Universidade Estadual de Campinas

Faculdade de Engenharia Elétrica e Computação

Departamento de Comunicações

SIMULADOR DE REDES ORIENTADO A EVENTOS PARA O SISTEMA OPERACIONAL LINUX

Dissertação apresentada a Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas, como parte dos requisitos exigidos para a obtenção do título de Mestre em Engenharia Elétrica.

AUTOR

Ricardo Luís Külzer

Graduado em Engenharia Elétrica pela Universidade Federal de Santa Maria, em 1999

Graduado em Ciência da Computação pela Universidade Federal de Santa Maria em 2001

ORIENTADOR

Leonardo de Souza Mendes

PhD em Engenharia Elétrica pela Syracuse University em 1992

BANCA EXAMINADORA:

Prof. Dr. Leonardo de Souza Mendes - DECOM/FEEC/UNICAMP
Prof. Dr. João Carlos Furtado – Departamento de Informática/UNISC
Prof. Dr. Mauricio Ferreira Magalhães - DCA/FEEC/UNICAMP
Prof. Dr. Luís Geraldo Pedroso Meloni - DECOM/FEEC/UNICAMP

Campinas, 6 de agosto de 2003

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP

K96s

Külzer, Ricardo Luís
Simulador de redes orientado a eventos para o sistema operacional Linux / Ricardo Luís Külzer.-- Campinas, SP: [s.n.], 2003.

Orientador: Leonardo de Souza Mendes
Dissertação (mestrado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Redes de computação. 2. Telecomunicações. 3. Linux (Sistema operacional de computador). 4. Simulação (Computadores). 5. TCP/IP (Protocolo de rede de computação). I. Mendes, Leonardo de Souza. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

RESUMO

Neste trabalho foi desenvolvido o simulador ***lsimnt*** para o sistema operacional Linux. Este simulador usa a técnica de orientação a eventos, e poderá ser utilizado para simular diferentes tipos de redes de comunicações, devido a sua característica modular. O núcleo do simulador, responsável pelo controle e execução da simulação, é um aplicativo separado dos modelos, os quais são implementados como bibliotecas dinâmicas. Sendo assim, o simulador consiste em uma ferramenta útil para uso em ensino, pesquisa, projeto e análise de redes de comunicações.

Como exemplo de aplicação, este trabalho também aborda a implementação de modelos de rede com a implementação do protocolo TCP/IP, e a simulação de um cenário de rede TCP/IP.

ABSTRACT

In this work was developed the simulator ***lsimnt*** for Linux operating system. This simulator uses the event-driven technique, and can be used to simulate different types of communication networks, due to its modular characteristic: the simulator kernel, responsible for control and execution of the simulation, it is an application separate from the models, which are implemented as dynamic libraries. So, the simulator consists of a useful tool for use in teaching, research, project and analysis of communication networks.

As an application example, this work also approaches the implementation of networks models with implementation of the TCP/IP protocol, and simulation of a TCP/IP network scenery.

A meus pais Ewaldo Carlos. e Joana Nelsinda.

Aos meus irmãos Ana Maria, José Carlos, Helena Maria e Rosa Maria.

A minha esposa Gláucia Giovana.

ÓRGÃO FINANCIADOR

Conselho Nacional de Desenvolvimento Científico e Tecnológico

AGRADECIMENTOS

Aos meus pais que me deram a luz da vida, e sempre estiveram comigo. Quero registrar aqui o quanto eu os amo. A minha gratidão para com eles será eterna.

Aos meus irmãos que tanto me ajudaram, e sempre serviram como fonte de apoio e inspiração.

A minha amada esposa, pelo amor, companheirismo, carinho, compreensão e incentivo.

Ao Prof. Leonardo de S. Mendes pela oportunidade, pelo apoio e sinceridade dada durante toda a orientação deste trabalho.

A todos os colegas do laboratório LARCOM/DECOM pela troca de idéias, pelo companheirismo e agradável convivência. Em especial, aos colegas Antônio, Mauricio, Gean, Davison, Meire, Henrique, Ana, e muitos outros.

Ao Prof. Nelson J. Schuch, meu orientador de iniciação científica durante a graduação, pelas suas valiosas lições de vida, e pelo incentivo a fazer o Mestrado.

Aos muitos amigos e colegas do laboratório LACESM/UFSM, e também do CRSPE/INPE, que estiveram comigo durante praticamente toda a minha graduação na UFSM.

Em especial, aos amigos Vincius F. Montagner, César A. Dartora, e Gláucia Heckler, pela amizade e ótima convivência.

Ao CNPq, pelo suporte financeiro.

Enfim, a todos os meus familiares e amigos que contribuíram para que eu chegasse até aqui, agradeço de coração.

"Muitas pessoas são ilógicas, desconcertantes.

Assim mesmo, dê a elas um voto de confiança.

Praticando o bem, haverá quem acuse você
de interesses ocultos, de promoção pessoal.

Assim mesmo, plante bondade, alegria.

Quem triunfa granjeia falsos
amigos e o despeito dos outros.

Assim mesmo, continue lutando.

A honestidade e a franqueza
nos tornam vulneráveis.

Assim mesmo, seja honesto, transparente.

Um trabalho de longos anos pode ser derrubado,
num instante, pelas mãos da inveja, da ingratidão.

*Assim mesmo, continue construindo, dando
o melhor de si mesmo, com desprendimento
e generosidade. Sorrindo. Amando. Servindo.*

É necessário que alguns batalhem, sofram,
construam e acreditem, sem desertar
da luta, para que muitos sejam felizes.

Ser bom, generoso e leal ainda é
o melhor caminho, a melhor solução."

(Pe. Roque Schneider, SJ.)

GLOSSÁRIO DAS ABREVIATURAS UTILIZADAS

AF	Assured Forwarding
ATM	Asynchronous Transfer Mode
BA	Behaviour Agregate
BE	Best-Effort
CAD	Computer Aided Design
CAR	Committed Access Rate
CBQ	Class-based Queuing
CBR	Constant Bit Rate
Diffserv	Differentiated Services
DLL	Dynamic Link Library
DS	Differentiated Services
DSCP	Differentiated Services Code Point
EF	Expedited Forwarding
FIFO	First In First Out
IETF	Internet Engineering Task Force
Intserv	Integrated Services
IP	Internet Protocol
LAN	Local Area Network
MF	Multi-Field
MSS	Maximum Segment Size
OSI	Open Systems Integration
PHB	Per-Hop Behaviour
PPP	Point to Point Protocol
PQ	Priority Queuing
QoS	Quality of Service
RED	Random Early Discard
RSVP	Resource ReSerVation Protocol
RTT	Round Trip Time
SCFQ	Self Clocked Fair Queuing

SLA	Service Level Agreement
TCP	Transmission Control Protocol
ToS	Type of Service
UDP	User Datagram Protocol
VoIP	Voice over IP
WAN	Wide Area Network
WFQ	Weighted Fair Queuing
WRR	Weighted Round-Robin

SUMÁRIO

CAPÍTULO 1. INTRODUÇÃO 1

1.1 - CONSIDERAÇÕES INICIAIS 1

1.1.1 - O SIMULADOR LSIMNT 2

1.1.2 - O PROTOCOLO TCP/IP 5

1.2 - ORGANIZAÇÃO DO TRABALHO 7

CAPÍTULO 2. SIMULAÇÃO DE SISTEMAS DE COMUNICAÇÃO 9

2.1 - JUSTIFICATIVA DA SIMULAÇÃO 9

2.2 - TÉCNICAS DE SIMULAÇÃO 10

2.2.1 - SIMULAÇÃO ORIENTADA PELO TEMPO (TIME-DRIVEN SIMULATION) 10

2.2.2 - SIMULAÇÃO ORIENTADA POR EVENTOS (EVENT-DRIVEN SIMULATION) 10

2.2.3 - SIMULAÇÃO ORIENTADA POR DADOS (DATA-DRIVEN SIMULATION) 11

2.3 - SOFTWARES DE SIMULAÇÃO PARA REDES DE COMUNICAÇÃO 12

2.3.1 - LINGUAGENS GERAIS DE SIMULAÇÃO 12

2.3.2 - LINGUAGENS DE SIMULAÇÃO ORIENTADAS ÀS REDES DE COMUNICAÇÕES 13

2.3.3 - SIMULADORES ORIENTADOS ÀS REDES DE COMUNICAÇÕES 13

CAPÍTULO 3. PROTOCOLOS TCP/IP 15

3.1 - O PROTOCOLO IP (INTERNET PROTOCOL) 15

3.2 - O PROTOCOLO TCP (TRANSMISSION CONTROL PROTOCOL) 15

3.2.1 - FUNCIONAMENTO 16

3.2.2 - CONFIABILIDADE 19

3.2.3 - CONTROLES 20

3.2.4 - CONTROLE DE SESSÃO 21

3.2.5 - CONTROLE DE ERRO 23

3.2.6 - CONTROLE DE FLUXO 26

3.2.7 - CONTROLE DE CONGESTIONAMENTO 28

CAPÍTULO 4. QUALIDADE DE SERVIÇO NAS REDES TCP/IP 31

4.1 - QUALIDADE DE SERVIÇO NAS REDES TCP/IP 31

4.2 - ARQUITETURA DE SERVIÇOS DIFERENCIADOS 32

4.2.1 - COMPONENTES ESSENCIAIS 33

4.2.2 - COMPORTAMENTO POR HOP 35

4.3 - DISCIPLINAS DE ESCALONAMENTO E POLICIAMENTO DE TRÁFEGO 37

4.3.1 - MECANISMOS DE GERENCIAMENTO DE ROTEADORES 37

4.3.2 - MECANISMOS DE CONTROLE DE CONGESTIONAMENTO ____ *40*
4.3.3 - O ALGORITMO WRED ____ *44*
4.4 - QUALIDADE DE SERVIÇO NO PROCESSAMENTO DIGITAL DE SINAIS PARA TELEFONIA IP ____ 45
4.5 - CLASSIFICAÇÃO DOS PADRÕES DE CODIFICAÇÃO DE VOZ ____ 46
4.6 - AVALIAÇÃO DA QUALIDADE DOS ALGORITMOS DE COMPRESSÃO DE VOZ ____ 48
4.7 - QUALIDADE DO SINAL DE VOZ ____ 49

CAPÍTULO 5. DESENVOLVIMENTO DO SIMULADOR *LSIMNT* ____ 51

5.1 - ESTRUTURA PROPOSTA PARA O SIMULADOR DE REDES ____ 51
5.2 - O NÚCLEO DO SIMULADOR ____ 52
5.3 - EXECUÇÃO DE SIMULAÇÕES ____ 53
5.4 - O ESCALONAMENTO DE EVENTOS ____ 54
5.5 - CLASSES DISPONÍVEIS PARA CRIAÇÃO DE MODELOS ____ 55
5.6 - PROCESSO DA SIMULAÇÃO ____ 56
5.6.1 - ELEMENTOS DISPONÍVEIS PARA PROGRAMAÇÃO DE MODELOS ____ *59*
5.6.2 - CONEXÃO ENTRE MODELOS ____ *60*
5.7 - ESTRUTURAS DE DADOS AUXILIARES ____ 62
5.7.1 - MODELOS ____ *63*

CAPÍTULO 6. EXEMPLO DE APLICAÇÃO ____ 65

6.1 - CENÁRIO DE SIMULAÇÃO: REDE *DIFFSERV* ____ 65
6.2 - MODELOS DESENVOLVIDOS PARA SIMULAR O CENÁRIO ____ 67
6.3 - MODELO DE APLICAÇÃO ____ 67
6.4 - MODELO DE TERMINAL ____ 68
6.4.1 - MODELO DA CAMADA DE TRANSPORTE TCP ____ *70*
6.4.2 - MODELO DA CAMADA DE REDE (IP) DO TERMINAL ____ *74*
6.4.3 - MODELO DA CAMADA DE ENLACE DO TERMINAL ____ *75*
6.5 - MODELO DE ROTEADOR ____ 75
6.6 - INTERAÇÃO ENTRE OS BLOCOS ____ 78
6.7 - SIMULAÇÃO ____ 79
6.8 - ANÁLISE DOS RESULTADOS ____ 81
6.8.1 - VAZÃO OBTIDA COMPARADA À LARGURA DE BANDA RESERVADA ____ *83*
6.8.2 - VISÃO GERAL SOBRE O COMPORTAMENTO DA REDE ____ *85*
6.9 - DESEMPENHO DO SIMULADOR ____ 86

CAPÍTULO 7. COMPARATIVO DO DESEMPENHO DOS NÚCLEOS DOS SIMULADORES *LSIMNT* E *HYDRAGYRUM* ____ 89

7.1 - MODELO DESENVOLVIDO PARA COMPARAÇÃO DOS SIMULADORES 90

CAPÍTULO 8. CONCLUSÕES 93

8.1 - TRABALHOS FUTUROS 94

CAPÍTULO 9. REFERÊNCIAS BIBILIOGRÁFICAS 95

APÊNDICE A. A LINGUAGEM SCL 99

A.1 - COMANDOS DO AMBIENTE DE SIMULAÇÃO 99

A.1.1 - Comando: block 99

A.1.2 - Comando: connect block 99

A.1.3 - Comando: param global 99

A.1.4 - Comando: param block 99

A.1.5 - Comando: param layer 99

A.1.6 - Comando: param queue 100

A.2 - CARGA E EXECUÇÃO DE ARQUIVO DE SIMULAÇÃO 100

APÊNDICE B. API E RECURSOS DE PROGRAMAÇÃO 101

B.1 - INTERFACE DOS MODELOS 101

B.2 - FUNÇÕES DAS CLASSES BASE (BLOCK, LAYER E SQUEUE) 101

B.2.1 - FUNÇÕES PARA MANIPULAÇÃO DE IDENTIFICADORES 101

B.2.2 - ADIÇÃO, REMOÇÃO E ASSOCIAÇÃO DE ELEMENTOS NO BLOCK 102

B.2.3 - FUNÇÕES DE MANIPULAÇÃO DE ELEMENTOS CONTIDOS NO BLOCK 103

B.2.4 - FUNÇÕES DE ACESSO A ELEMENTOS DO KERNEL 104

B.2.5 - FUNÇÕES DE GERAÇÃO DE EVENTOS 105

B.2.6 - FUNÇÕES PARA A MANIPULAÇÃO DE PARÂMETROS 107

B.2.7 - AS FUNÇÕES DA CLASSE DE EVENTOS 113

B.2.8 - A CLASSE CONNECTION 113

B.2.9 - A CLASSE TABLE 116

APÊNDICE C. DETALHES DA IMPLEMENTAÇÃO DO SIMULADOR 119

C.1 - O NÚCLEO DO SIMULADOR 119

C.2 - AS CLASSES DO SIMULADOR 121

C.2.1 - A Classe LW_Kernel 121

C.2.2 - A Classe LW_Command 122

C.2.3 - A Classe EventManager 123

C.2.4 - A Classe Block 125

C.2.5 - A Classe Layer 126

C.2.6 - A Classe Squeue 127

C.2.7 - A Classe Connection _____ *128*
C.2.8 - A Classe Event _____ *129*
C.2.9 - A Classe SMessage _____ *130*
C.3 - RELACIONAMENTO ENTRE AS CLASSES _____ 131

ÍNDICE DAS FIGURAS

Figura 1 - Campos dos pacotes TCP 17
Figura 2 - Confirmação positiva 19
Figura 3 - Confirmações com o uso da janela deslizante 20
Figura 4 - Handshake de três vias 22
Figura 5 - Encerramento de uma conexão 22
Figura 6 - ACKs duplicados 25
Figura 7 - Janela deslizante com nenhum segmento confirmado 27
Figura 8 - Janela deslizante com um segmento confirmado 28
Figura 9 - Janela de congestionamento 30
Figura 10 - Implementação do algoritmo RED 43
Figura 11 - Operação do algoritmo WRED 44
Figura 12 - Qualidade dos padrões de compressão 49
Figura 13 - Intervalos para o atraso e os seus respectivos graus de satisfação 50
Figura 14 - Estrutura dos eventos do *lsimnt* 54
Figura 15 - Execução de eventos dentro do ambiente de simulação. 55
Figura 16 - Relacionamento entre objetos do tipo *Block*, *Layer* e *Squeue* 56
Figura 17 - Etapas da simulação 58
Figura 18 - Conexão entre blocos 61
Figura 19 - Cenário de simulação 65
Figura 20 - Classes para o modelo de aplicação 68
Figura 21 - Classes que integram o modelo de terminal 69
Figura 22 - Listagem parcial do método *Run()* 73
Figura 23 - Classes que integram o modelo de roteador 78
Figura 24 - Interação entre os blocos no sentido servidor-cliente (fluxo de dados) 78
Figura 25 - Interação entre os blocos no sentido cliente-servidor (fluxo de ACKs) 79
Figura 26 - Vazão por fluxo de tráfego com pacotes de 128 bytes 82
Figura 27 - Vazão por fluxo de tráfego com pacotes de 256 bytes 82
Figura 28 - Vazão por fluxo de tráfego com pacotes de 512 bytes 83
Figura 29 - Vazão medida vs. largura de banda definida com pacotes de 128 bytes 84

Figura 30 - Vazão medida vs. largura de banda definida com pacotes de 256 bytes ___ 84
Figura 31 - Vazão medida vs. largura de banda definida com pacotes de 256 bytes ___ 85
Figura 32 - Método *Run* da classe Simple_Block ___ 90
Figura 33 - Classe LW_Kernel ___ 121
Figura 34 - Fluxograma básico do processamento ___ 122
Figura 35 - A classe LW_Command ___ 123
Figura 36 - A classe EventManager ___ 123
Figura 37 - Método para agendamento de eventos ___ 124
Figura 38 - Método para execução de eventos ___ 124
Figura 39 - A classe Block ___ 125
Figura 40 - Relacionamento entre objetos do tipo *Block*, *Layer* e *Squeue* ___ 126
Figura 41 - A classe *Layer* ___ 127
Figura 42 - A classe *Squeue* ___ 128
Figura 43 - A classe *Connection* ___ 129
Figura 44 - A classe *Event* ___ 129
Figura 45 - A classe *SMessage* ___ 130
Figura 46 – Extensão da classe *SMessage* para a criação de tipos de mensagens ___ 131
Figura 47 - Relacionamento entre as classes ___ 131

ÍNDICE DAS TABELAS

Tabela 1 - Codepoints para o grupo de PHBs AF 36
Tabela 2 - Comparação de Atributos de Codecs de Áudio.....47
Tabela 3 - Comparação dos MOS de Codecs de Áudio 48
Tabela 4 - Arquivos com o código do modelo de aplicação.....68
Tabela 5 - Arquivos com o código do modelo de terminal 70
Tabela 6 - Arquivos com o código do modelo de roteador 77
Tabela 7 - Reserva de recursos por classe 80
Tabela 8 - Vazão obtida com pacotes de 128 bytes.....80
Tabela 9 - Vazão obtida com pacotes de 256 bytes.....81
Tabela 10 - Vazão obtida com pacotes de 512 bytes.....81
Tabela 11 - Largura de banda agregada dos fluxos de cada classe.....86
Tabela 12 - Comparativo dos simuladores *lsimnt* e *Hydragyrum* 86
Tabela 13 - Arquivos com o código do modelo simple_block.....90
Tabela 14 – Comparativo: número de eventos processados por segundo 91
Tabela 15 - Arquivos com o código dos objetos desenvolvidos neste trabalho 119
Tabela 16 - Arquivos com os cabeçalhos dos objetos desenvolvidos neste trabalho.....120
Tabela 17 - Arquivos contendo código e cabeçalhos de estrutura auxiliares 120
Tabela 18 - Atributos da classe *Event*.....130

CAPÍTULO 1. INTRODUÇÃO

1.1 - CONSIDERAÇÕES INICIAIS

A simulação computacional de sistemas consiste na aplicação de conceitos físicos e matemáticos para criação de modelos computacionais que, processados por computadores, procuram reproduzir o funcionamento de um sistema real. Tais modelos computacionais são programados para imitar as operações e processos envolvidos no sistema. Usando tais modelos computacionais, a simulação nos permite compreender o comportamento de um sistema real e avaliar estratégias para sua operação.

O uso de simulação como ferramenta de estudo tem aumentado em diferentes áreas de conhecimento. Entre as principais causas deste crescimento estão a crescente complexidade dos problemas com os quais nos defrontamos, e a maior disponibilidade de recursos computacionais.

A etapa mais crítica no encaminhamento de um estudo de um sistema através de simulação consiste na modelagem do sistema. Esta modelagem requer um processo de criação e descrição, envolvendo um determinado grau de abstração que, na maioria das vezes, implica em uma série de simplificações sobre a organização e o funcionamento do sistema real.

São muitas as áreas passíveis de estudos baseados em simulação, como por exemplo [11]:

- Sistemas de produção e manufatura;
- Sistemas públicos;
- Sistemas de transporte;
- Sistemas de construção;
- Restaurantes e sistemas de entretenimento;
- Processos empresariais;
- Sistemas computacionais de alta performance, e muitos outros.

No desenvolvimento de modelos computacionais para estudo de conceitos envolvidos em processo tecnológicos, uma das técnicas mais empregadas consiste na

simulação orientada a eventos discretos, tipicamente caracterizada pela existência de um relógio que avança a simulação a intervalos variáveis de tempo, em função do próximo evento a ser executado. Neste caso, a simulação implica na realização de experimentos numéricos com modelos lógicos-matemáticos. Tais experimentos envolvem normalmente um grande volume de cálculos repetitivos, fazendo uso intensivo dos recursos computacionais.

Neste contexto, esta tese trata do projeto e da implementação de uma ferramenta para a simulação, chamada ***lsimnt***. Esta ferramenta, orientada a eventos, poderá ser utilizada para simular diferentes tipos de redes de comunicações, devido a sua modularidade. O núcleo do simulador, responsável pelo controle e execução da simulação, é um aplicativo separado dos modelos, os quais são implementados como bibliotecas dinâmicas.

Para exemplificar o uso do simulador, esta tese também trata da implementação de um modelo para análise do comportamento de uma rede TCP/IP, em que foram implementados o controle de erro, controle de fluxo e controle de congestionamento do protocolo TCP, e também de um modelo para a verificação e análise do desempenho do simulador.

1.1.1 - O SIMULADOR *LSIMNT*

A implementação do simulador ***lsimnt*** está baseada no desenvolvimento de outros softwares de simulação em nosso grupo de pesquisa. A primeira ferramenta de simulação desenvolvida por nosso grupo foi o ambiente de simulação de enlaces ópticos ***SimNT*** [3]. O ***SimNT*** atualmente é um ambiente de simulação de sistemas de comunicação que emprega a técnica de simulação *data-driven*. Na técnica *data-driven*, vetores de sinal são gerados e trocados entre os modelos. O processamento de vetores de sinais possui analogia direta com o processamento digital de sinais. Portanto, esta técnica é mais bem aplicada à simulação de sistemas como enlaces ópticos e enlaces de rádio, onde geralmente se pode trabalhar com amostras de sinais no tempo.

Em 1996, já com dois anos de andamento do projeto do ***SimNT*** (a partir de 1996 o ***SimNT*** tornou-se o projeto FAPESP n 0 95/4714-8), surgiu à necessidade de ampliar o

comportamento dos modelos e do ambiente de simulação de forma a incorporar outros áreas de simulação de interesse do nosso grupo de pesquisa, como a simulação de redes de alta velocidade. Para tanto, foi tomado como meta para o desenvolvimento do novo simulador a simulação de redes ATM. Assim surgiu o projeto do ***SimATM*** – Simulador de Redes ATM, desenvolvido em 1996 [2]. A primeira versão do ***SimATM*** (V0.61) foi concluída no final de 1997. Esta primeira versão operava apenas com uma interface de linha de comando e não possuía uma arquitetura modular e expansível para a definição de novos modelos como a que o ***SimATM*** possuía na época.

Em 1997, com a proposta original deste trabalho iniciamos o desenvolvimento de um ambiente de simulação de redes que seria modular e expansível, permitindo a adição de novos modelos por meio de DLLs sem recompilação do núcleo, e com uma interface gráfica de interação com o usuário. As idéias e os conceitos do ***SimNT*** foram amplamente empregados no desenvolvimento da nova versão do ***SimATM*** (v0.8), que começou em maio de 1997 e terminou em novembro de 1998, contando já com todos os elementos desejados (apesar de ter uma interface gráfica um tanto simplificada). Em 1999, a interface gráfica e o núcleo do ***SimATM*** (V0.8) foram aprimoradas utilizando-se uma série de modelos simplificados de elementos de rede ATM. Em setembro de 1999, com os testes da versão 0.8 do ***SimATM*** concluídos, resolvemos denominar o núcleo e a interface gráfica que compõe o ambiente de simulação de ***Hydragyrum*** – Ambiente de Simulação de Redes, para ressaltar o caráter genérico do ambiente como um ambiente de simulação de redes a eventos discretos (*event-driven*). Naquele estágio, a simulação de redes ATM era o enfoque do desenvolvimento de modelos para o ambiente de simulação [1].

Neste contexto, apesar da grande gama de simuladores de redes existentes, a ferramenta ***lsimnt*** desenvolvida neste trabalho apresenta o diferencial de ser o primeira implementação de um *software* para sistemas abertos (***Linux***) que permita reutilizar o legado de modelos já desenvolvidos para os simuladores ***SimNT***, ***SimATM*** e ***Hydragyrum***. Além disso, até então todos os modelos desenvolvidos para estes simuladores eram modelos para redes ópticas ou redes ATM, mas até então não existiam modelos que implementassem o protocolo TCP/IP. Desta forma, a implementação de modelos para simulação de redes TCP/IP vem a preencher uma lacuna existente há muito tempo em nosso grupo de pesquisa, e certamente será de grande utilidade em trabalhos futuros.

O software ***lsimnt*** desenvolvido neste trabalho consiste basicamente em uma ferramenta para simulação de redes de comunicações. Suas principais características são:

- Desenvolvido para o sistema operacional ***Linux***;
- Usa a técnica de simulação orientada a eventos
- Possui um interpretador de comando para leitura de arquivos de configuração de cenários de rede a serem simulados; e
- Possui um núcleo, responsável pela carga dos modelos e pelo controle e execução de todo o processo de simulação.

O ***lsimnt*** foi desenvolvido basicamente a partir do trabalho [1] em que implementou-se o simulador de redes ***Hydragyrum***, um simulador de redes orientado a eventos discretos para o sistema operacional ***Microsoft Windows***, com enfoque em redes orientadas a conexão, como por exemplo redes ATM.

O ***Hydragyrum*** vem sendo usado como ferramenta didática em disciplinas ministradas na FEEC, e também como ferramenta para criação de modelos de simulação por alunos em seus trabalhos para o mestrado da FEEC. Contudo, o ***Hydragyrum*** foi desenvolvido especificamente para o sistema operacional ***Windows***, e sua API (interface para programação) permite apenas a criação de modelos na forma de bibliotecas dinâmicas para ***Windows*** (DLLs). Isto tem sido visto pelo nosso grupo de estudos em simulação de redes como uma lacuna para a comunidade de usuários do sistema operacional ***Linux***, que atualmente se encontra em expansão. Neste contexto, este trabalho visou preencher esta lacuna, não só com a criação de um simulador compatível com a estrutura do simulador ***Hydragyrum***, mas também com uma implementação de um núcleo mais enxuto, melhor estruturado e mais eficiente que o núcleo do simulador ***Hydragyrum***.

No ***lsimnt***, os modelos de elementos de rede são implementados em bibliotecas dinâmicas para ***Linux***, de forma totalmente independente do núcleo do simulador. Esta separação do núcleo e do interpretador de comando facilita a criação de novos modelos, de forma que este simulador poderá ser empregado como:

- Ferramenta de aprendizado, análise e projeto de redes, para implementação de protocolos, de algoritmos de gerenciamento de recursos, mecanismos de controle fluxo, mecanismos de controle de congestionamento, etc;

- Ferramenta de análise, para avaliação de desempenho de protocolos, algoritmos e mecanismos utilizados em redes;
- Ferramenta de projeto de redes, para teste de redes sobre diferentes condições de carga e disposições geográficas; e
- Ferramenta de validação de novos modelos matemáticos, como por exemplo, modelos de geração de tráfego e de geração de falhas de transmissão.

1.1.2 - O PROTOCOLO TCP/IP

Como exemplo de aplicação, este trabalho também aborda a implementação de modelos para a simulação de um cenário de rede TCP/IP.

A operação do protocolo TCP (*Transmission Control Protocol*) sobre o protocolo IP (*Internet Protocol*) constitui o alicerce de funcionamento da Internet [24].

Entre as principais causas do sucesso desta família de protocolos estão:

- A ampla disponibilidade: implementações do TCP/IP estão disponíveis em praticamente todos os sistemas operacionais populares;
- Seu código fonte é amplamente disponível em várias implementações;
- Fabricantes de equipamentos de redes oferecem suporte para o TCP/IP na maior parte dos seus produtos.

A invenção do protocolo TCP/IP foi fundamental para o sucesso da Internet. Porém, a concepção do protocolo originalmente não levou em conta mecanismos para que o serviço de troca de dados seja realizado com garantias mínimas de requisitos de atrasos, variações de atraso e perdas de dados. Ao contrário, o protocolo funciona segundo a filosofia do melhor esforço: cada usuário da rede envia seus dados e compartilha a largura de banda com todos os fluxos de dados dos outros usuários. Os fluxos chegam ao destino da melhor forma possível, conforme as rotas definidas e a largura de banda que estiver disponível. No caso de ocorrência de congestionamentos, pacotes são descartados sem distinção. Não há garantia de que o serviço será realizado com sucesso, nem mesmo de desempenho.

Existem diversas aplicações que requisitam garantias de desempenho da rede. Exemplos são aplicações que geram tráfego de voz e imagem para encapsulamento e transmissão usando protocolo IP, como difusão de rádio na Internet, áudioconferências videoconferências, etc. As redes capazes de oferecer níveis de garantia e segurança às aplicações para Internet são ditas redes com qualidade de serviço, ou redes com QoS (*Quality of Service*) .

Com QoS é possível aos provedores de serviços de rede vender novos serviços de acordo com quantidade de largura de banda que se possa ou deseje contratar. Define-se assim, uma nova forma de comércio na Internet, donde já se pode vislumbrar um grande interesse em torno dessa tecnologia [33].

A implementação de QoS pode ser na rede de transporte, ou também nos próprias aplicações. Um exemplo é o controle de jitter a nível de processamento de sinais[17]. Já a nível de rede de transporte, temos dois mecanismos básicos disponíveis para implementar QoS em redes IP [38]:

- **Reserva de recursos**: os recursos da rede são divididos de acordo com os requisitos de QoS da aplicação, e sujeitos à política de administração de largura de banda. Esta estratégia requer a introdução de uma sinalização específica. Esta proposta é adotada pela Arquitetura de Serviços Integrados (***IntServ***).
- **Priorização**: o tráfego da rede é classificado e os recursos de rede são divididos de acordo com critérios de políticas de administração de largura de banda. Para habilitar QoS os mecanismos de classificação dão tratamento preferencial a aplicações identificadas como tendo requisitos mais exigentes. Esta estratégia requer a redefinição dos 8 bits do campo TOS, para que todos os roteadores dispensem um tratamento uniforme aos datagramas de acordo com este campo. Esta proposta é adotada pela Arquitetura de Serviços Diferenciados (***DiffServ***).

Neste trabalho foram implementados como exemplos de modelos para o simulador ***lsimnt***, modelos de rede que utilizam a arquitetura *DiffServ* para prover QoS.

1.2 - ORGANIZAÇÃO DO TRABALHO

Esta dissertação está dividida em capítulos e apêndices.

- No capítulo 1 é apresentada uma introdução, descrevendo a organização do trabalho e seus objetivos.
- No capítulo 2 discute-se de uma maneira geral a simulação de redes de comunicações.
- No capítulo 3 descreve-se a operação básica da família dos protocolos TCP/IP.
- No capítulo 4 são apresentadas definições e conceitos relacionados a QoS (qualidade de serviço) em redes TCP/IP. Também são abordados alguns conceitos sobre QoS para telefonia IP.
- No capítulo 5 apresenta-se uma visão geral do desenvolvimento do simulador ***lsimnt***. Descreve-se a arquitetura proposta para o simulador e o funcionamento do mesmo.
- No capítulo 6 é proposto um cenário para simulação com o ***lsimnt***. Neste item são definidos e analisados os modelos necessários para simulação deste cenário. Aqui descreve-se a implementação dos modelos de rede desenvolvidos. Também é apresentada uma simulação em que se empregam o simulador ***lsimnt*** e os modelos desenvolvidos, e são obtidos alguns resultados.
- No capítulo 7 descreve-se a implementação de um modelo construído especialmente para análise de desempenho do simulador. Mas o mais importante neste capítulo é a análise do desempenho do simulador propriamente dita, comparado ao desempenho do simulador ***Hydragyrum*** que deu origem a este trabalho.
- No capítulo 8 são apresentadas as conclusões do trabalho e sugestões para trabalhos futuros.
- No capítulo 9 são apresentadas a bibliografia e as referências bibliográficas.
- No apêndice A é descrita a linguagem de comandos do simulador.
- No apêndice B são apresentados a API (interface para programação) e os recursos de programação do ***lsimnt***.
- No apêndice C são apresentados mais detalhes internos da implementação do simulador ***lsimnt***.

CAPÍTULO 2. SIMULAÇÃO DE SISTEMAS DE COMUNICAÇÃO

2.1 - JUSTIFICATIVA DA SIMULAÇÃO

A simulação exerce um papel vital na caracterização do comportamento de uma rede de computadores, e dos possíveis efeitos que uma proposta de alteração em alguma tecnologia ou protocolo pode causar na operação desta rede. Formalmente, simulação pode ser definida como o processo de projetar um modelo de um sistema real e realizar experiências com estes modelos, com o propósito de compreender o comportamento deste sistema, ou avaliar diferentes estratégias para a operação do sistema.

A partir desta definição, podemos compreender as principais vantagens advindas da simulação. Primeiramente, com simulações é possível reproduzir o comportamento de sistemas reais via hardware e software. Isto permite-nos entender a sensibilidade do sistema a diversos parâmetros, e a melhor configuração dos mesmos para a obtenção de melhores resultados em uma implementação prática.

Em segundo lugar, a simulação permite estudar um problema em vários níveis de abstração. Isto permite um melhor entendimento do comportamento e das interações de todos os componentes que fazem parte do sistema. Os componentes do sistema podem ser projetados isoladamente, otimizados, e então unidos, representando o sistema todo. Todo o sistema pode ser construído com base nesta técnica *Top-Down*, e soluções alternativas podem ser investigadas sem a implementação física do sistema. A desvantagem é que o projeto e a implementação dos modelos para simulação muitas vezes são tão complexos quanto os sistemas reais que são os alvos da simulação.

Em terceiro lugar, a simulação pode ser usada como ferramenta para apresentação e demonstração de conceitos. Isto é verificado principalmente em simuladores que utilizam de maneira inteligente recursos de computação gráfica e animação.

A simulação também permite ajudar aos pesquisadores em sua intuição científica. O desenvolvimento de modelos de rede é uma técnica útil para o projeto de novas tecnologias

e configurações de redes. Ainda, os modelos resultantes também podem ser usados como um importante instrumento didático para o ensino de redes de computadores.

2.2 - TÉCNICAS DE SIMULAÇÃO

Vários fatores devem ser levados em consideração quando se escolhe a técnica de simulação a ser adotada em um simulador de redes. A técnica escolhida deve ser eficiente, deve permitir a troca flexível de informações entre os modelos de simulação, deve possibilitar a modelagem do fluxo e ainda deve permitir a modelagem flexível da arquitetura e dos componentes das redes.

Atualmente, várias técnicas são utilizadas no desenvolvimento de softwares de simulação de sistemas de comunicação. Estas técnicas basicamente diferem na forma como os blocos do sistema simulado são chamados (acionados). Entre as principais técnicas temos [1]: simulação orientada pelo tempo (*Time-Driven Simulation*), simulação orientada por eventos (*Event-Driven Simulation*) e simulação orientada por dados (*Data-Driven Simulation*). Veremos na seqüência mais detalhes destas técnicas.

2.2.1 - SIMULAÇÃO ORIENTADA PELO TEMPO (TIME-DRIVEN SIMULATION)

No modo mais simples de implementação desta técnica, cada bloco do modelo de simulação é chamado uma vez a cada intervalo de avanço do tempo de simulação (*simulation clock*). Ou seja, a cada incremento no tempo de simulação, todos os blocos do modelo são chamados, atualizam seus estados internos para corresponder ao tempo atual, independente da existência ou não de alguma tarefa a ser executada. Isto torna esta técnica bastante ineficiente do ponto de vista computacional [1]. O tempo global de simulação é atualizado através de um incremento constante.

2.2.2 - SIMULAÇÃO ORIENTADA POR EVENTOS (EVENT-DRIVEN SIMULATION)

Nesta técnica de simulação, os blocos agendam chamadas, que são armazenadas em uma fila na forma de eventos. Cada evento possui um tempo de execução. O núcleo do

simulador (*kernel*), gerencia os eventos, retirando os eventos da fila no instante de tempo apropriado, entregando-o para o bloco para o qual o evento se destina.

Toda vez que um evento é retirado da fila, o tempo global de simulação é avançado para o tempo deste evento. Vários eventos podem ser agendados para o mesmo tempo de execução. Neste caso, o núcleo do simulador deve executar os eventos na ordem em que eles foram agendados, ou seja, segundo a disciplina FIFO (*First-in first-out*).

Quando um bloco é chamado, ele executa as suas funções específicas. Ao término do processamento dessas funções, o fluxo da simulação é retornado para o núcleo do simulador, que retira e interpreta outro evento da fila. A simulação prossegue até que não haja mais eventos para serem executados, ou até que o tempo final de simulação seja alcançado.

A troca de informações entre blocos pode ser facilmente implementada utilizando-se a simulação orientada por eventos. Por exemplo, um bloco pode agendar um evento para outro bloco contendo as informações a serem trocadas. Assim, quando o núcleo do simulador interpretar este evento, ele poderá chamar o bloco de destino e entregar as informações que lhe pertencem. Neste caso o núcleo do simulador funciona também como um intermediador para a troca de mensagens entre blocos.

O simulador ***lsimnt*** desenvolvido neste trabalho usa esta técnica de simulação.

2.2.3 - SIMULAÇÃO ORIENTADA POR DADOS (DATA-DRIVEN SIMULATION)

Nesta técnica de simulação, um bloco só é chamado se todos os dados necessários na sua entrada estiverem disponíveis. Desta forma, se em cada bloco do modelo de simulação for conhecida de antemão a disponibilidade de tais dados, é possível a construção de uma seqüência de chamada de blocos antes do início da simulação. Caso contrário, a disponibilidade de tais dados deve ser verificada durante a simulação e os blocos deverão ser chamados dinamicamente.

2.3 - SOFTWARES DE SIMULAÇÃO PARA REDES DE COMUNICAÇÃO

Com a simulação, é possível verificar o comportamento de um sistema real, analisando a sensibilidade dos parâmetros. A melhor configuração destes parâmetros pode ser investigada sem a implementação física do sistema.

Os softwares de simulação para redes de comunicação estão classificados em categorias conforme o grau de flexibilidade e o propósito ao qual o software está destinado. Segundo [9], as três principais categorias de softwares para simulação de redes de comunicação são:

- Linguagens gerais de simulação;
- Linguagens de simulação orientadas às redes de comunicações; e
- Simuladores orientados às redes de comunicações.

2.3.1 - LINGUAGENS GERAIS DE SIMULAÇÃO

Uma linguagem geral de simulação pode ser utilizada na simulação de qualquer tipo de sistema. Algumas destas linguagens podem inclusive incluir um conjunto de modelos específicos para simulação de redes de comunicação.

Nesta categoria de software, um modelo é escrito em uma linguagem de simulação usando estruturas disponíveis na linguagem, incluindo entidades (mensagens), atributos (tipos de mensagens), recursos (nós ou conexões) e filas (buffers). A grande vantagem aqui é a possibilidade de modelar praticamente todos os tipos de redes de comunicações, embora requeira conhecimentos em programação por parte dos usuários, e trabalho no desenvolvimento do código dos modelos.

Exemplos desta categoria de software são os simuladores ***Arena*** [45] e ***SLAM System*** [42].

2.3.2 - LINGUAGENS DE SIMULAÇÃO ORIENTADAS ÀS REDES DE COMUNICAÇÕES

As linguagens de simulação orientadas às redes de comunicações são linguagens construídas especialmente para a simulação de redes de comunicação. Como vantagens temos a possível redução do tempo de programação pela utilização de modelos construídos especificamente para os sistemas de comunicação.

Um exemplo desta categoria de software é o simulador ***OPNET Modeler*** [46].

2.3.3 - SIMULADORES ORIENTADOS ÀS REDES DE COMUNICAÇÕES

Os simuladores orientados às redes de comunicações apresentam-se basicamente como um pacote de software que permite a simulação de uma rede de comunicação de um tipo específico sem a necessidade de programação. A rede específica que está sendo simulada (dentro do âmbito das redes abrangidas pelo pacote) é construída através da escolha de componentes para construir uma topologia.

No caso de existirem modelos disponíveis, a grande vantagem de um simulador orientado a redes de comunicação é que ele permite uma grande redução no tempo necessário para realizar uma simulação. No caso de não existirem modelos disponíveis para uma simulação, esta categoria de software tipicamente permitem a adição de novos modelos, e neste caso, requer conhecimentos em programação por parte dos usuários, e trabalho no desenvolvimento do código dos modelos.

Exemplos desta categoria de software são os simuladores ***NS2*** [44], ***Hydragyrum*** [1], e o ***lsimnt*** desenvolvido neste trabalho.

CAPÍTULO 3. PROTOCOLOS TCP/IP

3.1 - O PROTOCOLO IP (INTERNET PROTOCOL)

IP é um protocolo que providência a entrega de pacotes para o protocolo TCP. O IP oferece um sistema de entrega de dados sem conexão. Isto é, os datagramas IP não são garantidos de chegarem ao seu destino, nem de serem recebidos na ordem em que foram enviados. O *checksum* do IP confirma apenas a integridade dos cabeçalhos dos datagramas. Desta maneira, a responsabilidade pelos dados contidos nos datagramas do IP (e sua sequência) é tarefa de protocolos de mais alto-nível, como por exemplo, o TCP.

Uma dos motivos do sucesso deste protocolo é sua simplicidade. O princípio fundamental de projeto do IP é derivado do "argumento fim-a-fim" (*end-to-end*) que põe a inteligência nas extremidades da rede (no nó fonte e no nó destino) deixando os nós intermediários, no núcleo da rede, com pouca inteligência. Deste modo, os roteadores IP intermediários somente comparam o endereço de destino dos datagramas IP a serem roteados contra uma tabela de roteamento para determinar o próximo nó (*next hop*). Se a fila de saída para o próximo nó for longa, o datagrama poderá ser atrasado. Se a fila estiver cheia ou indisponível, o roteador pode descartar o datagrama. Desta forma, o serviço provido pelo IP é do tipo melhor esforço, sujeito a atrasos imprevisíveis e perdas de datagramas [24].

3.2 - O PROTOCOLO TCP (TRANSMISSION CONTROL PROTOCOL)

O TCP é um protocolo de transporte de *stream* de dados, orientado à conexão, e que fornece a garantia de um serviço de entrega confiável. O TCP é um protocolo complexo, pois foi desenvolvido para oferecer confiabilidade trabalhando com diversos ambientes de rede, nas mais diferentes plataformas e com os mais variados aplicativos [24].

Para o TCP, um *stream* de dados é uma seqüência de bits dividida em octetos (oito bits). A aplicação fornece ao protocolo de transporte os dados em forma de *stream*, e o

protocolo é livre para segmentar o *stream* da forma que for melhor para a transmissão. Pode-se dizer então que o TCP transporta segmentos de *stream* de dados. Os segmentos que o TCP transporta possuem o tamanho variável e o tamanho máximo do segmento é definido no momento em que a conexão é estabelecida.

Por ser um protocolo orientado à conexão, antes de iniciar a transferência de dados, ele precisa estabelecer uma conexão com o *host* destino, para que os dois possam fazer uma série de ajustes, como sincronização, cálculo de tempo, tamanho máximo do segmento, entre outros.

O TCP se baseia no conceito de portas de comunicação. O TCP identifica cada conexão por *end-points*. Cada *end-point* é visto como (*host*, porta), onde *host* é o número IP do *host* de destino e porta é o número da porta TCP deste *host*. Cada porta de comunicação corresponde a uma fila de mensagens, a um buffer individual. Isto permite que um *host* mantenha múltiplas conexões, utilizando uma porta para cada conexão. Existem algumas RFCs que definem portas específicas para serviços conhecidos, como TELNET, FTP, HTTP, POP, SMTP, entre outros.

As conexões que o TCP estabelece são *full duplex*, ou seja, a transferência dos dados pode ocorrer em ambos os sentidos, possibilitando que os dois *hosts* conversem durante a transmissão. Se um dos *hosts* encerrar a conexão, esta torna-se *half duplex*, com o fluxo trafegando em apenas um sentido. A vantagem da conexão *full duplex* em relação a *half duplex*, é o *piggyback*, isto é, o protocolo receptor pode retornar informações de controle para o transmissor, reduzindo o tráfego na rede.

Para garantir um serviço de entrega confiável, o TCP requer mais CPU e mais largura de banda que outros protocolos de transporte.

3.2.1 - FUNCIONAMENTO

Cada segmento TCP é composto de duas partes:

- Cabeçalho, onde são enviadas as informações de controle necessárias;
- Dados a serem transportados.

O cabeçalho do TCP possui um tamanho fixo de 20 bytes e mais um campo opcional de tamanho variável. O tamanho do cabeçalho é medido por múltiplos de 32 bits. O formato do segmento do TCP, que será visto a seguir, é definido na RFC 793 [29]. A Figura 1 mostra o formato do segmento TCP.

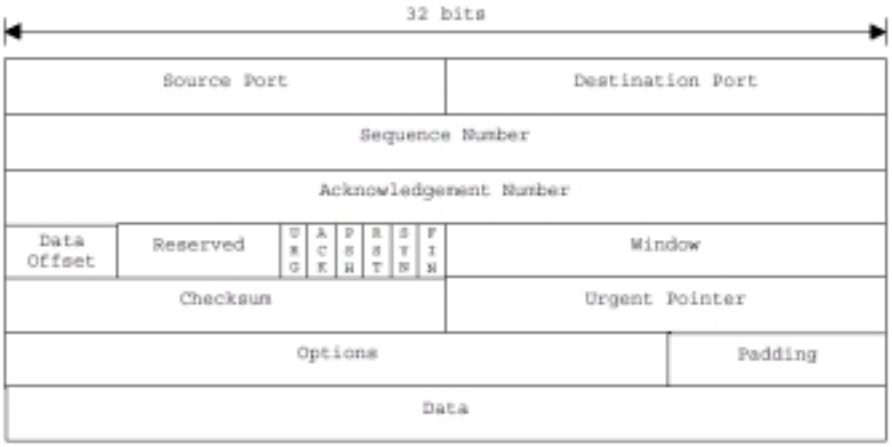

Figura 1 - Campos dos pacotes TCP

Os campos do cabeçalho são:

- ***Source Port* (16 bits) e *Destination Port* (16 bits)**: estes campos indicam o número da porta de origem e o número da porta de destino que serão usadas para estabelecerem a conexão. O TCP já tem definido portas padrão para diversos tipos de serviço.
- ***Sequence Number* (32 bits)**: campo que indica o número de seqüência dos pacotes que permite a remontagem dos dados no destino. O número de seqüência inicial é definido no momento em que a conexão estiver sendo estabelecida.
- ***Acknowledgement Number* (32 bits)**: número de reconhecimento, que significa o próximo byte aguardado pelo destino. Informa que até o byte anterior foi tudo recebido corretamente.
- ***Data Offset* (4 bits)**: chamado também, por alguns autores de *Header Length* (comprimento do cabeçalho). Indica onde os dados iniciam. Contém

um número inteiro medido em múltiplos de 32 bits. Devem ser incluídos zeros no final do cabeçalho até que ele se torne um múltiplo de 32 bits.

- ***Reserved* (6 bits)**: campo reservado para futura utilização.
- ***Control Bits* (6 bits)**: seis bits de controle, que servem para determinar a finalidade e o conteúdo do segmento. Indicam como interpretar outros campos do cabeçalho. São eles:
    - URG = indica que o campo *Urgent Pointer* é válido.
    - ACK = indica que o campo *Acknowledgement Number* é válido.
    - PSH = indica que o receptor deve entregar os dados para a aplicação imediatamente, sem esperar encher o buffer.
    - RST = serve para reiniciar ou rejeitar uma conexão.
    - SYN = utilizado para solicitar uma conexão.
    - FIN = utilizado para encerrar uma conexão.
- **Window (16 bits)**: tamanho da janela oferecido pelo *host*. Indica quantos bytes podem ser enviados a partir do byte confirmado (*Acknowledgement Number*).
- ***Checksum* (16 bits)**: campo que faz a checagem da integridade dos dados e do cabeçalho. A soma de todos os bytes mais o *Checksum* deve ser igual a zero.
- ***Urgent Pointer* (16 bits)**: campo que indica a posição no segmento onde os dados urgentes terminam.
- ***Options***: campo opcional e variável. Uma de suas funções é comunicar o tamanho máximo dos segmentos (MSS, *Maximum Segment Size*) que pode ser recebido pelo *host*. A opção MSS contém 16bits.
- ***Padding***: campo variável, composto de zeros, usado para especificar o final do cabeçalho e o início dos dados.
- ***Data***: esta é a segunda parte do segmento. Logo após o final do cabeçalho, iniciam os dados que serão transportados.

3.2.2 - CONFIABILIDADE

A maior parte dos protocolos que oferece confiabilidade na entrega de dados usa uma técnica chamada confirmação positiva (*positive acknowledgement*), ou seja, para cada segmento transmitido o receptor envia um segmento para confirmar o recebimento. É através deste mecanismo que o TCP consegue fornecer confiabilidade de entrega [24]. Um exemplo do funcionamento de confirmação positiva está ilustrado na Figura 2.

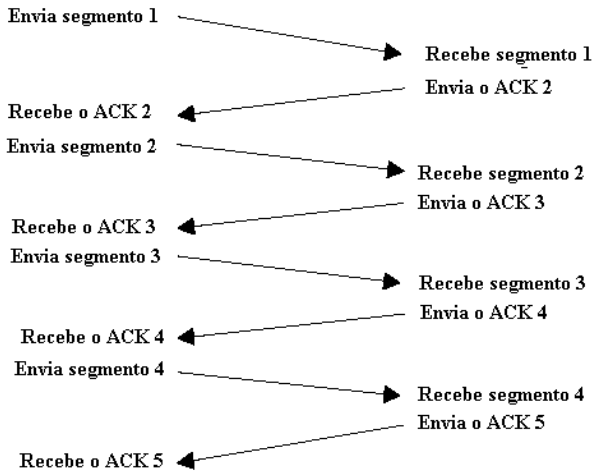

Figura 2 - Confirmação positiva

No exemplo da figura 2, somente após a chegada da confirmação do segmento anterior é que o transmissor pode enviar o próximo segmento. Na prática, este mecanismo gasta uma considerável largura de banda, pois ele espera a chegada da confirmação do segmento anterior para poder transmitir o próximo segmento. Isto diminui a vazão da rede, pois a rede permanece ociosa enquanto fica à espera da confirmação.

Existe um mecanismo que faz com que o processo de recebimento de confirmações seja mais eficaz. Tal mecanismo é conhecido como janela deslizante (*sliding window*). A janela deslizante permite que o transmissor envie múltiplos segmentos sem necessitar uma confirmação imediata. Por exemplo, se o tamanho da janela estiver estipulado para 4 segmentos, o transmissor pode enviar até 4 segmentos e, conforme chegarem as confirmações, novos segmentos poderão ser enviados. Isto aumenta o *throughput* na rede fazendo com que a banda seja melhor aproveitada. A Figura 3 ilustra as confirmações com o uso da janela deslizante.

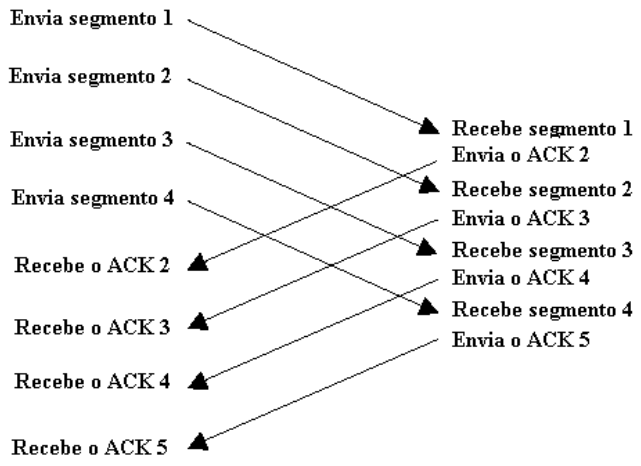

Figura 3 - Confirmações com o uso da janela deslizante

3.2.3 - CONTROLES

Como foi visto anteriormente, o TCP garante um serviço de entrega confiável utilizando técnicas de confirmação da chegada dos segmentos ao seu destino. Mas somente isto não é suficiente para garantir, além da confiabilidade, um bom funcionamento, uma boa performance e uma boa utilização da rede. Para tal, o protocolo requer uma série de controles, tais como: controle de erro, controle de fluxo, controle de congestionamento e controle de sessão [23].

O controle de erro faz com que o TCP certifique-se que os dados estão chegando corretamente no destino. Existem vários tipos de erro que podem ocorrer com os segmentos enviados durante uma transmissão, por isto o TCP deve detectar a ocorrência de um erro e tentar recuperar os segmentos que foram vítimas do erro, garantindo assim, a chegada correta dos dados no receptor.

O TCP deve preocupar-se também com o fluxo dos dados durante a transmissão para garantir melhor performance. O fluxo dos dados diz respeito à velocidade e a quantidade de dados que estão sendo enviados. Por exemplo, o transmissor deve saber qual é a capacidade de armazenamento e de processamento do receptor, pois se a capacidade do receptor for muito pequena, ele não dará conta de tratar muitos dados rapidamente, fazendo que o seu *buffer* se esgote e ocasionando o descarte de segmentos. Sendo assim, o

transmissor deve diminuir o fluxo de envio para que o receptor possa tratá-los de acordo com a sua capacidade.

O outro tipo de controle é o controle de congestionamento, que diz respeito à rede, como o estado dos roteadores que estão ao longo do caminho, entre o transmissor e o receptor. O TCP deve preocupar-se em evitar o congestionamento da rede, pois uma rede congestionada ocasiona não só o atraso, como a perda de pacotes, ou em casos mais extremos é causado o colapso de congestionamento, que pode fazer a rede parar.

O controle de sessão cuida do estado das conexões, controlando desde o estabelecimento até o encerramento de uma conexão.

3.2.4 - CONTROLE DE SESSÃO

3.2.4.1 - ESTABELECENDO UMA CONEXÃO

Para estabelecer uma conexão o TCP utiliza o *Three Way Handshake* [26], que significa, literalmente, "um aperto de mão em três vias", onde são necessários apenas 3 segmentos para que a conexão seja estabelecida. O *Three Way Handshake* funciona da seguinte maneira: o *host* que deseja fazer um pedido de conexão envia um segmento com a *flag* SYN ativa no campo *Control Bits*. Assim que o outro *host* recebe o pedido de conexão, ele envia um segmento contendo as *flags* ACK e SYN ativas, que significa aceitar a conexão. Para finalizar o estabelecimento da conexão, o primeiro *host* envia um segmento com a *flag* ACK, indicando ter aceitado a conexão com o *host*. A partir deste momento a conexão é considerada estabelecida pelo primeiro *host*. Quando o terceiro segmento é recebido pelo segundo *host*, este considera a conexão estabelecida. A partir do momento em que um *host* considera uma conexão estabelecida, ele pode iniciar a transferência de dados. A Figura 4 mostra o exemplo de um *Three Way Handshake*.

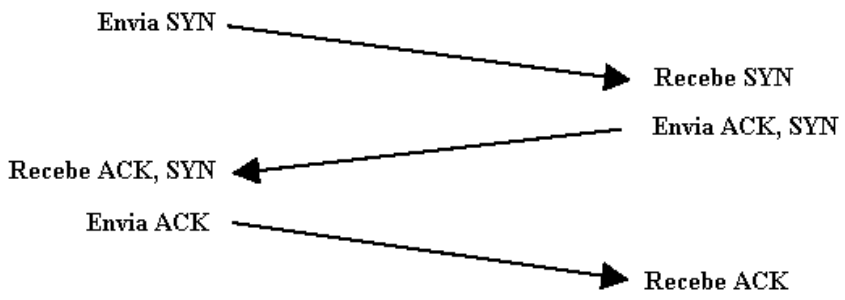

Figura 4 - Handshake de três vias

3.2.4.2 - ENCERRANDO UMA CONEXÃO

O método utilizado para encerrar uma conexão tamém é o *Three Way Handshake* [26]. A diferença é o campo *Control Bits*. Enquanto que para estabelecer uma conexão utiliza-se a *flag* SYN, para finalizar a conexão utiliza-se a *flag* FIN. Outra diferença é que o segundo segmento não inclui o FIN junto com o ACK. Primeiro, vai um segmento com a *flag* FIN, indicando que um dos *hosts* não tem mais dados para enviar e quer encerrar a conexão. Em resposta ao pedido de encerramento de conexão, o outro *host* envia um segmento com a *flag* ACK, avisando que recebeu o pedido. Este, deve então avisar a aplicação que não existem mais dados para receber e perguntar se pode encerrar a conexão. Caso a aplicação permita o encerramento da conexão, o TCP envia o segmento com a *flag* FIN para o outro, que por sua vez, envia um ACK finalizando a conexão. A Figura 5 apresenta o encerramento de uma conexão.

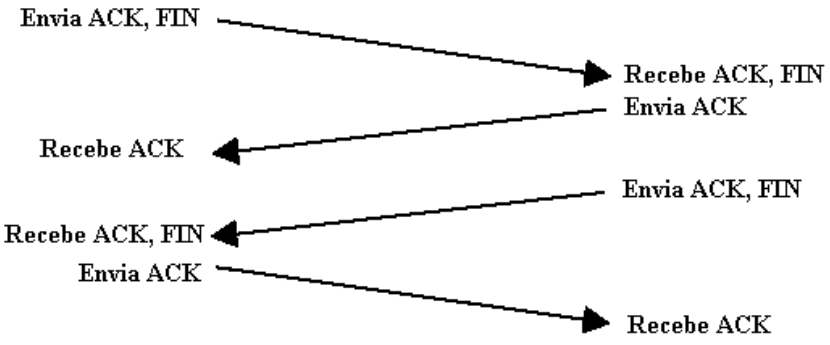

Figura 5 - Encerramento de uma conexão

3.2.5 - CONTROLE DE ERRO

3.2.5.1 - TIPOS DE ERRO

Os protocolos de transporte da pilha TCP/IP utilizam os serviços da camada inferior, IP, para a transmissão de datagramas até o *host* destino. Datagramas IP são encaminhados, *hop* a *hop*, de acordo com tabelas de roteamento. A transmissão de um datagrama IP de sua origem até seu destino final está sujeita a erros, tais como: entrega desordenada, pacotes replicados, pacotes corrompidos, pacotes perdidos e pacotes atrasados. O controle de erro do TCP é responsável pela detecção de tais erros e sua recuperação.

A entrega desordenada pode ser causada pela existência de vários caminhos para se chegar ao destino. Cada segmento é colocado na rede de forma seqüencial, mas eles podem percorrer diferentes caminhos dependendo do estado da rede. Como o TCP utiliza uma forma de conexão fim-a-fim (*end-to-end*), somente as pontas (*end-points*) conhecem a conexão. Isto quer dizer que a rede não tem conhecimento da conexão existente, ou seja, não é estabelecida uma conexão virtual, onde a rede sabe que todos os segmentos enviados por um *host* são da mesma conexão e que devem passar pelo mesmo caminho para chegar ao destino. Por exemplo, os roteadores trabalham com as suas tabelas de roteamento para saber qual é a melhor rota para se chegar a um determinado destino. Se, por acaso, um determinado roteador, que estiver entre uma ponta e outra da conexão, "sair do ar" por algum motivo, ou estiver congestionado, as tabelas de roteamento dos roteadores serão modificadas e o tráfego que passava por aquele roteador será desviado para outro, mudando o caminho da conexão, até que a rota anterior se normalize. A mudança das rotas pode ocorrer diversas vezes durante uma conexão, fazendo com que os últimos segmentos enviados cheguem antes dos primeiros. Isto faz com que os segmentos cheguem fora de ordem no destino. Para saber a ordem dos segmentos, o TCP utiliza o número de seqüência que vem especificado no campo *Sequence Number* do cabeçalho de cada segmento.

Pacotes replicados fazem parte de um tipo de erro no qual vários pacotes idênticos chegam no destino. A causa mais provável dos pacotes replicados é a retransmissão desnecessária, que ocorre quando o atraso na rede variar muito, ou quando a velocidade ou a capacidade de processar os dados entre o transmissor e o receptor for muito diferente. O

segmento realmente chega ao destino, mas o transmissor não sabe se chegou ou não devido a um atraso no recebimento da confirmação, então ele envia o segmento novamente. Isto faz com que cópias do mesmo segmento cheguem duas ou mais vezes no receptor. O receptor deve descartar as cópias destes segmentos. Pacotes replicados consomem, desnecessariamente, recursos da rede e dos *hosts* envolvidos.

Pacotes corrompidos são pacotes que não chegam completos no receptor, ou pela simples troca de um bit no decorrer do caminho, o cálculo da soma de verificação não confere e o pacote não pode ser reconstruído. Um *host* ou um roteador descarta o pacote corrompido, fazendo com que o mesmo tenha que ser retransmitido.

Já a perda de pacotes pode ser ocasionada pela própria rede. Isto é muito comum quando há um congestionamento e os roteadores necessitem descartar pacotes. Outro motivo que pode ocasionar perdas é o excesso de fluxo que pode estar fazendo com que o receptor esteja recebendo mais pacotes do que consegue dar conta, então o seu buffer de entrada esgota e os pacotes podem ser descartados.

O atraso da chegada de segmentos no receptor pode ser causado pelo mesmo motivo da perda, só que não chegam a ser descartados. Uma rede congestionada, ou com o fluxo muito alto, tende a ficar lenta, pois os pacotes ficam no buffer dos roteadores até que eles consigam encaminhá-los devidamente.

3.2.5.2 - DETECÇÃO DE ERRO

Uma das formas de detectar segmentos perdidos é através de um *timeout*, o RTO (*Retransmission Time Out*), que é o tempo que o transmissor tem para esperar a chegada de um ACK [22]. Se o RTO expirar antes da chegada do ACK, então o TCP assume que o segmento foi perdido. O RTO é definido através do RTT (*Round Trip Time*), que significa o tempo entre o envio do segmento e o recebimento do ACK. Estima-se, assim, o tempo entre a ida e a volta de um segmento. A amostra do RTT é feita iniciando um timer para cada segmento enviado. Quando chega a confirmação deste segmento, o timer é cancelado. O valor retirado deste timer é o RTT estimado. O RTO deve ser sempre maior que o RTT, do contrário o *timeout* poderá expirar antes da confirmação chegar. O valor do RTO é variável, adaptando-se de acordo com o desempenho da conexão. O TCP monitora cada

conexão e deduz valores razoáveis para o *timeout*. À medida que o desempenho de uma conexão muda, o TCP revisa o seu valor de *timeout* adaptando-se a mudança.

A RFC 1122 especifica que, no início de uma transmissão, quando ainda não tiver sido calculado o RTT, o RTO deve ser setado para 3 segundos. A RFC 2988 sugere que quando o RTO for calculado e o resultado for menor que um segundo, então ele deve ser arredondado para um segundo.

Outra maneira de detectar perdas é através de ACKs duplicados [26]. Suponha que o transmissor envie os segmentos 1, 2, 3 e 4, e o segmento 1 seja perdido. Assim que o segmento 2 chegar, o receptor enviará um ACK com o número de seqüência do segmento 1. Quando receber o segmento 3, ele enviará um ACK com o número de seqüência do segmento 1, e quando o segmento 4 chegar, ele enviará um ACK com o número de seqüência do segmento 1. Enquanto o segmento 1 não chegar, o receptor enviará ACKs com o número de seqüência deste segmento. A Figura 6 ilustra este exemplo.

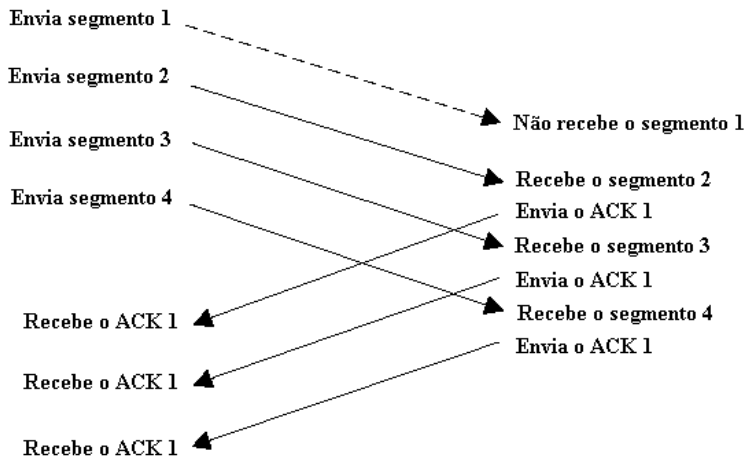

Figura 6 - ACKs duplicados

Se vários ACKs possuírem o mesmo número de seqüência, eles serão ACKs duplicados. Na verdade, o transmissor não consegue distinguir se o segmento foi perdido ou se simplesmente chegou fora de ordem, então ele assume que mais de 3 ACKs duplicados significam um segmento perdido.

A detecção de segmentos que chegam fora de ordem é feita e recuperada no próprio receptor, que os reordena através dos números de seqüência dos segmentos.

3.2.5.3 - RECUPERAÇÃO DE ERRO

Para que o TCP possa recuperar um segmento perdido ou corrompido, ele utiliza um mecanismo de retransmissão. Assim que o erro é detectado, o TCP providencia a retransmissão do segmento.

Com o algoritmo de Karn o TCP não deve utilizar como amostra o RTT de segmentos retransmitidos, ou seja, cada vez que um segmento for retransmitido o TCP não deve ligar o timer, ignorando o RTT do segmento retransmitido. Isto pode fazer com que, se o RTT real aumentar, o TCP não poderá perceber e continuará retransmitindo segmentos, pois o seu timeout estará desatualizado. Para que isto não aconteça deve ser utilizado o algoritmo de ***backoff* exponencial**.

O algoritmo de *backoff* exponencial parte do princípio que, se houve uma retransmissão, é por que o *timeout* expirou. Sendo assim, o *backoff* aumenta o valor do RTO para que, se houve um aumento do RTT, o *timeout* não expire novamente para o segmento retransmitido, que provocaria uma nova retransmissão. O cálculo utilizado para aumentar o *timeout* é:

$$novo_timeout = x \cdot timeout$$

onde x normalmente é igual a 2. Por exemplo, se o transmissor enviar um segmento e não receber confirmação em 3 segundos, ele faz uma retransmissão aumentando o *timeout* para 6 segundos. Se a confirmação não chegar, ele dobra o *timeout* para 12 segundos e assim sucessivamente, até ter sucesso ou encerrar a conexão.

3.2.6 - CONTROLE DE FLUXO

3.2.6.1 - JANELA DESLIZANTE

O TCP permite que o receptor controle a quantidade de dados que serão enviados pelo transmissor. Isto acontece através do campo *Window*, que é o tamanho da janela oferecida pelo receptor a cada ACK de confirmação, indicando o quanto de dados ele pode

receber. Isto equivale ao tamanho do buffer que o receptor possui para armazenar os dados. A janela indica o número máximo de bytes que o transmissor pode enviar.

Esta janela é chamada de janela deslizante [23], pois a cada confirmação que chega ela desliza abrindo espaço para novos segmentos serem enviados. O tamanho da janela deslizante do TCP é variável e o valor dela é medido em quantidade de bytes. O TCP permite que o tamanho da janela varie com o passar do tempo, pois cada ACK que o receptor envia para confirmar um segmento é acompanhado pelo tamanho da janela, então a cada ACK o receptor pode informar um novo tamanho de janela.

A cada ACK o receptor envia o valor da janela para o transmissor. Este valor corresponde ao tamanho do buffer disponível no receptor no momento em que o ACK foi enviado. O transmissor utiliza este valor para calcular o tamanho da janela utilizável, que é o tamanho da janela oferecida pelo receptor menos a janela do transmissor, que guarda os segmentos que foram enviados e ainda não foram confirmados. Como resultado deste cálculo, a janela utilizável deve ser sempre igual ou menor que a janela oferecida pelo receptor.

Se, por exemplo, o receptor oferecer uma janela de 2000 bytes, e o tamanho máximo de cada segmento esteja estipulado em 500 bytes, o transmissor pode utilizar esta janela para enviar 4 segmentos de 500 bytes cada. Depois que o receptor processar o primeiro segmento, ele tira de seu buffer os 500 bytes que correspondem a este segmento e envia um ACK para o transmissor com o valor do campo *Window* de 2000 bytes.

A Figura 7 ilustra uma situação em que temos uma janela oferecida pelo receptor com capacidade de 2000 bytes dividida em 4 segmentos de 500 bytes cada.

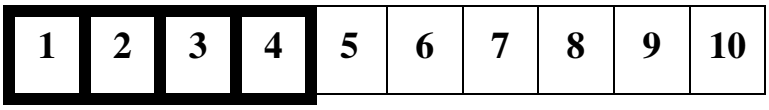

Figura 7 - Janela deslizante com nenhum segmento confirmado

Já a Figura 8 apresenta como a janela desliza após o recebimento do ACK do primeiro segmento de 500 bytes, possibilitando a transmissão de mais um segmento de 500 bytes.

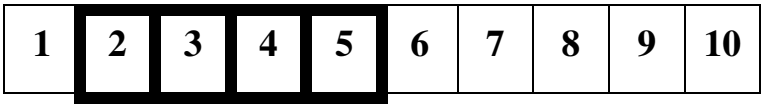

Figura 8 - Janela deslizante com um segmento confirmado

Os segmentos que ficam do lado esquerdo da janela são os segmentos que já foram enviados e confirmados. Os segmentos que estão do lado direito da janela são os segmentos que ainda não foram enviados. E os segmentos que estão dentro da janela são os que foram enviados e ainda não foram confirmados.

3.2.7 - CONTROLE DE CONGESTIONAMENTO

Para fazer o controle de congestionamento, o TCP precisa detectar quando há um congestionamento. Os mecanismos utilizados para a detecção de congestionamento são o *timeout* e os ACKs duplicados, abordados no capítulo anterior, pois o TCP originalmente não conta com notificação explícita de congestionamento (ECN, *Explicit Congestion Notification*). Quando um erro é detectado, através do *timeout*, o TCP assume que existe um roteador congestionado e que esta foi a causa da perda. Em outras palavras, a maioria das perdas é causada por congestionamento e não por corrupção.

Para diminuir o risco de colapso na rede, cada fluxo TCP é regido por uma política de controle de congestionamento. Isto assegura que as centenas de fluxos TCP que se cruzam em um roteador qualquer ocupem uma fatia "justa" da banda disponível (*fairness*).

3.2.7.1 - SLOW START

O *Slow Start* é um mecanismo de controle de congestionamento utilizado para estabelecer o equilíbrio de uma conexão [24]. O TCP deve iniciar uma transmissão inserindo segmentos na rede de forma gradual, para que ele possa conhecer a capacidade da rede. O equilíbrio da conexão é estabelecido cada vez que o TCP começa a inserir dados na rede, ou seja, a cada início de conexão ou logo após uma perda, fazendo com que a conexão volte ao normal.

A quantidade de dados que o TCP começa a transmitir é de acordo com o tamanho da janela do receptor. Quando o receptor oferece uma janela muito grande e a aplicação do transmissor fornece dados suficientes para preencher aquela janela, o TCP transmissor envia muitos segmentos ao mesmo tempo. Isto pode gerar perdas caso algum roteador esteja com muita carga, ocasionando retransmissões. O *Slow Start* faz com que os segmentos sejam inseridos na rede de uma forma lenta e gradual, melhorando o aproveitamento da banda e evitando o risco de um congestionamento na rede.

O *Slow Start* utiliza a janela de congestionamento (*cwnd*) que é o limite que o transmissor tem para enviar os dados quando inicia a transmissão ou depois de receber uma confirmação. A janela oferecida pelo receptor (*rwnd*) é o limite que o receptor tem para receber os dados. O limite que deve ser utilizado para a transmissão é o valor mínimo entre estas duas janelas, ou seja, se a *cwnd* for menor que a *rwnd*, então o transmissor deve enviar a quantidade de dados estipulada na *cwnd*. Outra variável incluída é a *Slow Start Threshold* (*ssthresh*), que serve para determinar se o algoritmo do *Slow Start* será utilizado ou não na transmissão dos dados.

A implementação do *Slow Start* implica nos seguintes passos: deve ser adicionada a *cwnd*, quando uma transmissão iniciar ou reiniciar após a perda de um segmento. Atribuir o valor da *cwnd* para 1 segmento. Para cada ACK que chegar, incrementar a *cwnd* com 1 segmento e quando o TCP for transmitir, deve transmitir o mínimo entre a *cwnd* e a *rwnd* [24].

Quando a janela de congestionamento é iniciada, ela contém o valor de 1 segmento. O transmissor envia este segmento e espera por sua confirmação. Quando a confirmação chega, aumenta-se a janela de congestionamento em mais 1 segmento, possibilitando a transmissão de 2 segmentos. Quando as 2 confirmações chegarem, aumenta-se mais 2 segmentos na janela, deixando-a com 4 segmentos. Enviando os 4 segmentos, chegarão 4 confirmações, fazendo com que a janela aumente para 8 segmentos e assim sucessivamente. Assim, a janela de congestionamento cresce de forma exponencial. Na prática, cada rajada confirmada duplica a janela de congestionamento, até atingir o valor do limitante, *ssthresh*. A partir deste ponto, o crescimento passa a ser linear, ou seja, *cwnd* aumenta em um segmento para cada **janela** confirmada, e não mais para cada **pacote** confirmado, como nos mostra a Figura 9.

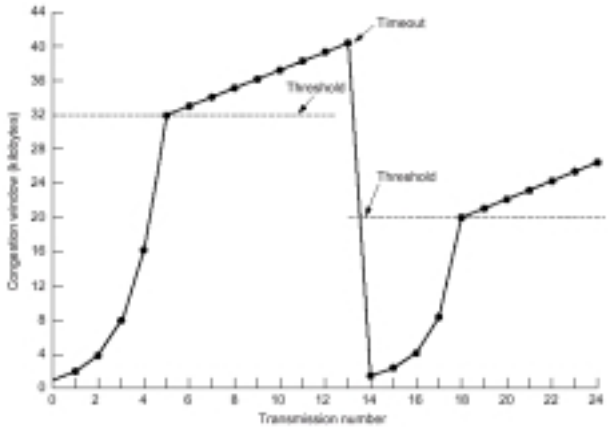

Figura 9 - Janela de congestionamento

Assim, a janela de congestionamento mantém seu crescimento até que ocorra um *timeout*. Neste instante, o valor de *ssthresh* é reduzido para a metade do valor corrente de *cwnd*, e *cwnd* é então reiniciado em um segmento.

CAPÍTULO 4. QUALIDADE DE SERVIÇO NAS REDES TCP/IP

4.1 - QUALIDADE DE SERVIÇO NAS REDES TCP/IP

Nada trouxe tantas transformações ao nosso dia a dia como a rede mundial de computadores, a Internet. A medida em que as redes ganham novas funcionalidades, com aplicações de áudio e vídeo tornando-se populares e compartilhando largura de banda com o tráfego convencional (transferência de arquivos, acesso a páginas HTML) e outras aplicações, faz-se necessária a implementação de técnicas que permitam um uso mais racional dos recursos existentes [38].

Assim, as redes convergem para uma infra-estrutura única compartilhada, capaz de suportar tráfego gerado por aplicações multimídia e, neste cenário, tipos diferentes de tráfego devem ser tratados de modo diferenciado, trazendo diferentes graus de satisfação aos usuários. Em resumo, as redes que permitem estes tratamentos diferenciados para deferentes tipos de tráfego são ditas redes capazes de prover Qualidade de Serviço (QoS) às aplicações.

A noção de QoS é fundamental para o projeto, a implantação e a operação de redes de computadores. Intuitivamente, qualidade de serviço expressa, em última análise, o grau de satisfação do usuário para com as aplicações que fazem uso da rede. O grau de satisfação do usuário pode ser subjetivo, como no caso de um usuário humano recebendo áudio e vídeo através da rede; ou objetivo como atrasos máximos de mensagens enviadas através da rede por um sistema de controle.

Redes com QoS são de fundamental importância quando se busca utilizar eficientemente canais de comunicação de redes de longa distância WAN (*Wide Area Network*), os quais geralmente apresentam baixa taxa de transferência e/ou alto custo. Em redes corporativas espalhadas por diversas localizações geográficas conectadas por uma WAN, pode ser imperativo dar maior prioridade ao tráfego multimídia em relação a outros tipos de tráfego.

Algumas aplicações, como as de áudio, têm exigências rígidas quanto ao atraso fim-a-fim, mas podem tolerar perdas mínimas de pacotes, enquanto outras como transferência de arquivos são sensíveis a este último aspecto, mas as exigências quanto ao atraso são pouco severas.

Algumas pesquisas, impulsionados pela evolução das redes de alto desempenho, desenvolvidas pelo ATM Fórum e IETF (*Internet Engineering Task Force*), apresentam várias alternativas para prover QoS às redes de computadores: o padrão ATM (*Asynchronous Transfer Mode*), o protocolo de reserva de recursos, RSVP (*Resource ReSerVation Protocol*), principal componente da arquitetura de serviços integrados (***IntServ***), e a arquitetura de serviços diferenciados (***DiffServ***), são soluções interoperáveis e complementares que buscam prover QoS às aplicações.

Serviços diferenciados tem sido o modelo mais utilizado para implementação de QoS. Ele exige menos dos roteadores, necessitando pouca atualização de software para prover bons métodos de classificação, policiamento, montagem e remarcação de pacotes.

No exemplo de simulação proposto neste trabalho, a abordagem escolhida foi a arquitetura de *DiffServ*, que será descrita a seguir.

4.2 - ARQUITETURA DE SERVIÇOS DIFERENCIADOS

O modelo de serviços diferenciados implementa QoS com base na definição de tipos de serviços. No cabeçalho de um pacote IP, existe um campo chamado TOS (*Type of Service*) que pode representar o tipo do serviço. No entanto, serviços diferenciados ampliam a representação de serviços e o tratamento que pode ser dado para encaminhar um pacote, definindo um novo layout para o TOS, passando a chamá-lo de *DS Field* (*Differentiated Service Field*). No *DS Field*, são codificadas as classes para serviços diferenciados. A marca de classificação *DiffServ* é conhecida como *DiffServ CodePoint*, ou DSCP. O valor do DSCP é marcado no subcampo "Precedência", que ocupa os três primeiros bits do campo TOS do protocolo IPv4 [33].

A arquitetura *DiffServ* parte do princípio que domínios adjacentes tenham um acordo sobre os serviços que serão disponibilizados entre os mesmos. Este acordo denomina-se SLA – *Service Level Agreement*. Um SLA determina as classes de serviços

suportadas e a quantidade de tráfego na banda entre os domínios. Os domínios podem definir um SLA estático ou dinâmico, sendo que, neste último caso, um protocolo de sinalização e controle será necessário para o gerenciamento da banda.

Com *DiffServ*, os próprios clientes podem marcar seus *DS Fields* e enviar para o receptor. No entanto, dessa forma, não há como saber se há recursos disponíveis para a comunicação, fazendo com que, por exemplo, o pacote chegando em um roteador que não provê QoS com o *DS Field* marcado, seja remarcado e passe a ser um pacote de um serviço de melhor esforço.

Um domínio *DiffServ* tem roteadores de borda e roteadores de núcleo. Os roteadores de borda são responsáveis pela classificação e condicionamento do tráfego que entra no domínio DS, e os de núcleo pelo encaminhamento do tráfego segundo uma política de QoS. Para cada fluxo de tráfego entrando no domínio pelos roteadores de borda, a política de QoS define qual terá um serviço diferenciado, como este deverá ser marcado nos roteadores de borda e como será tratado pelos nós interiores. Estes, por sua vez, examinam a marcação dos pacotes e atuam de acordo com as políticas definidas ou seu perfil de tráfego.

Todo fluxo de tráfego que entrar no domínio pelos roteadores de borda receberá um tratamento diferenciado segundo uma política de QoS. Esta política define como os pacotes deste fluxo deverão ser marcados nos roteadores de borda, e como serão servidos pelos roteadores de núcleo.

4.2.1 - COMPONENTES ESSENCIAIS

A arquitetura define dois importantes componentes para os nós *DiffServ*: os componentes de classificação e de condicionamento de tráfego. Estas funções são mais complexas em nós *DiffServ* de borda que em nós de interior, mas ambos os nós têm um **classificador** [39].

O classificador seleciona os pacotes em função de informações contidas no cabeçalho e regras de classificação. Existem dois tipos de classificadores:

- Classificadores de comportamento agregado (BA - *Behaviour Agregate*): estes classificadores somente verificam o codepoint dos datagramas; e

- Classificadores multi-campo (*Multi-Field* - MF): estes classificadores verificam múltiplos campos no cabeçalho IP.

Nos roteadores *DiffServ*, o classificador divide o fluxo de entrada em um conjunto de fluxos de saída através de filtros de tráfego, baseados no conteúdo do cabeçalho do pacote e/ou em diferentes atributos do pacote que podem ser implicitamente derivados.

Os demais componentes da arquitetura são responsáveis pela medição, condicionamento (*shapping*), policiamento de tráfego e remarcação de datagramas (alteração do DSCP presente no datagrama). Estas funções são desempenhadas pelo medidor, marcador e condicionador. Tais componentes visam assegurar que o tráfego ingressante no domínio está em conformidade com os limites impostos pelo Acordo de Nível de Serviço (SLA - *Service Level Agreement*).

Os **medidores** analisam as propriedades temporais do tráfego selecionado pelo classificador, comparando-as com os parâmetros especificados no SLA para a cada classe de serviço. Normalmente medidores implementam o algoritmo *tocken bucket* com parâmetros r (taxa em bytes/s) e b (tamanho da rajada em bytes).

Dependendo da conformidade as seguintes ações podem ser executadas:

- **Marcação**: **marcadores** atribuem um DSCP ao datagrama, ou então alteram o valor do DSCP previamente marcado. O DSCP atribuído ao datagrama é função do estado informado pelo medidor. Por exemplo, o marcador poderá atribuir um DSCP referente a um serviço do tipo melhor esforço para o excedente de tráfego de uma classe AF ou EF; e/ou
- **Formatação ou Policiamento**: executada por **condicionadores** de tráfego, esta ação modifica o tráfego de entrada para forçar um determinado perfil de saída. Altera a característica temporal de um tráfego não conforme, reconduzindo-o à conformidade especificada no SLA. A conformidade pode ser atingida pela introdução de atraso no tráfego excedente. Caso a conformidade plena não possa ser atingida por falta de recursos ou política de condicionamento, os condicionadores podem descartar parte ou todo o excedente do tráfego.

4.2.2 - COMPORTAMENTO POR HOP

Comportamentos por *hop* (*Per-Hop Behaviors* - PHBs) são especificados por RFCs segundo quinze critérios presentes na RFC 2475 [39], a qual descreve funcionalmente a arquitetura *DiffServ*. Exemplos destes critérios são resumidos a seguir e estipulam que a definição do PHB ou grupo de PHBs deve definir precisamente:

- Os *codepoints* empregados pelo PHB ou grupo;
- Tipo de serviço que o PHB ou grupo atende;
- As restrições para a implementação do PHB e as conseqüências caso as restrições sejam violadas;
- Informações sobre configuração, gerência e segurança de equipamentos que implementarão o PHB; e
- Se, e em quais condições a remarcação de datagramas é permitida.

Três grupos de PHBs foram especificados até a presente data: Seletor de Classe, Encaminhamento Garantido e Encaminhamento Expresso.

4.2.2.1 - O GRUPO SELETOR DE CLASSE

Este grupo se destina a manter compatibilidade com o subcampo Precedência que ocupa os três primeiros bits do campo TOS do protocolo IPv4. O objetivo é preservar os esquemas de roteamento e encaminhamento que levam em conta o subcampo Precedência do protocolo IPv4 [38].

O DSCP para este grupo possui o formato xxx000 onde “x” indica que o bit pode receber o valor 0 ou 1. O valor 000000 é reservado para o PHB default que representa o serviço de melhor esforço (*Best Effort* – BE). Quando o PHB de um datagrama não puder ser identificado pelo seu *codepoint*, o PHB *default* é empregado e o datagrama encaminhado de forma convencional segundo o critério de melhor-esforço.

A RFC 2474 define que pelo menos duas classes de encaminhamento independentes devem ser empregadas para o grupo Seletor de Classe: uma classe para o PHB default, e outra para o DSCP 11x000, sendo que esta preserva os valores de precedência 110 e 111 normalmente usados no Ipv4 para datagramas de controle. Datagramas com este DSCP devem receber encaminhamento preferencial em relação ao PHB *default*. Ainda, a RFC

2474 estipula que quanto maior o valor do DSCP, mais prioritário é o tráfego a ele associado.

4.2.2.2 - O PHB ENCAMINHAMENTO GARANTIDO

O serviço do tipo Encaminhamento Garantido (*Assured Forward*, ou AF) é especificado pela RFC 2597, e tem por objetivo fornecer entrega de pacotes com largura de banda assegurada, em quatro classes de transmissão, mas não oferece garantias quanto ao atraso [38]. Cada classe tem três precedências de descartes (*Drop Precedence*), as quais são utilizadas para determinar a importância do pacote. Assim, um roteador congestionado dá preferência para serem descartados, entre os pacotes de uma mesma classe, aqueles com maiores valores de precedência de descarte.

Os três primeiros bits do DSCP identificam a classe de transmissão, 001, 010, 011 e 100 e os três últimos bits definem a precedência de descarte - 010 para a precedência mais baixa (ou seja, ultimo a ser descartado), 100 para precedência média e 110 para a mais alta precedência de descarte (ou seja, primeiro a ser descartado). Na Tabela 1 são apresentados os DSCP recomendados para o grupo AF.

Precedência Descarte	**Classe AF1**	**Classe AF2**	**Classe AF3**	**Classe AF4**
Baixa	001010	010010	011010	100010
Média	001100	010100	011100	100100
Alta	001110	010110	011110	100110

Tabela 1 - Codepoints para o grupo de PHBs AF

Uma implementação do PHB AF deve buscar minimizar os congestionamentos duradouros, porém permitindo congestionamentos curtos resultantes de rajadas.

Ou seja, o mecanismo de gerenciamento de filas deve detectar e reagir a congestionamentos duradouros dentro de cada classe e tolerar rajadas de pacotes que provocam congestionamentos curtos.

A RFC 2597 sugere o algoritmo de gerenciamento ativo de filas RED - *Random Early Detection* para controle de congestionamento [38]. Veremos mais detalhes do RED mais adiante.

4.2.2.3 - O PHB ENCAMINHAMENTO EXPRESSO

Um PHB denominado Encaminhamento Expresso (*Expedited Forward* - EF) é documentado na RFC 2598, tendo por objetivo fornecer entrega de pacotes com baixas taxa de perda, latência e *jitter* (variação de atraso), além de banda garantida [38].

O PHB EF é também denominado serviço premium ou de canal dedicado, e pode ser usado para tráfego com requisitos de baixa perda, baixo atraso, baixo *jitter* e garantia de largura de banda. Para que tais requisitos sejam alcançados, equipamentos que implementam este PHB devem escalonar o tráfego agregado neste PHB de forma a manter as filas descongestionadas nas interfaces de saída. Em outras palavras, o roteamento deve ser "expresso" para esta classe de serviço, mantendo o datagrama o menor tempo possível no interior do equipamento. Os dispositivos de borda do domínio *DiffServ* devem policiar o tráfego EF para assegurar a taxa de bits definida. Pacotes em excesso devem ser descartados.

O PHB Encaminhamento Expresso tem o DSCP igual a 101110.

4.3 - DISCIPLINAS DE ESCALONAMENTO E POLICIAMENTO DE TRÁFEGO

No contexto de QoS em redes IP e, em particular na arquitetura *DiffServ*, é permitida ao usuário a escolha das disciplinas de enfileiramento e encaminhamento de pacotes, bem como o ajuste dos parâmetros de configuração dos métodos de policiamento do tráfego [39].

Por esta razão, neste capítulo descreve-se os principais algoritmos e métodos utilizados em redes IP para essa finalidade.

4.3.1 - MECANISMOS DE GERENCIAMENTO DE ROTEADORES

Uma forma dos elementos controlar uma rajada de pacotes na entrada de uma interface superior a capacidade de atendimento consiste em usar uma disciplina de escalonamento para ordenar o tráfego e, então, determinar alguma forma de priorizar o encaminhamento de pacotes de fluxos de maior relevância para uma interface de saída. Conceitualmente, enfileirar é a ação de armazenar pacotes em um local onde eles

permaneçam até serem processados. Tipicamente, o enfileiramento ocorre dentro de roteadores quando os pacotes são recebidos pelo processador de interface do dispositivo (fila de entrada), e ocorre também antes da transmissão dos pacotes para uma outra interface (fila de saída) do mesmo dispositivo.

Um roteador básico é composto por [33]:

- Um conjunto de processos de entrada, que remontam os pacotes na forma como foram recebidos, verificando a integridade de sua estrutura básica;
- Um ou mais processos de encaminhamento, que determinam a interface destino do pacote; e
- Processos de saída, que estruturam e transmitem os pacotes para o próximo nó da rede.

A união dos processos de entrada, de encaminhamento e de saída compreendem o processo de gerenciamento de filas de pacotes. É importante entender o papel que as estratégias de enfileiramento desempenham no provimento de serviços diferenciados. As redes não orientadas à conexão, onde as redes IP são as principais representantes, operam segundo o paradigma *store-and-forward* (os pacotes são armazenados para posterior transmissão) e todos os pontos finais estão distantes uns dos outros mais do que um *hop*. Portanto, deve-se entender a complexidade do enfileiramento de tráfego, o efeito que o enfileiramento tem no sistema de roteamento e aplicações e porque mecanismos de gerenciamento de filas adequados são cruciais para QoS.

O gerenciamento de filas depende basicamente do algoritmo de enfileiramento dos pacotes e do tamanho máximo da fila [39]. A escolha do algoritmo e o tamanho máximo da fila podem, a primeira vista, ser relativamente simples, porém esta escolha pode ser extremamente difícil em função do padrão de comportamento do tráfego na rede ser aparentemente randômico. Se um tamanho elevado é imposto à fila, introduz-se um atraso e *jitter* no RTT (*Round-Trip-Time*), que pode interromper aplicações e protocolos de transporte fim-a-fim. Por outro lado se a fila for muito pequena, pode ocorrer o envio de dados em taxas maiores que a taxa de serviço da fila, resultando em perdas de pacotes.

Em tráfego de fluxo confiável, como quando se utiliza o protocolo TCP, tais pacotes descartados devem ser identificados e retransmitidos. Em fluxos de tempo-real não

confiável, como quando se utiliza o protocolo UDP, tais como áudio e vídeo, os pacotes perdidos acabam por representar uma degradação do sinal.

A seguir são descritas algumas disciplinas de escalonamento para tratar o enfileiramento nas interfaces de saídas, pois estas são as estratégias predominantes para tráfego do tipo *store-and-forward* e enfileiramentos relacionados a QoS.

4.3.1.1 - FILA SIMPLES: FIFO

Esta é a disciplina de despacho mais simples. Nenhum conceito de prioridade ou classe de tráfego é utilizado, com todos os pacotes sendo tratados igualmente. Existe uma única fila de saída, os pacotes recebidos são armazenados e enviados na mesma ordem em que chegaram.

Neste tipo de fila, fontes de tráfego mal comportadas podem consumir toda a largura de faixa disponível, tráfegos em rajada podem causar atrasos inaceitáveis em tráfegos sensíveis a atraso, e pacotes pertencentes a tráfegos de maior importância podem ser perdidos devido a *overflow* do buffer, causado possivelmente por tráfegos de menor importância.

4.3.1.2 - FILA COM PRIORIDADE: PQ

Nesta técnica existem filas distintas para diferentes classes de prioridades. Por exemplo, podemos ter quatro níveis de prioridade (alta, média, normal e baixa), com uma fila associada a cada nível.

Os pacotes, uma vez classificados, são encaminhados para a fila com nível de prioridade correspondente. Durante a transmissão, o algoritmo de despacho dá tratamento preferencial absoluto à fila de maior prioridade, em detrimento das filas de menor prioridade. Ou seja, no momento da transmissão o algoritmo sempre busca um pacote na fila de alta prioridade; caso não haja pacote nesta fila, um pacote da fila de média prioridade é buscado, e assim sucessivamente até se chegar à fila de baixa prioridade.

Um volume de tráfego elevado de maior prioridade pode reter os tráfegos de menor prioridade por um tempo inaceitavelmente elevado; este fenômeno é conhecido como

"*starvation*". No pior caso, devido a um grande volume de tráfego de alta prioridade, por exemplo, o tráfego de baixa prioridade pode nunca ser transmitido. Para se evitar esta situação, pode-se utilizar ferramentas de formatação de tráfego ou CAR (*Committed access rate*), de modo a restringir a taxa de chegada de tráfego de alta prioridade [41].

4.3.2 - MECANISMOS DE CONTROLE DE CONGESTIONAMENTO

Técnicas de controle de congestionamento monitoram o tráfego na rede no sentido de antecipar e evitar a ocorrência de congestionamento, usualmente através do descarte de pacotes.

As duas técnicas mais empregadas para este controle são o emprego dos algoritmos RED (*Random Early Detection*) e sua versão com ponderação, o WRED (*Weighted Random Early Detection*). Veremos estes algoritmos a seguir.

4.3.2.1 - MECANISMO DE GERENCIAMENTO ATIVO DE FILAS: RED

Quando ocorre um *timeout* no TCP transmissor, o protocolo reduz o tamanho da janela de transmissão e inicia o processo de partida lenta (*slow start*), onde o tamanho da janela vai sendo aumentado gradativamente à medida em que o transmissor vai recebendo reconhecimentos positivos do receptor.

Se um pacote de um fluxo TCP é descartado pela rede, ocorrerá um *timeout*, e o procedimento descrito no parágrafo anterior tem início. Como conseqüência da redução do tamanho da janela de transmissão, temos a redução na taxa de transmissão de pacotes.

Se a perda de pacote é devida a congestionamento no roteador, a redução da taxa de transmissão de pacotes por parte do TCP transmissor, resultante desta perda, irá aliviar a situação de congestionamento. Se a situação de congestionamento leva ao descarte de pacotes de vários fluxos distintos, teremos vários transmissores reduzindo suas janelas de transmissão e iniciando o processo de partida lenta, eliminando o congestionamento. No entanto, estes transmissores irão aumentando suas janelas de transmissão conjuntamente, até voltar ao tamanho original e, conseqüentemente, teremos novamente uma situação de

congestionamento, que resultará em novos descartes de pacotes e no reinicio do processo. Este ciclo é conhecido como Problema de Sincronização Global.

O algoritmo RED tenta evitar este problema atuando de forma preventiva, e não reativa, ao congestionamento; ou seja, o RED tenta evitar que o congestionamento ocorra. No algoritmo RED, quando uma situação de tendência de congestionamento é detectada (o tamanho da fila ultrapassa um determinado limiar, por exemplo), inicia-se um processo de descarte aleatório de pacotes, onde a probabilidade de descarte é função da taxa de ocupação da fila.

O RED só funciona adequadamente em conjunto com protocolos de transporte que sejam robustos quanto à perda de pacotes, como o TCP. Se o protocolo de transporte não reage à perda de pacotes com a diminuição da taxa de transmissão de pacotes, como é o caso do UDP, o RED não terá nenhum efeito positivo, podendo inclusive deteriorar o desempenho do sistema pelo aumento da taxa de perda de pacotes [35].

4.3.2.1.1 - DESCRIÇÃO DO ALGORITMO

O algoritmo RED consiste de duas etapas. Na primeira, é feita a estimativa do tamanho médio da fila e na segunda é tomada a decisão de marcar ou descartar os pacotes que chegam [25].

O cálculo do tamanho médio das filas é feito utilizando-se uma fórmula de média ponderada móvel (*weighted moving average*), ou seja, um filtro passa-baixa é utilizado no cálculo do tamanho médio da fila para filtrar do cálculo da média congestionamentos passageiros. O valor obtido para o tamanho médio da fila é utilizado na segunda fase do algoritmo para decidir o que fazer com os pacotes.

O valor do tamanho médio da fila é comparado com dois *thresholds* min_{th} *e* max_{th}. Estes *thresholds* são utilizados para definir três zonas. Se o valor estiver abaixo de min_{th}, o algoritmo está na zona normal de operação, nenhum pacote é marcado ou descartado e todos os pacotes que chegam são aceitos. Caso o valor esteja entre os dois *thresholds*, o algoritmo está na zona de prevenção de congestionamento e cada pacote que chega é marcado ou descartado com uma probabilidade p_a. Se o valor estiver acima de max_{th}, o algoritmo está na zona de controle de congestionamento e todos os pacotes que chegam são descartados.

A probabilidade p_a cresce de acordo com dois fatores: um contador que é incrementado a cada vez que um pacote chega no roteador, sendo armazenado na fila e é reinicializado quando um pacote é descartado; uma probabilidade intermediária p_b, cujo valor máximo, max_p, é atingido quando o valor do tamanho médio da fila é igual a max_{th}. Quanto maior o crescimento do tamanho médio da fila, maior é a probabilidade de descarte p_a. Se o tamanho médio da fila é mantido constante, todos os pacotes que chegam têm a mesma probabilidade de descarte. Desta forma, RED descarta pacotes em proporção à utilização de banda de conexão.

O cálculo do tamanho médio da fila utiliza uma fórmula de média ponderada móvel com peso, W_q, para filtrar do cálculo congestionamentos passageiros. No algoritmo apresentado a seguir, pode-se verificar que o cálculo do tamanho médio da fila leva em consideração o período em que a fila ficou vazia (*idle period*), através da estimativa do número de pacotes que foram transmitidos durante este período.

O valor W_q determina a forma como o algoritmo responde às mudanças no tamanho da fila. Se W_q for muito alto, o uso da média não consegue filtrar os congestionamentos passageiros. Se ao contrário, o valor escolhido for muito baixo, a reação ao congestionamento é muito lenta. A escolha do valor correto depende do valor de min_{th} e da quantidade de pacotes em rajada que se deseja suportar. Desta forma, dado a quantidade de tráfego em rajada que se deseja suportar em número de pacotes, P e min_{th}, W_q pode ser calculado por:

$$P + 1 + \frac{\left(1 - W_q\right)^{(P+1)} - 1}{W_q} < min_{th}.$$

Isto significa que se W_q for escolhido de forma adequada, o roteador pode aceitar uma rajada de até P pacotes. Segundo [36] na maioria dos casos estudados o valor utilizado é $W_q = 0.002$. Este trabalho também sugere utilizar um max_{th} igual a pelo menos o dobro de min_{th}.

Dado o conceito, a implementação do algoritmo é bastante simples. O algoritmo RED implementado no modelo de roteador desenvolvido neste trabalho é a implementação proposta em [36]. Abaixo, na Figura 10, temos um resumo de sua implementação:

Inicialização:

avg ←0

count ←−1

Para cada pacote que chega:

Calcula o novo tamanho médio da fila:

Se a fila não está vazia

$avg \leftarrow (1 - W_q)avg + W_q.\ q$

senão

$m \leftarrow f\ (time_now - q_time\ )$

$avg \leftarrow (1 - W_q).m.\ avg$

se $min_{th} \leq avg < max_{th}$

incrementa count

calcula a probabilidade pa:

$p_b \leftarrow max_p(avg - min_{th})/(max_{th} - min_{th})$

$p_a \leftarrow p_b\ /(1 - count \cdot p_b\ )$

com probabilidade p_a*:*

marca o pacote que chega

count ←0

senão se $max_{th} \leq avg$

marca o pacote que chega

count ←0

senão

count ←−1

quando a fila ficar vazia

q_time ←time_now

Variáveis armazenadas:

avg: tamanho médio da fila

q_time: instante em que a fila fica vazia

count: pacotes que chegaram desde que o último pacote foi marcado

Parâmetros fixos:

W_q: peso da fila

min_{th} : valor minimo do limitante da fila

max_{th} : valor máximo do limitante da fila

max_p: valor máximo de p_b

Outros:

p_a: corrente probabilidade de marcar o pacote

q: tamanho da fila atual

time_now: instante de tempo atual

f (t): uma função linear de t

Figura 10 - Implementação do algoritmo RED

4.3.3 - O ALGORITMO WRED

No algoritmo WRED, a probabilidade de um pacote entrante ser descartado será definida pela taxa de ocupação da fila, e por um peso associado ao fluxo (ou classe de fluxo) ao qual o pacote pertence. O que se busca com o WRED é que pacotes de maior prioridade tenham menor probabilidade de descarte. Por exemplo, uma probabilidade de descarte menor pode ser associada a fluxos de pacotes com paior prioridade (determinada pelo conteúdo dos bits de Precedência do campo ToS do cabeçalho IP), ou fluxos de pacotes que fizeram reserva de recursos (através do protocolo RSVP). A Figura 11 ilustra a operação do algoritmo WRED [35].

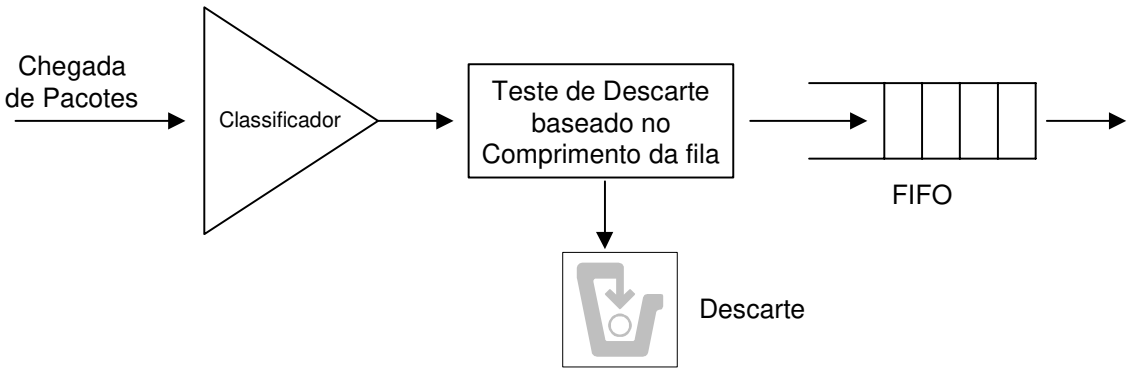

Figura 11 - Operação do algoritmo WRED

4.3.3.1 - WRED E VoIP

O uso do algoritmo WRED não resulta na priorização estrita que o tráfego de voz requer. No entanto, o WRED pode prover um tratamento preferencial aos pacotes de voz durante situações de congestionamento, minimizando a perda destes pacotes pelo descarte antecipativo de pacotes de dados, aos quais se atribui uma maior probabilidade de descarte. Deve-se lembrar que o descarte de um pacote de voz não reduzirá o fluxo de chegada deste tipo de pacote, uma vez que o UDP não reage à perda de pacotes. Portanto, um fluxo muito

pesado de tráfego de voz pode causar o overflow em uma fila WRED e conseqüentemente uma elevada taxa de perda de pacotes. Ainda, se a probabilidade de descarte, pelo WRED, associada a pacotes de voz não for muito baixa, podemos ter uma taxa de pacotes inaceitável para este tipo de tráfego e a consequente perda do nível de QoS [35].

4.4 - QUALIDADE DE SERVIÇO NO PROCESSAMENTO DIGITAL DE SINAIS PARA TELEFONIA IP

A telefonia IP promete conquistar as comunicações telefônicas, assim como o protocolo IP conquistou as comunicações de dados. Entre os fatores que nos levam a acreditar nessa afirmação podemos citar:

- Os custos envolvidos em uma conversação telefônica através de voz sobre IP são inferiores ao de uma conversação via rede telefônica pública comutada.
- Possibilidade de compartilhamento da rede de voz com a rede de dados.
- Redução de custos operacionais através da unificação das redes de transporte, sinalização e gerência.
- Possibilidade de supressão de silêncio e compactação da voz digitalizada, permitindo um uso mais eficiente da largura de banda disponível.
- Reutilização da infra-estrutura de rede de dados já instalada.
- Facilidade para oferecer outros serviços tais como o correio de voz, call center via Internet, segunda linha virtual, e até mesmo serviços como e-mail e fax.

A telefonia IP consiste no transporte do fluxo de voz gerado por uma conversação telefônica através da rede IP. Para que seja possível este transporte, em ambos os terminais os sinais analógicos de voz devem ser codificados/decodificados em cadeias de bits, e estes são o objeto transportado pela rede de dados.

Para que seja possível uma conversação de voz sobre IP, são necessários protocolos de controle e sinalização para a execução de tarefas como localização do usuário a ser

chamado, notificação de chamada, início da transmissão de voz, desconexão, etc. Muitas soluções proprietárias existem para estas funções, mas verificamos hoje o advento de dois padrões: o padrão H.323, proposto pela ITU-T (International Telecommunication Union), e o padrão SIP (Session Initiation Protocol), proposto pelo IETF (Internet Engineering Task Force).

4.5 - CLASSIFICAÇÃO DOS PADRÕES DE CODIFICAÇÃO DE VOZ

Para a conversão da voz em sinais digitais e vice-versa, são empregados codificadores de voz (*vocodes*) capazes de codificar a voz humana.

A compressão é a chave para permitir a tecnologia capaz de prover novos serviços de voz baseados em redes de dados. Os processadores de sinal digital (DSP – *Digital Signal Processors*) têm progredido a ponto de suportar milhões de instruções por segundo, o que permite atingir uma boa qualidade na voz. Apesar da tecnologia de análise de voz e sintetização de voz através de *vocoders* ter sido muito utilizada, esta tem um problema, que é a produção de um sinal de voz metálico robotizado. Entretanto, isto tem mudado dramaticamente nos últimos anos. Um grande esforço tem sido feito para testar a qualidade de voz produzida pelos algoritmos propostos [16].

Basicamente, utilizam-se duas técnicas para codificação da voz humana: uma com base na forma de onda (*Waveform Coding*), e outra com base nos padrões de voz (*Vocoding*).

Na codificação da forma de onda (ou codificação não paramétrica), o processo é mais simples e direto: a idéia é codificar o sinal de voz de forma a ter um sinal reproduzido cuja forma de onda se assemelha ao máximo à do sinal original. Esta técnica é usada quando uma qualidade elevada do sinal é requerida.

Na codificação da fonte (ou codificação paramétrica), o processo considera quase que apenas as características do sinal de voz, sem se preocupar com a forma de produção do sinal amostrado. Entretanto, como o processo da produção da voz humana é complexo, a modelagem aproximada e simplificada se traduz em um processo de codificação com

qualidade inferior ao da codificação não paramétrica, mas com a vantagem de permitir uma considerável redução das taxas de transmissão.

A tabela abaixo mostra uma comparação entre os métodos de codificação mais utilizados sobre redes de dados.

Método de Compressão	Taxa (Kbps)	Retardo (ms)	Complexidade
G.723.1	5,3-6,3	30	Muito alta
G.729/G.729A	8	10	Alta
G.728	16	2,5	Média
G.727	32	1	Baixa
G.711	64	0,75	Muito baixa

Tabela 2 - Comparação de Atributos de Codecs de Áudio

No padrão G.711, o sinal de voz na faixa de 100 a 3400 Hz é amostrado a 8 KHz com resolução de 13 (lei *u*) ou 14 (lei A), sofrendo uma compressão logarítmica do tipo *u*-low ou A-low para 8 bits. Este tipo de compressão é adequado, uma vez que a sensibilidade acústica é logarítmica e não linear, sendo que os sinais de baixa amplitude são mais frequentes e mais importantes.

No padrão G.723 emprega-se a técnica de codificação paramétrica. Neste caso, o codificador de voz analisa o espectro do sinal de voz identificando os parâmetros que são entendidos pelo ouvido. Tais parâmetros são transmitidos e usados na recepção para sintetizar o padrão da voz. A forma da onda do sinal de voz resultante pode não se assemelhar ao original, mas as diferenças são aceitáveis para a aplicação.

O padrão G.723 usa o modelo de predição, isto é, geração de códigos baseada em análise de amostras dos padrões repetitivos conhecidos como *pitches*. A cada janela de 30 milissegundos do sinal de voz, são analisadas 240 amostras de 16 bits do sinal de voz (amostrados a 8 KHz) para identificação dos *pitches* que melhor representam a amostra, gerando dez ou doze códigos de 16 bits, conforme a configuração do algoritmo, para uma taxa de 6,3 ou 5,3 Kbps.

4.6 - AVALIAÇÃO DA QUALIDADE DOS ALGORITMOS DE COMPRESSÃO DE VOZ

A medida usada para comparar estes algoritmos procura aferir de que maneira a voz soa sob condições ideais, como uma voz pura, sem erros de transmissão e somente um tipo de codificação.

O teste mais usado é o chamado ACR (*Absolute Category Rating*). Neste um ouvinte escuta cerca de 8 a 10 segundos de uma amostra de voz, após o quê, classifica a qualidade do material que ele ouviu. Normalmente é utilizada uma escala de pontuação para representar os níveis de qualidade apurados, desde 5 (excelente) até 1 (ruim). Através da associação de valores numéricos a cada categoria, é computado o que se chama de *Mean Opinion Score* (MOS) para cada método de codificação, através da média desta pontuação. Normalmente, o teste inclui uma seleção diversa de material. A tabela a seguir mostra o MOS dos algoritmos de compressão de voz citados anteriormente [16].

Método de Compressão	**Taxa (Kbps)**	**MOS**
G.711	64	4,1
G.727	32	3,8
G.728	16	3,6
G.729	8	3,9
G.729A	8	3,7
G.723.1	5,3-6,3	3,9

Tabela 3 - Comparação dos MOS de Codecs de Áudio

É razoável de se esperar que quanto mais o sinal de voz é comprimido, maior é a perda da sua inteligibilidade, como podemos verificar na tabela acima. Desde o G.711 até o G.728 temos um aumento de compressão de 2:1, com a respectiva queda de qualidade, segundo o critério MOS de 4,1 para 3,8. Isto é comum em algoritmos de compressão do tipo Waveform Coding.

Entretanto, o emprego da compressão nas técnicas de Vocoding, não causa uma perda tão significativa da qualidade do sinal de voz, como é o caso do padrão G.723.1 cuja compressão em relação ao PCM (64 Kbps) aumenta para 5,3/6,3 e o índice MOS cai apenas de 4,1 para 3,9. Portanto para aplicações que transmitem voz sobre redes de dados, as técnicas de compressão com Vocoding são as mais indicadas.

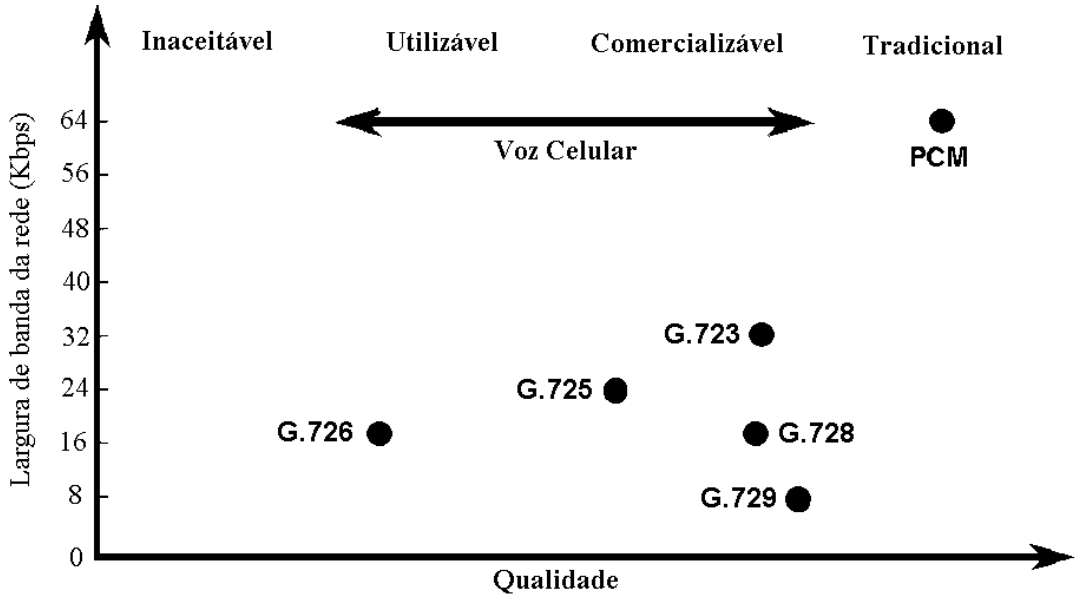

Figura 12 - Qualidade dos padrões de compressão

4.7 - QUALIDADE DO SINAL DE VOZ

O atraso e a variação de atrasos percebidos em um ambiente de telefonia IP são causados pelo acúmulo de uma série de pequenos atrasos. Os principais destes pequenos atrasos são [15]:

- Atraso da rede de transporte.
- Atraso de formação de pacote.

O atraso da rede está vinculado ao processo de transmissão dos pacotes pela rede de transporte, e inclui:

- Atraso de acesso ao meio: por exemplo, cada byte tem um tempo de transmissão de 0,35 segundos em um modem analógico de 28,8 Kbps.
- Atraso de rotamento: atraso dos pacotes que aguardam serviço nas filas dos roteadores.
- Atrasos em firewall e Proxy: visando adicionar segurança a uma rede, equipamentos do tipo firewall e proxy podem estar presentes no caminho

dos dados. Tais equipamentos adicionam mais filas ao sistema e identificam todos pacotes que passam por eles, o que pode demandar tempo considerável dependendo do volume de tráfego e da capacidade destes equipamentos.

O atraso de formação do pacote é o tempo demandado para o preenchimento de um pacote de voz a ser transmitidos. O tamanho do pacote correspondem ao tamanho do vetor ou célula gerado pelo vocoder multiplicado pelo número de células que compõem um pacote (a maior parte dos sistemas usam uma célula por pacote). Atrasos médios são da ordem de 20 a 30 milissegundos para a formação de uma célula [15].

A ITU-T estabelece os seguintes intervalos, em milissegundos, para o atraso e os seus respectivos graus de satisfação:

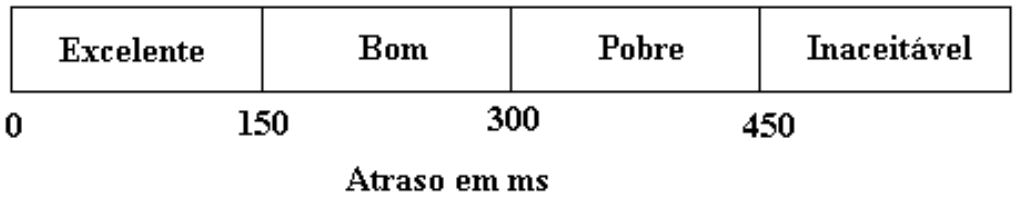

Figura 13 - Intervalos para o atraso e os seus respectivos graus de satisfação

CAPÍTULO 5. DESENVOLVIMENTO DO SIMULADOR *LSIMNT*

5.1 - ESTRUTURA PROPOSTA PARA O SIMULADOR DE REDES

Na seqüência, serão vistos alguns detalhes do simulador de redes para ***Linux***, o ***lsimnt***, implementado neste trabalho.

O ***lsimnt*** foi implementado a partir do trabalho [1], em que implementou-se o simulador de redes ***Hydragyrum***, um simulador de redes orientado a eventos discretos para o sistema operacional ***Windows***, com enfoque em redes orientadas a conexão, como por exemplo redes ATM. Porém, o ***Hydragyrum*** foi desenvolvido especificamente para o sistema operacional ***Windows***, e sua API permite apenas a criação de modelos na forma de bibliotecas dinâmicas para ***Windows*** (DLLs). Isto tem sido visto pelo nosso grupo de estudos em simulação de redes como uma lacuna para a comunidade de usuários do sistema operacional ***Linux***.

Visando preencher esta lacuna, foi desenvolvido o ***lsimnt***. Este simulador foi construído de forma modular, e a sua estrutura pode ser dividida basicamente em duas partes:

- **Núcleo (*kernel*) do simulador**: é responsável pelo gerenciamento e execução da simulação. O núcleo está embutido no aplicativo ***lsimnt***, o qual é responsável também pela interpretação do arquivo que descreve a topologia da rede a ser simulada, e pela carga dos modelos dos elementos de rede da topologia; e
- **Modelos dos elementos de rede**: formam os blocos do simulador responsáveis pelo comportamento dos modelos. Todos os modelos do ***lsimnt*** são construídos como bibliotecas dinâmicas para o ***Linux*** (arquivos .so).

5.2 - O NÚCLEO DO SIMULADOR

O núcleo é o componente mais importante do simulador. O núcleo é implementado no programa principal do simulador. Entre as principais funções do núcleo estão:

- **Instância dos modelos**: como os modelos são construídos em classes da linguagem C++, através da interpretação dos comandos lidos de um arquivo de configuração da simulação, o núcleo é responsável pela carga das bibliotecas dinâmicas correspondentes aos modelos, e pela criação das instâncias dos objetos que representam os elementos de redes modelados para a simulação;
- **Gerenciamento dos eventos**: sendo o simulador orientado a eventos, o núcleo é responsável por manter a fila de eventos, e pela entrega dos eventos ao objeto de destino no instante de tempo de simulação apropriado. Cabe destacar que para o tratamento dos eventos, o núcleo mantém um objeto escalonador (*Scheduler*), e um relógio para contar o tempo de simulação; e
- **Interpretação de comandos**: a topologia da rede é descrita em um arquivo com a extensão .SCL. A descrição da topologia é descrita usando uma linguagem de simulação denominada SCL (*Simulation Command Language*), a mesma linguagem usada para configurar as simulações no simulador ***Hydragyrum***, cuja sintaxe é descrita no apêndice A. O núcleo do simulador é responsável pela leitura e interpretação do arquivo, e pela conseqüente execução da simulação.

O simulador também contém as estruturas de dados e classes auxiliares disponíveis para utilização na construção de modelos para o simulador através da API descrita no apêndice B. Estes elementos podem ser divididos em categorias:

- Elementos de gerenciamento e controle;
- Elementos de modelagem (classes *Block*, *Layer* e *Squeue*);
- Elemento de conexão (classe *Connection*); e
- Elementos para representação de tipos e estruturas de dados (classes *Parameter*, *Table*, *Event, SMessage* e outras).

Mais detalhes destes elementos serão vistos no próximo item.

5.3 - EXECUÇÃO DE SIMULAÇÕES

O simulador consiste em um aplicativo para o sistema operacional ***Linux***. Os modelos usados na simulação consistem em objetos C++, e são implementados na forma de bibliotecas dinâmicas para o sistema ***Linux*** (arquivos com a extensão .so), análogas às bibliotecas dinâmicas do sistema operacional ***Microsoft Windows*** (arquivos com a extensão .dll), o que facilita o desenvolvimento de novos modelos para o simulador.

Para a configuração da simulação, um arquivo descritor do cenário de simulação deve ser previamente criado usando a linguagem SCL, que como vimos, é a mesma linguagem usada para configurar as simulações no simulador ***Hydragyrum***, e sua sintaxe é descrita no apêndice A. O núcleo é responsável pela verificação de sintaxe, tratamento de erros e pela chamada das rotinas associadas aos comandos lidos do arquivo .SCL.

Na construção de modelos é usada necessariamente a classe *Block*. Assim, sempre que for necessária a criação de uma instância de um modelo de rede, uma linha de comando deve ser inserida no arquivo .SCL com dois elementos cruciais:

- Um rótulo usado para identificação única daquele objeto pelo núcleo; e
- O nome do arquivo .so (biblioteca dinâmica) que contém o modelo.

Assim, com interpretação desta linha, o núcleo carrega a biblioteca, e cria uma instância do modelo contido na referida biblioteca.

Sendo a simulação orientada a eventos, toda a simulação é controlada pelo escalonador, o qual retira os eventos da fila no instante de simulação correto, e entrega-os para o objeto destino, onde tais eventos são então tratados pelo respectivo método de manipulação (*handler*). Tipicamente, na inicialização dos modelos, ainda antes da execução da simulação, modelos como fontes de tráfego agendam eventos de laço (*loop events*) para geração de tráfego. No tratamento destes eventos, os objetos podem efetuar algum processamento, escrever o valor de alguma variável em arquivo e/ou agendar novos eventos, contendo ou não mensagens para outros objetos.

Assim, a simulação mantém continuidade, sendo então finalizada em duas situações:

- O instante atual do relógio da simulação corresponde ao tempo total de simulação recebido como argumento pelo aplicativo; ou

- A fila de eventos do escalonador está vazia.

5.4 - O ESCALONAMENTO DE EVENTOS

Como o simulador usa o método de simulação orientado a eventos discretos, o agendador de eventos do simulador é o principal elemento de controle existente no núcleo do simulador. O agendador implementa um fila para os eventos a serem executados durante a simulação. Os eventos na fila de prioridades são ordenados em ordem crescente de tempo de simulação, sendo que o evento que é mantido no início da fila é aquele que possui o menor tempo de simulação agendado. Desta forma, quando for necessário processar um evento, é só retirá-lo do início da fila.

A fila de eventos armazena os eventos que aguardam por execução no *kernel*. Os eventos são ordenados conforme o tempo agendado para a execução. O evento com menor tempo de execução é o primeiro a ser executado. Se vários eventos estiverem agendados para um mesmo tempo de execução, o primeiro evento que foi agendado na fila será executado.

Os eventos são requisições de atendimento, agendadas para um dado instante de tempo, que tem por objetivo executar um processo predefinido. A Figura 14 mostra a estrutura dos eventos no ***lsimnt***.

Time	**Type**	**Source Block**	**Destination Block**	**Source Layer**	**Destination Layer**	**Event Message**

Figura 14 - Estrutura dos eventos do *lsimnt*

Os eventos do ***lsimnt*** possuem os seguintes campos de informações:

- **Time** – Tempo programado para a execução (em segundos). Repare que como a técnica de simulação orientada a eventos é fundamentada nos autômatos estocásticos, não é necessário manter a informação da duração do evento [11];
- **Type** – Este campo serve para identificar o tipo de mensagem que o evento está carregando;
- **Source Block** – Nome do bloco onde o evento foi gerado;

- **Destination Block** – Nome do bloco a quem o evento se destina. Este campo deve ser sempre preenchido, caso contrário, o evento será ignorado pelo *kernel*;
- **Source Layer** – Nome da camada ou sistema de fila onde o evento foi gerado;
- **Destination Layer** – Nome da camada ou sistema de fila ao qual o evento se destina; e
- **EventMessage** – Os eventos são usados para transportar mensagens entre camadas para outra, nos blocos da rede.

Durante a execução da simulação o escalonador retira do início da fila o evento que possui o menor tempo de execução agendado. Este evento é então enviado para o modelo de destino. Quando o evento chega ao modelo de destino, ele é executado e agora o modelo detêm o controle da simulação. Quando termina a execução do evento, o controle da simulação é devolvido ao escalonador, que inicia novamente o ciclo de execução. O esquema do processamento de eventos no simulador está resumido na Figura 15.

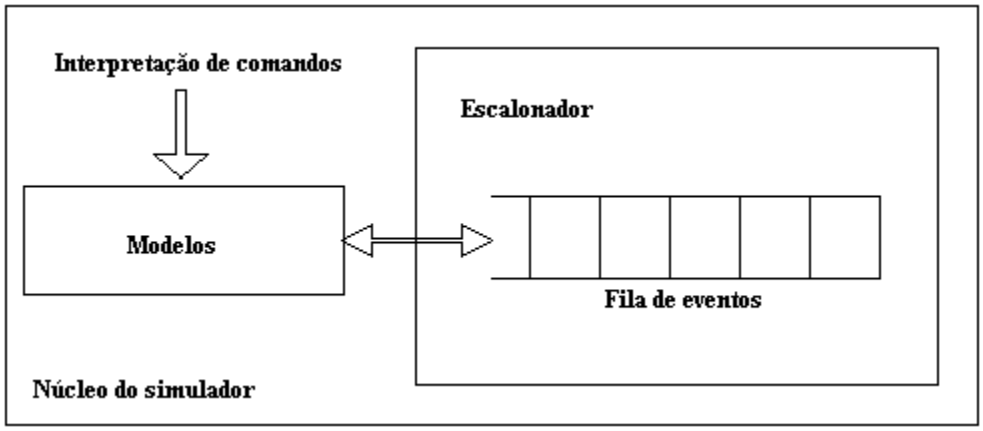

Figura 15 - Execução de eventos dentro do ambiente de simulação.

5.5 - CLASSES DISPONÍVEIS PARA CRIAÇÃO DE MODELOS

As classes disponíveis para a criação de novos modelos são: *Block*, *Layer* e *Squeue*. Todos os modelos são implementados na forma de classes derivadas da classe base *Block*, ou seja, para cada modelo de elemento de rede, o núcleo manterá uma instância de um

objeto derivado de *Block*. Esta classe pode agregar instâncias de objetos derivados das classes base *Layer* e *Squeue*, como mostra a Figura 16.

Figura 16 - Relacionamento entre objetos do tipo *Block*, *Layer* e *Squeue*

5.6 - PROCESSO DA SIMULAÇÃO

As classes base *Block*, *Layer* e *Squeue* possuem alguns métodos virtuais, cuja implementação é mandatória em todas as classes derivadas delas. Tais métodos possuem ordem de invocação relacionada ao comportamento do modelo em uma determinada fase ou procedimento da simulação. Estes métodos são:

- ***virtual int Setup()*:** invocado sempre que um modelo, derivado da classe *Block*, é adicionado à simulação. Para as classes derivadas de *Layer* e *Squeue* a função é chamada no instante em que os objetos destas classes são adicionados a um objeto derivado da classe *Block*. Como o método é chamado logo após a criação de um objeto bloco, camada ou fila, ela propõe-se ao fornecimento de um estado básico (*default*) para os modelos baseados nestes objetos no momento do carregamento do modelo.
- ***virtual int OnConnect(Sstring ConnecName)*:** este método é invocado em objetos derivados das classes *Block* e *Layer*. Isto ocorre porque os objetos das classes *Block* e *Layer* são os que definem os objetos da classe auxiliar *Connection*, que é usada pelo núcleo para representar as conexões existentes entre os modelos de rede. A invocação é efetuada quando uma conexão é criada pelo núcleo, após a leitura de uma linha de comando do arquivo .SCL que instrua a criação da respectiva conexão. Uma conexão é um objeto *Connection* que possui referências a seus blocos (*Block*) e camadas (*Layer*) de iniciais e finais. Na criação da conexão, o método *OnConnect*

(Sstring ConnecName) é chamado em todos este blocos e camadas, levando consigo como argumento o nome da conexão validada, que poderá ser utilizado dentro dos blocos ou camadas, para processamento em tabelas de dados do modelo ou para consultar maiores informações sobre a conexão criada. Outro emprego é, por exemplo, a criação e destruição de novos elementos do modelo, como filas associadas às novas conexões criadas entre modelos. A classe *Connection* será vista com mais detalhes no decorrer do texto.

- ***virtual int Initiate()*:** invocado antes da execução de cada simulação nos modelos (*Block*), em suas camadas (*Layer*) e em seus sistemas de filas (*Squeue*). Este método serve como local para colocação de estados e valores padrão (*default*), que precisam ser inicializados antes da execução de cada simulação. O método *Initiate()* é também o local indicado para a realização de procedimentos como verificação da integridade dos parâmetros da simulação, alocação de variáveis necessárias à simulação, limpeza de contadores e atualização de valores padrão dos parâmetros modificados pelo usuário através de comandos inseridos no arquivo .SCL.
- ***virtual int Run(Event * _Event)*:** o método *Run* de cada modelo (*Block*), camada (*Layer*) e sistemas de filas (*Squeue*) é o local onde se processam os eventos nos objetos que compõem os modelos. O método *Run()* é onde os algoritmos e a lógica dos modelos podem ser definidos. Devido à representação modular e em camadas dos sistemas de redes, em geral a maior parte do código da simulação tende a se concentrar no método *Run* dos objetos que representam as camadas (*Layer*) e os sistemas de filas (*Squeue*) de um modelo (*Block*). No método *Run()*, o evento (*Event*) recebido pelo objeto de destino pode ser aberto e decomposto para retirar e processar toda a informação que o evento transporta, tal como o tempo de ocorrência, o tipo do evento, a mensagem que o evento transporta e em qual conexão de rede ou conexão de dados o evento está sendo transportado.
- ***virtual int Posprocessing()*:** o método *PosProcessing()* é chamada nos objetos do modelo (*Block*, *Layer* e *Squeue*) ao final da simulação. O final da execução da simulação ocorre quando não existem mais eventos a serem processados ou quando a simulação atinge o tempo limite especificado pelo usuário. O objetivo do método *PosProcessing()* é prover a capacidade de realizar uma análise do estado das

variáveis dos modelos logo ao final da simulação e, se necessário, realizar processamentos adicionais como fechamento de arquivos de coleta estatística, geração de sumários das estatísticas de simulação de um modelo e limpeza de memória alocada pelo modelo.

Resumidamente, durante a execução da simulação no ***lsimnt***, duas etapas são processadas:

- **Etapa 1:** Carga dos modelos e configuração da rede para simulação
- **Etapa 2:** Execução propriamente dita da simulação

As etapas são ilustradas na Figura 17. Na etapa 1, é feita a leitura do arquivo .SCL. Inicialmente são criadas instâncias dos objetos que representam os elementos de rede. Imediatamente após a criação das instâncias dos modelos, o método virtual *Setup()* de todos os objetos derivados dos tipos *Block*, *Layer* e *Squeue* é invocado. Neste método tipicamente é feita pelo programador a inicialização de variáveis do modelo.

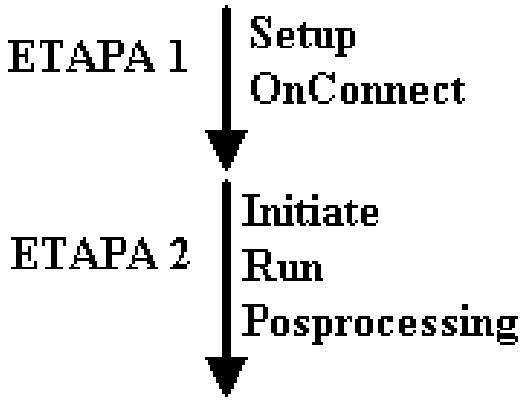

Figura 17 - Etapas da simulação

O próximo passo dentro da etapa 1 consiste na criação das conexões de rede. Tais conexões são também descritas em linhas de comandos existentes no arquivo .SCL, e veremos na seqüência do texto que para cada conexão entre blocos o núcleo cria um objeto

auxiliar do tipo *Connection*, e nele são mantidos ponteiros para os objetos terminais da respectiva conexão. A cada conexão criada, é feita a invocação do método virtual *OnConnect()* dos objetos terminais da conexão, passando como argumento uma referência para o objeto auxiliar do tipo *Connection* da respectiva conexão. Este artifício pode ser utilizado por exemplo para a inicialização de tabelas de rotas.

Ainda na etapa 1, são feitas modificações dos parâmetros por comandos lidos do arquivo .SCL. Tipicamente, valores padrão de variáveis são setados pelo programador no método *Setup()*. Porém, o usuário pode modificar valores padrão de variáveis sem necessitar recompilar sua biblioteca, como por exemplo uma variável usada para configurar a taxa de geração de uma fonte de tráfego. Isto pode ser feito através de comandos para manipulação de parâmetros, que serão vistos na seqüência do texto. Esta estrutura de parâmetros é uma contribuição dada originalmente por [3], e vem sendo usada pelo nosso grupo de pesquisa em simulação há alguns anos.

Já na etapa 2 é feita a execução da simulação propriamente dita. Aqui, o método virtual *Initiate()* de todos os objetos derivados dos tipos *Block*, *Layer* e *Squeue* é invocado. Em seguida, é invocado o método *Run()*, em que é executada a simulação propriamente dita, e finalmente, o método *PosProcessing()* .

5.6.1 - ELEMENTOS DISPONÍVEIS PARA PROGRAMAÇÃO DE MODELOS

As classes *Block*, *Layer* e *Squeue* disponibilizam através de herança uma série de métodos para os modelos, para funcionalidades tais como:

- Manipulação de estruturas de dados;
- Criação, envio e processamento de eventos;
- Troca de mensagens entre modelos; e
- Manipulação de parâmetros do modelo.

Todos os modelos são implementados na forma de classes que estendem a classe *Block*. Cada modelo pode criar novas camadas (classes derivadas da classe *Layer*) e sistemas de filas (classes derivadas da classe *Squeue*). Assim, um modelo pode ser visto

como um bloco em que são agregadas camadas e sistemas de filas, sendo que o comportamento deste aglomerado simula o comportamento do elemento modelado.

Na classe derivada de *Layer* deverão ser implementados os algoritmos que retratam o comportamento do modelo na simulação. Na classe derivada de *Squeue* deverão ser implementados os elementos de armazenamento e serviço das mensagens na simulação.

A divisão das funções originalmente atribuídas a cada classe baseia-se em um modelo genérico para elementos de rede. Na concepção original, estão presentes o elemento de rede (*Block*), seus algoritmos (*Layer*) e suas estruturas de armazenamento (*Squeue*). Entretanto, nada impede que outras funções possam ser desempenhadas por estas classes dentro da estrutura do modelo, uma vez que a definição do comportamento e do algoritmo que será implementado em cada classe é de responsabilidade do projetista do modelo. Portanto, quando se respeitam as funções básicas e a interface das classes de modelagem, qualquer outra funcionalidade não prevista originalmente pode ser desempenhada pelas classes derivadas.

5.6.2 - CONEXÃO ENTRE MODELOS

Após a construção dos modelos a partir da classe base *Block*, vem a necessidade de representar internamente a interconexão entre os blocos, análogo a interconexão entre equipamentos de rede para troca de dados existente em uma rede real. Para tal, foi criada uma classe de interconexão. Os objetos desta classe são criados através de comandos lidos do arquivo .SCL. A interconexão é sempre entre camadas (objetos derivados de *Layer*).

Para cada interconexão a ser criada, deverá existir uma linha no arquivo .SCL, contendo necessariamente 5 elementos:

- Um rótulo usado para identificação única daquela interconexão pelo núcleo do simulador;
- Dois rótulos identificando um dos lados da conexão: um rótulo identifica o bloco de origem, e o outro identificando a camada contida naquele bloco que constitui o lado origem da conexão.

- Dois rótulos identificando o outro lado da conexão: um rótulo identifica o bloco terminal, e o outro identificando a camada contida naquele bloco que constitui o lado terminal da conexão.

Assim, a classe *Connection* pode ser usada, por exemplo, para representar conexões físicas ou de enlaces entre os elementos de redes modelados.

Os objetos de interconexão representam as relações topológicas entre os modelos. Representam pares de conexão bloco fonte e camada fonte ligada ao bloco de destino e camada de destino.

A criação destes objetos é dada na etapa 2 (configuração da simulação), ilustrada na figura 13. A cada objeto *Connection* criado, os métodos *OnConnect()* das camadas (*Layer*) inicial e terminal da respectiva conexão são invocados. O rótulo identificador da conexão é passado como argumento na invocação, como ilustra a Figura 18.

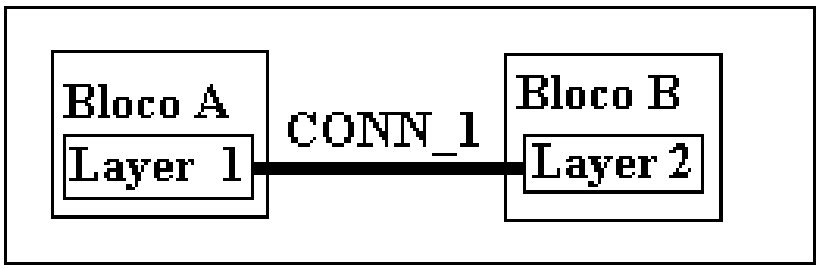

Figura 18 - Conexão entre blocos

No exemplo da figura acima, uma linha é lida do arquivo .SCL contendo o comando abaixo:

create connection CONN_1 Bloco_A Layer_1 Bloco_B Layer_2

O núcleo, ao ler tal comando, cria a instância de um objeto *Connection*, e invoca os métodos *OnConnect()* dos objetos *Bloco_A*, *Layer_1*, *Bloco_B*, e *Layer_2*, passando como argumento a cadeia de caracteres "*CONN_1*".

A maior utilidade da invocação destes métodos é a possibilidade dada aos modelos de saber a quais outros elementos de rede os blocos se encontram conectados, ainda na inicialização, antes da execução da simulação em si. Sendo o simulador orientado a

eventos, os modelo podem usar esta informação para troca de mensagens direta entre blocos conectados, através de métodos disponíveis aos modelos para agendar eventos no núcleo do simulador.

5.7 - ESTRUTURAS DE DADOS AUXILIARES

Algumas estruturas de dados tem sido usadas em nosso grupo de pesquisa na área de simulação há alguns anos, e foram usadas novamente aqui neste trabalho. São elas as classes *Sstring*, *SMessage*, *Param*, *Parameter*, *Joker* e *Table*.

A classe *Sstring* é uma abstração de dados para manipulação de cadeias de caracteres (*strings*), usada na maioria das trocas informações entre os modelos.

As classes de parâmetros (*Param*, *Parameter* e *Joker*) são usadas para armazenar parâmetros usados pelos modelos do simulador. Com eles, variáveis como por exemplo a taxa de geração de mensagens de uma fonte de tráfego podem ser setados através de comandos lidos do arquivo .SCL. Estas estruturas de dados são uma contribuição do trabalho ***SimNT*** [3][4], e tem se mostradas eficientes no tratamento de atualização de variáveis de modelos de outros trabalhos, tais como o ***SimATM*** [2] e ***Hydragyrum*** [1]. Mais detalhes sobre o funcionamento e a API para manipulação de parâmetros constam nos apêndices deste documento.

Na classe *Table* é implementada uma tabela para armazenamento de valores dos tipos *long*, *int*, *double*, *Sstring* e *unsigned int*. Sua estrutura permite que estes tipos de dados sejam indexados por duas chaves de procura na tabela. Esta estrutura é útil para o armazenamento de tabelas de roteamento e associações de sistemas de filas (*Squeue*) e camadas (*Layer*) dentro dos modelos. A estrutura da classe *Table* foi uma contribuição do trabalho da primeira versão do ***SimATM*** [2].

O simulador, orientado a eventos, faz uso de eventos para troca de mensagens entre os elementos da simulação. A classe *Event*, como vimos anteriormente, é usada para representar os eventos trocados na simulação. Os objetos do tipo *Event* contêm informações tais como o tempo em que um dado evento deve ser processado, um ponteiro para o objeto que deverá manipular o evento, um rótulo identificando o tipo do evento, e um ponteiro para a mensagem transportado pelo evento. Este último é um objeto derivado da classe base

SMessage. Os eventos podem ser agendados pelos modelos através de métodos herdados das classes base *Block*, *Layer* e *Squeue*, sendo tais métodos descritos no apêndice B.

A classe *SMessage* é a classe base para a criação de novos tipos de mensagens a serem transportadas pelos eventos do simulador. Exemplos típicos de mensagens que podem ser criadas com a extensão desta classe são objetos para representar pacotes TCP, datagramas IP, ou quadros *Ethernet*, trocados entre elementos de rede.

Uma característica importante a ressaltar na arquitetura do simulador é a forma como as mensagens são definidas, usando o polimorfismo da linguagem C++. No simulador, as mensagens são derivadas da classe base *SMessage* e transportadas dentro dos eventos usando um campo de ponteiro para a classe *SMessage* presente na estrutura do evento (objeto da classe *Event*). Esta característica da linguagem C++ permite que, no ponto de recepção do evento, a mensagem seja extraída do evento e convertida para o tipo de dado apropriado (*casting*). Assim, em termos da linguagem C++, o ponteiro da classe base de mensagens (*SMessage*) é convertido para o ponteiro da classe derivada apropriada, permitindo o acesso direto a todos os dados presentes na classe derivada. O ponto importante desta abordagem de conversão de ponteiros e polimorfismo é que qualquer tipo ou estrutura de dados pode ser passado entre modelos ou dentro de um modelo. Portanto, podemos trocar uma classe inteira com todos os seus elementos de dados, e trabalhar com eles diretamente usando o ponteiro convertido. Esta característica permite uma grande flexibilidade na implementação de protocolos e modelos de redes.

5.7.1 - MODELOS

Os modelos para o simulador são implementados na forma de objetos derivados das classes base *Block*, *Layer* e *Squeue*. O objeto estendido de *Block* é que define o modelo, e nele são agregados os seus componentes, que podem ser objetos derivados das classes *Layer* e *Squeue*. Assim, qualquer elemento de rede poderá ser modelado de forma modular, o que facilita sua implementação.

Tipicamente, elementos de rede como terminais, roteadores IP, comutadores ATM e outros equipamentos de rede podem ser modelados no ***lsimnt*** como classes derivadas de

Block. Tais classes serão compiladas na forma de bibliotecas dinâmicas, para serem carregadas pelo simulador na execução da simulação.

Em programação, entende-se por bibliotecas, arquivos que contêm um conjunto de módulos ou membros de códigos pré-compilados reutilizáveis. Esses códigos podem ser usados por vários programas sem a necessidade de detalhes de sua implementação. A grande vantagem de usar uma biblioteca, é que uma vez fabricada, não será mais preciso compilar, bastando simplesmente usá-la do modo desejado. Dessa forma, existe uma grande vantagem em usar bibliotecas, pois uma vez implementada ou adquirida de terceiros, o desenvolvedor pode se abstrair dos detalhes e concentrar-se somente no problema principal.

Em ***Linux***, pode-se desenvolver dois tipos de bibliotecas. A biblioteca de ligação estática (arquivos com a extensão .a), e a biblioteca de ligação dinâmica (arquivos com a extensão .so). As bibliotecas dinâmicas são ligadas em tempo de execução, ou seja, a ligação ocorre por demanda. Portanto, não fazem parte do programa principal, reduzindo assim o tamanho do arquivo executável. Quando se faz uso de bibliotecas compartilhadas, o programa é dividido em um módulo principal e em um ou mais módulos que serão ligados dinamicamente [18]. No ***lsimnt***, os modelos de rede são implementados como objetos derivados da classe base *Block* compilados na forma de bibliotecas dinâmicas. As bibliotecas são carregadas pelo núcleo no processo de execução da simulação.

Na classe *Block* são agregados os componentes do modelo, objetos derivados de *Layer* e *Squeue*. No modelo de um roteador IP, por exemplo, as camadas de rede e de enlace podem ser derivadas da classe base *Layer*, e o sistema de filas do roteador poderá ser modelado com o auxílio de objetos derivados de *Squeue*. Os quadros trocados entre os modelos pela camada de enlace podem ser objetos derivados da classe base *SMessage*. Tais quadros podem ter em um de seus campos um ponteiro para um objeto do tipo datagrama IP, sendo este também um objeto derivado de *SMessage*, e estes datagramas usados pra troca de mensagens ao nível de camada de rede.

A interface dos modelos com o núcleo do simulador é constituída por uma série de métodos virtuais disponibilizados aos objetos derivados das classes *Block*, *Layer* e *Squeue*. Tais métodos atuam em um contexto de orientação a eventos, e são invocados pelo núcleo quando um comando ou ação especifica é enviada ao núcleo por algum comando lido do arquivo .SCL, ou através de evento agendado no núcleo por algum modelo.

CAPÍTULO 6. EXEMPLO DE APLICAÇÃO

6.1 - CENÁRIO DE SIMULAÇÃO: REDE *DIFFSERV*

Para elaborar o cenário a ser simulado, foi adaptada a idéia do CENÁRIO I implementado no trabalho [33], onde o autor apresenta uma avaliação sobre a implementação de QoS em redes IP através de medições.

O cenário elaborado para ser simulado neste trabalho pode ser representado pela Figura 19.

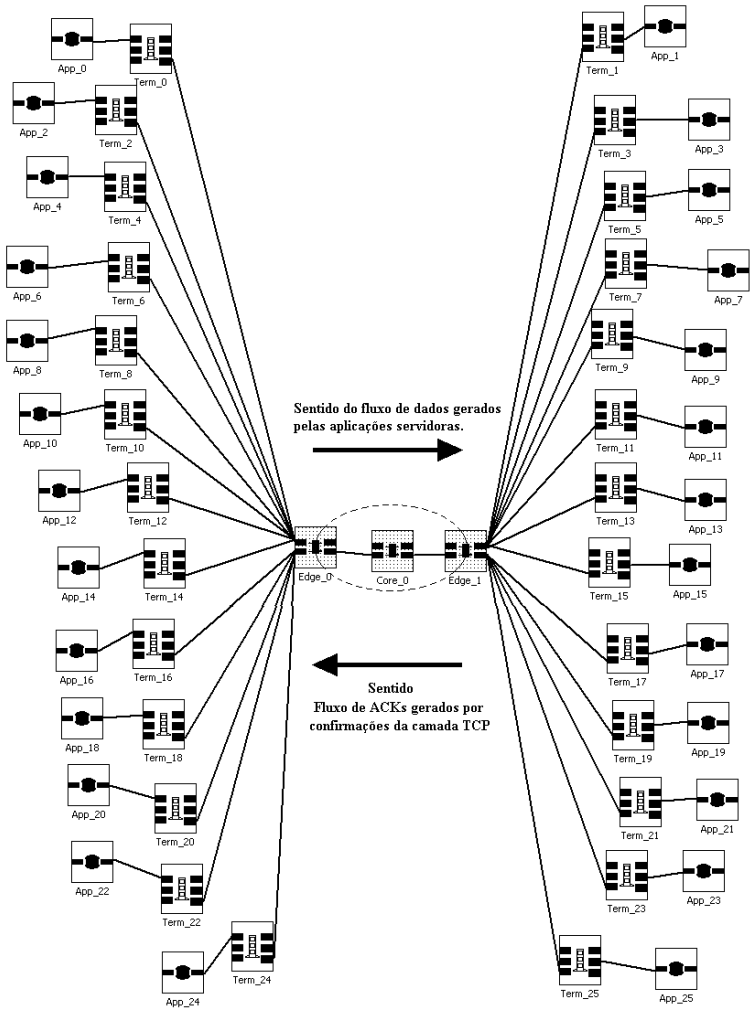

Figura 19 - Cenário de simulação

A topologia é composta de 26 aplicações de rede executadas em 26 terminais TCP/IP. Metade destes terminais estão em uma rede local do lado esquerdo (LEFT-LAN), e os outros 13 terminais restante estão dispostos em outra rede local do lado direito (RIGHT-LAN). As duas redes locais são interconectadas por três roteadores.

As duas redes locais são interconectadas por uma rede *DiffServ*, formada por três roteadores: dois roteadores de borda do domínio *DiffServ*, e um roteador de núcleo.

Para cada terminal existente na rede local do lado esquerdo existe uma aplicação gerando tráfego, conectada via protocolo TCP/IP, a uma aplicação receptora de tráfego situada em um terminal situado na rede local do lado direito. Ou seja, para cada aplicação [i] do lado esquerdo, existe uma correspondente aplicação [i+1] conectada do lado direito, com i variando de 0 a 24. Desta forma, temos um total de 13 conexões TCP estabelecidas através do domínio *DiffServ*.

Considerou-se as conexões entre os roteadores de 50 Kbps, e as conexões existentes nas redes locais de 100 Mbps. Ainda, considerou-se o enlace físico sem perdas.

Para verificarmos os benefícios que arquitetura *DiffServ* apresenta, foram feitas simulações com o *DiffServ* habilitado e simulações sem *DiffServ* (melhor esforço, sem tratamento diferenciado de conexões privilegiadas) em situação de congestionamento. O tráfego de dados unidirecional gerado pelas aplicações situadas na rede do lado esquerdo é gerado em intervalos exponencialmente distribuídos.

Desta forma, dispondo os servidores todos na rede local do lado esquerdo, percebe-se claramente que o gargalo desta rede é a interface de saída de 50 Kbps do roteador de borda Edge_0, pois o fluxo de dados da aplicação é no sentido da esquerda para direita, e o fluxo no sentido contrário é constituído somente por ACKs gerados para confirmações de pacotes recebidos, gerados pela camada TCP dos terminais.

Assumimos também que as conexões TCP já estejam estabelecidas no inicio da simulação, e que nunca sejam finalizadas durante a execução da simulação (situação em regime).

6.2 - MODELOS DESENVOLVIDOS PARA SIMULAR O CENÁRIO

Para tornar possível simular o cenário proposto na Figura 19, percebeu-se a necessidade de implementar os seguintes modelos:

- **Aplicação** (implementado na biblioteca dinâmica **application.so**);
- **Terminal** (implementado na biblioteca dinâmica **terminal.so**); e
- **Roteador** (implementado na biblioteca dinâmica **router.so**).

No desenvolvimento destes modelos para o cenário proposto, buscou-se tomar apenas os elementos essenciais para compreensão da dinâmica dos protocolos envolvidos. Veremos na seqüência mais detalhes de todos os modelos desenvolvidos.

6.3 - MODELO DE APLICAÇÃO

Este modelo tem como principal função sinalizar a entidade TCP presente no modelo de Terminal ao qual a aplicação está vinculada, se o mesmo funcionará como cliente ou servidor. Além disso, este modelo sinaliza para o Terminal as informações de endereço IP e porta TCP dos pontos terminais da conexão TCP usada por esta aplicação, bem como a classe de serviço utilizada.

Para cada bloco exercendo a função de aplicação servidora, haverá necessariamente um bloco com a função de aplicação cliente correspondente. Assumiu-se tráfego de dados TCP sempre unidirecional, com a **aplicação servidora enviando dados para a aplicação cliente**. O único tráfego existente no sentido contrário (do cliente para o servidor) é o gerado pelas confirmações do TCP (ACKs).

O usuário configura através de comandos no arquivo .SCL qual o papel que a respectiva fonte assumirá (cliente ou servidora), bem como as informações de endereço IP e porta TCP dos pontos terminais da conexão TCP usada por esta aplicação, e a classe de serviço utilizada. Estas informações serão usadas pelas camadas de transporte e de rede existentes na implementação do TCP/IP feita no bloco Terminal.

O modelo de aplicação possui seu código distribuído nos arquivos listados na Tabela 4:

ARQUIVO	DESCRIÇÃO
libApplication.cpp	Implementação dos métodos da classe Application_Block, o nosso modelo de aplicação. Ele é constituído de uma única camada, chamada Application_Layer
application_block.h	Declaração da classe Application_Block
application_layer.cpp	Implementação dos métodos da classe Application_Layer, a camada de aplicação
application_layer.h	Declaração da classe Application_Layer
application_buffer.h	Declaração e implementação da classe Application_Buffer, que é o tipo de mensagem que a aplicação usa para se comunicar com outros blocos

Tabela 4 - Arquivos com o código do modelo de aplicação

As classes e os relacionamentos deste modelo são ilustrados na Figura 20. No lado esquerdo temos as classes que integram o modelo de aplicação, e no lado direito as mensagens utilizadas para comunicação deste modelo (buffer da aplicação).

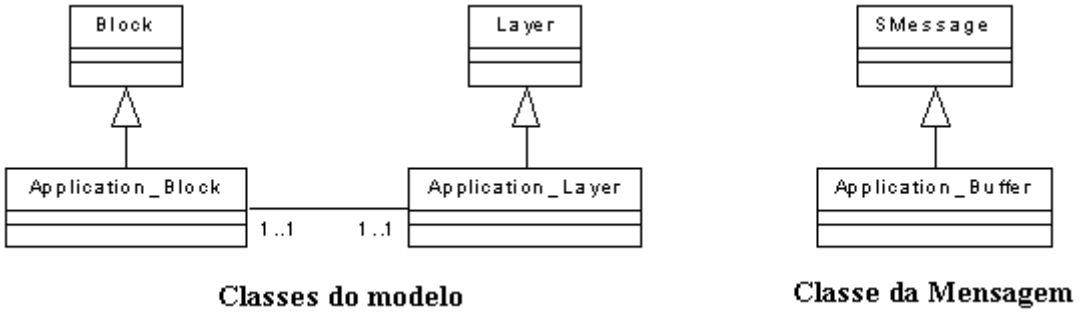

Figura 20 - Classes para o modelo de aplicação

6.4 - MODELO DE TERMINAL

No modelo de terminal, buscou-se implementar algumas das funcionalidades existentes nas camadas existentes no protocolo TCP/IP:

- **Transporte (TCP)**: foi implementado uma classe, derivada de *Layer*, para simular o processamento do protocolo TCP no lado do terminal servidor (classe *TCP_Server*), e outra para simular o processamento do protocolo TCP no lado do terminal cliente (classe *TCP_Client*),
- **Roteamento (IP):** foi implementado uma classe, derivada de *Layer*, para simular o processamento do protocolo IP nos terminais (classe *IP_Layer*),
- **Enlace:** foi implementado uma classe, derivada de *Layer*, para simular o processamento a nível de enlace nos terminais (classe *Link_Layer*),

As classes e os relacionamentos deste modelo são ilustrados na Figura 21. No lado esquerdo temos as classes que integram o modelo Terminal (camadas TCP, IP e enlace), e no lado direito as mensagens utilizadas para comunicação deste modelo (pacote TCP, datagrama IP e quadro).

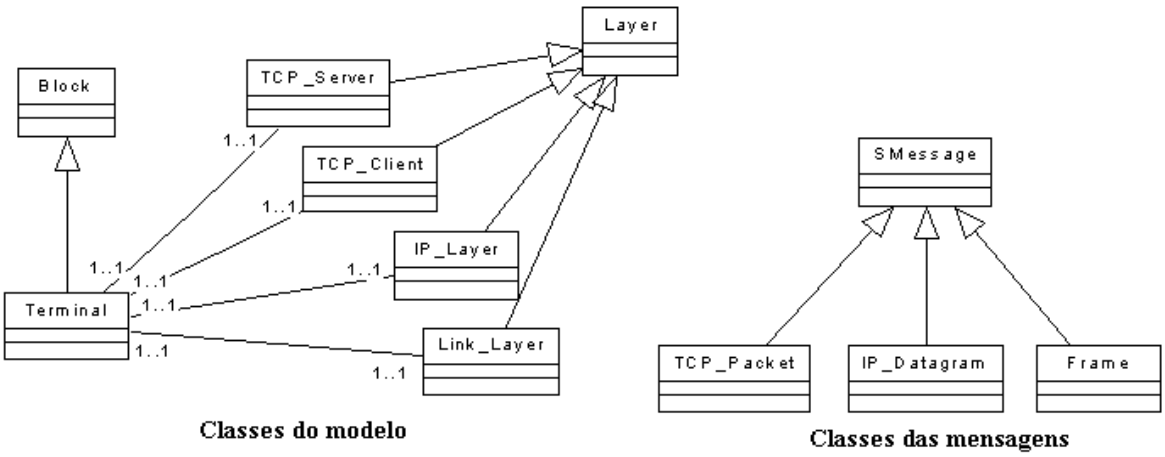

Figura 21 - Classes que integram o modelo de terminal

O modelo de terminal possui seu código distribuído nos arquivos listados na Tabela 5.

ARQUIVO	DESCRIÇÃO
libTerminal.cpp	Implementação dos métodos da classe Terminal, o nosso modelo de terminal.
terminal.h	Declaração da classe Terminal
tcp_layer_server.cpp	Implementação dos métodos da classe TCP_Server, que assume o papel da entidade TCP em um terminal do tipo servidor
tcp_layer_server.h	Declaração da classe TCP_Server
tcp_layer_client.cpp	Implementação dos métodos da classe TCP_Client, que assume o papel da entidade TCP em um terminal do tipo cliente
tcp_layer_client.h	Declaração da classe TCP_Client
ip_layer.cpp	Implementação dos métodos da classe IP_Layer, na qual é implementado o processamento do protocolo IP em um terminal
ip_layer.h	Declaração da classe IP_Layer
link_layer.cpp	Implementação dos métodos da classe Link_Layer, em que é implementada a camada de enlace em um terminal
link_layer.h	Declaração da classe Link_Layer
tcp_packet.h	Declaração e implementação da classe TCP_Packet, que é o tipo de mensagem que o terminal usa para se comunicar a nível de camada TCP
ip_datagram.h	Declaração e implementação da classe IP_Datagram, o tipo de mensagem que o terminal usa para se comunicar a nível de camada IP
frame.h	Declaração e implementação da classe Frame, o tipo de mensagem que o terminal usa para se comunicar a nível de enlace

Tabela 5 - Arquivos com o código do modelo de terminal

Veremos mais detalhes das camadas do modelo de terminal na seqüência.

6.4.1 - MODELO DA CAMADA DE TRANSPORTE TCP

O processamento do protocolo TCP é a implementação mais complexa dentro dos modelos desenvolvidos neste trabalho. Como vimos, foram implementadas duas camadas para simular o processamento do protocolo TCP: uma para a entidade TCP receptora de tráfego, presente no lado do cliente (*TCP_Client*), e uma entidade TCP geradora de tráfego, presente no lado do servidor (*TCP_Server*).

Na implementação deste protocolo, assumiu-se que as conexões TCP já estejam estabelecidas no inicio da simulação, e que nunca sejam finalizadas durante a execução da simulação. Também, assumiu-se que o receptor sempre tenha buffer disponível para armazenamento de dados recebidos (janela de recepção infinita)

6.4.1.1 - ENTIDADE GERADORA DE TRÁFEGO - *TCP_SERVER*

Nesta camada são gerados os pacotes transmitidos para a entidade TCP cliente. O modelo foi construído de forma tal que todos os pacotes transmitidos tenham o mesmo tamanho (um MSS), e a distribuição estatística dos **intervalos entre pacotes** sucessivos obedece a uma distribuição de probabilidade **exponencial negativa** [10] [12].

Cada pacote transmitido é marcado com um *timeout*. Caso o pacote não seja confirmado a tempo, ele será retransmitido. O *timeout* é atribuído ao campo *rtime* do pacote, com o valor do tempo corrente acrescido de duas vezes o RTT (*Round Trip Time*) da conexão.

O tamanho da janela da transmissão, mantido na variável ***cwnd***, é iniciado em um MSS. Ao ser transmitido o primeiro segmento, se o mesmo for confirmado antes de ocorrência de *timeout*, a janela é incrementada de um MSS, e o transmissor transmite então o equivalente a dois MSS (dois pacotes). À medida que cada um dos pacotes é confirmado, a janela de congestionamento é aumentada eu um MSS.

Assim, a janela de congestionamento mantém seu crescimento exponencial até que ocorra um *timeout*, ou quando o valor do limitante (*threshold*) é alcançado:

- se ocorrer *timeout*, o valor do limitante cai para a metade do valor da janela de transmissão (*cwnd*), e *cwnd* é reiniciado em um MSS;
- se o valor do limitante (*threshold*) é alcançado, o crescimento exponencial de *cwnd* é interrompido, e passa a ser linear (aumenta agora em um MSS para cada **janela** reconhecida, e não mais em um MSS para cada **pacote** reconhecido). Ou seja, a cada confirmação de pacote recebida, o valor da janela de transmissão *cwnd* é aumentada. Este aumento é de um MSS para cada pacote confirmado se *cwnd* estiver abaixo do limitante. Caso contrário é de um MSS para cada janela confirmada, e neste caso o valor de *cwnd* só é incrementado se o ACK recebido é o correspondente ao último pacote da janela transmitida.

Todos os pacotes em trânsito (transmitidos, porem não confirmados) são mantidos em uma lista, *unacked_list*. Periodicamente, o evento *timeout* ocorre nesta camada, e no seu

processamento a lista de pacotes não confirmados (*unacked_list*) é percorrida. Todos os pacotes cujo campo *rtime* é maior que o instante de tempo corrente são retransmitidos, e o valor do *timeout* do pacote (campo *rtime*) é dobrado.

A cada confirmação de pacote recebida, o pacote correspondente é removido da lista de pacotes não confirmados (*unacked_list*), e o valor do RTT é atualizado. Para tal, determina-se o atraso (*delay*) decorrido entre o instante da transmissão do pacote e o recebimento da confirmação, e aplica-se a fórmula [29]:

$$RTT = 0.9 * RTT + 0.1 * (Delay)$$

Seguindo a RFC 1122, nos primeiros pacotes enviados pelo transmissor, quando ainda não tiver sido calculado o RTT, o tempo usado para *timeout*, atribuído ao campo *rtime*, é de 3 segundos.

ACKs duplicados recebidos pelo transmissor são simplesmente descartados.

6.4.1.1.1 - IMPLEMENTAÇÃO DA CAMADA TCP_SERVER

O tamanho da janela da transmissão, mantido na variável ***cwnd***, é iniciado em um MSS no método *Initiate()*. Também são inicializados o limitante (***trsh***), o contador de pacotes transmitidos mas não reconhecidos ainda (***una***), e o contador para indicar a seqüência do último pacote transmitido.

O método *Run()*, responsável pela manipulação de eventos, é listado parcialmente na Figura 22. O evento *WRITE_APPLICATION* é agendado pela aplicação a qual o terminal está vinculada. O evento *TRANSMITION_TIMEOUT* ocorre periodicamente a cada 100 ms, e nele são retransmitidos todos os pacotes cujo *timeout* tenha expirado. O evento *READ_TCP* é agendado pela camada IP do terminal sempre que for recebido um pacote do enlace.

```
Run(Event * Evnt)
{
   if (Evnt == "READ_TCP")
         Acknowledge( E->GetEventMessage() );
   if (Evnt == "WRITE_APPLICATION")
         Transmit();
   if (Evnt == "TRANSMITION_TIMEOUT") {
         CreateLoopEvent (time_now + NEXT_TIMEOUT, "TRANSMITION_TIMEOUT");
         Time_Out();
   }
}
```

Figura 22 - Listagem parcial do método *Run()*

O método *Transmit()* é invocado inicialmente na manipulação do evento *WRITE_APPLICATION*. No método, ocorre a transmissão dos pacotes correspondentes ao tamanho da janela de transmissão (***cwnd***). No método, um laço gera os ***cwnd*** pacotes para transmissão, com intervalos entre sucessivos pacotes sendo exponencialmente distribuídos. Os pacotes são inserindos na lista de pacotes transmitidos (*unacked_list*), e transmitidos com o agendamento de um evento para a camada IP.

O método *Acknowledge()*, é invocado na manipulação do evento *READ_TCP*. Este método somente é executado no lado do servidor. Como a transmissão de dados da aplicação é sempre unidirecional, do servidor para o cliente, este método apenas processa os ACKs recebidos do cliente (quadros não contém dados, mas somente os cabeçalhos TCP/IP).

No processamento do método *Acknowledge()*, os pacotes reconhecidos são removidos da lista *unacked_list*. Os ACKs válidos são usados para atualizar o valor de RTT (*Round Trip Time*), e o algoritmo *Slow Start* é aplicado:

- Todos os ACKs recebidos implicam no aumento de 1 (um) MSS no valor da janela de transmissão (***cwnd***) caso o tamanho dela esteja abaixo do limitante (***trsh***). Se ***cwnd*** for superior a ***trsh***, o aumento de ***cwnd*** somente ocorre se o ACK corresponde ao último de uma janela transmitida. Desta forma, se o valor do limitante (***trsh***) for atingido, o incremento da janela de transmissão (***cwnd***), antes exponencial (aumentava a cada pacote reconhecido) passa a ser linear (aumenta agora para cada janela reconhecida).

- Se uma janela previamente transmitida for inteiramente reconhecida, o TCP desbloqueia e transmite a próxima janela, invocando novamente o método *Transmit()*.
- Se for identificada uma retransmissão, assume-se um pacote ter sido perdido, e neste caso, usa-se o algoritmo *Slow Start*: o limitante (***trsh***) assume o valor da metade da janela de transmissão (***cwnd***), e esta é reduzida para 1 (um) MSS.

O método *Time_Out()*, é invocado na manipulação do evento *TRANSMITION_TIMEOUT*. No método, a lista dos pacotes em trânsito (*unacked_list*) é percorrida. Todos os pacotes, cujo campo em que é marcado o instante de tempo para *timeout*, for maior que o instante de tempo corrente, são retransmitidos, e o *timeout* do pacote é então dobrado (*back-off exponencial)*.

6.4.1.2 - ENTIDADE RECEPTORA DE TRÁFEGO - *TCP_CLIENT*

Esta camada simplesmente verifica se os pacotes recebidos do servidor estão na seqüência esperada. Caso positivo, simplesmente responde com um ACK para o servidor. Caso contrário, o pacote é descartado.

6.4.2 - MODELO DA CAMADA DE REDE (IP) DO TERMINAL

A camada de rede recebe os pacotes TCP da camada de transporte, e através das informações de endereço IP de origem e destino da conexão, sinalizadas pelo modelo de aplicação, a camada de rede monta objetos do tipo datagrama IP. Estes objetos serão repassados para camada inferior, de enlace, contendo:

- Os endereços IP de origem e de destino da conexão;
- A classe de serviço (DS *field*); e
- Um ponteiro para o pacote TCP transportado na carga útil do datagrama.

O endereço IP de origem, e o campo DS *field* são usados pelos roteadores para consulta na tabela de rotas, e para o tratamento diferenciado, respectivamente. O ponteiro

para o pacote TCP é usado pela entidade de rede presente no terminal receptor: ela extrai o pacote TCP e entrega-o para a camada superior, a camada de transporte.

6.4.3 - MODELO DA CAMADA DE ENLACE DO TERMINAL

A camada de enlace foi modelada com a introdução de um valor de atraso de processamento fixo nas mensagens transmitidas.

A camada de enlace recebe os datagramas IP da camada de rede, e monta objetos do tipo *frame*. Estes *frames* possuem apenas um ponteiro para o datagramas IP transportado na carga útil do datagrama. A camada então recebe um datagrama, adiciona um atraso de processamento fixo e repassa os *frames*, contendo o datagrama recebido, para o próximo *hop*. O próximo *hop* recebe o *frame* na camada (*Layer*), extrai o datagrama e entrega-o para a camada superior, a camada de rede daquele modelo.

6.5 - MODELO DE ROTEADOR

O modelo de roteador é composto de uma camada de rede (*IP_Router*), e de uma ou mais camadas de enlace (*Link_Router*), sendo uma para cada interface do roteador. Também, cada camada de enlace possui uma fila para cada classe de serviço.

O roteamento é baseado em consulta a tabela de rotas estática (RIP), sendo que a tabela de rotas é também configurável por comandos existentes no arquivo .SCL.

No processamento, a camada de enlace recebe, de algum modelo qualquer, um *frame* na camada de enlace, e extrai dele o datagrama, entregando-o para a camada superior, a camada de rede daquele modelo.

Os pacotes tratados na camada de rede, com base no campo ToS, são encaminhados para uma das filas da interface para o próximo nó de rede, onde aguardam serviço. Cada interface possui uma fila para cada classe de serviço. As filas são servidas pelo roteador, numa taxa de serviço proporcional a largura de banda reservada para a classe de serviço daquela fila.

A taxa de serviço de cada uma das filas equivale à banda reservada para a sua respectiva classe de serviço. Se uma das classes atingir taxas altas de geração de tráfego,

poderá ocorrer a situação em que sua fila não tenha mais espaço para armazenar novos pacotes, sendo então a classe penalizada com o descarte de pacotes. O tamanho das filas é configurável através de comando para manipulação de parâmetros (arquivo .SCL). O valor padrão adotado foi de 12 datagramas.

Os datagramas IP são tratados da seguinte forma na camada de rede do roteador:

- **Datagramas da classe BE:** são inseridos em uma fila FIFO, onde ficam aguardando serviço. A taxa de serviço desta fila corresponde à largura de banda reservada para esta classe de serviço.
- **Datagramas da classe EF:** são inseridos em outra fila FIFO, onde ficam aguardando serviço. A taxa de serviço desta fila corresponde à largura de banda reservada para esta classe de serviço.
- **Datagramas da classe AF:** são encaminhados para a sua respectiva fila que implementam o algoritmo RED (Random Early Detection). Como vimos, o RED visa prevenir a situação de congestionamento, descartando pacotes da fila mesmo sem ela estar cheia. A probabilidade de descarte de pacotes é uma função do tamanho médio da fila. Tal algoritmo evita uma reação brusca a sobrecargas médias. O roteador possui uma fila RED para cada uma das três categorias de serviços AF implementadas: uma para AF1, outra para AF2, e outra para AF3.

Repare que na implementação do roteador, não se levou em conta mecanismos para prover garantias de valores de atraso e variações de atraso no tratamento dos pacotes. Esta simplificação adotada tornou mais fácil a implementação do modelo, mas precisa ser levada em conta em uma análise mais rigorosa de resultados gerados pelos modelos desenvolvidos neste trabalho. Como o escopo deste trabalho não é a análise de desempenho de redes que empregam a arquitetura *DiffServ,* mas sim o estudo e a compreensão desta arquitetura de uma forma mais genérica a fim de ilustrar uma aplicação para o simulador desenvolvido, para o nosso caso esta simplificação é válida. Segundo [38], este serviço pode ser implementado com políticas de escalonamento de filas baseadas em prioridades (PQ), round robin ponderado (WRR), WFQ (Weighted Fair Queuing) e CBQ (Class-based Queuing). Futuros trabalhos deverão contemplar de forma mais rigorosa o tratamento de pacotes da classe EF, levando em conta esta observação.

Já no tratamento de datagramas da classe AF, o algoritmo RED detecta uma tendência de congestionamento monitorando o tamanho da respectiva fila. Quando este tamanho ultrapassa um determinado limiar, inicia-se um processo de descarte aleatório de pacotes, onde a probabilidade de descarte é função da taxa de ocupação média da fila. Devido ao mecanismo de operação do TCP, conseqüentemente teremos uma reversão da tendência de congestionamento.

O modelo de roteador possui seu código distribuído nos arquivos listados na Tabela 6:

ARQUIVO	DESCRIÇÃO
libRouter.cpp	Implementação dos métodos da classe Router, o nosso modelo de roteador
libRouter.h	Declaração da classe Router
ip_router.cpp	Implementação dos métodos da classe IP_Router, na qual implementamos o processamento do protocolo IP no roteador
ip_router.h	Declaração da classe IP_Router
router_ip_queue.cpp	Implementação dos métodos da classe Router_Output_Queue, na qual implementamos a fila usada no processamento do protocolo IP no roteador.
router_ip_queue.h	Declaração da classe Router_Output_Queue
link_router.cpp	Implementação dos métodos da classe Link_Router, em que implementamos o nível de enlace do roteador
link_router.h	Declaração da classe Link_Router
ip_datagram.h	Declaração e implementação da classe IP_Datagram, o tipo de mensagem que o roteador usa para se comunicar a nível de camada IP
frame.h	Declaração e implementação da classe Frame, o tipo de mensagem que o roteador usa para se comunicar a nível de enlace

Tabela 6 - Arquivos com o código do modelo de roteador

Note que os arquivos ip_datagram.h e frame.h são os mesmos usados pelo modelo de terminal.

As classes são ilustradas na Figura 23. No lado esquerdo temos as classes que integram o modelo Roteador (camadas IP e enlace), e no lado direito as mensagens utilizadas para comunicação deste modelo (datagrama IP e quadro).

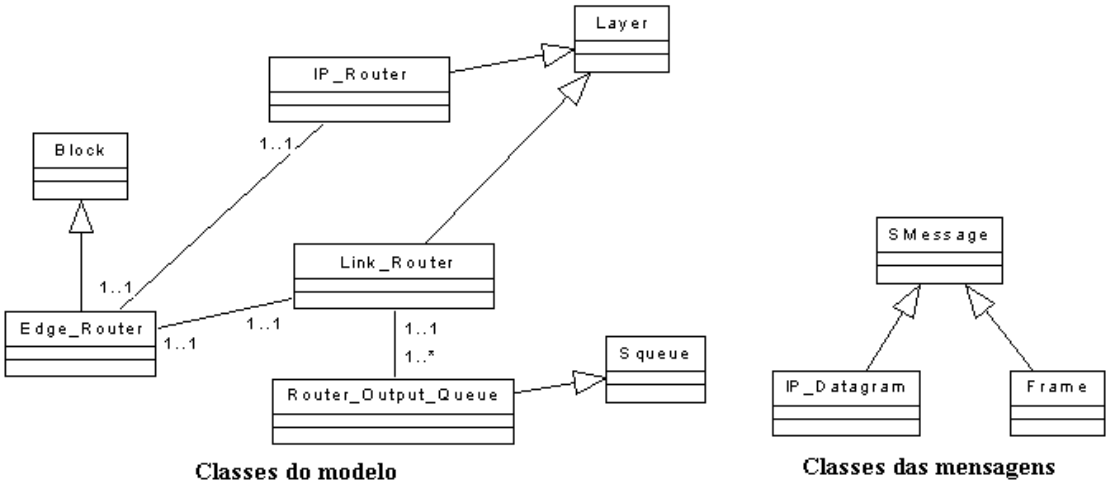

Figura 23 - Classes que integram o modelo de roteador

6.6 - INTERAÇÃO ENTRE OS BLOCOS

Na Figura 24 temos um diagrama de eventos de referência para o funcionamento dos modelos desenvolvidos, no tocante ao fluxo de eventos para a transmissão de dados no sentido servidor-cliente.

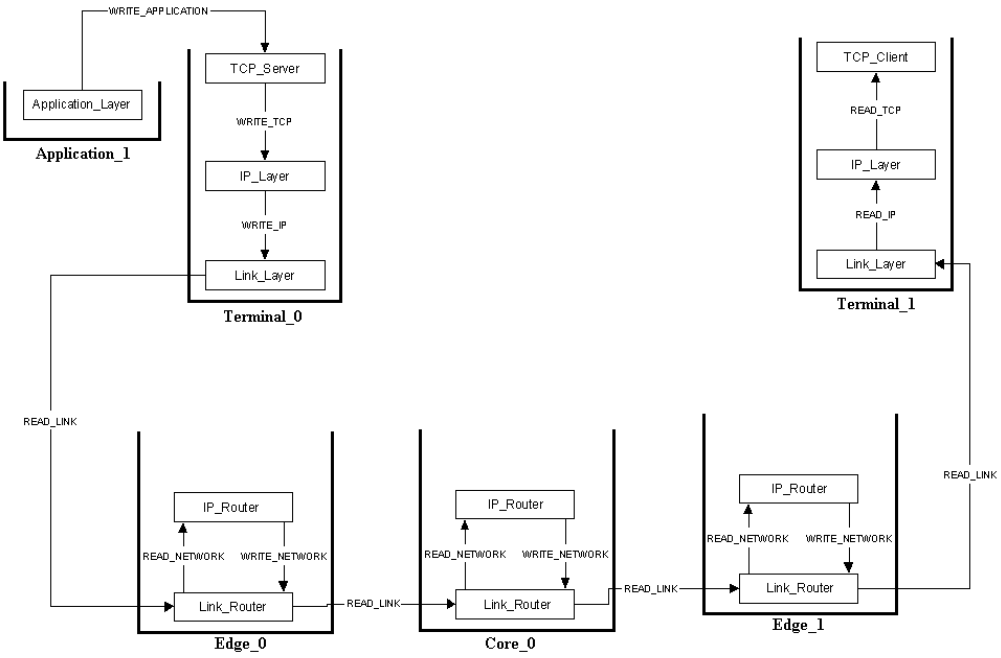

Figura 24 - Interação entre os blocos no sentido servidor-cliente (fluxo de dados)

Nos sentido contrário, ou seja, do Terminal_1 para o Terminal_0, temos o fluxo de eventos gerado pela confirmação dos pacotes na camada TCP (fluxo de ACKs),como ilustra a Figura 25.

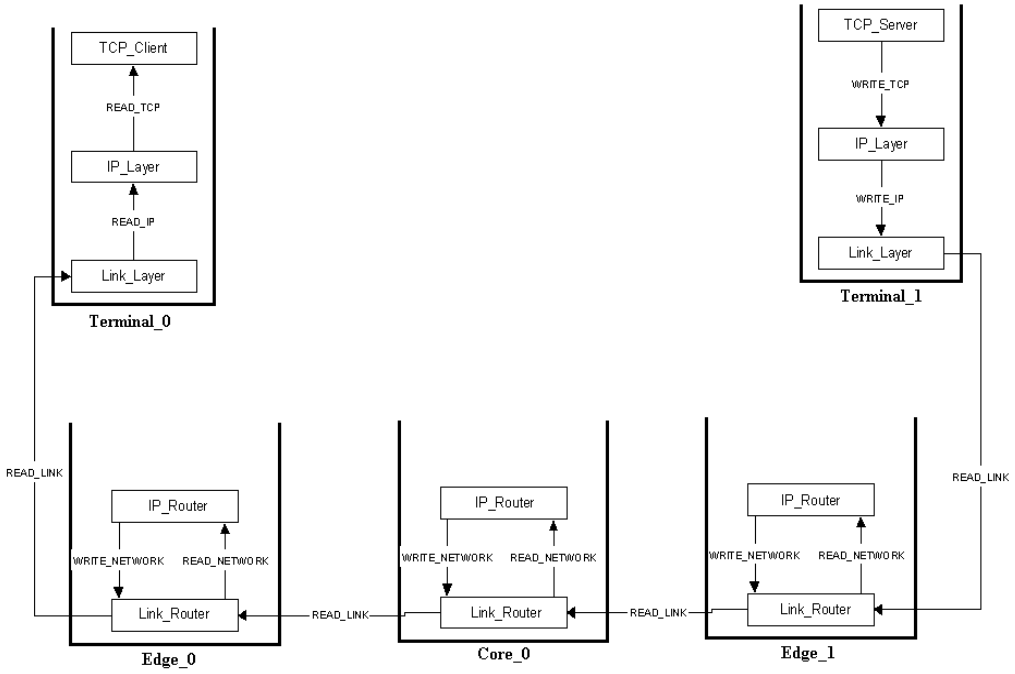

Figura 25 - Interação entre os blocos no sentido cliente-servidor (fluxo de ACKs)

6.7 - SIMULAÇÃO

Sendo o cenário simulado o ilustrado na figura 30, na **primeira simulação** realizada, configurou-se o roteador de ingresso para reservar recursos para as classes de serviço. As reservas que foram configuradas são ilustradas na Tabela 7:

Classe	ToS	Banda Reservada (bps)
BE	0	6400
AF1	1	15360
AF2	2	9600
AF3	3	5760
EF	4	12160

Tabela 7 - Reserva de recursos por classe

Na **segunda simulação** a reserva de recursos foi desabilitada, ou seja, todos os datagramas foram tratados pelo roteador segundo a primitiva de melhor esforço.

Foram feitas simulações com pacotes de tamanhos 128, 256 e 512 bytes. Em todas as simulações, o intervalo de geração de pacotes (fonte de tráfego com intervalos entre pacotes transmitidos obedecendo a distribuição exponencial) foi configurado com uma taxa média de 0,1s.

A seguir, na Tabela 8, temos os valores da vazão obtida, em bits por segundo, tomando pacotes de 128 bytes, nas simulações com *DiffServ* habilitado, e com *DiffServ* desabilitado.

CLASSE	COM DIFFSERV	SEM DIFFSERV
EF	10330,29	3673,88
AF1	5175,48	3864,55
AF1	4988,13	3677,07
AF1	4857,2	3675,78
AF2	2930,51	4169,55
AF2	3099,92	3732,98
AF2	3299,05	3777,51
AF3	1821,61	3685,88
AF3	1873,87	3695,97
AF3	1893,95	3615,56
BE	2084,28	3655,79
BE	2096,22	3729,59
BE	2033,64	3733,94

Tabela 8 - Vazão obtida com pacotes de 128 bytes

Com pacotes de tamanho 256 bytes, os valores obtidos para vazão na simulação estão ilustrados na Tabela 9:

CLASSE	COM DIFFSERV	SEM DIFFSERV
EF	10255	3692,55
AF1	5045,35	3715,52
AF1	5100,13	3794,84
AF1	4979,36	3741,79
AF2	3214,59	3811,78
AF2	3152,69	3967,25
AF2	3008,7	3915,77
AF3	1869	4032,87
AF3	1867,37	3788,3
AF3	1978,02	3989,93
BE	2281,87	4036,55
BE	1680,78	3972,17
BE	1638,89	3912,23

Tabela 9 - Vazão obtida com pacotes de 256 bytes

Com pacotes de tamanho 512 bytes, os valores obtidos para vazão estão ilustrados na Tabela 10:

CLASSE	COM DIFFSERV	SEM DIFFSERV
EF	10285,6	3732,67
AF1	4808,91	4197,95
AF1	5206,12	3801,79
AF1	4978,79	3969,01
AF2	3150,58	3836,96
AF2	3061,9	3767,33
AF2	3313,66	3883,42
AF3	1884,89	3880,59
AF3	1819,88	3761,33
AF3	2058,12	3742,16
BE	2052,58	3718,57
BE	2219,55	4013,47
BE	2055,79	4001,71

Tabela 10 - Vazão obtida com pacotes de 512 bytes

6.8 - ANÁLISE DOS RESULTADOS

A partir dos valores obtidos das simulações, podemos avaliar de forma geral o comportamento da diferenciação de serviços no cenário de rede simulado.

O gráfico apresentado na Figura 26 ilustra um comparativo entre os valores da vazão obtida por fluxo transmitido nas situações de rede com *DiffServ* habilitado, e com *DiffServ* não habilitado, considerando o tamanho dos pacotes TCP de 128 bytes.

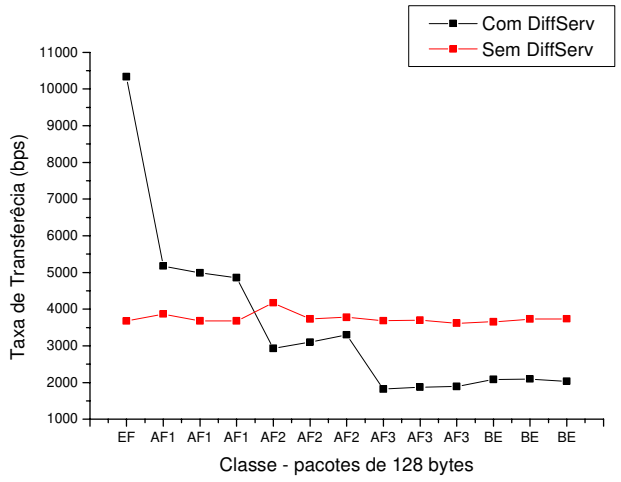

Figura 26 - Vazão por fluxo de tráfego com pacotes de 128 bytes

O gráfico apresentado na Figura 27 ilustra um comparativo entre os valores da vazão obtida por fluxo transmitido nas situações de rede com *DiffServ* habilitado, e com *DiffServ* não habilitado, considerando o tamanho dos pacotes TCP de 256 bytes.

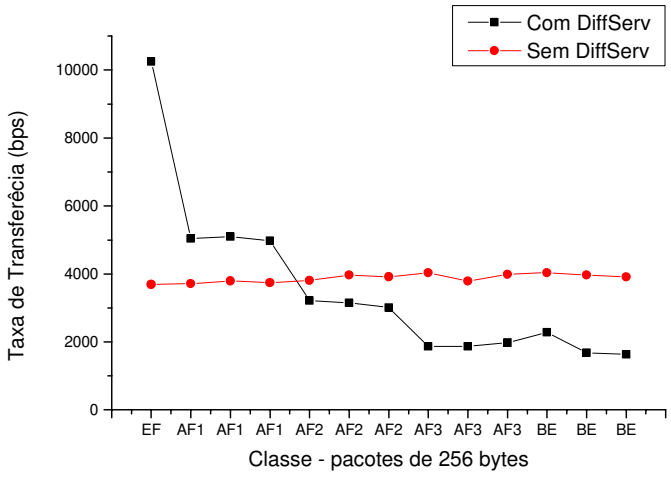

Figura 27 - Vazão por fluxo de tráfego com pacotes de 256 bytes

Já o gráfico apresentado na Figura 28 ilustra um comparativo entre os valores da vazão obtida por fluxo transmitido nas situações de rede com *DiffServ* habilitado, e com *DiffServ* não habilitado, considerando o tamanho dos pacotes TCP de 512 bytes.

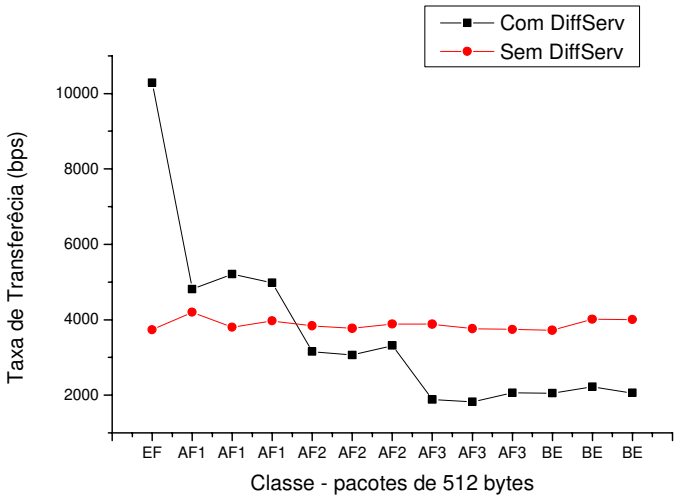

Figura 28 - Vazão por fluxo de tráfego com pacotes de 512 bytes

Podemos verificar que nas simulações em que o *DiffServ* foi desabilitado, a vazão obtida para todos os fluxos têm valor semelhantes.

Nas simulações em que o *DiffServ* foi habilitado, verifica-se que a vazão obtida dos fluxos agregados das classes EF, AF1, AF2, AF3 e BE é proporcional a largura de banda reservada para cada classe.

6.8.1 - VAZÃO OBTIDA COMPARADA À LARGURA DE BANDA RESERVADA

O gráfico apresentado na Figura 29 mostra um comparativo entre os valores obtidos da vazão em cada uma das 13 conexões, e os valores de largura de banda reservada para estas classes de serviço, considerando pacotes TCP de 128 bytes.

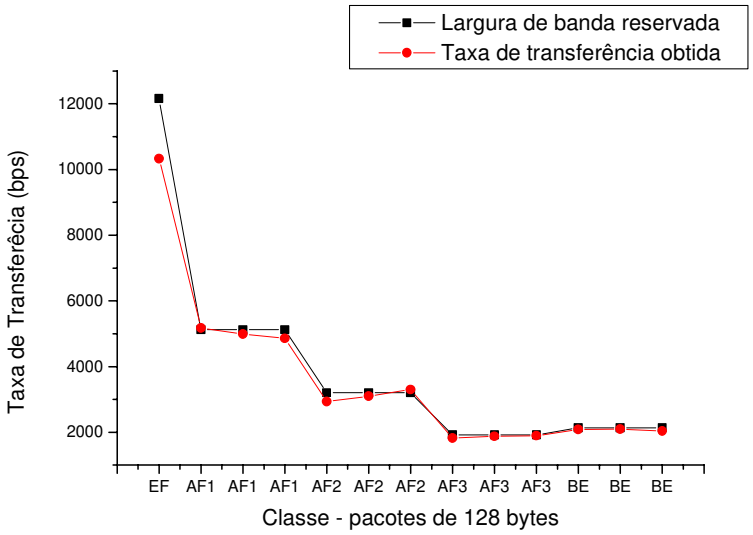

Figura 29 - Vazão medida vs. largura de banda definida com pacotes de 128 bytes

O gráfico apresentado na Figura 30 mostra um comparativo entre os valores obtidos da vazão em cada uma das 13 conexões, e os valores de largura de banda reservada para estas classes de serviço, considerando pacotes TCP de 256 bytes.

.

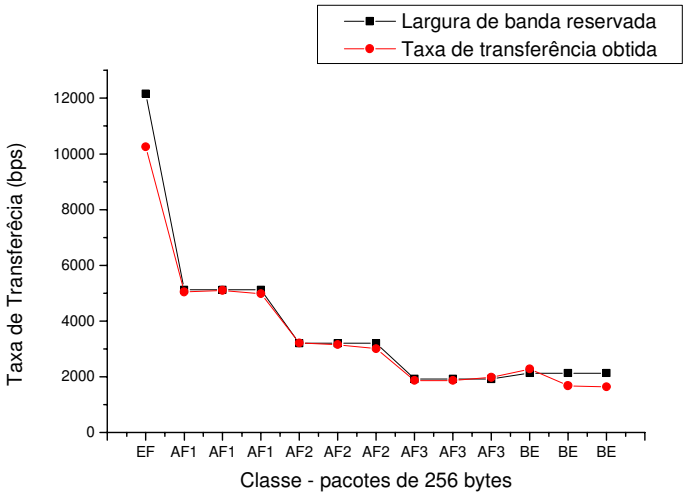

Figura 30 - Vazão medida vs. largura de banda definida com pacotes de 256 bytes

O gráfico apresentado na Figura 31 mostra um comparativo entre os valores obtidos da vazão em cada uma das 13 conexões, e os valores de largura de banda reservada para estas classes de serviço, considerando pacotes TCP de 512 bytes.

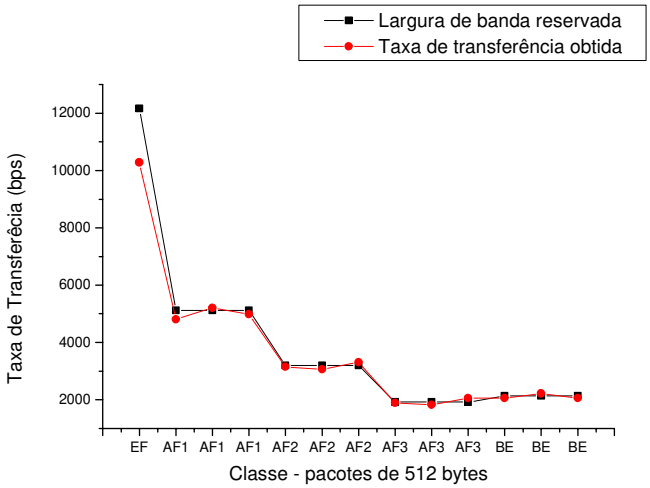

Figura 31 - Vazão medida vs. largura de banda definida com pacotes de 256 bytes

Pode-se observar que os valores obtidos para a vazão de cada um dos fluxos estão bastante próximos do que havia sido reservado para os mesmos.

6.8.2 - VISÃO GERAL SOBRE O COMPORTAMENTO DA REDE

Na Tabela 11, temos os valores de largura de banda reservada para o fluxo agregado de cada uma das classes comparada aos valores da vazão medida nas simulações feitas com pacotes de tamanhos 128, 256 e 52 bytes. Verificou-se que independentemente do tamanho dos pacotes, a vazão obtida sempre esteve muito próxima dos valores reservados.

Classe	Banda Reservada (bps)	Banda Usada (bps) (pacotes de 128 bytes)	Banda Usada (bps) (pacotes de 256 bytes)	Banda Usada (bps) (pacotes de 512 bytes)
EF	12160	10330,29	10255	10285,6
AF1	15360	15020,81	15124,84	14993,82
AF2	9600	9329,48	9375,98	9526,14
AF3	5760	5589,43	5714,39	5762,89
BE	6400	6214,14	5601,54	6327,92

Tabela 11 - Largura de banda agregada dos fluxos de cada classe

6.9 - DESEMPENHO DO SIMULADOR

Foram analisados os desempenhos dos simuladores ***lsimnt*** e ***Hydragyrum*** para a simulação do cenário de rede relatada neste documento. O mesmo código fonte, em linguagem C++, escrito para implementação dos modelos na forma de bibliotecas dinâmicas para o ***lsimnt*** (***Linux***), foi usado na compilação dos modelos para o simulador ***Hydragyrum*** (***Windows***). O microcomputador utilizado para esta análise possui um processador AMD K-6 de 500 MHz, com 128 Mbytes de memória RAM. Os sistemas operacionais usados foram o ***Windows*** 98, no caso do ***Hydragyrum***, e o ***Linux*** Mandrake 8.0, no caso do ***lsimnt***. Em todas as análises o tempo de simulação utilizado foi de 7300 s, um tempo de aproximadamente 2 horas de atividade na rede simulada, considerado suficiente para a análise aqui descrita. Os resultados obtidos são resumidos na Tabela 12.

Simulador	***Lsimnt*** (***Linux***)	***Hydragyrum*** (***Windows***)
Tempo gasto para completar a simulação	118 s	203 s
Número total de eventos processados	20.137.045 eventos	20.136.300 eventos
Eventos processados por segundo pelo simulador	170.652,9 eventos/s	99.193,4 eventos/s

Tabela 12 - Comparativo dos simuladores *lsimnt* e *Hydragyrum*

A diferença do número de eventos justifica-se pela aleatoriedade existente no intervalo entre pacotes gerados pelos terminais.

O tempo total gasto para simulação foi 118 segundos usando o ***lsimnt***, e 203 segundos usando o ***Hydragyrum***.

Podemos observar que o simulador ***lsimnt*** processou uma média de 170.652,9 eventos por segundo, ao passo que o ***Hydragyrum*** processou somente 99.193,4 eventos por segundo. Neste caso, o ***lsimnt*** apresentou uma capacidade de processamento da simulação na ordem de 72 % acima da capacidade do ***Hydragyrum***.

CAPÍTULO 7. COMPARATIVO DO DESEMPENHO DOS NÚCLEOS DOS SIMULADORES *LSIMNT* E *HYDRAGYRUM*

Como vimos, o ***lsimnt*** foi desenvolvido a partir do simularor ***Hydragyrum***. Neste capítulo iremos mostrar um comparativo do desempenho dos núcleos destes simuladores. Em ambos os simuladores, o núcleo possui uma fila para armazenar os eventos ordenados pelo tempo. Assim, no instante de tempo adequado, o núcleo extrai o evento e entrega-o para o objeto de destino.

Vimos uma comparação do desempenho no item - 6.9 - . Porém, aquela simulação singular não permite afirmar se de fato o núcleo do ***lsimnt*** apresenta um desempenho superior ao do simulador ***Hydragyrum¸*** pois naquelas simulações foram usados compiladores diferentes para geração dos códigos dos simuladores, e é claro, sistemas opercionais diferntes (***Windows*** e ***Linux***). A fim de realizar um comparativo apenas dos núcleos, o código do núcleo do simulador ***Linux*** desenvolvido em C++ foi compilado no mesmo compilador usado para compilar o simulador ***Hydragyrum¸*** o Borland C++ 5.02. Uma pequena alteração foi necessária somente no código referente a criação da instância dos modelos, que na implementação do ***lsimnt*** é baseada na carga de bibliotecas dinâmicas. Para tornar possível a execução do ***lsimnt*** o código dos modelos foram ligados junto com o código do núcleo do ***lsimnt***, gerando assim um único programa executável para ***Windows*** contendo o núcleo do ***lsimnt*** e os modelos.

Na simulação em que foi empregado o núcleo do simulador ***Hydragyrum¸*** usamos o aplicativo ***simscl*** [1], que é a versão de linha de comando do simulador ***Hydragyrum***. O ***simscl*** contém o núcleo do simulador ***Hydragyrum*** e cria instâncias dos modelos a partir da carga de bibliotecas dinâmicas para ***Windows*** (DLLs). Este fato não imlicará em erro na análise do desempenho dos simuladores, pois o tempo necessário para o ***simscl*** ocorre na inicialização (***Setup***), e não durante a execução da simulação em si (***Run***).

Como estamos interessados apenas na verificação do desempenho do núcleo, foi criado um novo modelo, descrito a seguir.

7.1 - MODELO DESENVOLVIDO PARA COMPARAÇÃO DOS SIMULADORES

Este novo modelo que nada mais faz do que agendar eventos no núcleo do simulador de maneira aleatória.

O modelo de aplicação possui seu código distribuído nos arquivos listados na Tabela 13:

ARQUIVO	DESCRIÇÃO
Simple_block.cpp	Implementação dos métodos da classe Simple_Block, o bloco.
Simple_block.h	Declaração da classe Simple_Block.
simple _layer.cpp	Implementação dos métodos da classe Simple_Layer, a única camada presente no bloco
simple _layer.h	Declaração da classe Simple_Layer.

Tabela 13 - Arquivos com o código do modelo simple_block

Para agendar os eventos, foi implementado o método *Run()* na classe *Simple_Layer* , mostrado na Figura 32. Este evento nada mais faz do que reagendar o mesmo evento na fila de eventos do núcleos, usando intervalos randômicos.

```
int Simple_Layer::Run(Event * _Event){
  double Time_Now = _Event->GetEventTime();
  double r= (double) (rand() % 100)/1000;  // Número Randômico
  CreateLoopEvent(Time_Now+r,"LOOP_EVENT");
  count++;
  return 0;
}
```

Figura 32 - Método *Run* da classe Simple_Block

O bloco *Simple_Block* foi compilado para os dois simuladores, o ***Hydragyrum*** e o ***lsimnt***. Um arquivo .SCL foi criado de forma a criar um total de **300** instâncias do objeto Simple_Block, um número considerável para a nossa análise. Assim, teremos 300 entidades concorrentes agendando eventos em intervalos aleatórios no núcleo do simulador.

Para termos um número considerável de amostras foram executadas 10 simulações, em cada um dos simuladores, anotando a quantidade de eventos processada por segundo, da

mesma forma que foi feito no item - 6.9 - deste documento. Os valores coletados estão na Tabela 14.

SIMULADOR	***Lsimnt***	***Hydragyrum***
Número de eventos processados por segundo (eventos/s) em cada simulação	315139,4	223533,5
	326738,2	226014,9
	319015,1	226036,6
	322844,1	229548,3
	323785,7	226373,7
	328932,8	232518,6
	326461,4	228659,3
	319389,2	229139,6
	323211,8	224726,9
	322671,7	232375,4
Média (eventos/s)	**322818,94**	**227892,68**

Tabela 14 – Comparativo: número de eventos processados por segundo

Aqui, o núcleo do ***Lsimnt*** apresentou um desempenho cerca de 41% acima do desempenho do núcleo do ***Hydragyrum***. Este resultado é bastante conclusivo, o que permite afirmar que o desempenho do núcleo do simulador ***Lsimnt*** é superior ao desempenho do núcleo do simulador ***Hydragyrum***.

CAPÍTULO 8. CONCLUSÕES

Neste trabalho foi desenvolvido o ***lsimnt***, uma ferramenta genérica para simulação, para uso em ensino, pesquisa, projeto e análise de redes de comunicações. Esta ferramenta foi desenvolvida para o sistema operacional ***Linux***, vindo a preencher uma lacuna existente, não só com a criação de um simulador para ***Linux*** compatível com a estrutura do simulador ***Hydragyrum*** [1], mas também com a implementação de um *kernel* mais enxuto, melhor estruturado, e com desempenho superior ao do simulador em que foi baseado, o ***Hydragyrum*** para ***Windows***. Desta forma, este novo *software* contribui também para a evolução da cultura já existente nas técnicas de simulação de redes empregadas no nosso grupo de simulação.

O ***lsimnt***, a grande contribuição deste trabalho, permite implementar modelos e cenários compatíveis com o ***Hydragyrum***, de forma que o código em C++ dos modelos desenvolvidos para o ***lsimnt*** podem ser compilados para geração de bibliotecas dinâmicas para o simulador ***Hydragyrum***.

Outra contribuição deste trabalho é uma implementação do protocolo de transporte TCP. Até então, os trabalhos desenvolvidos em nosso grupo de simulação trataram de simulações na área de redes óticas, com base no trabalho do ***SimNT*** [3], ou de redes de comunicação orientadas a conexão, com base nos trabalhos do ***SimATM*** [2] ou ***Hydragyrum*** [1]. Porém, nenhum trabalho contemplou a implementação do complexo protocolo TCP. A implementação feita do protocolo TCP será de grande valor pra outros trabalhos, alguns já em andamento dentro do nosso grupo de simulação, que exploram também o uso do protocolo TCP para comunicação.

Também são contribuições, no contexto de programação e modelagem de redes, as implementações dos elementos de rede: aplicação, terminar, roteador de borda e roteador de núcleo.

Já no contexto teórico, a contribuição extraída deste trabalho é uma pesquisa bibliográfica importante, baseada em livros, artigos publicados, RFCs e notas de aula. A partir desta pesquisa, foram trabalhados conceitos e fundamentos relacionados a, principalmente, três temas:

- Simulação de redes.

- Protocolo TCP/IP.
- Arquitetura *DiffServ*.

8.1 - TRABALHOS FUTUROS

Concluído este trabalho, identificou-se a possibilidade de maiores aprofundamentos em alguns pontos. Assim, sugestões para possíveis trabalhos futuros são:

- Implementação de mecanismos para garantias de requisitos de atraso e variação de atraso no tratamento dos pacotes da classe EF no modelo de roteador.
- Implementação de modelos de elementos de rede baseados no protocolo TCP/IP desenvolvidos neste trabalho. São bons candidatos: elementos de rede para simulação de redes baseadas na arquitetura de serviços integrados, serviços diferenciados, redes MPLS e aplicações de voz sobre redes IP.
- Implementação de uma interface gráfica para o ***lsimnt*** no ambiente ***Linux***.
- Implementação de uma interface de programação para simulação distribuída usando o ***lsimnt***.

CAPÍTULO 9. REFERÊNCIAS BIBILIOGRÁFICAS

[1] NETO, Ernesto L. A.: **Ambiente de Simulação de Redes a Eventos Discretos**. Tese de Doutorado. Faculdade de Engenharia Elétrica e de Computação. UNICAMP. Setembro, 2001.

[2] ALBERTI, A. M.: **SimATM: Um Ambiente para a Simulação de Redes ATM**. Tese de Mestrado. Faculdade de Engenharia Elétrica e de Computação. UNICAMP. Abril, 1998.

[3] KLEIN, J.: **SIMNT - Uma ferramenta para a simulação de sistemas de Comunicação**. Tese de Mestrado. Faculdade de Engenharia Elétrica e de Computação, UNICAMP. Agosto, 1995.

[4] KLEIN, J.: **Desenvolvimento de uma ferramenta CAD/CAM para Simulação, Projeto e Ensino de Sistemas de Comunicação**. Tese de Doutorado. Faculdade de Engenharia Elétrica e de Computação, UNICAMP. Março, 1999.

[5] BOTTOLI, M.: **Implementação de Modelos de Amplificadores a Fibra Dopada com Érbio para Simulação de Enlaces Ópticos**. Tese de Mestrado. Faculdade de Engenharia Elétrica e de Computação, UNICAMP. Abril, 1999.

[6] CAMARGO, H. A. M.: **Estudo de métodos de controle de admissão de chamadas em redes ATM com análise de desempenho**. Tese de Mestrado. Faculdade de Engenharia Elétrica e de Computação, UNICAMP. Outubro, 1999.

[7] HEIDEMANN, J.: **Expanding confidence in network simulations**. 1999.

[8] LAW, A. e McCOMAS, M.: **Simulation Software for Communications Networks: The State of the Art**. IEEE Communications Magazine. 1994.

[9] LAW, A. M. e KELTON, W. David: **Simulation Modeling And Analysis**. Editora McGraw-Hill. 1982.

[10] CASSANDRAS, C. G. e LAFORTUNE, S.: **Introduction to Discrete Event Systems**. Editora Kluwer Academic Publishers. 1999.

[11] BANKS, J., CARSON II, J. S. e NELSON, B. L.: **Discrete Event System Simulation**. Editora Prentice-Hall. 1996.

[12] PAPOULIS, A.: **Probability, Random Variables, and Stochastic Processes**. Terceira edição. Edigora McGraw-Hill. 1991.

[13] ROBERTAZZI, Thomas G.: **Computer Networks and Systems: Queueing Theory and Performance Evaluation**. Segunda edição. Editora Springer-Verlag. 1994.

[14] SHANMUGAN, S. K.: **Simulation and Implementation Tools for Signal Processing and Communication Systems**. IEEE Communications Magazine. Julho, 1994.

[15] OLIVEIRA, Sergio de: **Telefonia IP para Ambientes Móveis**. Documento disponível em http://www.dcc.ufmg.br/~sergiool/

[16] FONSECA, José L. A. da Fonseca: **Aplicações Multimídia na Internet.** Documento disponível em http://www.geocities.com/jluizf/Multimidia/COD.html

[17] **Understanding Jitter in Packet Voice Networks**. Documento disponível em http://www.cisco.com/warp/public/788/voice-qos/jitter_packet_voice.html

[18] **Fox House – UML**. Documento disponível em http://www.foxbrasil.com.br/paulino/projetos/oop/UML.htm

[19] **C++ Programming HOW-TO.** Documento disponível em http://www.Linux.org/docs/ldp/howto/C++Programming-HOWTO-1.html

[20] STROUSTRUP, B.: **The C++ Programming Language**. Terceira Edição. Editora Addison-Wesley. 1997.

[21] COMER, D. e STEVENS D.: **Interligação em Rede com TCP/IP – Volume I: Princípios, Protocolos e Arquitetura**. Tradução da terceira edição. Editora Campus. Rio de Janeiro, 1998.

[22] COMER, D. e STEVENS D.: **Internetworking with TCP/IP – Volume II: Design, Implementation and Internals**. Terceira edição. Editora Prentice Hall. New Jersey. 1995.

[23] CARVALHO, T. C. M. de Brito: **Arquiteturas de Redes de Computadores OSI e TCP/IP.** Editora Makron Books. São Paulo. 1997.

[24] TANENBAUM, S. A.: **Computer Networks.** Terceira edição; Editora Prentice Hall. 1996.

[25] STALLINGS, W.: **High-Speed Networks TCP/IP and ATM Design Principles**. Editora Prentice Hall. 1998.

[26] BECKER, M. S.: **Transmission Control Protocol**. Monografia de especialização. UNISINOS. Março de 2001.

[27] RFC 675. **Specification of Internet Transmission Control Program**. Dezembro, 1974.

[28] RFC 761: **DOD Standard Transmission Control Protocol**. Janeiro, 1980.

[29] RFC 793: **Transmission Control Protocol**. Setembro, 1981.

[30] DUNKELS, A.: **Design and Implementation of the lwIP TCP/IP Stack**. Swedish Institute of Computer Science. Fevereiro, 2001.

[31] RFC 813: **Window and Acknowledgement Strategy in TCP**. Julho, 1982.

[32] JACOBSON, V. e KARELS M.: **Congestion Avoidance and Control**. SIGCOMM Symposium on Communications Architectures and Protocols. 1988.

[33] MELO, E. T. L.: **Qualidade de Serviço em Redes IP com DiffServ: Avaliação através de Medições.** Tese de Mestrado. UFSC. Maio, 2001.

[34] FLOYD, Sally e PAXSON, Vern: **Difficulties in Simulating the Internet**. AT&T Center for Internet Research at ICSI.

[35] BRITO, José M. C.: **Voz sobre IP – Tecnologias e Aplicações**. INATEL; junho de 2000.

[36] FLOYD, Sally e JACOBSEN, V.: **Random Early Detection Gateways for Congestion Avoidance.** IEEE/ ACM Transactions on Networking, Vol. 1, No. 4, Aug. 1993, pp. 397-413.

[37] ANDREOZZI, Sergio: **DiffServ simulations using the network simulator: requirements, issues and solutions**. 2001.

[38] MAGALHÃES, Mauricio F. e CARDOZO, Eleri: **Qualidade de Servico na Internet**. Draft. Faculdade de Engenharia Elétrica e de Computação, UNICAMP. Outubro, 1999.

[39] RFC 2475: **An Architecture for Differentiated Services**. 1998.

[40] RFC 2998: **A Framework for Integrated Services Operation over Diffserv Networks**. Novembro, 2000.

[41] **Quality of Service Solutions Configuration Guide, Congestion Management Overview.** Documento disponível em http://www.cisco.com

[42] **LBL-TCP-3 - Two Hours of Wide-Area TCP Traffic**. Documento disponível em http://ita.ee.lbl.gov/html/contrib/LBL-TCP-3.html

[43] PRITZKER, A. A. B.: **Introduction to Simulation and SLAM II**. Wiley, New York, 1995.

[44] **NS2 Simulator.** Documento disponível em http://www.isi.edu/nsnam/ns/

[45] **Arena Software.** Documento disponível em http://www.arenasimulation.com

[46] **OPNET Modeler.** Documento disponível em http://www.mil3.com/products/modeler

Apêndice A. A linguagem SCL

Neste apêndice, é descrita a linguagem de comandos do simulador. A linguagem de comandos pode ser utilizada para a composição de diferentes cenários de simulação.

A.1 - COMANDOS DO AMBIENTE DE SIMULAÇÃO

A.1.1 - Comando: block

Sintaxe: **block** *blockname modelfilename*

Descrição: cria na simulação o bloco *blockname* utilizando o modelo contido no arquivo *modelfilename.* (Obs: os nomes dos blocos devem ser únicos em uma mesma simulação).

A.1.2 - Comando: connect block

Sintaxe: **connect block** *connecname sourceblock destinyblock sourcelayer destinylayer*

Descrição: cria a conexão *connecname* entre o bloco *sourceblock* e sua camada *sourcelayer* com o bloco *destinyblock* e sua camada *destinylayer*

A.1.3 - Comando: param global

Sintaxe: **param global** *paramname paramtype rows cols value description option*

Descrição: carrega para o parâmetro global *paramname* os dados do parâmetro *(paramtype rows cols value description option)*

A.1.4 - Comando: param block

Sintaxe: **param block** *blockname paramname paramtype rows cols value description option*

Descrição: carrega no bloco *blockname* o parâmetro *paramname* e seus dados *(paramtype rows cols value description option)*

A.1.5 - Comando: param layer

Sintaxe: **param layer** *blockname layername paramname paramtype rows cols value description option*

Descrição: carrega para a camada *layername* do bloco *blockname* o parâmetro *paramname* e seus dados *(paramtype rows cols value description option)*

A.1.6 - Comando: param queue

Sintaxe: **param queue** *blockname queuename paramname paramtype rows cols value description option*

Descrição: carrega para a fila *queuename* do bloco *blockname* o parâmetro *paramname* e seus dados *(paramtype rows cols value description option)*.

A.2 - CARGA E EXECUÇÃO DE ARQUIVO DE SIMULAÇÃO

Na carga de um arquivo de simulação é feita via linha de comando em terminal, através do programa executável ***lsimnt***.

Por exemplo, o arquivo denominado *arquivo.scl*, configurado com comandos para carga dos blocos, conexões e manipulação de parâmetros, é passado como argumento para o ***lsimnt***, juntamente com o tempo de simulação. Supondo este tempo ser de 100 segundos, o comando fica assim:

lsimnt *arquivo.scl 100*

Apêndice B. API e recursos de programação

Este apêndice descreve os recursos de programação para a implementação de novos modelos para o simulador. Estes recursos abrangem as funções presentes nas classes de modelagem e as estruturas de dados disponíveis para troca de dados entre o núcleo do simulador e os modelos. O emprego de praticamente todos os métodos é idênticos ao do simulador ***Hydragyrum*** [1], cuja API serviu da base para a elaboração deste apêndice B.

B.1 - INTERFACE DOS MODELOS

Os métodos virtuais da interface dos modelos com o simulador são listadas abaixo:

- **virtual int Setup();**
- **virtual int Initiate();**
- **virtual int Run(*Event* *_Event);**
- **virtual int Posprocessing();**
- **virtual int OnConnect(*Sstring* ConnecName);**

B.2 - FUNÇÕES DAS CLASSES BASE (BLOCK, LAYER E SQUEUE)

Esta seção detalha as funções que pertencem às classes base de modelagem. Funções que pertencem a só uma determinada classe base são especificadas. Caso contrário, as funções são comuns a todas as classes base de modelagem.

B.2.1 - FUNÇÕES PARA MANIPULAÇÃO DE IDENTIFICADORES

Sstring GetName()

Retorna o nome do *Block*, *Layer* ou *Squeue*.

void SetName (Sstring & _Name)

Atribui o nome de um *Block*, *Layer* ou *Squeue*.

void SetModel (Sstring _Model)

Atribui o nome do modelo associado ao *Block*. Este método só está disponível na classe *Block*.

Sstring GetModel()

Retorna o nome do modelo associado ao *Block*. Este método só está disponível na classe *Block*.

void SetType(Sstring _Type)

Atribui o tipo da *Layer* ou *Squeue*. Este método só está disponível nas classes *Layer* e *Squeue*.

Sstring GetType()

Retorna o tipo da *Layer* ou *Squeue*. Este método só está disponível nas classes *Layer* e *Squeue*.

Sstring GetBlockName()

Retorna o nome do *Block* que contêm o objeto da classe *Layer* e *Squeue*. Este método só está disponível nas classes *Layer* e *Squeue*.

B.2.2 - ADIÇÃO, REMOÇÃO E ASSOCIAÇÃO DE ELEMENTOS NO BLOCK

B.2.2.1 CAMADAS – LAYERS

int AddLayer (Layer * NewLayer)

Adiciona uma *Layer*, previamente alocada, apontada por ***NewLayer*** ao *Block*. Este método só está disponível na classe *Block*.

B.2.2.2 FILAS – SQUEUES

unsigned int AddSqueue (Squeue * NewSqueue)

Adiciona uma *Squeue*, previamente alocada, apontada por ***NewSqueue*** ao *Block*. Este método só está disponível nas classes *Block* e *Layer*.

B.2.3 - FUNÇÕES DE MANIPULAÇÃO DE ELEMENTOS CONTIDOS NO BLOCK

Estas funções retornam ponteiros para os objetos contidos dentro da estrutura de um modelo.

O modelo é um objeto de uma classe derivada da classe *Block*. Os ponteiros retornados por esta função permitem a manipulação direta dos objetos apontados.

Block * GetBlock( )

Retorna um ponteiro para o *Block* que armazena a *Layer* ou *Squeue* que fez a chamada da função. Este método só está disponível nas classes *Layer* e *Squeue* .

int GetLayer(Sstring SelectedLayer, Layer * & _Layer)

Obtém a *Layer* denominada ***SelecteLayer*** e retorna um ponteiro para ela em ***_Laye***r. Se a *Layer* não for encontrada o código de erro retornado é 1. A operação normal da função retorna 0. A busca das camadas nos objetos derivados das classes *Layer* ou *Squeue* é sempre realizada nas listas de *Layers* dos objetos derivados da classe *Block*.

Layer * GetLayer(Sstring SelectedLayer)

Obtém a *Layer* denominada ***SelectedLayer*** e retorna um ponteiro para ela. Se a *Layer* não for encontrada o código de erro retornado é NULL (0). A operação normal da função retorna o valor do ponteiro. A busca das camadas nos objetos derivados das classes

Layer ou *Squeue* é sempre realizada nas listas de *Layers* dos objetos derivados da classe *Block*.

int GetSqueue(Sstring SelectedSqueue, Squeue * & _Squeue)

Obtém a *Squeue* denominada ***SelectedSqueue*** e retorna um ponteiro para ela em ***_Squeue***. Se a *Squeue* não for encontrada o código de erro retornado é 1. A operação normal da função retorna 0. A busca das filas nos objetos derivados das classes *Layer* ou *Squeue* é sempre realizada nas listas de *Squeues* dos objetos derivados da classe *Block*.

Squeue * GetSqueue(Sstring SelectedSqueue)

Obtém a *Squeue* denominada ***SelectedSqueue*** e retorna um ponteiro para ela. Se a *Squeue* não for encontrada o código de erro retornado é NULL (0). A operação normal da função retorna o valor do ponteiro. A busca das filas nos objetos derivados das classes *Layer* ou *Squeue* é sempre realizada nas listas de *Squeues* dos objetos derivados da classe *Block*.

Connection * GetConnection(Sstring SelectedConnection)

Obtém a *Connection* denominada ***SelectedConnection*** e retorna um ponteiro para ela. Se a *Connection* não for encontrada o código de erro retornado é NULL (0). A operação normal da função retorna o valor do ponteiro. A busca das conexões é realizada nas lista de conexões dos objetos derivados do classe *Layer* ou classe Block *Squeue*. O local da procura depende de chamar a função em uma ou outra classe. Este método só está disponível nas classes *Block* e *Layer*.

B.2.4 - FUNÇÕES DE ACESSO A ELEMENTOS DO KERNEL

Connection * GetConnectionFromKernel (Sstring _Connection)

Retorna um ponteiro para o objeto *Connection* denominado ***_Connection***, caso contrário retorna NULL (0).

Block * GetBlockFromKernel (Sstring _Block)

Retorna um ponteiro para o objeto *Block* denominado ***_Block***, caso contrário retorna NULL (0).

B.2.5 - FUNÇÕES DE GERAÇÃO DE EVENTOS

B.2.5.1 EVENTOS DE LAÇO (LOOP)

int CreateLoopEvent(double time, Sstring type)

Gera um evento especificado pelo tipo ***type*** para ser executado no instante ***time***. O evento é enviado para o mesmo objeto que o gerou.

int CreateLoopEvent(double time, Sstring type, SMessage * _SMessage)

Gera um evento especificado pelo tipo ***type*** para ser executado no instante ***time***, transportando a mensagem *_SMessage*. O evento é enviado para o mesmo objeto que o gerou.

B.2.5.2 EVENTO LOCAL (LOCAL)

int CreateLocalLayerEvent(double time, Sstring type, Sstring DestinyLayerName)

Gera um evento especificado pelo tipo ***type*** para ser executado no instante ***tim***e. O evento é enviado para a *Layer* ***DestinyLayerName*** contida no mesmo *Block* do objeto que o gerou o evento.

int CreateLocalLayerEvent(double time, Sstring type, Sstring DestinyLayerName, SMessage *_SMessage).

Gera um evento especificado pelo tipo ***type*** para ser executado no instante ***tim***e, transportando a mensagem *_SMessage*. O evento é enviado para a *Layer* ***DestinyLayerName*** contida no mesmo *Block* do objeto que o gerou o evento.

int CreateLocalSqueueEvent(double time, Sstring type, Sstring DestinySqueue)

Gera um evento especificado pelo tipo ***type*** para ser executado no instante ***time***. O evento é enviado para a *Squeue* ***DestinySqueue*** contida no mesmo *Block* do objeto que o gerou o evento.

int CreateLocalSqueueEvent(double time, Sstring type, Sstring DestinySqueue, SMessage *_SMessage)

Gera um evento especificado pelo tipo ***type*** para ser executado no instante ***tim***e, transportando a mensagem *_SMessage*. O evento é enviado para a *Squeue* ***DestinySqueue*** contida no mesmo *Block* do objeto que o gerou o evento.

int CreateLocalBlockEvent(double time, Sstring type);

Gera um evento para o *Block* que contem a *Layer* ou *Squeue* que esta gerando o evento. O evento possui o tipo **type** e será executado no instante **time**. Este método só está disponível nas classes *Layer* e *Squeue*.

int CreateLocalBlockEvent(double time, Sstring type, SMessage * _SMessage);

Gera um evento para o *Block* que contem a *Layer* ou *Squeue* que esta gerando o evento. O evento possui o tipo **type** e será executado no instante **time** com a mensagem ***_SMessage***. Este método só está disponível nas classes *Layer* e *Squeue*.

OBS: As seguintes funções de eventos locais são similares as descritas anteriormente. Apenas as variam os elementos transportados no evento e o objeto local ao qual o evento se destina.

int CreateLocalLayerEvent(double time, Sstring type, Sstring DestinyLayer);

int CreateLocalLayerEvent(double time, Sstring type, Sstring DestinyLayer, SMessage *_SMessage);

int CreateLocalSqueueEvent(double time, Sstring type, Sstring DestinySqueue);

int CreateLocalSqueueEvent(double time, Sstring type, Sstring DestinySqueue, SMessage *_SMessage);

B.2.5.3 EVENTOS DE CONTROLE (CONTROL)

int CreateBlockControlEvent(double time, Sstring type, Sstring DestinyBlock, SMessage *_SMessage)

Gera um evento especificado pelo tipo ***type***, a ser executado no instante ***time***, transportando a mensagem ***_SMessage***. O evento é enviado para o bloco ***DestinyBlock*** que pode ser qualquer *Block* existente na simulação.

int CreateLayerControlEvent(double time, Sstring type, Sstring DestinyBlock, Sstring DestinyLayerName, SMessage * _SMessage)

Gera um evento especificado pelo tipo ***type***, a ser executado no instante ***time***, transportando a mensagem ***_SMessage***. O evento é enviado para o bloco ***DestinyBlock*** que pode ser qualquer *Block* existente na simulação e para a camada de destino ***DestinyLayerName*** presente no bloco de destino.

int CreateSqueueControlEvent(double time, Sstring type, Sstring DestinyBlock, Sstring DestinySqueue, SMessage * _SMessage)

Gera um evento especificado pelo tipo ***type***, a ser executado no instante ***time***, transportando a mensagem ***_SMessage***. O evento é enviado para o bloco ***DestinyBlock*** que pode ser qualquer *Block* existente na simulação e para a fila de destino ***DestinySqueue*** presente no bloco de destino.

B.2.6 - FUNÇÕES PARA A MANIPULAÇÃO DE PARÂMETROS

O objeto parâmetro é também um recurso extraído do trabalho [1] que permite a configuração de parâmetros e variáveis de entrada para a simulação nos arquivos .SCL. O objeto parâmetro possui várias funções de interface que variam de acordo com a

localização do objeto parâmetro. Existem cinco possíveis localizações para um parâmetro dentro do ambiente de simulação.

- **Parâmetro Local**: o parâmetro está localizado no mesmo objeto *(Block*, *Layer* ou *Squeue*) que realiza a chamada da função.
- **Parâmetro Global**: o parâmetro está localizado no *Kernel* e é acessível aos modelos.
- **Parâmetro do *Block***: o parâmetro está localizado em um *Block* que não está executando a chamada da função.
- **Parâmetro da *Layer***: o parâmetro está localizado em uma *Layer* que não está executando a chamada da função.
- **Parâmetro da *SQueue***: o parâmetro está localizado em uma *Squeue* que não está executando a chamada da função.

B.2.6.1 PARÂMETROS LOCAIS

As funções para manipular parâmetros locais são:

void PutParam(Sstring Name, SIM_TYPE Valor, Sstring Description="",Sstring Option="");

Coloca o parâmetro ***Name*** no conjunto de parâmetros localizado em um *Block*, *Layer* ou *Squeue*. O valor do parâmetro é ***Valor*** do tipo ***SIM_TYPE*** (os tipos suportados atualmente são **int**, **double**, **long** and **Sstring**). O parâmetro possui uma descrição description de sua função armazenada em ***Description*** e um campo de opções ***Option*** não utilizado no ***lsimnt***, mas mantido por razões de compatibilidade com o simulador ***Hydragyrum*** [1].

void PutParam(Sstring Name, SIM_TYPE* Valor, long Rows, Sstring Description="", Sstring Option="");

Coloca o parâmetro ***Name*** no conjunto de parâmetros localizado em um *Block*, *Layer* ou *Squeue*. O valor do parâmetro é ***Valor*** que é um vetor do tipo ***SIM_TYPE*** (os tipos suportados atualmente são **int**, **double**, **long** and **Sstring**) com o tamanho ***Row***s. O

parâmetro possui uma descrição description de sua função armazenada em ***Description*** e um campo de opções ***Option*** não utilizado no ***lsimnt***, mas mantido por razões de compatibilidade com o simulador ***Hydragyrum*** [1].

void PutParam(Sstring Name,SIM_TYPE** Valor, long Rows, long Cols, Sstring Description="", Sstring Option="");

Coloca o parâmetro ***Name*** no conjunto de parâmetros localizado em um *Block*, *Layer* ou *Squeue*. O valor do parâmetro é ***Valor*** que é uma matriz do tipo ***SIM_TYPE*** (os tipos suportados atualmente são **int**, **double**, **long** and **Sstring**) com o tamanho de ***Rows*** por ***Cols***. O parâmetro possui uma descrição description de sua função armazenada em ***Description*** e um campo de opções ***Option*** não utilizado no ***lsimnt***, mas mantido por razões de compatibilidade com o simulador ***Hydragyrum*** [1].

int GetParam(Sstring Name, SIM_TYPE& Valor)

Obtém o parâmetro ***Name*** e retorna ele para uma variável de tipo apropriado ***(SIM_TYPE***) denominada ***Valor***.

int GetParam(Sstring Name, SIM_TYPE* Valor)

Obtém o parâmetro ***Name*** e retorna ele para uma variável que é um vetor do tipo apropriado ***(SIM_TYPE***) denominada ***Valor***.

int GetParam(Sstring Name, SIM_TYPE** Valor)

Obtém o parâmetro ***Name*** e retorna ele para uma variável que é uma matriz do tipo apropriado ***(SIM_TYPE***) denominada ***Valor***.

int GetSizeParam(Sstring Name,long& Rows,long& Cols)

Obtém as dimensões do parâmetro denominado ***Name*** retornando suas ***Rows*** e ***Cols***.

Sstring GetTypeParam(Sstring Name)

Obtém o tipo do parâmetro denominado ***Name***.

B.2.6.2 PARÂMETROS GLOBAIS

As funções que manipulam parâmetros globais são similares as funções para a manipulação de parâmetros locais exceto que o parâmetro global é armazenado no *Kernel* e pode ser acessado por qualquer elemento da simulação. A descrição dos argumentos das funções apresentadas a seguir é a mesma da dos parâmetros locais. Por isto, só a interface será mostrada. Para uma descrição dos argumentos refira a função de parâmetros local equivalente.

void PutGlobalParameter(Sstring Name, SIM_TYPE Valor, Sstring Description, Sstring Option)

void PutGlobalParameter (Sstring Name, SIM_TYPE* Valor, long Rows, Sstring Description="", Sstring Option="");

void PutGlobalParameter (Sstring Name,SIM_TYPE** Valor, long Rows, long Cols, Sstring Description="", Sstring Option="");

int GetGlobalParameter(Sstring Name,SIM_TYPE& Valor)

int GetGlobalParameter (Sstring Name,SIM_TYPE* Valor)

int GetGlobalParameter (Sstring Name,SIM_TYPE** Valor)

B.2.6.3 PARÂMETROS DO BLOCK

As funções que manipulam parâmetros do *Block* são similares as funções para a manipulação de parâmetros locais. Neste caso o parâmetro está armazenado em um *Block* que não é o objeto que faz a chamada a função. A descrição dos argumentos das funções apresentadas a seguir é a mesma da dos parâmetros locais, o único argumento adicional é o nome do *Block* aonde a função será aplicada, por isto, só a interface será mostrada. Para uma descrição dos argumentos refira a função de parâmetros local equivalente.

void PutBlockParameter(Sstring BlockName, Sstring Name,SIM_TYPE Valor, Sstring Description, Sstring Option).

void PutBlockParameter (Sstring BlockName, Sstring Name, SIM_TYPE* Valor, long Rows, Sstring Description="", Sstring Option="");

void PutBlockParameter (Sstring BlockName, Sstring Name,SIM_TYPE** Valor, long Rows long Cols, Sstring Description="", Sstring Option="");

int GetBlockParameter(Sstring BlockName, Sstring Name,SIM_TYPE& Valor)

int GetBlockParameter (Sstring BlockName, Sstring Name,SIM_TYPE* Valor)

int GetBlockParameter (Sstring BlockName, Sstring Name,SIM_TYPE** Valor)

B.2.6.4 PARÂMETROS DA LAYER

As funções que manipulam parâmetros da *Layer* são similares às funções para a manipulação de parâmetros locais. Neste caso o parâmetro está armazenado em uma *Layer* que não é o objeto que faz a chamada a função. A descrição dos argumentos das funções apresentadas a seguir é a mesma da dos parâmetros locais, os únicos argumentos adicionais são nome do *Block* e da camada (*Layer*) aonde a função será aplicada, por isto, só a interface será mostrada. Para uma descrição dos argumentos refira a função de parâmetros local equivalente.

void PutLayerParameter (Sstring BlockName, Sstring LayerName, Sstring Name, SIM_TYPE Valor, Sstring Description, Sstring Option)

void PutLayerParameter (Sstring BlockName, Sstring LayerName, Sstring Name, SIM_TYPE *Valor, long Rows, Sstring Description="", Sstring Option="");

void PutLayerParameter (Sstring BlockName, Sstring LayerName, Sstring Name, SIM_TYPE **Valor, long Rows, long Cols, Sstring Description="", Sstring Option="");

int GetLayerParameter (Sstring BlockName, Sstring LayerName, Sstring Name,SIM_TYPE& Valor);

int GetLayerParameter (Sstring BlockName, Sstring LayerName, Sstring Name,SIM_TYPE *Valor)

int GetLayerParameter (Sstring BlockName, Sstring LayerName, Sstring Name,SIM_TYPE **Valor)

B.2.6.5 PARÂMETROS DA SQUEUE

As funções que manipulam parâmetros da *Squeue* são similares às funções para a manipulação de parâmetros locais. Neste caso o parâmetro está armazenado em uma *Squeue* que não é o objeto que faz a chamada a função. A descrição dos argumentos das funções apresentadas a seguir é a mesma da dos parâmetros locais, os únicos argumentos adicionais são nome do *Block* e da *Squeue* aonde a função será aplicada, por isto, só a interface será mostrada. Para uma descrição dos argumentos refira à função de parâmetros local equivalente.

void PutSqueueParameter (Sstring BlockName, Sstring SqueueName, Sstring Name, SIM_TYPE Valor, Sstring Description, Sstring Option)

void PutSqueueParameter (Sstring BlockName, Sstring SqueueName, Sstring Name, SIM_TYPE* Valor, long Rows, Sstring Description="", Sstring Option="");

void PutSqueueParameter (Sstring BlockName, Sstring SqueueName, Sstring Name, SIM_TYPE** Valor, long Rows, long Cols, Sstring Description="", Sstring Option="");

int GetSqueueParameter (Sstring BlockName, Sstring SqueueName, Sstring Name,SIM_TYPE &Valor)

int GetSqueueParameter (Sstring BlockName, Sstring SqueueName, Sstring Name,SIM_TYPE *Valor)

int GetSqueueParameter (Sstring BlockName, Sstring SqueueName, Sstring Name,SIM_TYPE** Valor).

B.2.7 - AS FUNÇÕES DA CLASSE DE EVENTOS

Os eventos transportam a informação e a mensagem entre os elementos da simulação. Quando o evento chega a um local onde deve ser processado, a função *Ru*n() ou *Post()*, a informação e mensagem que transporta devem ser extraída. As funções para a extração de dados do evento são:

double GetEventTime ()

Obtém o tempo do evento. Na função *Run()* representa o tempo corrente da simulação.

Sstring GetEventType ()

Obtém o tipo do evento.

SMessage*GetEventMessage()

Obtém um ponteiro para a mensagem transportada pelo evento. A mensagem precisa ser convertida para o tipo apropriado da classe derivada de *SMessage* que representa a mensagem antes de ser manipulada.

B.2.8 - A CLASSE CONNECTION

A classe *Connection* representa uma conexão entre um *Block* e *Layer* para outro *Block* e *Layer* em outro modelo ou no mesmo modelo. A interface da classe *Connection* inclui o conjunto de funções para recuperar informação da classe *Connection*.

Sstring GetName()

Obtém o nome da conexão.

Sstring GetType()

Obtém o tipo da conexão.

Sstring GetBeginBlockName();

Obtém o nome do *Block* de início da conexão.

Sstring GetEndBlockName();

Obtém o nome do *Block* de fim da conexão.

Sstring GetBeginLayerName();

Obtém o nome da *Layer* de início da conexão no *Block* de início.

Sstring GetEndLayerName();

Obtém o nome da *Layer* de fim da conexão no *Block* de fim.

Sstring GetBeginLayerType();

Obtém o tipo da *Layer* de início da conexão no *Block* de início.

Sstring GetEndLayerType();

Obtém o tipo da *Layer* de fim da conexão no *Block* de fim.

Sstring GetLayerName(Sstring BlockName);

Obtém o nome da *Layer* de uma conexão que está associada ao *Block* ***BlockName*** na mesma extremidade da conexão.

Sstring GetBlockName(Sstring LayerName);

Obtém o nome do *Block* de uma conexão que está associado a *Layer* ***LayerName*** na mesma extremidade da conexão.

Sstring GetConnectedLayerName(Sstring LayerName);

Obtém o nome da *Layer* de uma conexão que está conectada a *Layer* ***LayerName*** na outra extremidade da conexão.

Sstring GetConnectedBlockName(Sstring BlockName);

Obtém o nome do *Block* de uma conexão que está conectado ao *Block* ***BlockName*** na outra extremidade da conexão.

Block * GetBeginBlock();

Obtém um ponteiro para o *Block* de início da conexão.

Block * GetEndBlock();

Obtém um ponteiro para o *Block* de fim da conexão.

Layer * GetBeginLayer();

Obtém um ponteiro para a *Layer* de início da conexão.

Layer * GetEndLayer();

Obtém um ponteiro para a *Layer* de fim da conexão.

Layer * GetLayer(Sstring LayerName);

Obtém um ponteiro para a *Layer* da conexão dado o nome da *Layer* ***LayerName***.

Block * GetBlock(Sstring BlockName);

Obtém um ponteiro para o *Block* da conexão dado o nome do *Block* ***BlockName***.

Layer * GetLayerOfBlock(Sstring BlockName);

Obtém um ponteiro para a *Layer* da conexão dado o nome do *Block* ***BlockName*** associado.

Block * GetBlockOfLayer(Sstring LayerName);

Obtém um ponteiro para o *Block* da conexão dado o nome da *Layer* ***LayerName*** associada.

B.2.9 - A CLASSE TABLE

A tabela é uma classe que fornece um dicionário duplamente indexado com duas chaves *Sstring* que armazenam um valor do tipo SYM_TYPE (os valores de SYM_TYPE são **int**, **long**, **double** e **Sstring**). A estrutura da tabela está mostrada na Tabela B-1.

Primeira Chave	**Segunda Chave**	**Valor**
Sstring	Sstring	SYM_TYPE
Exemplo		
"BTE_1"	"Connection_1"	23
"Source_1"	"CD_1"	"OFF"

Estrutura de armazenamento e procura usada na classe Table.

A classe *Table* como a *Parameter* são classes base das classes de modelagem *(Block*, *Layer* e *Squeue*)

A classe *Table* tem uma interface composta das seguintes funções:

void Delete(Sstring Key1, Sstring Key2)

Apaga a entrada armazenada com ***Key1*** e ***Key2*** da tabela.

PutTableData(Sstring Key1, Sstring Key2, SIM_TYPE Valor)

Armazena o valor ***Valor*** na tabela com as chaves ***Key1*** e **Key**2.

GetTableData(Sstring Key1, Sstring Key2, SIM_TYPE &Valor)

Obtém o valor ***Valor*** da tabela que possui as chaves ***Key1*** e ***Key***2.

GetTableFirstKey(Sstring Key2, SIM_TYPE Valor, Sstring &Key1)

Dada a segunda chave ***Key2*** e o valor ***Valor*** recupera a primeira chave associada ***Key***1.

GetTableFirstKeys(Sstring Key2, SIM_TYPE Valor, bvector<Sstring> &V)

Dada a segunda chave ***Key2*** e o valor ***Valor*** recupera todas as primeiras chaves associada V.

GetTableFirstKey(Sstring Key2A, SIM_TYPE ValorA, Sstring Key2B, SIM_TYPE ValorB, Sstring &Key1).

Dada a segunda chave ***Key2A*** e o valor ***ValorA*** e a segunda chave ***Key2B*** e o valor ***ValorB*** recupera a primeira ocorrência da primeira chave com esta combinação de parâmetros ***Key1***.

GetTableFirstKeys (Sstring Key2A, SIM_TYPE ValorA, Sstring Key2B, SIM_TYPE ValorB, bvector<Sstring> &V)

Dada a segunda chave ***Key2A*** e o valor ***ValorA*** e a segunda chave ***Key2B*** e o valor ***ValorB*** recupera todas as primeiras chaves com esta combinação de parâmetros V.

OBS: As seguintes funções de manipulação de parâmetros podem ser chamadas de classe de modelagem para acessar parâmetros em outros objetos.

int PutBlockTableData(Sstring BlockName, Sstring Key, Sstring Name, SIM_TYPE Valor)

int GetBlockTableData(Sstring BlockName, Sstring Key, Sstring Name, SIM_TYPE &Valor)

int PutLayerTableData(Sstring BlockName, Sstring LayerName, Sstring Key, Sstring Name, SIM_TYPE Valor)

int GetLayerTableData(Sstring BlockName, Sstring LayerName, Sstring Key, Sstring Name, SIM_TYPE &Valor).

Apêndice C. DETALHES DA IMPLEMENTAÇÃO DO SIMULADOR

Veremos na seqüência alguns detalhes do núcleo do simulador.

C.1 - O NÚCLEO DO SIMULADOR

O núcleo do simulador possui seu código distribuído em arquivos, os quais podemos agrupar em três grupos:

1. Arquivos com codificação dos objetos que foram desenvolvidos neste trabalho. Estes arquivos estão listados na Tabela 15.

ARQUIVO	DESCRIÇÃO
lw_main.cpp	Contém, além da função *main()* do aplicativo lsimnt, a declaração e a implementação da classe *LW_Kernel*. Esta classe constitui o núcleo do simulador
lw_command.cpp	Implementação dos métodos da classe *LW_Command*, na qual implementa-se o interpretador de comandos mantido pelo núcleo do simulador
lw_event_manager.cpp	Implementação dos métodos da classe *EventManager*, o gerenciador de eventos mantido pelo núcleo
lw_block.cpp	Implementação dos métodos da classe *Block*, base para criação de novos modelos
lw_layer.cpp	Implementação dos métodos da classe *Layer*, base para criação de camadas presentes em modelos
lw_squeue.cpp	Implementação dos métodos da classe *SMessage*, base para criação de tipos de mensagens para comunicação entre modelos
lw_connect.cpp	Implementação dos métodos da classe *Connection*, usada para representar as conexões entre blocos
lw_message.cpp	Implementação dos métodos da classe *Layer*, base para criação de camadas presentes em modelos
lw_event.cpp	Implementação dos métodos da classe *Event*, que representa um evento agendado. Um evento pode carregar uma mensagem (objeto *SMessage*)

Tabela 15 - Arquivos com o código dos objetos desenvolvidos neste trabalho

2. Arquivos de cabeçalhos com a definição (protótipos) dos objetos que foram desenvolvidos neste trabalho. Estes arquivos estão listados na Tabela 16.

ARQUIVO	DESCRIÇÃO
lw_main.h	Contém a declaração de variáveis e inclusão de arquivos de cabeçalho usados nas funções presentes no arquivo lw_main.cpp
lw_command.h	Declaração da classe *LW_Command*, o interpretador de comandos mantido pelo núcleo do simulador
lw_event_manager.h	Declaração da classe *EventManager*, o gerenciador de eventos mantido pelo núcleo
block.h	Declaração da classe Block, base para criação de novos modelos
layer.h	Declaraçãoda classe *Layer*, base para criação de camadas presentes em modelos
squeue.h	Declaração da classe *SMessage*, base para criação de tipos de mensagens para comunicação entre modelos
lw_connect.h	Declaração da classe *Connection*, usada para representar as conexões entre blocos
smessage.h	Declaração da classe *Layer*, base para criação de camadas presentes em modelos
lw_event.h	Declaração da classe *Event*, que representa um evento agendado. Um evento pode carregar uma mensagem (objeto *SMessage*)

Tabela 16 - Arquivos com os cabeçalhos dos objetos desenvolvidos neste trabalho

3. Arquivos contendo código e cabeçalhos de estrutura auxiliares, tais como as classes *Sstring, SMessage, Param, Parameter, Joker e Table*. Tais estruturas de dados tem sido usadas em nosso grupo de pesquisa na área de simulação há alguns anos, e foram usadas novamente aqui neste trabalho. Tais estruturas são descritas em [4]. Os nomes dos arquivos com o código destas estruturas estão listados na Tabela 17.

Joker.cpp	bhash.h	bstack.h	param.h
param.cpp	biterator.h	btable.h	parameter.h
sstring.cpp	blist.h	btree.h	simdefs.h
Table.cpp	bpriorque.h	bvector.h	sstring.h
	bqueue.h	joker.h	table.h

Tabela 17 - Arquivos contendo código e cabeçalhos de estrutura auxiliares

C.2 - AS CLASSES DO SIMULADOR

A seguir temos uma descrição das principais classes que compõem o simulador.

C.2.1 - A Classe *LW_Kernel*

Nesta classe é implementado o núcleo do simulador em si. A função *main()* do aplicativo lsimnt, existente no arquivo lw_main.cpp, cria uma instância desta classe, e invoca o método *simulate()*, repassando a ele os argumentos fornecidos pelo usuário na linha de comando.

A classe é ilustrada na Figura 33. A classe contém um objeto interpretador de comandos (*Command_Interpreter*), um objeto gerenciador de eventos (*Main_Scheduler*), e duas listas de objetos, uma para manter os blocos (*Main_Block_List*), e outra para manter as conexões entre blocos (*Main_Connection_List*) existentes na simulação corrente.

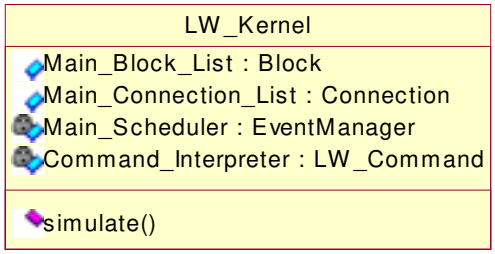

Figura 33 - Classe LW_Kernel

O ponteiro para uma lista de blocos contidos na simulação corrente (*Block *Main_Block_List*), como mostra a Figura 33, é um atributo público. Assim, a lista de blocos da simulação é construída pelo interpretador de comandos (*Command_Interpreter*) no processamento de linhas do arquivo .SCL que instrua a criação de um novo bloco. Igualmente, esta lista é usada também pelo escalonador de eventos (*EventManager*).

O ponteiro para uma lista de conexões contidas na simulação corrente (*Connection *Main_Connection_List*), como mostra a figura 1, é um atributo público. Assim, a lista de conexões da simulação é construída pelo interpretador de comandos (*Command_Interpreter*) no processamento das linhas do arquivo .SCL. Igualmente, esta lista é usada também pelo escalonador de eventos (*EventManager*).

Resumidamente, o simulador executa o processamento ilustrado na Figura 34.

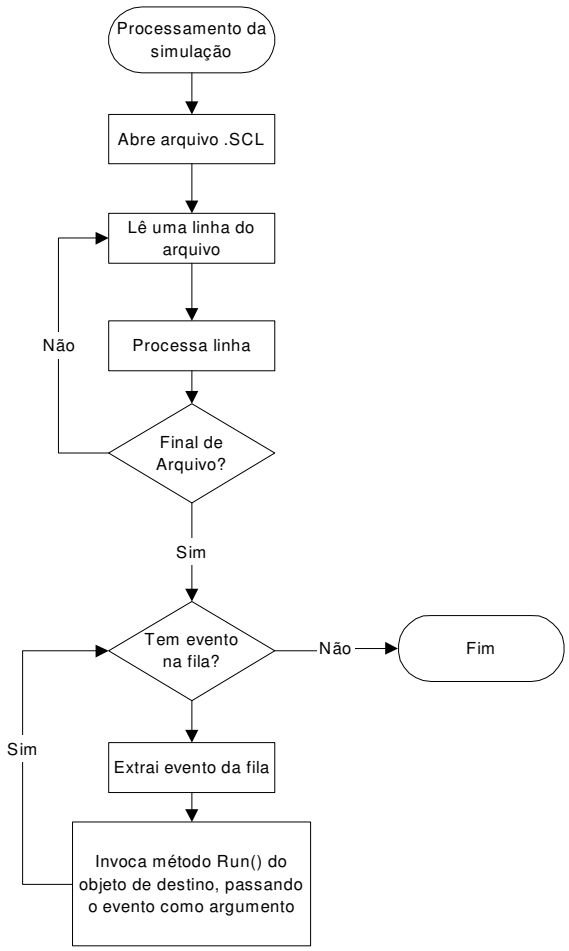

Figura 34 - Fluxograma básico do processamento

Os processos de "Abrir arquivo .SCL" e "Ler linha do arquivo" indicados na Figura 34 são executados pelo núcleo. Já a tarefa "Processa linha" é executada pelo objeto interpretador de comandos. Os processos restantes, indicados no fluxograma, são realizados pelo gerenciador de eventos.

Veremos mais detalhes dos objetos *Command_Interpreter* e *Main_Scheduler* na seqüência.

C.2.2 - A Classe *LW_Command*

Um objeto interpretador de comando (*LW_Command *Command_Interpreter*) responsável pela interpretação dos comandos lidos do arquivo .SCL. A cada linha lida, no

processamento do método *simulate()* de *LW_Kernel*, é invocado o método *Exec()* do objeto *Command_Interpreter*, passando como argumento a linha lida. O interpretador de comandos então interpreta o comando. A classe *LW_Command* é ilustrada na Figura 35.

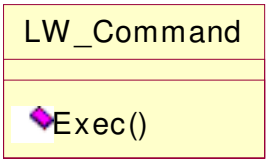

Figura 35 - A classe LW_Command

C.2.3 - A Classe *EventManager*

Um objeto escalonador de eventos (*EventManager *Main_Scheduler*), cuja classe é ilustrada na Figura 36. Ele usa uma lista de eventos (*Event *Event_List*) para armazenar os eventos agendados pelos modelos através do método *Schedule()*. O método *Run()* é invocado pelo núcleo ao final da leitura do arquivo .SCL, e é nele que se dá efetivamente o processo da simulação.

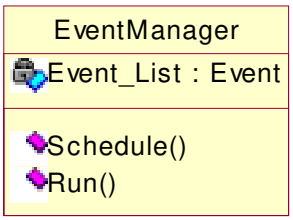

Figura 36 - A classe EventManager

O processamento do gerenciador de eventos é mostrado nos fragmentos de código do método *Schedule()* na Figura 37, e do método *Run()* na Figura 38. Repare que, no método *Schedule()* os novos eventos são inseridos na lista de eventos (*Event_List*) de forma tal que esta lista mantém todos os eventos ordenados pelo tempo. Já no método *Run()* do objeto gerenciador de eventos, os eventos são retirados da lista de eventos, e em cada evento, o objeto destino tem seu método de manipulação invocado. Note que o nome do método de manipulação dos objetos receptores de eventos também é *Run()*.

```
int EventManager::Schedule(Event * New_Event){
    Event *pt_anterior;
```

```
    Event *pt_atual;
    if (Event_List == NULL) {  // Lista de eventos está vazia
        Event_List = New_Event;
        return 0;
    }
    Event *pt_lista = Event_List;
    if ( (New_Event->Time) <= (pt_lista->Time) ) { // Novo evento é o primeiro
        New_Event->prox = pt_lista;
        Event_List = New_Event;
    } else {   // Varre a lista até encontrar o ponto para inserir o  evento
        pt_atual = pt_lista;
        pt_anterior = pt_atual;
        while ((pt_atual != NULL) && (pt_atual->Time < New_Event->Time)) {
            pt_anterior = pt_atual;
            pt_atual = pt_atual->prox;
        }
        pt_anterior->prox = New_Event;
        New_Event->prox = pt_atual;
        Event_List = pt_lista;
    }
    return 0;
}
```

Figura 37 - Método para agendamento de eventos

```
int EventManager::Run(double Final_Time){
    for (;;) {
        Event *pt_atual = Event_List;       //Toma o primeiro da fila
        Event_List = Event_List->prox;      //Avança a fila
         if (pt_atual == NULL)                //Fila de eventos está vazia
             return(0);
         pt_atual->DestinationLayer->Run(pt_atual);
         if (pt_atual->Time > Final_Time)  //Tempo final de simulação atingido
             return(0);
         delete(pt_atual);
    }
}
```

Figura 38 - Método para execução de eventos

Tipicamente, os modelos agendam eventos na inicialização, antes da invocação do método *Run()* mostrado na Figura 38, de forma que ao executar tal método a fila de eventos não esteja vazia no instante da sua invocação. O método *Run()* simplesmente retira o primeiro evento da fila (os eventos são mantidos na lista ordenados pelo tempo de execução do evento), identifica o objeto destinatário, e então invoca o método *Run()* daquele objeto (que pode ser um objeto derivado da classe *Block*, *Layer* ou *Squeue*), passando como argumento o próprio objeto *Event*. Tal informação pode ser usada pelo objeto destinatário na manipulação do evento, pois contém informações tais como o instante de ocorrência do evento, um ponteiro para o objeto que originou o evento, e opcionalmente alguma mensagem (derivada da classe base *Smessage*), usada para comunicação.

Todos os objetos derivados de *Block*, *Layer* e *Squeue* mantém uma referência ao escalonador de eventos, a partir da qual são agendados eventos pelos modelos desenvolvidos.

C.2.4 - A Classe *Block*

A classe *Block* é ilustrada na Figura 39.

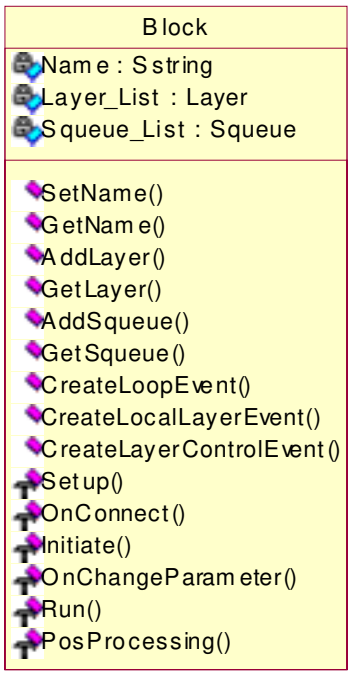

Figura 39 - A classe Block

Cada bloco é identificado por um nome (atributo *Name*, manipulado através dos métodos públicos *SetName()* e *GetName()*), e mantém duas listas: uma para manter as camadas (objetos derivados da classe *Layer*), agregados no bloco, e outra para manter as filas (objetos derivados da classe *Squeue*) contidas no bloco. Todos os modelos são implementados na forma de classes derivadas da classe base *Block*, ou seja, para cada modelo de elemento de rede, o núcleo manterá uma instância de um objeto derivado de *Block*. Esta classe pode agregar instâncias de objetos derivados das classes base *Layer* e *Squeue*, como mostra a Figura 40.

Figura 40 - Relacionamento entre objetos do tipo *Block*, *Layer* e *Squeue*

Os métodos *SetName()*, *AddLayer()* e *SetSqueue()* são usados pelo interpretador de comandos para configurar os blocos durante a leitura do arquivo .SCL. Os métodos *GetName()*, *GetLayer()* e *GetSqueue()* são disponibilizadas aos objetos derivados de *Block* para programação de modelos.

Os métodos *CreateLoopEvent()*, *CreateLocalLayerEvent()* e *CreateLayerControlEvent()* são disponibilizadas aos objetos derivados de *Block* para o agendamento de eventos nos modelos desenvolvidos.

Os métodos *Setup(), OnConnect(), Initiate(), OnChangeParameter(), Run()* e *PosProcessing()* são métodos virtuais, e sua implementação é obrigatória em todos os objetos derivados de *Block*.

C.2.5 - A Classe *Layer*

A classe *Layer* é ilustrada na Figura 41.

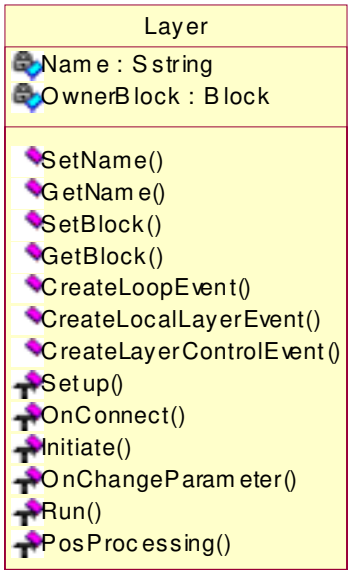

Figura 41 - A classe *Layer*

Cada camada é identificada por um nome (atributo *Name*, manipulado através dos métodos públicos *SetName()* e *GetName()*), e mantém um ponteiro para o bloco ao qual a camada pertence (atributo *OwnerBlock*, manipulado através dos métodos públicos *SetBlock()* e *GetBlock()*).

Os métodos *CreateLoopEvent(), CreateLocalLayerEvent()* e *CreateLayerControlEvent()* são disponibilizadas aos objetos derivados de *Layer* para o agendamento de eventos nos modelos desenvolvidos.

Os métodos *Setup(), OnConnect(), Initiate(), OnChangeParameter(), Run()* e *PosProcessing()* são métodos virtuais, e sua implementação é obrigatória em todos os objetos derivados de *Layer*.

C.2.6 - A Classe *Squeue*

A classe Squeue é ilustrada na Figura 42.

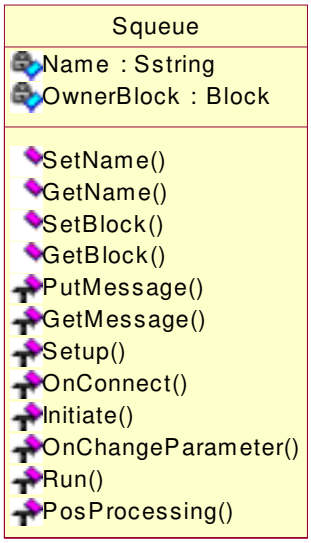

Figura 42 - A classe *Squeue*

Cada fila é identificada por um nome (atributo *Name*, manipulado através dos métodos públicos *SetName()* e *GetName()*), e mantém um ponteiro para o bloco ao qual a fila pertence (atributo *OwnerBlock*, manipulado através dos métodos públicos *SetBlock()* e *GetBlock()*).

Na programação da fila, os métodos virtuais *PutMessage()* e *GetMessage()* devem ser implementados para inserir mensagem e retirar mensagens das filas, respectivamente.

Os métodos *Setup(), OnConnect(), Initiate(), OnChangeParameter(), Run()* e *PosProcessing()* são métodos virtuais, e sua implementação é obrigatória em todos os objetos derivados de *Squeue*.

C.2.7 - A Classe *Connection*

A classe Connection é ilustrada na Figura 43.

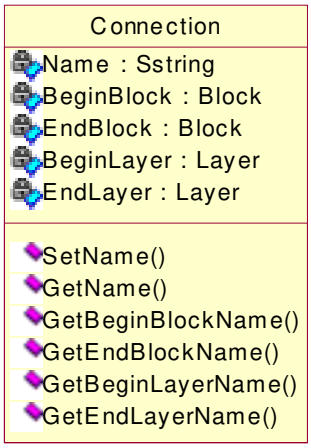

Figura 43 - A classe *Connection*

Como vimos, a conexão sempre se dá ao nível de camadas (*Layer*). Cada conexão é identificada por um nome (atributo *Name*), e referências a seus blocos (*Block*) e camadas (*Layer*) iniciais e finais.

A classe *Connection* pode ser usada, por exemplo, para representar conexões físicas ou de enlaces entre os elementos de redes modelados.

C.2.8 - A Classe *Event*

A classe *Event*, a qual é usada para representar os eventos do simulador ***lsimnt***, é ilustrada na Figura 44.

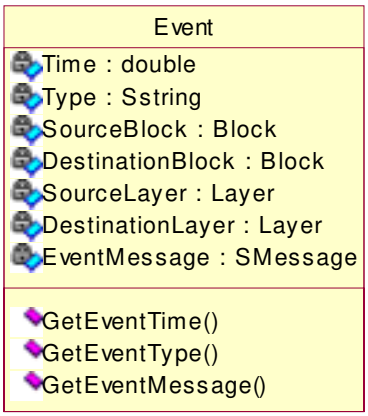

Figura 44 - A classe *Event*

Os atributos da classe *Event* são descritos na Tabela 18:

ATRIBUTO	DESCRIÇÃO
Time	Tempo programado para a execução (em segundos). Um modelo pode obter este tempo invocando o método *GetEventTime()* do objeto *Event* recebido como argumento em *Run()*
Type	Este campo serve para identificar o tipo de mensagem que o evento está carregando. Um modelo pode obter o tipo de evento com o método *GetEventType()* do objeto *Event* recebido como argumento em *Run()*
SourceBlock	Nome do bloco onde o evento foi gerado
DestinationBlock	Nome do bloco a quem o evento se destina. Este campo deve ser sempre preenchido, caso contrário, o evento será ignorado pelo núcleo
SourceLayer	Nome da camada onde o evento foi gerado
DestinationLayer	Nome da camada ao qual o evento se destina
EventMessage	Os eventos são usados para transportar mensagens entre camadas. Este atributo é usado para armazenar a informação a ser trocada, um objeto derivado da classe *SMessage*. Um modelo pode recuperar a mensagem com o método *GetEventMessage()* do objeto *Event* recebido como argumento em *Run()*

Tabela 18 - Atributos da classe *Event*

Os eventos são agendados pelos modelos através da invocação de métodos tais como *CreateLoopEvent(), CreateLocalLayerEvent()* e *CreateLayerControlEvent()*.

C.2.9 - A Classe *SMessage*

A classe *SMessage*, cujos objetos derivados podem ser usados para representar mensagens trocadas entre modelos durante uma simulação, é ilustrada na Figura 45.

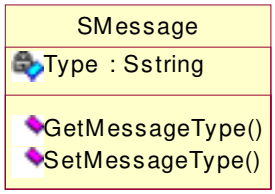

Figura 45 - A classe *SMessage*

A classe *SMessage* é a classe base para a criação de novos tipos genéricos de mensagens a serem transportadas pelos eventos do simulador. Exemplos típicos de mensagens que podem ser criadas com a extensão desta classe são objetos para representar pacotes TCP, células ATM, ou quadros Ethernet, trocados entre elementos de rede, como ilustra a Figura 46.

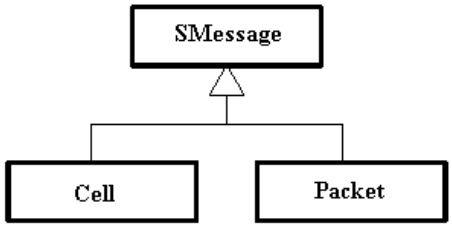

Figura 46 – Extensão da classe *SMessage* para a criação de tipos de mensagens

C.3 - RELACIONAMENTO ENTRE AS CLASSES

Na Figura 47 temos a representação que resume o relacionamento entre as classes apresentadas até aqui.

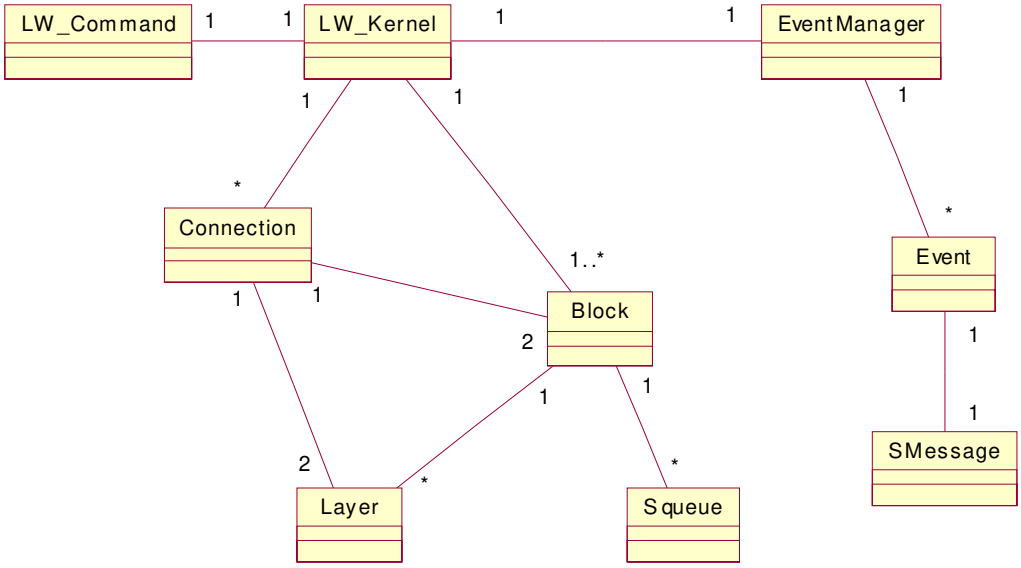

Figura 47 - Relacionamento entre as classes

Os números indicados ao lado das associações indicam a cardinalidade [18]. Como podemos observar, o lsimnt possui um objeto do tipo *LW_Kernel*, o qual contém:

- Um objeto interpretador de comandos (*LW_Command*);
- Um objeto gerenciador de eventos (*EventManager*);
- De zero a N (representado por *) conexões, representadas por objetos do tipo *Connection*; e
- N objetos do tipo *Block*.

Cada objeto do tipo *Connection* possui dois objetos do tipo *Block*, e dois objetos do tipo *Layer* (existe uma camada no lado inicial, e outra camada no lado terminal da conexão, sendo as duas camadas pertencentes a blocos diferentes).

Cada objeto do tipo *Block* possui de zero a N objetos do tipo *Layer*, e de zero a N objetos do tipo *Squeue*.

O objeto gerenciador de eventos (*EventManager*) possui em sua agenda de zero a N eventos, representados por objetos do tipo *Event*. Cada objeto do tipo *Event* carrega uma mensagem, representada por um objeto do tipo *SMessage*.