologia, apesar de estar longe de ser aleatória ilustra que a performance em uma rede apresenta *hotspots* (nós de grau quatro que interconectam dois triângulos), assim como rotas alternativas (os triângulos). Ela serve

Figura 5.9: Topologia triangular.

para mostrar que mesmo redes pequenas, a exemplo de redes sob controle administrativo de um único provedor, podem se beneficiar da deflexão. Sob condições de tráfego leve todos os três algoritmos de roteamento apresentam taxas de perdas similares (não mostrado). Quando a carga é mais alta, o roteamento deflexivo apresenta taxas de perdas menores com *buffers* pequenos nos roteadores, como pode ser visto na Figura 5.10, para uma carga equivalente a 100 sessões VoIP simultâneas. Nesta particular topologia as melhores (mais curtas) rotas para todos os nós passam através dos nós de numera-

Figura 5.10: Resultados para a topologia triangular: (a) fração de canais VoIP com QoS aceitável e (b) atraso fim-a-fim.

Figura 5.11: Fração das sessões VoIP com QoS aceitável em função do LSUP para diferentes coeficientes de frequência (FC) na topologia triangular com 100 sessões simultâneas e roteamento QoS.

ção par, e quando os *links* entre estes nós ficam congestionados eles descartam pacotes no roteamento clássico. Com deflexão, os *links* entre nós com numeração par e ímpar são automaticamente utilizados para distribuir a carga, o que resulta em uma taxa de transmissão agregada mais alta e, portanto, perdas menores. Sob condições de tráfego mais intensas este efeito é ainda mais acentuado.

A Figura 5.11 mostra que a performance do roteamento QoS na rede triangular melhora com a troca dos estados dos *links* uma vez que isto permite explorar os benefícios das rotas alternativas disponíveis.

5.4.3 Topologias irregulares

Finalmente, verificamos se padrões de performance similares aparecem em topologias irregulares. Usamos *nem* [81] para gerar diversas topologias irregulares com tamanhos variando de 35 a 750 nós, com todos os comprimentos dos *links* calculados a partir das posições aleatórias dos nós geradas por *nem*. Os resultados apresentados na Figura 5.12(a) mostram a fração de canais VoIP com QoS aceitável em uma rede de 35 nós com 120 sessões VoIP simultâneas usando um *playout threshold* equivalente a 4 *slots*. O atraso para esta mesma topologia é mostrado na Figura 5.12(b). A QoS observada é muito similar à QoS que foi vista em topologias regulares. Mover os *buffers* do núcleo para a periferia da rede ainda aumenta o número de sessões que podem ser suportadas, mas somente sob o regime da deflexão. O atraso também é melhorado no caso da deflexão. A razão por trás dos resultados observados pode estar ligada ao fato que nas topologias irregulares não existem muitos caminhos alternativos entre a origem e

Figura 5.12: Resultados para uma topologia irregular com 35 nós: (a) fração de sessões VoIP com QoS aceitável e (b) latência fim-a-fim.

o destino. De fato, apesar das topologias irregulares geradas por *nem* apresentarem um grau médio igual a 2,6, poucos nós têm um grau alto e muitos têm grau baixo, sendo que grande parte destes têm grau igual a um. Desta forma fica claro que o roteamento através das rotas mais curtas rapidamente satura os melhores caminhos e raramente um segundo melhor caminho existe, como as curvas $k = 2$ indicam. É a capacidade de usar, quando se faz necessário (mesmo que de forma transiente), um *link* pouco utilizado que leva a um aumento de performance, mesmo quando não existem rotas totalmente disjuntas.

5.5 Conclusões

Foram estudados aspectos de QoS percebidos por sessões VoIP concorrentes em redes operando sob três diferentes esquemas de roteamento:

- roteamento clássico por único caminho através do melhor esforço
- roteamento QoS com dois caminhos mais curtos
- deflexão assíncrona com pequenos *buffers* nos roteadores

Nossos experimentos mostraram que, especialmente sob condições de tráfego intensas, a deflexão apresenta uma performance pelo menos tão boa quanto a performance apresentada pelas duas outras formas de roteamento, e frequentemente ela chega a apresentar características de QoS melhores à aplicação. A melhor distribuição de carga na rede obtida com deflexão resulta em perdas e atrasos menores.

Argumentamos que ao contrário do que se costuma acreditar, o paradigma da deflexão não é necessariamente inferior (do ponto de vista das garantias de serviço) que alternativas mais previsíveis envolvendo rotas determinísticas pré-arranjadas. Apesar da diferença na aparência, nenhum paradigma é de fato determinístico e previsível, e a percepção que a camada de aplicação tem de sua operação não necessariamente impõe determinismo no núcleo da rede.

Deflexão aparenta oferecer mais flexibilidade na camada de rede com relação a congestionamentos e balanceamento de carga, por causa da sua resposta imediata e automática a condições de sobrecarga intermitentes. Esta flexibilidade também se aplica na utilização de *buffers* no núcleo da rede: o espaço dos *buffers* passa a ser efetivamente compartilhado e junto com os *links* subutilizados torna-se um meio global para acomodar tráfego local excessivo. Mesmo se os benefícios diretos da deflexão parecem um tanto enfraquecidos nas redes irregulares mais realísticas (por causa do reduzido número de rotas alternativas), uma vantagem que se mantém é a redução do tamanho dos *buffers* nos roteadores e sua utilização mais econômica como recurso global.

As pesquisas que desenvolvemos com redes deflexivas resultaram em dois trabalhos aceitos em congressos internacionais, o Networking 2005 [97] e o SPECTS 2005 [98]. Os anais do primeiro congresso foram publicados eletronicamente pela Springer-Verlag nos *Lecture Notes in Computer Science* [34].

CAPÍTULO

6

Trabalhos futuros

O estudo dos aspectos de QoS mais relevantes a transmissões em tempo real nas linhas de pesquisa apresentadas nos capítulos 4 e 5 mostrou que diferentes técnicas podem ser empregadas com sucesso para obtenção de QoS na WAN. Em particular, foi mostrado que tanto a solução proposta baseada em gerenciadores de tráfego como o roteamento deflexivo em redes sem conexão poderiam ser aplicados na prática. Além dos bons resultados obtidos nestas duas soluções, há outras vantagens claras que se destacam. Dentre as principais devemos apontar o fato de que ambas as soluções são baseadas em *hardware* comercialmente disponível no mercado, sendo que as novas funcionalidades são todas implementadas em *software*. Isto faz com que o desenvolvimento seja mais rápido e barato do que se *hardware* dedicado tivesse que ser desenvolvido no lugar do *software*. Além disso, também deve ser mencionado que há muito mais flexibilidade nas soluções baseadas em *software*, pois estas permitem atualizações ou mesmo modificações sem maiores dificuldades, o que já não costuma ocorrer com soluções baseadas inteiramente em *hardware*.

É natural observar que as duas soluções propostas neste trabalho poderiam convergir para uma solução única da qual ambas fizessem parte. Esta solução única poderia usufruir das vantagens que cada uma das soluções independentes. Ela poderia ainda ser integrada na implementação do MPLS no *kernel* do Linux, que já está em fase avançada de desenvolvimento [76]. Assim seria possível converter a sinalização das aplicações multimídia para a abertura de um novo canal entre gerenciadores em sinalização para a abertura de um canal MPLS. Este canal MPLS por sua vez poderia fazer uso das características de QoS do ATM, quando redes desta tecnologia fossem usadas, ou usar múltiplas rotas em redes IP pelas quais poderia ser feito roteamento deflexivo. Até o presente momento (Setembro/2005) a implementação do MPLS no Linux ainda não está integrada com a implementação do ATM no *kernel*, mas seria muito interessante que isto fosse feito. Também ainda não é possível usar roteamento deflexivo com roteadores baseados em Linux, mas não seria muito difícil fazer as modificações necessárias no mecanismo de roteamento do *kernel* para tornar isto realidade. Os clas-

sificadores de tráfego na borda da rede MPLS também poderiam tentar reconhecer os fluxos de áudio e vídeo automaticamente, sem o auxílio das aplicações. Provavelmente iptables [19] seria uma ferramenta adequada para atingir este objetivo. Se os fluxos pudessem ser reconhecidos automaticamente, a abertura de canais poderia se dar de forma ainda mais automática e transparente.

Contudo, uma vez que os principais objetivos expostos no início deste projeto já foram alcançados com os trabalhos até aqui desenvolvidos, deixamos todas estas idéias como interessantes propostas para trabalhos futuros.

CAPÍTULO

7

Conclusões gerais

Apesar dos muitos trabalhos sobre QoS já publicados, os problemas desta área ainda não foram totalmente resolvidos. Em parte isto se deve à falta de reconhecimento da real necessidade de prover garantias mínimas de QoS a certas aplicações que sem isto não conseguem operar de modo satisfatório. Ainda hoje a grande maioria das aplicações funciona a contento com o paradigma do "melhor esforço", mas evidências sugerem que em um futuro próximo aplicações de voz e vídeo se tornarão onnipresentes, e devido às suas características algo melhor que o "melhor esforço" será necessário para satisfazê-las. Não existe uma alternativa única para atingir este objetivo, e algumas têm sido consideradas mais viáveis que outras. Como exemplo, redes deflexivas têm tipicamente sido desprezadas como alternativa viável para obter QoS em WANs, mas conforme mostramos neste trabalho, o paradigma da deflexão não é necessariamente inferior (do ponto de vista da QoS) que alternativas mais previsíveis envolvendo rotas determinísticas pré-arranjadas, servindo portanto como solução possível.

O fato de não existir uma solução única para resolver o problema da QoS é uma das razões que levam à falta de consenso quanto à solução a adotar. Como muitos já afirmaram, ATM poderia ser "a solução" pois ela é capaz de garantir que os requisitos de QoS das aplicações sejam atendidos de fim-a-fim e ao mesmo tempo permite às aplicações existentes continuar usando o já tão bem aceito protocolo IP. Isto levou à especulação de que gradualmente o ATM substituiria as outras tecnologias, a começar pelo *backbone* [119, 48, 125, 108].

Por uma razão ou outra a adoção do ATM pelo mercado acabou não sendo tão grande como previsto [44, 71, 127, 114, 132]. Apesar de tudo, algumas idéias defendidas pelo ATM foram adotadas em tecnologias mais novas como MPLS e têm boas chances de sobreviverem no futuro. Como exemplo podemos citar o fato de que os rótulos MPLS usados para identificar os requisitos de QoS dos fluxos, assim como os identificadores de canal (VCI) e caminho (VPI) no ATM, têm significado unicamente local ao equipamento a repassar o fluxo adiante, e são substituídos a cada *hop*.

Talvez uma das maiores dificuldades a ser enfrentada resida na integração das di-

ferentes tecnologias. Isto foi verdade quando ATM foi lançado, por exemplo, devido à grande mudança filosófica na operação das redes, que até então ofereciam um serviço sem conexão baseado em IP e passariam a operar em modo orientado à conexão com a adoção do protocolo ATM. Foi um grande passo que muitos não quiseram dar. Hoje o desafio da integração continua, tanto em *hardware*, com no exemplo da integração do *Firewire* com o *Ethernet* mencionado na seção 1.1, como em *software*, pela busca de uma maior sinergia entre as tecnologias de rede e as aplicações multimídia.

Com este trabalho foi possível estudar as dificuldades envolvidas na reserva de recursos nos mais diferentes níveis, do *hardware* ao *software*. Uma solução para integração de redes *Ethernet* e ATM com algumas características inéditas foi proposta e implementada com sucesso na plataforma Linux. Apesar da escolha pelo ATM no *backbone*, que se deu principalmente pela disponibilidade deste equipamento no Instituto onde o trabalho foi realizado, outra tecnologia poderia ser usada em seu lugar, de modo que a solução proposta pode ser vista de maneira bem mais ampla do que pode parecer inicialmente. O estudo de transmissões de voz sobre redes IP (VoIP) empregando roteamento deflexivo foi inédito por aliar um algoritmo de *playout* simples, porém eficaz, ao roteamento deflexivo para reduzir a latência eficaz de comunicação a níveis aceitos por aplicações VoIP. Com isto mostrou-se que é possível oferecer boa QoS a aplicações em tempo real mesmo com WANs desprovidas de QoS.

Referências Bibliográficas

- [1] IEEE 802.3 – LAN/MAN CSMA/CD access method.
<http://standards.ieee.org/getieee802/802.3.html>.
- [2] *in* [95], chapter 4.
- [3] *in* [95], chapter 7.
- [4] *in* [42], page 93.
- [5] *in* [130], chapter 5.
- [6] *in* [130], pages 279–281.
- [7] *in* [130], page 434.
- [8] *in* [130], page 412.
- [9] *in* [130], pages 278–279.
- [10] LAN/MAN overview and architecture.
<http://standards.ieee.org/getieee802/802.html>.
- [11] Wikipedia’s definition for Ethernet.
<http://en.wikipedia.org/wiki/Ethernet>.
- [12] ATM Forum. ATM Forum addressing: Reference guide, February 1999.
- [13] ATM Forum. the history of ATM, October 2003.
<http://www.atmforum.com/aboutatm/history.html>.
- [14] Conexant Systems, Inc. Conexant Systems home page, February 2004.
<http://www.conexant.com>.
- [15] Hauppauge Computer Works, Inc., February 2004.
<http://www.hauppauge.com>.
- [16] Apple Computer, Inc., 2005.
<http://www.apple.com>.
- [17] Hauppauge Computer Works, Inc. Personal video recorders from Hauppauge, 2005.
http://www.hauppauge.com/Pages/prods_pvrs.html.
- [18] Microsoft Corporation. Xbox game console, 2005.
<http://www.xbox.com>.

- [19] The netfilter/iptables project, September 2005.
<http://www.netfilter.org/>.
- [20] Projeto ToRM, 2005.
http://mri.if.sc.usp.br/paginas/port/index_torm.html.
- [21] Sony Computer Entertainment Inc. Playstation game console, 2005.
<http://www.playstation.com>.
- [22] Sony Corporation of America, 2005.
<http://www.sony.com>.
- [23] 3Com. Pace technology, 2000.
http://www.pulsewan.com/data101/pace_basics.htm.
- [24] D. Adimi and A. Hachicha. A marking method for congestion avoidance in ATM-based multimedia networks. In *Singapore ICCS '94*, volume 2, pages 545–548, November 1994.
- [25] W. Almesberger. High-speed ATM networking on low-end computer systems. In *Conference Proceedings of the 1996 IEEE Fifteenth Annual International Phoenix Conference on Computers and Communications*, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. Laboratoire de Réseaux de Communication (LRC), IEEE Computer Society Press.
- [26] W. Almesberger. Linux ATM API version 0.4, July 1996. EPFL, LRC
<http://linux-atm.sourceforge.net/doc.php#api>.
- [27] W. Almesberger, C. Williams, M. Blank, P. B. Schroeder, and S. Russell. ATM on Linux: The distribution, April 2003.
<http://linux-atm.sourceforge.net/dist.php>.
- [28] G. Apostolopoulos, R. Guerin, S. Kamat, and S. K. Tripathi. Quality of service based routing: a performance perspective. In *Proceedings of the ACM SIGCOMM '98 conference on applications, technologies, architectures, and protocols for computer communication*, pages 17–28. ACM Press, 1998.
- [29] C. Baransel, W. Dobosiewicz, and P. Gburzyński. Routing in multi-hop switching networks: Gbps challenge. *IEEE Network Magazine*, (3):38–61, 1995.
- [30] M. A. Bender, S. Chakrabarti, and S. Muthukrishnan. Flow and stretch metrics for scheduling continuous job streams. In *SODA '98: Proceedings of the ninth annual ACM-SIAM symposium on discrete algorithms*, pages 270–279, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics.
- [31] S. Blake, D. L. Black, M. A. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differentiated services, December 1998. RFC 2474.
- [32] F. Borgonovo and E. Cadorin. Locally-optimal deflection routing in the bidirectional Manhattan network. In *Proceedings of IEEE INFOCOM'90*, pages 458–464, 1990.

- [33] F. Borgonovo and L. Fratta. Deflection networks: architectures for metropolitan and wide area networks. *Computer Networks and ISDN Systems*, (24):171–183, 1992.
- [34] R. Boutaba, K. C. Almeroth, R. Puigjaner, S. X. Shen, and J. P. Black, editors. *NETWORKING 2005: Networking Technologies, Services, and Protocols; Performance of Computer and Communication Networks; Mobile and Wireless Communication Systems, 4th International IFIP-TC6 Networking Conference, Waterloo, Canada, May 2-6, 2005, Proceedings*, volume 3462 of *Lecture Notes in Computer Science*. Springer, 2005.
- [35] B. Braden, L. Zang, S. Berson, S. Herzog, and S. Jamin. Resource reSerVation Protocol (RSVP), September 1997. RFC 2205.
- [36] J. C. Brustoloni and P. Steenkiste. Evaluation of data passing and scheduling avoidance. In *Proc. Intl. Workshop on Network and Operating System Support for Digital Audio and Video*, May 1997.
- [37] A. Campbell, G. Coulson, and D. Hutchison. A quality of service architecture. *SIGCOMM Comput. Commun. Rev.*, 24(2):6–27, 1994.
- [38] C.-J. Chang, C.-H. Yu, C.-S. Chang, and L.-F. Lin. Intelligent leaky bucket algorithms for sustainable-cell-rate usage parameter control in ATM networks. *IEEE Transactions on Multimedia*, 6(5):749–759, October 2004.
- [39] D. Chaplin. Glade - GTK+ User Interface Builder, November 2003.
<http://glade.gnome.org/index.html>.
- [40] S. Cherry. The largest players rule the media playground. *IEEE Spectrum*, 39(7):32–33, July 2002.
- [41] A. Choudhury and N. Maxemchuk. Effect of a finite reassembly buffer on the performance of deflection routing. *Conference Record of the International Conference on Communications (ICC)*, 3:1637–1646, 1991.
- [42] D. E. Comer. *Internetworking With TCP/IP - Principles, Protocols, and Architecture*, volume 1. Prentice-Hall, 3rd edition, 1995.
- [43] D. E. Comer and D. L. Stevens. *Internetworking with TCP/IP*, volume 2, chapter 8. Prentice-Hall Inc., 2nd edition, 1994.
- [44] S. Crosby, S. Rooney, R. Isaacs, and H. Bos. A perspective on how ATM lost control. *SIGCOMM Comput. Commun. Rev.*, 32(5):25–28, 2002.
- [45] M. Cumming and D. Elstner. gtkmm, November 2003.
<http://gtkmm.sourceforge.net>.
- [46] P. de Cuetos and K. W. Ross. Optimal streaming of layered video: joint scheduling and error concealment. In *MULTIMEDIA '03: Proceedings of the eleventh ACM international conference on Multimedia*, pages 55–64, New York, NY, USA, 2003. ACM Press.

- [47] P. DeLeon and C. J. Sreenan. An adaptive predictor for media playout buffering. In *IEEE International Conference on Acoustics, Speech, and Signal Processing. ICASSP '99*, volume 6, pages 3097–3100, March 1999.
- [48] R. C. Dixon. Cells-in-frames: a system overview. *IEEE Network*, 10(4):9–17, July/August 1996.
- [49] W. Dobosiewicz and P. Gburzyński. A bounded-hop-count deflection scheme for Manhattan-street networks. In *Proceedings of IEEE INFOCOM '96*, pages 172–179, San Francisco, March 1996.
- [50] P. Gburzyński and J. Maitan. Deflection routing in regular MNA topologies. *Journal of High Speed Networks*, 2(2):99–131, 1993.
- [51] GCC Team. GCC front ends, October 2003.
<http://gcc.gnu.org/frontends.html>.
- [52] GNU. GNU Autoconf, April 2001.
<http://www.gnu.org/software/autoconf>.
- [53] GNU. GNU Make, April 2002.
<http://www.gnu.org/software/make/make.html>.
- [54] GNU. GNU Automake, November 2003.
<http://www.gnu.org/software/automake>.
- [55] GNU. GNU Common C++, January 2003.
<http://www.gnu.org/software/commonc++/CommonC++.html>.
- [56] B. Goode. Voice over internet protocol (VoIP). *Proceedings of the IEEE*, 90(9):1495–1517, September 2002.
- [57] T. Henderson and S. Bhatti. Networked games: a QoS-sensitive application for QoS-insensitive users? In *RIPQoS '03: Proceedings of the ACM SIGCOMM workshop on Revisiting IP QoS*, pages 141–147, New York, NY, USA, 2003. ACM Press.
- [58] M. Hluchyj and N. Yin. A second-order leaky bucket algorithm to guarantee QoS in ATM networks. In *Global Telecommunications Conference, 1996. GLOBECOM '96. 'Communications: The Key to Global Prosperity*, volume 2, pages 1090–1096. IEEE, 1996.
- [59] J. Ho, H. Uzunalioglu, and I. Akyildiz. Cooperating leaky bucket for average rate enforcement of VBR video traffic in ATM networks. In *Fourteenth Annual Joint Conference of the IEEE Computer and Communications Societies Bringing Information to People. INFOCOM '95*, volume 3, pages 1248–1255, April 1995.
- [60] H.-Y. Huang, T. Robertazzi, and A. Lazar. A comparison of information based deflection strategies. *Computer Networks and ISDN Systems*, 27:1399–1407, 1995.

- [61] IEEE. IEEE 802.1Q-2003 – IEEE standards for local and metropolitan area networks–virtual bridged local area networks, May 2003.
<http://standards.ieee.org/getieee802/802.1.html>.
- [62] IEEE. IEEE 802.1D-2004 – IEEE standard for local and metropolitan area networks–media access control (MAC) bridges, 2004.
<http://standards.ieee.org/getieee802/802.1.html>.
- [63] ITU. ITU-T Recommendation E.164/I.331. Numbering plan for the ISDN era, May 1997.
- [64] ITU-T. B-ISDN ATM adaptation layer specification, March 1993. Recommendation I.363.
- [65] ITU-T. B-ISDN ATM adaptation layer specification: Type 5 AAL, 1996. Recommendation I.363.5.
- [66] T. Jehaes, D. D. Vleeschauwer, T. Coppens, B. V. Doorselaer, E. Deckers, W. Naudts, K. Spruyt, and R. Smets. Access network delay in networked games. In *NETGAMES '03: Proceedings of the 2nd workshop on Network and system support for games*, pages 63–71, New York, NY, USA, 2003. ACM Press.
- [67] Y. Jia, I. Nikolaidis, and P. Gburzyński. Scalable QoS routing using alternative paths. *International Journal of Communication Systems*, 17(1):1–26, 2004.
- [68] R. Jones. Netperf homepage, November 2003.
<http://www.netperf.org>.
- [69] N. M. Josuttis. *The C++ Standard Library: a Tutorial and Reference*. Addison-Wesley, 1999.
- [70] S.-W. Jung, E.-S. Kim, and D.-H. Lee. Design and implementation of an enhanced personal video recorder for dtv. *IEEE Transactions on Consumer Electronics*, 47(4):915–920, November 2001.
- [71] C. Kalmanek. A retrospective view of ATM. *SIGCOMM Comput. Commun. Rev.*, 32(5):13–19, 2002.
- [72] M. G. Katevenis, I. Mavroidis, G. Sapountzis, E. Kalyvianaki, I. Mavroidis, and G. Glykopoulos. Wormhole IP over (connectionless) ATM. *IEEE/ACM Transactions on Networking*, 9(5):650–661, October 2001.
- [73] S. Kim and M. Lee. Server based QoS routing with implicit network state updates. In *Ninth IEEE International Conference on Networks*, pages 511–516, October 2001.
- [74] C. Ko, O. Yang, L. Orozco-Barbosa, and H. Mouftah. Using token allocations in a leaky bucket scheme. In *Conference Record, 1994 IEEE Military Communications Conference, 1994. MILCOM '94.*, volume 1, pages 82–86, October 1994.

- [75] A. Krishna and B. Hajek. Performance of shuffle-like switching networks with deflection. In *Proceedings of IEEE INFOCOM'90*, pages 473–480, San Francisco, CA, June 1990.
- [76] J. R. Leu and R. Casellas. Project: MPLS for Linux: Summary.
<http://sourceforge.net/projects/mpls-linux/>.
- [77] Z. Liang and M. Ali. A modified leaky bucket policing mechanism. In *IEEE Canadian Conference on Electrical and Computer Engineering, 1998.*, volume 1, pages 345–348, May 1998.
- [78] F.-S. Lin. Optimal real-time admission control algorithms for the video-on-demand (VOD) service. *IEEE Transactions on Broadcasting*, 44(4):402–408, December 1998.
- [79] S. Lin and N. McKeown. A simulation study of IP switching. In *SIGCOMM '97: Proceedings of the ACM SIGCOMM '97 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 15–24, New York, NY, USA, 1997. ACM Press.
- [80] D. Magoni and J.-J. Pansiot. Evaluation of internet topology generators by power law and distance indicators. In *10th IEEE International Conference on Networks. ICON 2002*, pages 401–406, August 2002.
- [81] D. Magoni and J.-J. Pansiot. Internet topology modeler based on map sampling. In *Proceedings of the Seventh International Symposium on Computers and Communications (ISCC'02)*, pages 1021–1027. IEEE Computer Society, July 2002.
- [82] M. Maki, S. Hamada, M. Tokuda, Y. Shimoshio, and N. Kuwabara. Home information wiring system using UTP cable for IEEE1394 and Ethernet systems. *IEEE Transactions on Consumer Electronics*, 47(4):921–927, November 2001.
- [83] Y. Mansour, B. Patt-Shamir, and O. Lapid. Optimal smoothing schedules for real-time streams (extended abstract). In *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 21–29, New York, NY, USA, 2000. ACM Press.
- [84] K. Mase and Y. Toyama. End-to-end measurement based admission control for VoIP networks. In *IEEE International Conference on Communications (ICC 2002)*, volume 2, pages 1194–1198, May 2002.
- [85] N. Maxemchuk. The Manhattan Street Network. In *Proceedings of GLOBE-COM'85*, pages 255–261, New Orleans, LA, December 1986.
- [86] N. Maxemchuk. Routing in the Manhattan Street Network. *IEEE Transactions on Communications*, 35(5):503–512, May 1987.
- [87] N. Maxemchuk. Twelve random access strategies for fiber optic networks. *IEEE Transactions on Communications*, 36(8):942–950, August 1988.

- [88] N. Maxemchuk. Comparison of deflection and store-and-forward techniques in Manhattan-street network and shuffle-exchange networks. In *Proceedings of IEEE INFOCOM'89*, pages 800–809, 1989.
- [89] N. Maxemchuk. Problems arising from deflection routing. In Pugolle, editor, *High Capacity Local and Metropolitan Networks*, pages 209–233. Springer Verlag, 1991.
- [90] N. Maxemchuk and R. Krishnan. A comparison of linear and mesh topologies—DQDB and the Manhattan street network. *IEEE Journal on Selected Areas in Communications*, 11(8):1278–1301, October 1993.
- [91] M. Mbaye, B. Tohio, Y. Savaria, and S. Pierre. Performance of a Firewire-Ethernet protocols conversion on an ARM7 embedded processor. In *Canadian Conference on Electrical and Computer Engineering. IEEE CCECE 2003*, volume 2, pages 1267–1270, May 2003.
- [92] C. Y. Metz. *IP Switching - Protocols and Architectures*. McGraw-Hill, 1999.
- [93] A. Mink. *Operating Principles of MultiKron II Performance Instrumentation for MIMD Computers*. National Institute of Standards and Technology, Gaithersburg, MD 20899, December 1994.
- [94] A. Mink. *Operating Principles of the PCI Bus MultiKron Interface Board*. National Institute of Standards and Technology, Gaithersburg, MD 20899, March 1997.
- [95] M. Mitchell, J. Oldham, and A. Samuel. *Advanced Linux Programming*. New Riders Publishing, 1st edition, June 2001.
- [96] A. Muezerie. Caracterização de desempenho de uma rede ATM. Master's thesis, Instituto de Física de São Carlos. Universidade de São Paulo, Setembro 2001.
- [97] A. Muezerie, I. Nikolaidis, and P. Gburzyński. Attaining VoIP-grade QoS via deflection: A buffer space tradeoff study. In Boutaba et al. [34], pages 1457–1460.
- [98] A. Muezerie, I. Nikolaidis, and P. Gburzyński. Buffer space tradeoffs for VoIP QoS in deflection networks. In *2005 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS'05)*, pages 381–390, Philadelphia, PA, July 2005.
- [99] A. Muezerie and J. F. W. Slaets. Uma proposta de gerenciamento dinâmico de tráfego multimídia em redes híbridas ATM e Ethernet. In *IX Workshop de Gerência e Operação de Redes e Serviços (WGRS 2004)*, Gramado, Brasil, Maio 2004.
- [100] L. Neal, R. Perez, and D. Miller. eLearning and fun. In *CHI '05: CHI '05 extended abstracts on Human factors in computing systems*, pages 2043–2044, New York, NY, USA, 2005. ACM Press.

- [101] P. Newman, G. Minshall, and T. L. Lyon. IP switching - ATM under IP. *IEEE/ACM Trans. Netw.*, 6(2):117–129, 1998.
- [102] K. Nichols, S. Blake, F. Baker, and D. Black. Definition of the differentiated services field (DS field) in the IPv4 and IPv6 headers, December 1998. RFC 2474.
- [103] Nintendo. Gamecube game console, 2005.
<http://www.nintendo.com>.
- [104] The Multikron project, 1999.
<http://www.multikron.nist.gov/software.html>.
- [105] W. Olesinski and P. Gburzyński. Real-time traffic in deflection networks. In *Proceedings of WMC'98: Communication Networks and Distributed Systems, Modeling and Simulation*, pages 23–28, San Diego, California, January 1998.
- [106] W. Olesinski and P. Gburzyński. Service guarantees in deflection networks. In *Proceedings of MASCOTS'01*, pages 267–274, Cincinnati, Ohio, August 2001.
- [107] A. Oliphant and B. Scudamore. TV gets personal [personal video recorders]. *IEE Review*, 47(5):9–13, September 2001.
- [108] E. Pelikan, A. Ganser, E. Kotter, U. Schrader, and U. Timmermann. Experience with PACS in an ATM/Ethernet switched network environment. *IEEE Transactions on Information Technology in Biomedicine*, 2(1):26–29, March 1998.
- [109] Q. Peng and J. Jing. System-on-chip design for TV-centric home networks. In *First IEEE Consumer Communications and Networking Conference (CCNC 2004)*, pages 501–506, January 2004.
- [110] C. Petig. glade-- (gtk add-on), November 2003.
<http://home.wtal.de/petig/Gtk/>.
- [111] T. Pulkkinen, K. Nelson, and M. Cumming. libsigc++, November 2003.
<http://libsigc.sourceforge.net>.
- [112] Y. Rekhter, B. Davie, D. Katz, E. Rosen, and G. Swallow. Cisco systems' tag switching architecture overview, February 1997. RFC 2105.
- [113] Y. Rekhter, B. Davie, E. Rosen, G. Swallow, D. Farinacci, and D. Katz. Tag switching architecture overview. In *Proceedings of the IEEE*, volume 85, pages 1973–1983, December 1997.
- [114] M. J. Riezenman and W. Sweet. Communications [technology analysis and forecast]. *IEEE Spectrum*, 36:29–34, January 1999.
- [115] B. O. T. Rosa. Análise de sistemas de comunicação para computação paralela em clusters. Master's thesis, Instituto de Física de São Carlos. Universidade de São Paulo, 2002.
- [116] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol label switching architecture, January 2001. RFC 3031.

- [117] A. Rubini and J. Corbet. *Linux Device Drivers*, chapter 14. O'Reilly, 2nd edition, June 2001. Available on-line <http://www.xml.com/ldd/chapter/book>.
- [118] A. Saad and D. Smith. An IEEE 1394 - Firewire - based embedded video system for surveillance applications. In *Proceedings of the IEEE Conference on Advanced Video and Signal Based Surveillance*, pages 213–218, July 2003.
- [119] M. Sadiku, S. Subramanian, and A. Bhadra. Gigabit Ethernet and ATM in high speed arena. In *IEEE Southeastcon'98*, pages 295–297, Orlando, Florida, April 1998.
- [120] H. Schildt. *The Annotated ANSI C Standard*. McGraw-Hill, 1990. ANSI/ISO 9899-1990.
- [121] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A transport protocol for real-time applications, January 1996. RFC 1889.
- [122] C. Severance. Linking computers and consumer electronics. *Computer*, 30(2):119–121, February 1997.
- [123] A. Shallwani and P. Kabal. An adaptive playout algorithm with delay spike detection for real-time VoIP. In *Canadian Conference on Electrical and Computer Engineering. IEEE CCECE 2003*, volume 2, pages 997–1000, May 2003.
- [124] S. Shenker, C. Partridge, and R. Guerin. Specification of guaranteed quality of service, September 1997. RFC 2212.
- [125] M. Shore, J. Veronneau, T. Parker, and D. Cogger. Cells in frames: ATM over legacy networks. In *1st IEEE International Conference on ATM (ICATM-98)*, pages 418–422, June 1998.
- [126] G. Sidler, B. Stiller, C. Nadig, and B. Plattner. ATM over shared media networks. In *Proceedings of the 1998 International Zurich Seminar on Broadband Communications*, pages 225 – 236, February 1998.
- [127] W. D. Sincoskie. Broadband packet switching: a personal perspective. *IEEE Communications Magazine*, 40(7):54–66, July 2002.
- [128] G. Stone and V. Williams. Quality of service mapping and related issues for bridging 1394 and Ethernet. In *IEEE International Conference on Consumer Electronics (ICCE 2003)*, pages 92–93, June 2003.
- [129] H. Taale, P. van Bekkum, and M. van Rij. Evaluation of traffic management by the traffic police. In *Proceedings of the 12th IEE International Conference on Road Transport Information and Control (RTIC 2004)*, pages 106–111, April 2004.
- [130] A. S. Tanenbaum. *Computer Networks*. Prentice Hall PTR, 4th edition, 2003.
- [131] The KDevelop Team. KDevelop, November 2003. <http://www.kdevelop.org>.

- [132] K. Tolly. The great networking correction: Frames reaffirmed. *IEEE Internet Computing*, 1(5):16–23, October 1997.
- [133] K. Toyoshima, H. Tsuboi, H. Fujiwara, Y. Watanabe, H. Miyao, and H. K. High-speed IP packet/Ethernet frame forwarding engine (CAFE) for ATM-based network router/bridge. In *IEEE International Conference on Communications (ICC 98)*, volume 2, pages 1213–1218, June 1998.
- [134] D. van Heesch. Doxygen home page, December 2003.
<http://www.stack.nl/~dimitri/doxygen>.
- [135] A. Vogel, B. Kerhervé, G. v. Bochmann, and J. Gecsei. Distributed multimedia applications and quality of service: a survey. In *CASCON '94: Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*, page 71. IBM Press, 1994.
- [136] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th Annual Symposium on Operating System Principles*, pages 40–53, Copper Mountain Resort, Colorado, December 1995.
- [137] X. Xiao and L. M. Ni. Internet QoS: A big picture. *IEEE Network*, 13(2):8–18, March 1999.
- [138] T. Yuk, K. Tse, M. Leung, and B. Kamali. Simulation of admission control in ATM network. volume 2, pages 714–718, June 1992.
- [139] W. Zhao and S. K. Tripathi. Bandwidth-efficient continuous media streaming through optimal multiplexing. In *SIGMETRICS '99: Proceedings of the 1999 ACM SIGMETRICS international conference on measurement and modeling of computer systems*, pages 13–22, New York, NY, USA, 1999. ACM Press.
- [140] École Centrale Paris. VLC - Video LAN, October 2003.
<http://www.videolan.org>.
- [141] École Centrale Paris. VLC - Video LAN, December 2003.
<http://www.videolan.org/vlc/features.html>.

APÊNDICE

A

Hardware utilizado

Neste trabalho foram utilizados diversos equipamentos, alguns bastante conhecidos, outros nem tanto. Neste capítulo são brevemente descritos os equipamentos usados, com alguma ênfase nos menos usuais.

A.1 A Interface MultiKron II

A.1.1 Introdução

Produzida pelo Instituto Nacional de Padrões e Tecnologia americano ([NIST](#)), a MultiKron II consiste de uma placa de computador que permite medidas de tempo em computadores multiprocessados ou de um único processador. Em versões para barramentos [VME](#), Sbus e [PCI](#), há suporte para sistemas Solaris 2.x, SunOS 4.1.x, [DEC](#) - ALPHA, Windows NT, [BSDI](#), FreeBSD e Linux. Apesar do seu principal *chip*, o MultiKron II ser um dispositivo de 64 bits, a interface se configura de maneira automática e transparente para funcionar em modo de 32 ou 64 bits através do protocolo PCI. Um exemplar para barramento PCI é mostrado na Figura [A.1](#).

A meta do NIST ao produzir esta interface era prover a preço acessível suporte de *hardware* para medições de performance com perturbação tolerável tanto para a aplicação sendo executada quanto para a arquitetura na qual ela é executada¹. A interface MultiKron II alcança estes objetivos sendo mapeada na memória do computador. Desta maneira todas as interações são leituras e escritas em endereços específicos de memória [[93](#)].

O *software* necessário à operação da interface MultiKron pode ser encontrado no *website* do NIST [[104](#), [94](#)], consistindo basicamente do *driver* e do *software* de captura de amostras armazenadas na memória da placa MultiKron (`mk_rdsample.c`) e outro

¹Perturbação tolerável refere-se à não introdução de distorções significativas no ato da medida.

Figura A.1: Interface MultiKron II para barramento PCI.

de geração de uma tabela das amostras capturadas (`mk_expand.c`). O *script* de instalação do *driver* cria o arquivo especial `/dev/mk0`, usado para acessar o dispositivo. O *software* `mk_rdsample.c` lê o conteúdo da memória da interface MultiKron e gera um arquivo binário a partir destes dados. Já `mk_expand.c` utiliza este arquivo binário como entrada para gerar uma tabela legível que lista os eventos registrados pela interface.

A.1.2 Registro de Eventos

O registro preciso dos instantes em que eventos ocorreram durante a execução de um processo no computador pode ser usado para calcular com precisão o tempo transcorrido e a latência, entre outras coisas. Aos dados necessários para especificar um registro de evento chamamos de *trace sample*.

Monitoramento passivo de endereços de instruções buscadas externamente para determinar a ocorrência de um evento é insuficiente, já que caches internos mascaram as instruções executadas. Além disso, endereços externos são endereços físicos, enquanto que os compiladores produzem código em termos de endereços virtuais. Devido a estes problemas, uma visão interna da atividade do programa é necessária para determinar quando um evento ocorre de maneira que dados de medida para um *trace sample* possam ser coletados. Para perturbação mínima do programa sendo executado, o NIST adotou uma solução híbrida para registro de eventos na qual pouco código é adicionado ao *software* para sinalizar ao *hardware* quando este deve registrar um evento. Ao construir o *hardware* de captura como um dispositivo mapeado na memória, este

Figura A.2: Relógio central utilizado pelas placas MultiKron II com botão de reset simultâneo.

código pode ser tão pequeno quanto uma única operação de escrita na memória por evento. A execução deste código informa ao *hardware* de medição a ocorrência de um evento e ainda informa os dados especificados pelo usuário, que podem ser armazenados como parte de um *trace sample*, fornecendo meios de identificar um evento, ou mais precisamente, onde na execução do programa o evento ocorreu. A identificação de um evento pode ser especificada por um número que tem significado apenas para a pessoa que realiza a medida, como “27”, que poderia indicar a posição no código onde uma mensagem proveniente de outro aplicativo acaba de ser recebida.

Tempo instantâneo é um dado importante a coletar na ocorrência de um evento. Para isso é necessário um contador de tempo (*Timestamp Counter*) que tenha uma resolução de tempo compatível com a velocidade de execução do computador e precisão suficiente para que ele não recomece a contagem em zero durante o experimento a ser realizado. A MultiKron II é dotada de um contador de 56 bits e todos eles são incluídos em um *trace sample*.

Para permitir correlação das amostras tomadas por várias interfaces MultiKron os contadores devem ser todos sincronizados. O método de sincronização utilizado emprega um relógio e um sinal de reset (que inicializa o contador de tempo em zero) comum a todas as interfaces. Este relógio central é mostrado na Figura A.2.

A.2 Interface ATM ForeRunner LE 155

Fabricada pela Marconi (antiga Fore Systems), a interface **PCI** ForeRunner LE obedece ao padrão ATM sendo capaz de operar a uma taxa sustentada *full-duplex* de 155 Mbps *wire-speed*. Sua implementação é baseada no *chip* IDT77211 NICStAR TM, sendo que este *chip* é responsável pela camada de adaptação **AAL**, pela camada de segmentação e remontagem (**SAR**) e pela camada ATM, sem a necessidade de um processador acoplado. São suportadas pelo *hardware* as classes de serviço **CBR**, **VBR** e **UBR**, assim como células de operação e manutenção (células **OAM**).

Com o objetivo principal de reduzir custos, sua arquitetura foi desenvolvida de maneira a manter os **PDU**s da subcamada de convergência (**CS**) na memória do com-

putador ao invés de armazená-los em uma memória na própria interface. Os dados são acessados pelo barramento PCI através de DMA para reduzir a utilização de CPU no computador. O NICStAR controla todas as operações das camadas AAL, SAR e ATM e também as transferências de dados com a memória do computador como um PCI *bus master*. Ao *driver* da placa fica a responsabilidade de alocar e gerenciar espaços de memória destinados aos *buffers* a serem utilizados pela placa. Estruturas com ponteiros para estes *buffers* são fornecidas pelo *driver* à placa durante a utilização da mesma, na medida em que ela faz tais solicitações através de interrupções.

O fato dos PDUs serem armazenados diretamente na memória do computador, além da redução de custo, traz também uma redução na latência de comunicação pois é evitada uma cópia dos dados de uma memória local da placa para a memória do computador. A técnica de armazenar os PDUs em uma memória próxima à camada SAR é mais utilizada em sistemas que têm um barramento com menor largura de banda sustentada que a conexão ATM. Assim, os CS-PDUs são transferidos a uma taxa pequena através do barramento para então serem transmitidos em uma rajada através da rede. Quando os PDUs são armazenados na memória local da placa, há uma limitação física (sem usar um algoritmo complexo) no número de PDUs que podem ser manipulados simultaneamente [25].

O NICStAR possui uma FIFO para receber células ATM com capacidade para 315 células. Ao receber uma célula ATM, o NICStAR realiza um processo de *lookup table* utilizando o identificador de conexão presente no cabeçalho da célula para determinar onde na memória do computador colocar o *payload* da célula. Desta maneira, os *payloads* das células ATM são colocados diretamente na memória do computador para formar os CS-PDUs.

Na transmissão, os CS-PDUs são enfileirados na memória do computador pelo *driver* da placa ATM. O próprio NICStAR, ao realizar a segmentação dos CS-PDUs em *payloads* de células ATM, adiciona os respectivos cabeçalhos para criar as células de 53 *bytes*. O *byte* do campo *Header Error Check* (HEC) é inserido pelo NICStAR apenas como um *padding*. As células são então transmitidas ao dispositivo de camada física (PHY), através de um barramento chamado UTOPIA, de acordo com uma tabela de escalonamento, como ilustrado na Figura A.3. O PHY calcula e substitui o *byte* do HEC para formar uma célula ATM de 53 *bytes* completa. É ele também que na recepção faz a verificação do HEC, descartando a célula em caso de erro.

A.3 Chaves ATM

Foram utilizadas duas chaves ATM neste trabalho. Uma de modelo ForeRunner LE 155 e outra ASX-200BX, ambas produzidas pela empresa Marconi (antiga Fore Sys-

Figura A.3: Interface de rede utilizando o *chip* NICStAR.

tems). Suas principais características físicas são apresentadas a seguir.

A.3.1 Chave ATM ForeRunner LE 155

Possui 14 portas ATM de 155 Mbps para cabo UTP e 2 portas ATM para fibra multimodo de 155 Mbps, além de uma porta serial utilizada para configuração inicial da chave e atribuição de número IP. Configurações adicionais podem ser posteriormente feitas pela rede ATM.

A.3.2 Chave ATM ForeRunner ASX-200BX

Oferece uma solução mais flexível que a do modelo LE 155, pois esta chave é composta por um chassi com um *backplane* de 2,5 Gbps de capacidade onde podem ser inseridos diferentes módulos, à escolha do proprietário. Existem vários módulos que oferecem diferentes tecnologias, velocidades e quantidades. Alguns destes módulos (existem muitos outros) possuem as seguintes características:

- 1 porta ATM de 622 Mbps para fibra monomodo
- 1 porta ATM de 622 Mbps para fibra multimodo
- 4 portas ATM de 155 Mbps para fibra multimodo
- 4 portas ATM de 155 Mbps para cabo UTP
- 4 portas *Ethernet* de 10/100 Mbps para cabo UTP
- 4 portas *Frame Relay* para cabo UTP

Nesta chave utilizamos um módulo com quatro portas ATM de 155 Mbps para fibra e outro módulo com quatro portas ATM de 155 Mbps para cabo UTP. Uma das portas de fibra foi usada para conectar esta chave à LE 155. As portas UTP foram usadas para conectar computadores.

A.4 Chave Fast Ethernet

A chave *Fast Ethernet* utilizada foi a de modelo DES-1016D, produzida pela D-Link. Ela possui um *backplane* de 3,2 Gbps, 16 portas *Ethernet* 10/100 Mbps *full-duplex* com auto-negociação e 4 Mbits de RAM para armazenamento temporário de dados. Emprega chaveamento *Store-and-Forward*.

A.5 Placa de aquisição WinCast TV

Produzida pela Hauppauge [15] e baseada no *chip* BT848 da antiga BrookTree (comprada pela Conexant Systems, Inc [14]), a WinCast TV é uma placa PCI que possui um sintonizador de TV e um digitalizador de vídeo. Ela é capaz de exibir vídeo ao vivo em uma janela redimensionável na tela do PC. O vídeo é enviado diretamente pelo barramento PCI para a memória da placa de vídeo instalada no computador. A qualidade do vídeo exibido é muito boa pois apesar do processo de digitalização, não é feita uma compactação propriamente dita do vídeo². A interface WinCast TV é mostrada na Figura A.4.

Suas principais características são:

- TV em uma janela "ao vivo" em monitores VGA em resolução de 640x480 com até 16 milhões de cores;
- Sintonizador de TV embutido com varredura automática de canais que suporta os sistema PAL M, NTSC e SECAM;
- Possibilidade de redimensionar a janela em qualquer tamanho, até tela cheia;
- Entradas RCA auxiliares de vídeo e áudio em estéreo;
- Saída de áudio;

²Apesar do modelo de placa de captura usado não permitir a compactação do vídeo, outros modelos, a exemplo da placa WinTV PVR 350 da própria Hauppauge [17], são capazes de fazer compactação do vídeo em *hardware* em tempo real usando formato MPEG 1 ou 2, gerando um fluxo de dados cuja intensidade pode ser ajustada por *software* para um valor entre 1 e 12 Mbps, dependendo da qualidade solicitada.

Figura A.4: Interface WinCast TV da Hauppauge baseada no *chip* BT848.

- Possibilidade de captura de imagens com qualidade profissional com 24 bits de cor e digitalização 4:2:2;
- Suportado nos sistemas Linux, FreeBSD e Windows 9x/ME/NT/2000;

Ferramentas utilizadas

Durante a execução do projeto foram usadas diversas ferramentas, algumas das quais são dependentes de outras. Neste capítulo é feita uma breve descrição das principais delas. Não é intenção aqui explicar detalhadamente o funcionamento de cada uma, apenas passar uma idéia do que fazem e como são usadas.

B.1 GCC

GCC é a sigla para *GNU Compiler Collection*¹. O GCC, através de *front ends*, é capaz de compilar programas escritos em diversas linguagens, como C, C++, Objective-C, FORTRAN, Java e Ada. O GCC traz também consigo bibliotecas para estas linguagens (*libstdc++*, *libgcj*, etc.). Existem ainda *front ends* para outras linguagens que ainda não foram integrados na distribuição principal do GCC mas que podem vir a ser integrados no futuro [51].

Ao compilar programas escritos na linguagem C++, é usual chamar o compilador de G++. Como há apenas um compilador, é também correto chamá-lo de GCC, independentemente da linguagem sendo utilizada. O termo "G++" apenas enfatiza que a linguagem sendo utilizada é C++. O mesmo é feito para outras linguagens como FORTRAN (*g77*), Java (*gcj*), Ada (*gnat*) e Pascal (*gpc*).

O G++ é de fato um compilador, e não meramente um pré-processador. Ele gera um código objeto diretamente do programa fonte C++, sem fazer uso de uma versão intermediária do programa na linguagem C.

B.2 STL

A biblioteca padrão da linguagem C++, comumente chamada de STL, do inglês *Standard Template Library*, estende a linguagem C++ fornecendo alguns componentes de

¹Antigamente a mesma sigla era conhecida por *GNU C Compiler*.

uso geral através de um conjunto de classes e interfaces comuns. Entre as diversas facilidades oferecidas pela biblioteca estão:

- Tipos *string*;
- Diferentes estruturas de dados (como vetores dinâmicos, listas ligadas, árvores binárias, etc.);
- Diferentes algoritmos (como diferentes algoritmos de ordenação);
- Classes numéricas;
- Classes para realizar E/S.

Estes componentes são de uso comum na maioria dos programas, e componentes mais complexos podem ser construídos com base neles quando necessário. Maiores detalhes sobre STL podem ser encontrados em [69].

B.3 Threads

Ao desenvolver os primeiros programas para a execução deste projeto observou-se que o uso de *threads* facilitava significativamente a implementação de programas que tinham que fazer transmissão e recepção simultaneamente. A maneira usual para lidar com este tipo de situação era criar dois programas diferentes, um servidor para recepção de dados e outro cliente, capaz de enviar dados ao servidor. Para que a comunicação seja possível ambos devem ser executados. Com *threads*, pode-se fazer um único programa que engloba as partes comuns aos programas cliente e servidor, e os trechos de código que não são comuns aos dois, que são de fato responsáveis pelo envio e recepção de dados, são colocados dentro do corpo das *threads*. Ao executar este programa, ele pode iniciar apenas uma das *threads* para se comportar como um servidor ou outra para se comportar como cliente, ou ambas para possibilitar comunicação bidirecional com outra máquina. Além desta facilidade, as *threads* também permitem acesso a regiões de memória compartilhada entre elas, o que facilita o acesso do programa principal a informações que as *threads* podem disponibilizar em tempo real, através de simples acessos à memória.

Os primeiros programas foram escritos em linguagem C. Neste estágio usou-se a API de *threads* padrão POSIX do kernel do Linux (também conhecidas por *pthread*), devido à sua alta eficiência e *overhead* mínimo no chaveamento dos processos [2].

À medida que os programas foram ficando maiores e mais complexos tornou-se evidente que seriam grandes os benefícios decorrentes do uso de uma linguagem orientada a objetos. Com a utilização da linguagem C++ o uso direto da API POSIX

```
#include <pthread.h>
int pthread_create(pthread_t * thread, pthread_attr_t * attr,
                  void * (*start_routine)(void *), void * arg);
```

Figura B.1: Rotina da API POSIX responsável pela criação de uma nova *thread* cujo comportamento é o da rotina `start_routine` chamada com `arg` como argumento.

deixou de ser conveniente, pois como mostra a Figura B.1, para se criar uma nova *thread* a partir desta API é necessário passar para a rotina `pthread_create` um ponteiro para a rotina a ser usada como corpo da *thread*. Assim, um objeto não tem como iniciar uma nova *thread* cujo corpo é um método deste objeto, a menos que o método seja declarado `static`. Porém, métodos estáticos não têm como acessar diretamente os membros não estáticos do objeto, não havendo assim vantagem no uso da linguagem C++ com uso direto das *pthreads*.

A solução mais fácil e elegante para este problema foi passar a usar um *framework* que oferecesse suporte a *threads*. O *framework* escolhido foi o GNU Common C++ (descrito na seção B.4), que apesar de bastante portátil², acaba por utilizar no ambiente Linux a implementação de *threads* POSIX do *kernel*, não havendo assim prejuízo na sua utilização.

Neste *framework* existe uma classe *thread* que pode ser usada para criar classes derivadas. A rotina que dá "corpo" à *thread* é definida como `virtual`, cabendo ao usuário definir sua implementação. Para iniciar uma nova *thread*, basta criar uma nova instância do objeto `Thread`. Para que a nova *thread* recém criada seja iniciada deve-se chamar seu método `start`.

No desenvolvimento do projeto todos os programas em C++ que fazem uso de *threads* passaram a fazer uso deste *framework*.

B.4 GNU Common C++

O GNU Common C++[55] é um *framework* que oferece suporte para *threads*, *sockets*, acesso a arquivos, *daemons*, E/S serial, *parsing XML*, etc. com alta portabilidade. É licenciado sob o termos da GNU GPL.

O GNU Common C++ oferece a implementação de classes para *threading* e *sockets* tanto para UNIX (sistemas POSIX com suporte a *pthreads*) como para a API Windows de 32 bits. Ele usa vários conjuntos de macros autoconf para detecção automática dos vários níveis de conformância às *pthreads* na plataforma onde é instalado e ajusta-se automaticamente. Tem sido testado em *GNU/Linux*, *FreeBSD*, *Solaris* e *DEC Tru64*

²O *framework* GNU Common C++ implementa a mesma API em UNIX e Windows.

Unix. Trabalhos recentes estão sendo desenvolvidos para suportar HP/UX.

O objetivo principal do GNU Common C++ tem sido prover uma interface abstrata em C++ com baixíssimo *overhead* a tarefas comumente realizadas. Muitas escolhas feitas buscaram privilegiar a eficiência e a portabilidade.

Está disponível em [55] documentação em formato TeXInfo que fornece uma visão abrangente das classes do *framework*, além de instruções para a configuração e instalação do pacote.

B.5 AutoTools

Para maior produtividade durante o desenvolvimento dos programas usou-se o utilitário `make` [53] para compilar apenas os trechos dos programas que sofreram alterações.

O `make` necessita de um arquivo `Makefile` que descreve as dependências dos arquivos de um programa e os comandos a utilizar para atualizar cada um deles.

Inicialmente o `Makefile` foi gerado e atualizado manualmente. Porém, à medida que o número de arquivos pertencentes ao mesmo projeto aumentava, crescia a dificuldade da análise da dependência dos mesmos. Erros chegaram a ser cometidos, causados pelo esquecimento ocasional de uma dependência indireta de um arquivo objeto à definição de uma classe base, levando à geração de um código executável desatualizado em relação ao código fonte.

Com a ferramenta `automake`, que é parte integrante dos AutoTools, este problema desapareceu, pois ele mesmo se encarrega de verificar todas as dependências automaticamente para a geração de um arquivo `Makefile.in` que posteriormente é usado para a geração de um `Makefile` correto, conforme explicado na seção B.5.1.

B.5.1 Automake

O `automake` [54] é uma ferramenta que automaticamente gera arquivos `Makefile.in` a partir de arquivos chamados `Makefile.am`. Cada arquivo `Makefile.am` contém basicamente definições simples de variáveis que determinam quais são os arquivos necessários à criação dos arquivos executáveis de um projeto, assim como quais são as bibliotecas necessárias a estes executáveis.

Os arquivos `Makefile.in` gerados pelo `automake` obedecem ao padrão GNU para *makefiles*. O documento que especifica as regras deste padrão é longo, complicado e sujeito a mudanças. O objetivo do Automake é facilitar a manutenção dos *makefiles* de maneira que todas as preocupações com relação a estas regras fiquem a cargo do Automake, e não do desenvolvedor do projeto.

O Automake é usado em conjunto com o Autoconf, descrito na seção B.5.2.

Figura B.2: Janela principal do Glade e janela de aplicação sendo desenvolvida.

B.5.2 Autoconf

O Autoconf [52] é um pacote extensível de macros m4 que produzem *shell scripts* para configurar pacotes de *software* com código fonte automaticamente. Estes *scripts* podem adaptar os pacotes a muitos sistemas UNIX sem intervenção manual do usuário. O Autoconf cria um *script* de configuração (chamado *configure*) para um pacote a partir de um arquivo modelo que lista as características do sistema operacional que o pacote pode usar, na forma de macros m4.

Ao final da execução do *script* *configure* são gerados os *Makefiles* a partir dos arquivos *Makefile.in* criados pelo Automake, descrito na seção B.5.1.

B.6 Glade2

Glade2 [39] é um construtor de interface gráfica para GTK+ e GNOME, liberado sob a licença GPL.

Por si só o Glade produz código em linguagem C. Com ferramentas externas que processam os arquivos de descrição da interface em XML pode também produzir código nas linguagens C++, Ada95, Python e Perl, entre outras. Para utilização da linguagem C++ foi utilizado juntamente com o Glade o *glade++*, descrito na seção B.7.

A Figura B.2 mostra a principal janela do Glade. É nela que são especificadas as opções para o projeto, como a linguagem a ser empregada para gerar o código fonte. Ela também inclui uma listagem de cada janela separada que foi construída para a nova aplicação. No caso há apenas uma (*window1*).

A Figura B.3 mostra duas das principais janelas da ferramenta Glade. A janela da esquerda mostra algumas das *widgets* que podem ser utilizadas na construção de novas interfaces gráficas. Já a janela da direita permite modificar diversas propriedades das *widgets*, de maneira bastante prática.

Após construída a interface gráfica e definidos os sinais que podem ser gerados

Figura B.3: Janela com *widgets* e janela de edição de propriedades do Glade.

pelas *widgets* solicita-se através do botão *Build* da janela principal do Glade a criação do esqueleto do código fonte contendo as referências aos *callbacks*. Isto é tudo que o Glade faz. Posteriormente o programador edita este esqueleto com outro programa qualquer para que sejam definidas as ações dos *callbacks* e do restante do programa. Para isto utilizamos o Kdevelop [131], brevemente descrito na seção B.10.

B.7 Glade++

O Glade++ [110] é uma extensão ao Glade2 usado para criar códigos fontes em linguagem C++ que façam uso do gtkmm, descrito na seção B.8.

Ele processa os arquivos de descrição da interface em XML produzidos pelo Glade e produz automaticamente código na linguagem C++.

B.8 gtkmm

Antigamente conhecido como Gtk++, o gtkmm [45] é a interface em C++ oficial para a biblioteca GTK+. Ele é multi-plataforma e acompanha de maneira bastante próxima o desenvolvimento do GTK+. Entre suas características podemos destacar um conjunto

bem definido de classes de *widgets* que podem ser livremente combinadas para a rápida criação de interfaces complexas, inclusive com a criação de *widgets* personalizadas por meio de heranças. O sistema de sinais e conectores é parte de uma biblioteca externa chamada libsigc++, descrita na seção B.9.

B.9 libsigc++

O libsigc++ [111] implementa um sistema de *callback* para a linguagem C++ padrão. Ele permite definir sinais e conectar estes sinais a qualquer rotina *callback*, seja rotina global ou método estático de uma classe. É suportado por uma variedade de compiladores e plataformas.

B.10 KDevelop

O KDevelop [131] é um ambiente de desenvolvimento integrado para UNIX. Um *screenshot* deste aplicativo é mostrado na Figura B.4. Ele possui muitas características interessantes, dentre as quais destacam-se:

- gerenciamento de todas as ferramentas necessárias na programação em C++, como compilador, *linker* e *debugger*;
- editor integrado que permite a fácil edição de vários arquivos de um mesmo projeto;
- modo de destaque de sintaxe;
- auto-preenchimento para variáveis e métodos de classes, argumentos de funções, entre outros;
- quatro modos de visualização em árvore para chavear facilmente entre arquivos fonte, arquivos de cabeçalho, classes e documentação.

Usamos a versão 3.0 (última disponível na época) para facilitar o desenvolvimento dos aplicativos, mesmo aqueles com interface gráfica baseada em GTK+ previamente construídos com Glade (seção B.6).

B.11 Doxygen

O Doxygen [134] é um sistema de documentação para C++, C, Java, IDL (Corba e Microsoft) e com algumas limitações PHP e C#. Ele é capaz de gerar documentação

Figura B.5: Painel de controle do aplicativo VLC.

on-line em formato [HTML](#) ou um manual de referência *off-line* a partir de um conjunto de arquivos fonte documentados. Também há suporte para gerar saída em formato [RTF](#), PostScript, PDF com *hyperlinks*, HTML comprimido e *UNIX man pages*. A documentação é extraída diretamente dos arquivos fontes dos programas, o que torna muito mais fácil manter a documentação consistente com o código fonte.

O Doxygen também pode ser configurado para extrair a estrutura do código de um programa não documentado. Ele permite visualizar as relações entre os vários elementos através de grafos de dependência, diagramas de herança e diagramas de colaboração, que são todos gerados automaticamente.

Este programa foi desenvolvido em Linux, mas é bastante portátil. Ele funciona na maioria dos UNIX e executáveis existem para os sistemas Windows 9x/NT e Mac OS X.

B.12 VLC

O *VideoLan Client* ([VLC](#)) [140] nasceu inicialmente como um tocador multimídia, mas atualmente é também capaz de fazer *streaming unicast* e *multicast* em IPv4 e IPv6. Ele possui vários CODECs de áudio e vídeo, o que o torna capaz de reproduzir vários formatos diferentes, como MPEG-1, MPEG-2, MPEG-4, DivX, mp3, ogg, etc.

O VLC foi portado para diversos sistemas diferentes, a exemplo do Windows, Mac OS X, BeOS, FreeBSD, entre outros, mas o Linux foi adotado como sua principal plataforma de desenvolvimento, sendo que as novas características sempre aparecem primeiro na versão Linux. Para maiores detalhes sobre as características disponíveis do VLC em cada sistema operacional sugere-se consultar [141].

A Figura [B.5](#) apresenta o painel de operação do VLC. À esquerda do painel estão os botões que permitem selecionar rapidamente a fonte a ser usada para obtenção do áudio ou vídeo:

- arquivo acessível pelo sistema de arquivos (disco rígido, CD previamente montado, NFS)
- DVD com suporte a menus, DVD, VCD ou CD de áudio

Figura B.6: Janela de seleção de origem do conteúdo multimídia a ser exibido. Detalhes são mostrados para o caso da rede ser a origem.

- *streaming* pela rede, onde deve ser feita opção por um dos protocolos abaixo:
 - UDP/RTP
 - UDP/RTP Multicast
 - HTTP/FTP/MMS

A Figura B.6 apresenta as opções de protocolo quando se deseja usar a rede como fonte de um vídeo. Nota-se que para recebimento de tráfego *unicast* basta selecionar o protocolo UDP/RTP, informar a porta para a qual o tráfego está sendo enviado e clicar no botão OK. Logo em seguida o vídeo começa a ser exibido.

Quando a máquina está usando Linux com suporte à API *Video for Linux* (V4L) e existe uma placa de aquisição de vídeo instalada, é possível usá-la como fonte (opção "Satellite" na janela da Figura B.6).

Se a opção "Stream Output" for selecionada na janela da Figura B.6 o VLC abre uma nova janela, mostrada na Figura B.7. Nesta janela é possível configurar detalhes do envio do vídeo, assim como a máquina para a qual o vídeo deve ser enviado e a porta a ser utilizada na máquina destino.

Figura B.7: Janela do aplicativo VLC para configuração da forma de transmissão de conteúdo multimídia.

B.13 Netperf

O Netperf [68] é um *benchmark* usado para medir performance de redes de computadores. Neste trabalho ele basicamente foi utilizado para avaliar a largura de banda disponível com o uso dos protocolos UDP e TCP através de *sockets* BSD, mas ele é também capaz de mensurar latência fim-a-fim e usar outros protocolos.

Um aspecto interessante é que ele permite selecionar, além do protocolo, o tamanho dos pacotes e a duração dos testes. Isto o transforma em uma ferramenta prática para saturar *links* com tráfego UDP unidirecional por um intervalo bastante longo, a fim de permitir a realização de testes simultâneos com outros aplicativos que visam verificar os aspectos de QoS das transmissões multimídia em situações de congestionamento na rede.

B.14 Kernel do Linux

Usamos sempre, na medida do possível, a última versão de *kernel* estável disponível (versão 2.4.x na época em que a maior parte das experiências foi realizada) porque o desenvolvimento dos *drivers* ATM que fazem parte do *kernel* ainda estava em andamento. As versões mais novas além de implementarem novas correções e otimizações são consideradas também mais confiáveis e estáveis. No que se refere ao ATM, a série 2.6.x do *kernel* lançada após a realização da maior parte das experiências deste trabalho não apresenta grandes novidades com relação à série 2.4.x, sendo que a maior parte das modificações nos *drivers* limitou-se estritamente às adaptações necessárias para que eles pudessem compilar corretamente e funcionar com o novo *kernel*.

B.14.1 Suporte ao ATM no Linux

Pode de início parecer que muito tempo já se passou para que a maturidade dos *drivers* ATM fosse alcançada, haja visto que muitos outros *drivers* de rede já foram desenvolvidos para Linux e que já existe um procedimento padrão a ser seguido para o desenvolvimento de um *driver Ethernet* [117], por exemplo. Mas é preciso lembrar que a tecnologia ATM é muito diferente da *Ethernet*, o que acaba levando a problemas inéditos. Apenas para ilustrar, acompanhando a lista de discussão dos desenvolvedores de *drivers* ATM pude acompanhar um deles. Tratava-se basicamente sobre o gerenciamento de VCIs realizado no interior do *driver*. Sabe-se que para estabelecer comunicação em ATM é necessário primeiro abrir um canal ATM. Depois disso as células que saem da interface possuem em seu cabeçalho um VPI e um VCI válidos que farão com que esta célula seja chaveada corretamente até seu destino. O destino também decide o

que fazer com a célula baseando-se no seu par VPI/VCI. Muitos canais podem existir simultaneamente e por isso o gerenciamento de VCIs precisa ser bastante eficiente. Pelo que entendi, existia uma estrutura com todos os VCIs que era praticamente duplicada em outra região de memória por alguma razão muito forte, possivelmente por questões de eficiência ou diferenças na recepção e na transmissão. O fato é que em situações muito singulares podia acontecer do gerenciamento ficar inconsistente logo após o fechamento de um canal, pois isto levava à retirada do VCI correspondente de uma das estruturas mas permitia que ainda fossem recebidas algumas células da máquina na outra extremidade do canal. Algumas questões debatidas nesta lista foram então como impedir este estado inconsistente sem perda de desempenho e o que fazer com estas células (se deveriam ser descartadas ou ainda entregues às camadas superiores, mesmo após a solicitação de fechamento do canal). Mesmo entre desenvolvedores experientes questões similares podem ser polêmicas e causar longas discussões até que se opte por uma solução que seja a mais elegante ou às vezes, a mais viável (mas nem sempre a mais elegante). Não há situação análoga com *Ethernet*, uma vez que ele não usa o conceito de conexão.

B.14.2 Habilitando o suporte ao ATM no *kernel*

Como o desenvolvimento do ATM em Linux ainda é considerado experimental, há a necessidade de reconfigurar o *kernel* e recompilá-lo para obter o suporte ATM. *Kernels* pré-compilados que acompanham as distribuições não costumam trazer as partes experimentais compiladas (se bem que algumas, a exemplo da distribuição Fedora Core 3 já trazem os módulos ATM compilados). Mesmo que o façam, é recomendável verificar se o *kernel* que acompanha a distribuição corresponde a uma das últimas versões disponíveis, pelas razões já discutidas.

No processo de configuração de um novo *kernel* a ser compilado, antes de mais nada, é necessário ir no menu "Code maturity level options" e habilitar a opção "Prompt for development and/or incomplete code/drivers" para que partes do *kernel* que ainda estejam em desenvolvimento apareçam e sejam habilitáveis. No menu "Networking options" a opção "Asynchronous Transfer Mode (ATM) (EXPERIMENTAL)" deve ser habilitada. Depois que isto é feito surgem as sub-opções:

- Classical IP over ATM (NEW)
- LAN Emulation (LANE) support (NEW)
- RFC1483/2684 Bridged protocols (NEW)

Quando a opção do LANE é habilitada ainda surge uma outra nova opção: "Multi-Protocol Over ATM (MPOA) support (NEW)". Fizemos uso intensivo de [CLIP](#), uso

ocasional de [LANE](#) mas não usamos [MPOA](#) já que basicamente ele permite que dispositivos de borda como roteadores, *bridges* e máquinas ATM estabeleçam circuitos ATM virtuais diretos para fazer *bypass* em roteadores para aumentar a performance, situação que não exploramos.

Por fim, no menu "Network device support" deve-se ir para o sub-menu "ATM drivers" e selecionar os *drivers* a serem usados. As opções disponíveis até o momento são mostradas na Figura [B.8](#).

```
< > ATM over TCP (NEW)
< > Efficient Networks Speedstream 3010 (NEW)
< > Efficient Networks ENI155P (NEW)
< > Fujitsu FireStream (FS50/FS155) (NEW)
< > ZeitNet ZN1221/ZN1225 (NEW)
<M> IDT 77201 (NICStAR) (ForeRunnerLE) (NEW)
[*] Use suni PHY driver (155Mbps)
[ ] Use IDT77015 PHY driver (25Mbps)
< > IDT 77252 (NICStAR II) (NEW)
< > Madge Ambassador (Collage PCI 155 Server) (NEW)
< > Madge Horizon [Ultra] (Collage PCI 25 and Collage PCI 155 Client) (NEW)
< > Interphase ATM PCI x575/x525/x531 (NEW)
< > FORE Systems 200E-series (NEW)
< > ForeRunner HE Series (NEW)
```

Figura B.8: Opções de *drivers* para placas ATM no *kernel* 2.4.24.

Nós usamos o *driver* para a interface ForeRunnerLE. Ao selecionar o *driver* Nicstar deve-se selecionar uma das duas opções de *PHYsical layer chips*. No nosso caso a opção correta é "Use suni PHY driver (155Mbps)". A outra é usada em interfaces de 25Mbps.

A opção de ATM sobre TCP serve para emular ATM usando TCP. É útil para desenvolvimento, especialmente para quem não dispõe de *hardware* ATM real.

A Figura [B.9](#) resume as opções que foram habilitadas no *kernel* 2.4.24 para obter o suporte ao ATM em Linux.

B.14.3 Pseudodiretório /proc

O diretório /proc presente na raiz do sistema de arquivos de uma máquina Linux não corresponde a um diretório propriamente dito, pois os arquivos que nele se encontram na verdade não estão armazenados em nenhum dispositivo físico. Eles correspondem a entidades virtuais criadas pelo *kernel* para prover acesso a parâmetros, estruturas de dados e estatísticas do *kernel*. O conteúdo destes arquivos não é sempre estático, e pode ser gerado sob demanda pelo *kernel* no momento da leitura do arquivo. Certas configurações do *kernel* podem ser alteradas através de escritas em alguns destes

```

[Code maturity level options]
  [*] Prompt for development and/or incomplete code/drivers

[Networking options]
  <M> Asynchronous Transfer Mode (ATM) (EXPERIMENTAL)
  <M>   Classical IP over ATM
  [ ]   Do NOT send ICMP if no neighbour
  <M>   LAN Emulation (LANE) support
  < >   Multi-Protocol Over ATM (MPOA) support
  < >   RFC1483/2684 Bridged protocols

[Network device support]
  [ATM drivers]
    <M> IDT 77201 (NICStAR) (ForeRunnerLE) (NEW)
    [*]   Use suni PHY driver (155Mbps)
    [ ]   Use IDT77015 PHY driver (25Mbps)

```

Figura B.9: Resumo das opções habilitadas no *kernel* 2.4.24 para obter o suporte ao ATM em Linux.

arquivos.

Diversas informações estão disponíveis através dos arquivos em `/proc`, como por exemplo:

`/proc/cpuinfo`: informa quantos processadores estão disponíveis no computador e quais as características de cada um (modelo, frequência do relógio, tamanho da cache, quais os conjuntos de instruções implementados, etc.);

`/proc/meminfo`: informa quanta memória RAM está instalada no computador, quanta memória está disponível para *swap*, quanta memória está livre, etc.;

`/proc/pid/3`: é um diretório que contém diversas informações sobre o processo com identificador `pid`, como arquivos e *pipes* abertos pelo processo, etc.

Maiores informações sobre o pseudodiretório `/proc` podem ser encontradas em [3].

B.15 ATM Tools

O *kernel* do Linux implementa apenas os *device drivers* necessários para o correto funcionamento das interfaces. O pacote *ATM Tools* [27], que não faz parte do *kernel*, deve ser instalado separadamente quando se deseja usar a tecnologia ATM na plataforma Linux. Fazem parte do pacote diversos utilitários e *daemons*:

- Documentação em formato SGML DocBook

³`pid` corresponde ao identificador do processo.

- *daemon* de sinalização UNI 3.0, UNI 3.1 e UNI 4.0: *atmsigd*
- biblioteca de sinalização AAL (SSCOP, SSCF e SAAL)
- manipulação de mensagens semelhante ao ITU Q.2931
- *daemon* de registro de endereço ILMI: *ilmid*
- programas de manutenção ATM: *atmaddr*, *atmdiag*, *atmdump*, *atmloop*, *atmtcp*, *enitune*, *esi*, *sonetdiag*, *saaldump* e *zntune*
- Programas de teste: *align*, *aping*, *aread*, *awrite*, *br*, *bw*, *isp* e *ttcp_atm*
- *daemon* e ferramentas ATMARP: *atmarpd* e *atmarp*
- *daemon* para LANE: *zeppelin*
- servidores do LANE: *bus*, *lecs*, *les*
- *daemon* do MPOA: *mpcd*
- ferramentas para *debugging*: *delay*, *ed*, *encopy*, *endump*, *svctor*, *zndump* e *znth*
- bibliotecas para aplicações e *daemons*

Modelo de comunicação e protocolos de rede utilizados

A [API](#) ATM desenvolvida para o Linux é intencionalmente parecida com a interface dos *sockets* UNIX no [BSD release 4.3](#) [26].

Até o presente momento apenas as qualidades de serviço [CBR](#) e [UBR](#) são suportadas.

O paradigma fundamental de comunicação em ATM é a conexão. Este capítulo descreve os mecanismos para estabelecer, usar e liberar conexões ATM através da API.

C.1 Fases de uma conexão

Uma conexão tipicamente passa pelas seguintes fases:

- preparação da conexão
- criação da conexão
- troca de dados
- término da conexão

Durante a *preparação da conexão*, parâmetros são definidos e recursos locais (como descritores de *sockets*) são alocados. Nenhum recurso de rede local ou remoto é alocado durante a fase da preparação. Durante a *criação da conexão* recursos de rede locais (largura de banda, identificadores de conexão, *buffers*, etc.) são alocados. Alocação de recursos na rede pode ser feito por gerenciamento de rede ([PVCs](#)) ou pode ser feita como parte da criação da conexão ([SVCs](#)). Na *troca de dados*, dados são enviados por uma conexão previamente estabelecida. Finalmente, durante o *término da conexão* a comunicação é cessada e os recursos são desalocados.

Preparação da conexão, criação da conexão e troca de dados são tipicamente realizados nesta ordem. O término da conexão pode ser iniciado a qualquer momento. Pode até mesmo sobrepor-se a outras fases.

C.1.1 Fases para *sockets* PVC

Abrir um *socket* PVC significa anexar um *endpoint* a uma conexão que normalmente já existe na rede. O diagrama de estados na Figura C.1 ilustra o ciclo de vida dos *sockets* PVC.

Figura C.1: Diagrama de estados para a abertura de um *socket* PVC.

Sockets PVC são criados com a chamada de sistema `socket`. Posteriormente os requisitos de QoS são indicados, um descritor de endereço é criado, e `bind` ou `connect` é chamado (eles têm funcionalidades idênticas quando usados com PVCs) para abrir o VC. Se o VC não pode ser aberto a operação pode ser tentada novamente, possivelmente após mudar o descritor de endereço e/ou os parâmetros de tráfego.

Uma vez que o VC é criado, dados podem ser trocados até que o *socket* seja fechado com a chamada de sistema `close`.

C.1.2 Fases para *sockets* SVC

SVCs são conexões cuja criação na rede é controlada pelos elementos finais da comunicação. Um *endpoint* pode tanto aceitar uma comunicação entrante (*passive open*) como pode tentar criar uma nova conexão por si mesmo (*active open*). A Figura C.2 ilustra o ciclo de vida de um *active open*.

Figura C.2: Ciclo de vida de um *Active open* de um *socket* SVC.

A maneira mais comum de criar uma conexão é criar um *socket* SVC com `socket`, especificar os requisitos de QoS, preparar o descritor de endereço, requisitar a criação de uma conexão com um `connect` bloqueante para bloquear até que a rede aceite (ou rejeite) a conexão, trocar dados e eventualmente fechar o *socket* com `close`.

O *endpoint* local pode ser amarrado a um porto específico com a chamada de sistema `bind`. Ao usar uma conexão não-bloqueante o processo continua a executar enquanto a conexão é criada.

O *passive open* é ilustrado na Figura C.3. Como usual, o *socket* é criado com a chamada de sistema `socket` e os parâmetros de QoS são especificados. Depois o descritor de endereço especifica o tipo de conexão que deve ser atribuído a este *socket* e o *socket* é amarrado com a chamada de sistema `bind`. O *Service Access Point* (SAP) é registrado na entidade de sinalização local com a chamada de sistema

`listen`. A seguir o processo pode bloquear na chamada de sistema `accept` até que uma requisição de conexão seja recebida, ou ele pode ficar fazendo *polling* usando `accepts` não-bloqueantes ou usando `select`. Se uma conexão entrante combina com a especificação do SAP, a chamada de sistema `accept` sucede (ou o *socket* torna-se legível de maneira que `select` retorna e `accept` pode ser chamado. Quando `accept` sucede um novo *socket* é criado para a conexão e a troca de dados começa.

Figura C.3: Diagrama de estados para um *Passive open* de um *socket* SVC.

O *socket* usado com `listen` e o(s) *socket(s)* usado(s) para trocar dados podem ser individualmente fechados com `close`. Fechar um *socket* que foi usado com `listen` só remove o registro do SAP e não afeta as conexões que eventualmente tenham sido criadas.

C.2 Descritores de endereço

Um descritor de endereço especifica o endereço real de uma entidade destino. Ele é usado sempre que uma estrutura de endereço (`struct sockaddr`) é necessária. Sua estrutura geral é mostrada na Figura C.4, onde *tipo* é o tipo de endereço (`pvc` para PVCs e `svc` para SVCs) e *prefixo* é o prefixo do nome de campo correspondente, que segue a convenção usada para outras famílias de endereço. O prefixo é `sap` para PVCs e `sas` para SVCs. *prefixo_family* contém a família do endereço. *prefixo_addr* contém a informação de endereço real. O formato de endereço difere significativamente para PVCs e SVCs.

```
struct sockaddr_atmtipo {
    unsigned short prefixo_family;
    struct {
        ...
    } prefixo_addr;
};
```

Figura C.4: Estrutura geral do descritor de endereço ATM.

O endereçamento PVC é descrito na seção C.3 e o endereçamento SVC na seção C.4.

C.3 Endereçamento PVC

PVCs são endereçados especificando a interface local e o identificador de conexão (VPI e VCI) a usar nesta interface. A estrutura de endereço é definida em `/usr/include/linux/atm.h` e mostrada na Figura C.5.

```
struct {
    short itf;
    short vpi;
    int vci;
} sap_addr;
```

Figura C.5: Estrutura de um endereço PVC.

`itf` é o número da interface local (iniciando de zero). `vpi` é o identificador de caminho virtual na faixa de 0 a 255. `vci` é o identificador de canal virtual na faixa de 0 a 65535. Os VCIs de 0 a 31 são reservados pelo ITU-T e pelo ATM Forum e não devem ser usados por nenhuma aplicação. Por isso apenas processos privilegiados têm permissão para abrir VCIs usando VCIs abaixo de 32.

Dois *sockets* PVC podem compartilhar o mesmo endereço se um deles o usa somente na direção de recepção, isto é, `qos.txtp.class == ATM_NONE`, e o outro o usa somente na direção da transmissão.

C.3.1 Endereços PVC especiais

Os seguintes valores especiais são reconhecidos em endereços PVC:

`itf == ATM_ITF_ANY` seleciona a interface de número mais baixo na qual o par VPI/VCI é válido e ainda não está em uso.

`vpi == ATM_VPI_ANY` seleciona o VPI de número mais baixo no qual o VCI especificado é válido e ainda não está em uso.

`vci == ATM_VCI_ANY` seleciona o VCI livre de número mais baixo (não reservado) na interface especificada para a qual os VPIs correspondem.

`vpi == ATM_VPI_UNSPEC` não aloca nenhum par VPI/VCI e usa o número de interface somente para controle de recursos. `vpi == ATM_VPI_UNSPEC` também implica que o VCI não é especificado. Portanto, o valor de VCI é ignorado.

`vci == ATM_VCI_UNSPEC` não aloca nenhum par VPI/VCI e usa a interface e números VPI somente para controle de recursos.

Endereços contendo componentes `ATM_parte_ANY` são chamados de endereços coringa (*wildcards addresses*). Analogamente, endereços contendo componentes `ATM_parte_UNSPEC` são chamados de endereços especificados de maneira incompleta.

Um endereço pode conter mais de um componente coringa, por exemplo, `itf == ATM_ITF_ANY`, `vpi == ATM_VPI_ANY` e `vci == ATM_VCI_ANY` selecionaria o endereço PVC livre de número mais baixo. Ao usar mais de um componente coringa em um endereço, números de interface são mais significativos que números VPI, e números VPI são mais significativos que números VCI ao determinar o endereço mais baixo.

C.3.2 Representação textual

Os formatos para representação textual de endereços PVC são:

1. `vpi.vci`
2. `itf.vpi.vci`

`itf`, `vpi` e `vci` são números decimais não negativos sem zeros à esquerda, ou caracteres especiais `?` ou `*` indicando um endereço não especificado ou um endereço com componente coringa, respectivamente. Se o primeiro formato é usado o número de interface é assumido ser zero.

C.4 Endereçamento SVC

O endereçamento SVC pode ocorrer em três níveis:

- seleção de um *gateway* que repassa conexões entre uma rede pública e uma privada

- seleção de um sistema final dentro de uma rede pública ou privada
- seleção se um serviço no sistema final selecionado

Redes públicas usam números de endereçamento E.164 (ISDN/telefonia [63]). Redes privadas usam números NSAP privados do ATM Forum, conforme descrito em [12]. No que tange ao endereçamento, redes públicas que também usam endereços NSAP são tratadas como redes privadas.

C.4.1 Endereçamento do sistema final

Para selecionar um sistema final, a seguinte informação de endereço é necessária (os exemplos na Figura C.6 estão entre parênteses):

- se o destino está em uma rede privada: o endereço privado ($A \rightarrow B$)
- se o destino está em uma rede pública (usando endereços E.164): o endereço público ($A \rightarrow D, D \rightarrow E$)
- se o destino está em uma rede privada que é alcançada via uma rede pública usando endereços E.164: os endereços privado e público ($A \rightarrow F \rightarrow G$)

Figura C.6: Endereçamento SVC.

O serviço em um sistema final é identificado especificando tanto o protocolo de camada baixa (camadas OSI 2 ou 3), ou o protocolo de camada alta, ou ambos.

A estrutura de endereço SVC é definida em `/usr/include/linux/atm.h` e mostrada na Figura C.7.

```
struct {  
    unsigned char prv[ATM_ESA_LEN];  
    char pub[ATM_E164_LEN+1];  
    struct atm_blli *blli;  
    struct atm_bhli bhli;  
} sas_addr;
```

Figura C.7: Estrutura de endereço SVC.

O endereço privado `prv` é um endereço NSAP ATM Forum de um sistema final ATM em uma rede privada. Seu comprimento é sempre `ATM_ESA_LEN` (vinte) *bytes*. O endereço privado é omitido colocando seu primeiro *byte* em zero.

O endereço público `pub` é um endereço E.164. É representado por uma cadeia de caracteres [ASCII](#) terminada em zero de até `ATM_E164_LEN` dígitos decimais. O endereço público é omitido colocando seus primeiros *bytes* em zero.

`blli` e `bhli` identificam o SAP, ou seja, a aplicação chamante ou chamada ou uma parte funcional da última. Informação de camada alta é opcional (sua ausência é indicada colocando `bhli.hl_type` em `ATM_HL_NONE`. Informação de camada baixa também é opcional (sua ausência é indicada colocando `blli` em `NULL`)

C.5 Descritores de QoS

Os descritores de QoS são usados para expressar os requisitos de QoS ou parâmetros de uma conexão. Eles são codificados em uma estrutura de dados de tipo `struct atm_qos`, que é definida em `/usr/include/linux/atm.h`. Ela contém os seguintes campos:

```
struct atm_qos {  
    struct atm_trafprm txtp;  
    struct atm_trafprm rxtp;  
};
```

Figura C.8: Estrutura do descritor de QoS.

`txtp` e `rxtp` são os parâmetros de tráfego na direção de envio e recepção, respectivamente. Suas estruturas são descritas na seção seguinte.

C.5.1 Parâmetros de tráfego

Parâmetros de tráfego são adicionados ao descritor de QoS atribuindo valores aos campos correspondentes.

Parâmetros de tráfego são codificados em uma estrutura de dados de tipo `struct atm_trafprm` que contém os seguintes campos:

```
struct atm_trafprm {  
    unsigned char class;  
    int max_pcr;  
    int min_pcr;  
    int max_cdv;  
    int max_sdu;  
};
```

Figura C.9: Estrutura usada para armazenar parâmetros de tráfego.

`class` representa a classe de tráfego, indicando as propriedades gerais de tráfego. `min_pcr` e `max_pcr` indicam as taxas de células de pico (**PCR**) desejadas, em células por segundo. `max_cdv` representa a variação máxima da latência (**CDV**), em microssegundos. `max_sdu` representa o tamanho máximo do *Service Data Unit* (**SDU**), em bytes. O uso destes campos é descrito nas seções seguintes.

A compatibilidade da QoS requisitada com os recursos disponíveis não é verificada durante a preparação da conexão. Isto só é feito durante a criação da conexão. Além disso, já que apenas certos valores são suportados para alguns parâmetros, a conexão pode ser criada com parâmetros (ligeiramente) diferentes dos requisitados. Os parâmetros efetivos podem ser obtidos solicitando informações do *socket* com `SO_ATMQOS`, conforme detalhado na seção C.7.4.

Deve ser observado que garantias de serviço no processamento do tráfego CBR só se aplicam à rede. É responsabilidade da aplicação garantir que recursos de máquina suficientes estão disponíveis para gerar ou aceitar os fluxos de tráfego.

C.5.2 Classes de tráfego

As seguintes classes são definidas:

ATM_NONE: nenhum tráfego nesta direção

ATM_ANYCLASS: aceita qualquer classe

ATM_CBR: taxa de bits constante (CBR)

ATM_UBR: taxa de bits não especificada (UBR)

Quando nenhuma classe é especificada (campo é colocado em zero), ATM_NONE é usado. Quando a classe de tráfego é ATM_NONE, todos os demais parâmetros de tráfego são ignorados.

ATM_ANYCLASS é usado para especificar SAPs SVC que são compatíveis com qualquer classe de tráfego que é requisitada por quem solicita conexão.

Dependendo da classe de tráfego apenas alguns parâmetros são usados:

	ATM_NONE	ATM_CBR	ATM_UBR
max_pcr	—	Sim	—
min_pcr	—	Sim	—
max_cdv	—	Sim	—
max_sdu	—	Sim	Sim

Figura C.10: Parâmetros utilizados em cada classe de tráfego.

C.5.3 Taxa de células de pico

O PCR mínimo `min_pcr` e o PCR máximo `max_pcr` especificam a largura de banda que será consumida pela conexão. Eles são quantificados em células por segundo.

O *hardware* normalmente não permite especificar o PCR com uma resolução de uma célula por segundo. Como esta quantidade é discretizada, irá-se escolher uma taxa de acordo com as seguintes regras:

- se somente `min_pcr` é especificado (ou seja, não é zero), a próxima taxa possível acima de `min_pcr` é usada;
- se somente `max_pcr` é especificado (ou seja, não é zero nem `ATM_MAX_PCR`), é usada uma taxa possível menor que `max_pcr` e também menor que a largura de banda ainda disponível no *link*.
- se ambos `min_pcr` e `max_pcr` forem especificados, a próxima taxa possível acima de `min_pcr` é usada.

A operação falha quando a taxa selecionada excede a largura de banda ainda disponível ou o PCR máximo (se especificado).

No envio os suavizadores de tráfego são ajustados para nunca exceder a taxa de células selecionada. Na recepção o receptor prepara-se para aceitar células chegando a pelo menos esta taxa.

O PCR mínimo normalmente é zero ou um inteiro positivo. O PCR máximo normalmente é um inteiro positivo. O valor especial `ATM_MAX_PCR` indica uma taxa de células ilimitada, ou seja, a velocidade do *link*. `ATM_MAX_PCR` nunca é usado pelo sistema operacional para retornar uma taxa de células.

C.5.4 Variação da latência das células

A variação máxima da latência das células `max_cdv` especifica o limite superior para o número de microssegundos que uma célula pode chegar antes do seu devido instante (em relação às células anteriores do mesmo VC). O sistema operacional pode selecionar um CDV menor que o solicitado. No envio as células nunca serão emitidas num ritmo que exceda esta garantia. O receptor sempre se prepara para receber células que cheguem com pelo menos com o CDV especificado. O CDV máximo normalmente é um inteiro positivo. Colocando-o em zero indica um CDV não especificado.

C.5.5 Tamanho do SDU

O tamanho máximo do *Service Data Unit* (SDU) especifica a quantidade máxima de dados (em *bytes*) que serão enviados com uma única chamada de sistema. Para envio e recebimento os *buffers* são dimensionados de acordo. O SDU é um inteiro positivo. Quando colocado em zero o SDU máximo suportado pelo respectivo protocolo é assumido. O sistema operacional nunca retorna zero como tamanho máximo de SDU. Como o SDU máximo pode levar a desperdício na alocação de recursos, `max_sdu` deve sempre ser definido se o tamanho máximo da mensagem for conhecido.

C.6 Rotinas de biblioteca

Um conjunto de rotinas é fornecido para codificar, decodificar e comparar endereços e para converter taxas SDU para taxas de células.

Para usar a biblioteca, o arquivo de cabeçalho `/usr/include/atm.h` deve ser incluído. Além disso a biblioteca `libatm.a` deve ser linkada.

C.6.1 `text2atm`

`text2atm` converte uma representação textual de um endereço PVC ou SVC para sua codificação binária correspondente.

```
int txt2atm(const char *text, struct sockaddr *addr,
int length, int flags);
```

`text` aponta para uma string terminada em zero contendo a representação textual do endereço. `addr` aponta para uma estrutura grande o bastante para armazenar a estrutura de endereço resultante, que pode ser tanto um `sockaddr_atmpvc` como um `sockaddr_atmsvc`. `length` indica o comprimento da estrutura de dados. A conversão falha se for fornecido espaço insuficiente. `flags` é um conjunto de opções de processamento, conforme mostrado na Figura C.11.

```
#define T2A_PVC
#define T2A_SVC
#define T2A_UNSPEC
#define T2A_WILDCARD
#define T2A_NNI
#define T2A_NAME
```

Figura C.11: Opções de processamento na conversão da representação textual de um endereço ATM para sua codificação binária correspondente.

T2A_PVC e T2A_SVC habilitam os formatos de endereço correspondentes. Se nenhum dos dois é definido, a presença dos dois é assumida. T2A_UNSPEC e T2A_WILDCARD permitem o uso de partes não especificadas ou coringas, respectivamente, em endereços PVC. T2A_NNI permite o uso de VPIs com 12 bits ¹. T2A_NAME permite resolução usando um diretório de serviços. Resolução baseada em informação numérica tem precedência sobre resolução através de diretórios.

text2atm retorna o comprimento da parte NSAP em caso de sucesso (0 se não há parte NSAP, 160 se há uma, ou algo entre estes dois extremos quando são usados coringas), ou um inteiro negativo em caso de falha.

Em caso de sucesso, text2atm inicializa os seguintes campos em PVCs: sap_family e tudo em sap_addr. Em SVCs, text2atm inicializa sas_family, sas_addr.priv e sas_addr.pub. Em caso de falha text2atm pode bagunçar toda a estrutura de endereço.

C.6.2 atm2text

atm2text realiza a operação inversa de text2atm: converte um endereço codificado em bits para sua representação textual.

```
atm2text(char *buffer, int length, const struct
sockaddr *addr, int flags);
```

buffer aponta para o *buffer* onde a *string* resultante será armazenada. length indica o tamanho do *buffer*. Ele deve ser grande o bastante para armazenar toda a *string*, além da terminação 0. MAX_ATM_ADDR_LEN é o comprimento máximo de uma string (excluindo a terminação 0) que atm2text pode gerar. addr aponta para a estrutura de endereço a converter. flags é um conjunto de opções de processamento, conforme mostrado na Figura C.12.

A2T_PRETTY habilita a adição de pontos nos endereços ATM. A2T_NAME habilita buscas de diretório para traduzir o endereço para um nome.

atm2text retorna o comprimento da *string* resultante (sem a terminação 0) em

¹VPIs com tais valores são apenas permitidos na NNI e não na UNI.

```
#define A2T_PRETTY 1
#define A2T_NAME 2
```

Figura C.12: Opções de processamento na conversão de um endereço ATM codificado em bits para sua representação textual.

caso de sucesso e um inteiro negativo em caso de falha.

C.6.3 atm_equal

`atm_equal` é usado para verificar se dois endereços ATM são iguais:

```
int atm_equal(const struct sockaddr_atmsvc *a,
const struct sockaddr_atmsvc *b, int len, int flags);
```

`a` e `b` apontam para os dois endereços a serem comparados. `len` corresponde ao comprimento (em bits) do prefixo do endereço NSAP a considerar quando estiverem sendo usados coringas. `flags` corresponde a um conjunto de opções de processamento, conforme mostrado na Figura C.13.

```
#define AXE_WILDCARD 1
#define AXE_PRVOPT 2
```

Figura C.13: Opções de processamento na comparação de dois endereços ATM.

Quando `AXE_WILDCARD` é setado, somente `len` bits de um endereço NSAP são comparados.

Quando `AXE_PRVOPT` é setado, dois endereços ainda são considerados iguais se suas partes públicas combinam, mesmo quando apenas um deles tem uma parte privada. Endereços com a mesma parte privada são sempre iguais, independentemente da parte pública.

`atm_equal` retorna um inteiro não nulo se os endereços combinam, ou zero caso contrário ou quando uma comparação não é possível.

C.6.4 sdu2cell

`sdu2cell` calcula o número de células que seriam geradas enviando um conjunto de SDUs por um dado *socket*.

```
int sdu2cell(int s, int sizes, const int *sdu_size,
int *num_sdu);
```

`s` é o descritor de *socket*. `sizes` representa o número de tamanhos de SDUs descritos nos vetores `sdu_size` e `num_sdu`. Cada elemento de `sdu_size` indica o comprimento (em *bytes*) de um SDU. Cada elemento de `num_sdu` indica o número de vezes

que um SDU com o tamanho correspondente é enviado.

`sdu2cell` calcula o número total de células ATM que seriam enviadas pela conexão e retorna este número, ou -1 se ocorrer um erro (um *overflow*, por exemplo). `sdu2cell` não verifica se os tamanhos de SDU especificados são válidos para a conexão.

C.7 Preparação da conexão

A fase da preparação da conexão consiste nas seguintes partes:

- criação do *socket*
- inicialização do descritor de QoS
- inicialização do descritor de endereço
- especificação dos parâmetros de tráfego

Criação do socket e inicialização do descritor de QoS devem ser sempre realizadas antes da *especificação dos parâmetros de tráfego*.

C.7.1 Criação do *socket*

Sockets são criados com a chamada de sistema `socket`.

```
socket(int dominio,int tipo,int protocolo);
```

O *dominio* indica se deve ser usada sinalização para esta conexão (SVC) ou não (PVC). Os seguintes domínios são definidos:

- PF_ATMPVC conexão ATM PVC
- PF_ATMSVC conexão ATM SVC

O *tipo* seleciona as características gerais do protocolo da camada de transporte. Somente o tipo `SOCK_DGRAM` está disponível até o momento, especificando um transporte de datagramas não-confiável. Apesar de nenhuma garantia de seqüência explícita ser fornecida, o uso do ATM geralmente implica que tal seqüência será preservada.

Com *protocolo* é possível selecionar um protocolo específico. Atualmente, apenas os protocolos `ATM_AAL0` e `ATM_AAL5` estão definidos.

C.7.2 Alocação e inicialização dos descritores de endereço

O descritor de endereço é uma estrutura de dados que contém a informação de endereço real. Veja a seção C.2 para uma descrição detalhada.

Dependendo da família de endereços, um dos tipos de dados seguintes é usado para o descritor de conexão:

- `struct sockaddr_atmpvc` para um PVC
- `struct sockaddr_atmsvc` para um SVC

As estruturas de dados são definidas em `/usr/include/linux/atm.h` e mostradas nas Figuras C.14 e C.15.

```
struct sockaddr_atmpvc {
    unsigned short sap_family;
    struct {
        short itf;
        short vpi;
        short vci;
    } sap_addr;
    struct atm_trafprm sap_txtp;
    struct atm_trafprm sap_rxtp;
};
```

Figura C.14: Estrutura de dados usada como descritor de conexão PVC.

`sap_family` contém a família de endereço e deve ser setado para `AF_ATMPVC`. `sap_addr` é o endereço PVC como descrito na seção C.3. `sap_txtp` e `sap_rxtp` estão obsoletos e o suporte a eles será descontinuado no futuro.

```
struct sockaddr_atmsvc {
    unsigned short sas_family;
    struct {
        unsigned char prv[ATM_ESA_LEN];
        char pub[ATM_E164_LEN+1];
        struct atm_blli *blli;
        struct atm_bhli bhli;
    } sas_addr;
    struct atm_trafprm sas_txtp;
    struct atm_trafprm sas_rxtp;
};
```

Figura C.15: Estrutura de dados usada como descritor de conexão SVC.

Os campos são os mesmos que para PVCs, com a óbvia exceção que a família deve ser setada para `AF_ATMSVC` e que o endereço é um endereço SVC como descrito na

seção C.4. `sas_tntp` e `sas_rntp` estão obsoletos e suporte a eles será descontinuado no futuro.

O descritor de endereço é inicializado setando todos seus *bytes* em zero com a rotina de biblioteca C `memset` [120]. Simplesmente zerar todos os campos não é um modo válido para inicializar o descritor de endereço, porque novos campos que podem vir a ser adicionados no futuro não seriam inicializados. Depois disso os campos podem receber os valores desejados.

C.7.3 Alocação e inicialização dos descritores de QoS

O descritor de QoS é uma estrutura de dados que descreve as propriedades de tráfego (como taxa oferecida, tamanho do SDU, etc.) e requisitos da conexão (máximo CDV, etc.). Veja a seção C.5 para maiores detalhes.

O descritor de QoS é inicializado setando todos seus *bytes* em zero usando a rotina de biblioteca C `memset`. Simplesmente zerar todos os campos não é um modo válido para inicializar o descritor de endereço. Depois disso os campos podem receber os valores desejados.

C.7.4 Atribuição dos parâmetros de QoS

Os parâmetros de QoS de uma conexão são setados através da chamada de sistema `setsockopt`. De maneira análoga eles podem ser obtidos com `getsockopt`.

Para atribuir e obter parâmetros de QoS o nível `SOL_ATM` e o nome de opção `SO_ATMQOS` são usados. O valor da opção é uma `struct atm_qos` (veja seção C.5) e a opção `length` é seu tamanho em *bytes*.

Simplesmente atribuir os parâmetros de QoS não aloca nenhum recurso ou verifica sua disponibilidade.

Os parâmetros de QoS devem ser atribuídos antes de tentar estabelecer uma conexão ou esperar com `listen` por uma solicitação de conexão. É um erro tentar tal operação sem antes atribuir com sucesso os parâmetros de QoS.

Os parâmetros de QoS podem ser alterados quantas vezes for necessário antes de chamar `connect` ou `bind`, mas eles não devem ser alterados quando o estabelecimento da conexão está em andamento ou mesmo depois que ela foi estabelecida. Modificação dos parâmetros de QoS para conexões ativas pode vir a ser suportado no futuro.

C.7.5 Exemplo

É mostrado na Figura C.16 um trecho de código que ilustra o uso do descritor de QoS.

```
struct sockaddr_atmpvc addr;
struct atm_qos qos;
int s;

if ((s = socket(PF_ATMPVC, SOCK_DGRAM, ATM_AAL5)) < 0) {
    perror("socket");
    exit(1);
}

memset(&qos, 0, sizeof(qos));
qos.txtp.class = ATM_UBR;
qos.txtp.min_pcr = ATM_MAX_PCR;
qos.txtp.max_sdu = 8192;
qos.rxtp = qos.txtp;

if (setsockopt(s, SOL_ATM, SO_ATMQOS, &qos, sizeof(qos)) < 0) {
    perror("setsockopt(SO_ATMQOS)");
    exit(1);
}

memset(&addr, 0, sizeof(addr));
addr.sap_family = AF_ATMPVC;
```

Figura C.16: Trecho de código que ilustra o uso do descritor de QoS.

C.8 Criação da conexão

Para PVCs a criação da conexão consiste simplesmente em adicionar informação de endereço ao descritor da conexão e chamar `connect` ou `bind`. Para SVCs o conceito dos *sockets* INET envolvendo `bind`, `connect`, `listen` e `accept` é usado.

C.8.1 Chamadas de sistema para criação de um PVC

As chamadas de sistema `connect` e `bind` são usadas para criar conexões. O comportamento de ambas é idêntico para PVCs. A seguinte peculiaridade deve ser observada: uma conexão a um endereço não especificado completamente não está pronta para transportar dados depois de chamar `connect` pela primeira vez; ela deve ser conectada ou amarrada uma segunda vez sem nenhum componente não especificado. Nesta segunda chamada, só componentes de endereço previamente não especificados são usados. É um erro se ainda houver componentes não especificados. A primeira chamada a `connect` pode reservar alguns ou todos os recursos que foram explicitamente especificados. Realizar um `connect` em duas fases como descrito acima é chamado de *two-phase connect*.

Os valores seguintes de `errno` têm um significado especial ao usar `connect` ou

bind em *sockets* PVC:

- EADDRNOTAVAIL o endereço requisitado não pode ser atribuído à interface existente na configuração presente.
- EOPNOTSUPP na segunda chamada partes do endereço já especificado foram alteradas ou ainda há partes não especificadas.
- ENETDOWN a interface especificada existe mas não está operacional no momento.
- ENETUNREACH o critério de QoS requisitado não pôde ser alcançado.

C.8.2 Chamadas de sistema para criação de um SVC ativo

O endereço de quem chama (o endereço local) pode opcionalmente ser setado com a chamada de sistema `bind`. Parâmetros de tráfego e informações de camadas alta e baixa especificadas na chamada a `bind` são ignorados. O endereço deve corresponder a um dos endereços que o sistema reconhece como sendo local. Se o *socket* não é explicitamente amarrado a um endereço local um endereço arbitrário será passado nas mensagens de sinalização.

A conexão é criada com a chamada de sistema `connect`. Um `connect` não bloqueante (se o *socket* foi previamente setado como não bloqueante) retorna imediatamente com `errno` setado com alguma condição de erro ou com `EINPROGRESS`, indicando um início de criação de conexão bem sucedido.

O estado de uma conexão não bloqueante pode ser solicitado através de `select` ou chamando `connect` novamente. No último caso um valor negativo é retornado e `errno` é setado para `EALREADY` se a criação da conexão ainda estiver em andamento. Se a criação teve êxito ou falhou o resultado é indicado como um `connect` bloqueante.

C.8.3 Chamadas de sistema para criação de um SVC passivo

Um *passive open* é realizado em três passos. Primeiro o SAP local é definido usando `bind`. Parâmetros de tráfego e informação de SAP serão usados posteriormente para seleccionar chamadas entrantes. Se um endereço local é especificado, apenas chamadas com exatamente esta destinação serão atribuídas ao *socket*.

Após o `bind`, o SAP deve ser registrado usando `listen`. O `listen` pode retornar um erro se o SAP conflitar com outro SAP já registrado².

²Diferentemente dos *sockets* INET, *sockets* ATM SVC só ocupam o espaço de "endereços" SAP durante o `listen`. Por isso não é possível detectar conflitos com a chamada de sistema `bind`. `bind` só verifica a validade geral do endereço.

Depois que uma conexão entrante é atribuída a um *socket* ela pode ser aceita com a chamada de sistema `accept`.

C.8.4 Exemplo

A Figura C.17 contém um trecho de código que ilustra como pode ser realizado o `connect` em duas fases.

```
addr.sap_addr.itf = 0;
addr.sap_addr.vpi = ATM_VPI_UNSPEC;

if (connect(s, (struct sockaddr *) &addr, sizeof(addr)) < 0) {
    perror("connect(1)");
    exit(1);
}

/* algumas outras atividades */
addr.sap_addr.vpi = 0;
addr.sap_addr.vci = 42;

if (connect(s, (struct sockaddr *) &addr, sizeof(addr)) < 0) {
    perror("connect(2)");
    exit(1);
}
```

Figura C.17: Trecho de código que ilustra o *two-phase connect*.

C.9 Estado da conexão

Os endereços dos participantes comunicando através de um *socket* podem ser solicitados usando as chamadas de sistema `getsockname` e `getpeername`.

O endereço local pode ser obtido com `getsockname` depois de um `bind` bem sucedido, ou (para SVCs), depois de um `accept` ou `connect` bem sucedido. O endereço remoto pode ser obtido com `getpeername` depois do `bind` (ou `connect`) para PVCs, e depois de `accept` ou `connect` para SVCs.

Os parâmetros de tráfego de uma conexão podem ser obtidos por meio de `getsockopt` com a opção `SO_ATMQOS` (veja seção C.7.4).

`getsockname`, `getpeername` e `getsockopt` podem retornar informações parciais ou inválidas se usados prematuramente.

C.10 Término da conexão

Dois passos distinguem-se ao acabar com uma conexão:

- desligamento da conexão
- fechamento da conexão

O *desligamento da conexão* pára a transmissão em uma ou ambas as direções. Recursos associados a uma conexão são desalocados com o fechamento da conexão. O desligamento de uma conexão é realizado implicitamente ao fechar uma conexão.

C.10.1 Desligamento de uma conexão

Quando uma conexão é desligada para envio, dados não são mais aceitos para envio. Dados ainda podem ser recebidos a menos que a conexão também tenha sido desligada para recepção.

Quando uma conexão é desligada para recepção, dados não são mais aceitos da rede. Dados ainda podem ser enviados, a menos que a conexão também tenha sido desligada para envio.

A chamada de sistema `shutdown` é usada para desligar conexões.

A liberação de recursos alocados à conexão devido ao `shutdown` é dependente da implementação.

C.10.2 Fechamento

Quando uma conexão é fechada, nenhuma forma de acesso é possível e todos os recursos associados a ela são liberados.

Conexões são fechadas através do fechamento de todos os *sockets* que as referenciam através da chamada de sistema `close` ou terminando os processos que são donos destes *sockets*.

Um *socket* pode ser fechado em qualquer estado, exceto se ele ainda não foi criado ou se já foi fechado anteriormente.

C.11 Resumo do controle de conexão

Todos os passos do gerenciamento de uma conexão estão resumidos abaixo, com as principais chamadas de sistema e rotinas de biblioteca usadas em cada passo:

- preparação da conexão

- criação do *socket* (`socket`)
- inicialização do descritor de conexão (`memset`)
- inicialização do descritor de QoS (`memset`)
- especificação dos parâmetros de tráfego (atribuições)
- atribuição dos parâmetros de tráfego à conexão (`setsockopt`)
- inicialização do descritor de endereço (`memset`)
- criação da conexão
 - PVC
 - * endereçamento (atribuições)
 - * opcional: criação parcial (`bind`, `connect`)
 - * preenchimento final do endereço (atribuições)
 - * criação completa (`bind`, `connect`)
 - * opcional: verificação dos parâmetros de tráfego (`getsockopt`)
 - SVC, *active open*
 - * opcional: amarramento do lado local (endereçamento e `bind`)
 - * endereçamento do lado remoto (atribuições)
 - * criação da conexão (`connect`)
 - * opcional: usar `select` para fazer *polling* de `connect` não bloqueante
 - * opcional: verificação dos parâmetros de tráfego (`getsockopt`)
 - SVC, *passive open*
 - * endereçamento do lado local (atribuições)
 - * amarramento do lado local (`bind`)
 - * registro do SAP (`listen`)
 - * opcional: fazer *polling* com `select`
 - * aceite de conexão (`accept`)
- troca de dados
 - escalonamento de transferências (`select`)
 - envio e recepção (`read`, `write`, ...)
- Término da conexão
 - desligamento da conexão (`shutdown`)
 - fechamento (`close`)

A tecnologia [IEEE 1394](#), também conhecida por Firewire pela Apple [16] e por i.Link pela Sony [22], é um padrão de interface de comunicação serial para computadores e transmissão de vídeo digital que oferece transmissão bidirecional isócrona em tempo real em altas velocidades. A Apple já o vem utilizando há anos na maioria dos seus produtos como Power Macs, iMacs, eMacs, PowerBooks, iBooks e iPods enquanto que a Sony já o adotou como a tecnologia a ser utilizada para transmissão de vídeo em filmadoras digitais.

O Firewire foi desenvolvido inicialmente pela Apple no fim de 1980 e definido como o padrão IEEE 1394 pelo IEEE em 1995 como sendo uma interface digital para produtos de áudio e vídeo, computadores pessoais e dispositivos periféricos, capaz de prover atingir até 400 Mbps.

Uma atualização do padrão, formalmente conhecida por IEEE 1394b e popularmente chamada de Firewire 800, foi definida em 2002. Como o nome indica, dispositivos Firewire 800 suportam taxas de transmissão máximas de até 800 Mbps.

Firewire pode ser utilizado em aplicações que possuem necessidades especiais, pois é capaz de oferecer:

- garantias de largura de banda (modo isócrono), que é extremamente desejável para aplicações que fazem *streaming* de vídeo em tempo real ou máquinas de inspeção visual de alta definição;
- garantias de entrega (modo assíncrono), importante para aplicações que não são sensíveis a atrasos, mas que não toleram perdas de dados.

O padrão é flexível e pode ser usado em topologias do tipo árvore ou estrela, conectando diversos dispositivos em uma rede não supervisionada. Até 63 dispositivos podem compartilhar o mesmo barramento, em uma estrutura acíclica (diferentemente da estrutura linear do [SCSI](#)). Uma única rede pode acomodar até 1024 barramentos e 64512 dispositivos.

O gerenciamento do barramento é feito de forma automática pelo próprio Firewire, incluindo o gerenciamento de endereços e a conexão/desconexão dos dispositivos sem perda de dados. Não há necessidade de reinicialização de todo o sistema quando um dispositivo é adicionado ou removido. A comunicação na rede é do tipo ponto-a-ponto. Não há a necessidade de um computador, apenas um gerenciador de rede.

Existem cabos Firewire com dois tipos de conectores: um de 4 pinos e outro com 6 pinos. O modelo de 6 pinos pode ser usado para fornecer energia elétrica aos periféricos conectados, provendo até 30 V (em contraste aos 5 V máximos que o [USB](#) pode oferecer), permitindo que até mesmo dispositivos de armazenamento externo possam ser construídos sem a necessidade de fontes externas adicionais. Já os conectores de 4 pinos não fornecem energia.

APÊNDICE E USB

O *Universal Serial Bus* (USB) provê um padrão de comunicação serial para conectar dispositivos, geralmente a um computador, mas podendo ser também conectado a outros dispositivos como *set-top boxes*, *video games* e *Personal Digital Assistants* (PDAs).

Ele tem um funcionamento assimétrico, consistindo de um controlador e múltiplos dispositivos conectados em topologia de árvore usando *hubs*. Há um limite de cinco níveis para a árvore formada pelos *hubs* e no máximo 127 dispositivos podem ser conectados a um único controlador, incluindo o *hub*.

Uma das idéias iniciais do USB era eliminar a necessidade de adição de placas de expansão nos barramentos ISA e PCI dos computadores e melhorar as capacidades *plug-and-play* permitindo adição e remoção de dispositivos sem necessidade de reinicialização do computador. Ele tem sido usado para conectar periféricos como *mouse*, teclados, *scanners*, câmeras digitais, impressoras, discos rígidos e componentes de rede.

O USB foi padronizado pelo *USB Implementers Forum* (USBIF), um órgão de padronização formado por diversas companhias da indústria eletrônica e de computadores. O padrão 2.0 foi estabelecido no fim de 2001. Padrões anteriores foram 0.9, 1.0 e 1.1, sendo que cada padrão é compatível com o anterior.

Três velocidades são suportadas pelo USB:

Low Speed (USB 1.1): taxa de 1.5 Mbps que é mais comumente utilizado por dispositivos de entrada como teclados, mouses, *joysticks* e às vezes impressoras e *scanners*;

Full Speed (USB 1.1): taxa de 10 Mbps. Foi a maior velocidade até a especificação USB 2.0. Dispositivos *Full Speed* dividem a largura de banda disponível e não é incomum que vários dispositivos isócronos acabem ficando sem banda.

High Speed (USB 2.0): taxa de 480 Mbps. Nem todos os dispositivos USB 2.0 são

Hi-Speed, sendo que o USBIF certifica dispositivos e provê logos apropriados a cada velocidade.

O conector USB provê 5 V através de um dos seus fios através do qual dispositivos USB conectados podem ser alimentados. O barramento pode fornecer até 500 *mA* a serem compartilhados entre todos os dispositivos.

O USB tem sido usado para conectar dispositivos de armazenamento, com algumas vantagens sobre o [SCSI](#), como:

- custo mais baixo, especialmente devido ao cabo de interconexão que possui menos fios;
- possibilidade de adicionar ou remover discos sem a necessidade de desligar e/ou abrir o computador;

Os dispositivos USB conectados ao mesmo barramento são controlados por um único *host*. No USB 1.1 havia duas especificações de *host controllers*:

- *Universal Host Controller Interface* ([UHCI](#)) desenvolvido pela Intel, coloca maior responsabilidade no *software* barateando assim o *hardware*;
- *Open Host Controller Interface* ([OHCI](#)) desenvolvido por Compaq, Microsoft e National Semiconductor, coloca maior responsabilidade no *hardware* com objetivo de simplificar o *software*.

APÊNDICE

F

Mecanismo CSMA/CD

O mecanismo chamado de *Carrier Sense Multiple Access with Collision Detection* (CSMA/CD), internacionalmente padronizado pelas especificações [IEEE 802.3 \[1\]](#) e [ISO 8802.3](#), foi inicialmente utilizado nas redes *Ethernet* baseadas em barramento único e é usado até hoje nos dispositivos *Ethernet* conectados em modo *half-duplex*.

Para explicar mais claramente o funcionamento do mecanismo CSMA/CD vamos considerar um cabo coaxial constituindo um único segmento *Ethernet*. Nesta topologia de rede todas as máquinas são conectadas diretamente ao cabo usado para a transmissão dos dados e todas elas são capazes de detectar quando um datagrama é transmitido.

Nada impede que duas máquinas comecem a transmitir simultaneamente, corrompendo os dados de ambas as fontes. Para reduzir a probabilidade disso acontecer, antes de transmitir um datagrama a máquina transmissora verifica se um datagrama está sendo transmitido naquele momento. Se um sinal é detectado a máquina posterga sua transmissão pelo tempo equivalente à transmissão de 96 bits ($9.6\mu\text{s}$ para *Ethernet* 10 Mbps, 960ns para *Fast Ethernet*, etc.) após o término da transmissão em andamento. Só então a máquina tenta transmitir o seu datagrama. Mesmo assim ainda existe a possibilidade de duas máquinas detectarem que o meio de transmissão não está sendo usado e iniciarem suas transmissões simultaneamente. Neste caso diz-se que ocorreu uma colisão.

Uma máquina *Ethernet*, ao mesmo tempo que transmite também monitora o sinal presente no cabo. Se o sinal transmitido não for igual ao sinal monitorado assume-se que houve uma colisão. Para garantir que a outra máquina envolvida também esteja ciente da colisão, a primeira reforça a colisão através de um sinal de embaralhamento e aborta a transmissão. No caso de colisão ambas as máquinas devem esperar por um intervalo de tempo, chamado de "tempo de *backoff*" [\[9\]](#), antes de tentar começar pela etapa inicial de detecção do sinal no cabo.

O "tempo de *backoff*" é um múltiplo do "tempo de um *slot*", onde "*slot*" é o datagrama *Ethernet* mínimo de 64 bytes (512 bits) e o "tempo de um *slot*" é o tempo necessário para transmitir 64 bytes, ou $51,2\mu\text{s}$ para o *Ethernet* tradicional de 10 Mbps.

O múltiplo usado na determinação do intervalo de *backoff* é calculado como um número randômico r , onde $0 \leq r < 2^k$, onde $k = \text{MIN}(n, 10)$ e onde n é o contador de colisões. Deste modo, se o transmissor detectar uma colisão pela primeira vez ele vai usar um tempo de *backoff* entre 0 e 1 tempos de *slot* (0 a 51, $2\mu\text{s}$). Se isto resultar em uma segunda colisão o tempo de espera será entre 0 e 3 tempos de *slot* (0 a 153, $6\mu\text{s}$) e assim sucessivamente até a 10^{a} colisão, que resultará em um tempo aleatório entre 0 e 1024 tempos de *slot*. Após a 16^{a} tentativa frustrada o datagrama é descartado e o [MAC](#) informa o erro. Nos piores casos a latência pode ser de centenas de milissegundos.

Condicionador de tráfego

O gerenciamento de tráfego em uma rede que usa Serviços Diferenciados requer que seja estabelecido um acordo (**TCA**, do inglês *Traffic Conditioning Agreement*) entre o usuário e o provedor do serviço de acesso. É função do TCA:

- descrever as características do tráfego a ser enviado e recebido da **WAN**;
- definir as regras de policiamento para o volume e o tipo de tráfego a ser transmitido e recebido da WAN;
- definir a sequência de regras a serem seguidas para aderir ao TCA.

Um mecanismo de condicionamento de tráfego como o ilustrado na Figura **G.1** é geralmente empregado na periferia da rede para certificar que o tráfego segue as regras estabelecidas pelo TCA.

Um classificador lê o **DSCP** (do inglês *Differentiated Services Code Point*) ou uma combinação formada pelo DSCP e outros campos (endereço IP de origem, endereço IP de destino, etc.) e seleciona e roteia pacotes para um condicionador de tráfego. O condicionador de tráfego pode realizar as seguintes ações:

Figura G.1: Mecanismo de condicionamento de tráfego usado para certificar que o tráfego segue as regras estabelecidas pelo **TCA**.

- Monitorar a taxa instantânea de transmissão de cada fluxo (medidor) para verificar se ele está dentro do perfil requerido e remarcar, descartar ou suavizar se ele estiver fora do perfil;
- Mudar o DSCP, se necessário, para mudar o comportamento do reencaminhamento;
- Mudar o gerenciamento das filas (suavizador)
- Descartar o pacote (descartador)

Como um exemplo, um classificador pode selecionar um fluxo de pacotes indicando serviço que é classificado como "ouro" e rotear um pacote deste fluxo para o condicionador de tráfego. Se o tráfego "ouro" estiver dentro das características do perfil de tráfego o pacote pode ser agregado a outros pacotes em uma rota de alta prioridade e ser enviado imediatamente ao próximo *hop*. Entretanto, se o fluxo estiver fora do perfil, o DSCP do pacote pode ser mudado de forma que ele seja classificado como pertencente a uma categoria "prata", com características inferiores à categoria "ouro", ou o pacote pode ser colocado na fila novamente ou ser descartado, de acordo com o que foi estabelecido nas regras do TCA. Outra alternativa, se o TCA permitisse, seria remarcar os pacotes de classes inferiores ou eventualmente, descartá-los.

Nem sempre o condicionador de tráfego implementa todas estas funções. Geralmente são implementadas apenas as funções necessárias às regras de policiamento do TCA.