



Universidade do Porto

Faculdade de Engenharia

FEUP

Algoritmo de Escalonamento para Diferenciação de Qualidade de Serviço em Redes DiffServ/MPLS

DANIEL FILIPE DA SILVA RAMOS

Licenciado em Engenharia Electrotécnica e de Computadores
pela Faculdade de Engenharia da Universidade do Porto

Dissertação realizada sob a supervisão do professor

José António Ruela Simões Fernandes

Departamento de Engenharia Electrotécnica e de Computadores da Universidade do Porto

Porto, Maio de 2009

*Para a minha avó Rosa,
pelo seu amor e carinho.*

Avó estarás para sempre no meu coração!

Agradecimentos

Agradeço aos meus pais, Manuel e Ana Maria, por todo o esforço e dedicação ao longo da minha vida, sem os quais nunca teria sido possível chegar até aqui. Estou-lhes eternamente agradecido pela educação, amor, carinho e paciência.

À Carla por me fazer sentir especial, pelo amor e carinho, e pela inesgotável paciência, compreensão e força nos momentos mais difíceis. Sem ti não teria sido possível... O verdadeiro amor é aquele que ao longo do tempo nos consome e nos indica o verdadeiro caminho.

A toda a minha família e amigos, obrigado pelo apoio e por estarem sempre presentes.

Ao professor José Ruela pela oportunidade de realização deste trabalho, pelo seu apoio e orientação durante este projecto, e por sempre me ter mostrado o caminho certo para a obtenção dos resultados pretendidos. Professor obrigado pela amizade, pelo empenho que sempre demonstrou desde o primeiro dia.

Obrigado a todos.

Sumário

Autor	Daniel Filipe da Silva Ramos
Titulo	Algoritmo de Escalonamento para Diferenciação de Qualidade de Serviço em Redes
Ano	2009
Supervisão	Professor José António Ruela Simões Fernandes
Palavras Chave	Redes de Comunicação, Serviços Diferenciados, Qualidade de Serviço, Simulação, <i>NS - Network Simulation</i> , Escalonamento de Pacotes

Resumo

As redes de comunicações têm assumido um papel cada vez mais importante no quotidiano das pessoas, empresas e instituições. O constante desenvolvimento das redes de comunicações e a globalização da informação permitiu assistir, em menos de meia década, ao aumento dos débitos de transferência da informação no acesso à Internet através da linha do assinante de 56kbit/s para 30Mbits/s. As redes de comunicações são cada vez mais sofisticadas e complexas e os serviços suportados estão em constante renovação e crescimento.

A engenharia de tráfego assume uma importância fundamental neste cenário complexo em constante evolução, fornecendo metodologias para dimensionar os recursos e configurar os mecanismos de controlo de redes de forma eficiente. Uma das ferramentas utilizadas pela engenharia de tráfego é a simulação que permite estudar o desempenho das redes de comunicações em detalhe. Permite, também, avaliar e completar modelos teóricos, escolher entre as diferentes alternativas os vários mecanismos de controlo de rede e determinar os valores mais adequados dos seus parâmetros, dimensionar os recursos das redes de forma eficiente e estudar novas evoluções com base em estimativas de crescimento de tráfego.

Esta dissertação tem como objectivo idealizar um novo algoritmo de escalonamento que tire partido da combinação de mecanismos de engenharia de tráfego suportados em modelos já adoptados, de forma a ultrapassar as limitações de soluções existentes.

Para validar o novo algoritmo será utilizado o simulador de redes NS-2 (Network Simulator).

Abstract

Author	Daniel Filipe da Silva Ramos
Title	Algoritmo de Escalonamento para Diferenciação de Qualidade de Serviço em Redes
Year	2009
Supervisor	José António Ruela Simões Fernandes
Keywords	Communication Networks, Differentiated Services, Quality of Service, Simulation, <i>NS - Network Simulation</i> , Packet Scheduling

Resume

The role of communications networks is being increasingly important in the daily life of persons, companies and institutions. The constant development of the communications networks and the globalization of the information allowed us to attend, in less than one half of decade, to the increase of the information transmission rate in the Internet access through the subscriber liner, from 56Kbit/s to 30Mbit/s. The communications networks are increasingly sophisticated and complex and the supported services are constantly renewing and growing.

The Traffic Engineering is of fundamental importance in this complex environment in constant evolution, supplying methodologies for an efficient network dimensioning and choice of the correct parameters of the network control mechanisms.

One of the tools usually used by the Traffic Engineering is the simulation. The simulation allows the accomplishment of performance studies of the communications networks with significant detail, in general larger than the one provided by analytical models. The simulation enables to evaluate and enhance theoretical models, choose between different available control network mechanisms and evaluate the adequate values of its parameters, dimension the network resources in an efficient way and study the networks evolution with information on the traffic growth estimation.

The objective of this thesis is to create new scheduling algorithm which combine traffic engineering mechanisms supported in models already adopted, to solve the limitations of existing solutions.

To validate this new algorithm we used the network simulator NS-2 (Network Simulator-2).

Lista de Abreviaturas e Acrónimos

AF	Assured Forwarding
ATM	Asynchronous Transfer Mode
BA	Behavior Aggregate
BW	Bandwidth
CBR	Constant Bit Rate
CBQ	Class Based Queueing
CoS	Class of Services
DiffServ	Differentiated Services
DP	Drop Precedence
DS	Differentiated Services
DSCP	Differentiated Services Code Point
ECN	Explicit Congestion Notification
EF	Expedited Forwarding
FEC	Forwarding Equivalence Class
FIFO	First In First Out
FQ	Fair Queuing
FTP	File Transfer Protocol
HTB	Hierarchical Token Bucket
IESG	Internet Engineering Steering Group
IETF	Internet Engineering Task Force
ISP	Internet Service Provider
IntServ	Integrated Services
IP	Internet Protocol
IPDV	IP Packet Delay Variation
LAN	Local Area Network
LB	Largura de Banda
LDP	Label Distribution Protocol
LER	Label Edge Router

LSP	Label Switched Path
LSR	Label Switching Router
LLQ	Low Latency Queueing
MPLS	MultiProtocol Label Switching
NS	Network Simulator
OA	Ordered Aggregate
OSI	Open Systems Interconnection
OWD	One Way Delay
PDB	Per Domain Behavior
PHB	Per Hop Behavior
PQ	Priority Queuing
PSC	PHB Scheduler Classes
QoS	Qualidade de Serviço
QoS	Quality of Service
RED	Random Early Detection
RR	Round Robin
RSpec	Service Specification
RSVP	Resource ReSerVation Protocol
RTT	Round trip time
SCFQ	Self-Clocked Fair Queueing
SFQ	Start-time Fair Queueing
SLA	Service-Level Agreement
SLS	Service-Level Specification
SMTP	Simple Mail Transfer Protocol
TCA	Traffic Conditioning Agreement
TCP	Transmission Control Protocol
TCS	Traffic Conditioning Specifications
TE	Traffic Engineering
TELNET	TELecommunication NETwork
ToS	Type of Service
TSpec	Traffic descriptor

TTL	Time To Live
UDP	User Datagram Protocol
VoIP	Voice over IP
WFQ	Weighted Fair Queueing
WRED	Weighted Random Early Detection
WRR	Weighted Round Robin

Índice Geral

1.Introdução	27
1.1. Enquadramento	27
1.1.1. Qualidade de Serviço, QoS	32
1.2. Objectivos / Caracterização sucinta do problema	36
1.3. Organização da dissertação	37
2.Mecanismos de suporte à QoS	39
2.1. Controlo de admissão	39
2.2. Classificação	40
2.3. Marcação	40
2.4. Policiamento e <i>Shaping</i>	40
2.4.1. Leaky Bucket	41
2.4.2. Token Bucket	42
2.5. Escalonamento	43
2.5.1. Algoritmo First Come First Serve, FCFS ou First In First Out, FIFO	45
2.5.2. Algoritmo Priority Queueing, PQ	46
2.5.3. Algoritmo Generalized Processor Sharing, GPS	47
2.5.4. Algoritmo Round Robin, RR	49
2.5.5. Algoritmo Weighted Round Robin, WRR	51
2.5.6. Algoritmo Weighted Fair Queueing, WFQ	51
2.5.7. Algoritmo Self-Clocking Fair Queueing, SCFQ	53
2.5.8. Algoritmo Start-Time Fair Queueing, SFQ	54
2.5.9. Algoritmo Worst-Case Fair Weighted Fair Queueing, WF ² Q	54
2.5.10. Algoritmo Class Based Queueing, CBQ	57
2.5.11. Algoritmo Hierarchical Token Bucket, HTB	60
2.6. Descarte	63
3.Modelos de QoS / Engenharia de tráfego	65
3.1. Serviços Integrados, IntServ	65
3.2. Serviços Diferenciados, DiffServ	70
3.3. MultiProtocol Label Switching	77
3.3.1. Engenharia de Tráfego no MPLS	81
3.3.2. <i>MPLS Support of DiffServ</i>	83
4.O ambiente de simulação	89
4.1. A necessidade de um simulador	89
4.2. Network Simulator (NS)	91

4.3.	Arquitectura do NS	92
4.4.	Objectos presentes no NS	93
4.5.	Módulo Diffserv	100
4.6.	Tipos de fontes de tráfego	102
4.6.1.	Tráfego Constant Bit Rate, CBR	102
4.6.2.	Tráfego Exponencial	102
5.	Proposta de um novo algoritmo de escalonamento	105
5.1.	Introdução	105
5.2.	Objectivos	108
5.3.	Pressupostos e relações importantes	109
5.4.	Princípios do modelo	112
5.4.1.	Política de empréstimos	113
5.4.2.	Comparação com serviço de referência	117
5.4.3.	Algoritmo de escalonamento	122
6.	Simulação e avaliação do desempenho	129
6.1.	Análise e validação	129
7.	Conclusões	151
7.1.	Trabalho futuro	152
8.	Referências	153
	Anexos	161
	Anexo 1	162
	Anexo 2	177
	Anexo 3	192
	Anexo 4	208

Índice Figuras

Figura 1.1: Exemplo do <i>jitter</i> para as aplicações	34
Figura 2.1: Leaky Bucket	42
Figura 2.2: <i>Token Bucket</i>	43
Figura 2.3: Escalonador FIFO	45
Figura 2.4: Escalonador PQ	46
Figura 2.5: Escalonador GPS	49
Figura 2.6: Round Robin.....	50
Figura 2.7: Escalonador WFQ.....	52
Figura 2.8: Comparação entre os mecanismos de escalonamento GPS, WFQ, WF ² Q	56
Figura 2.9: Escalonador CBQ	57
Figura 2.10: Distribuição da capacidade da ligação.....	58
Figura 2.11: Distribuição da capacidade da ligação com estrutura hierárquica.....	59
Figura 2.13: Exemplo prático de uma árvore HTB.....	62
Figura 3.1: Arquitectura IntServ	67
Figura 3.2: Troca de mensagens PATH e RESV	68
Figura 3.3: Elementos de uma infra-estrutura de comunicações DiffServ.	70
Figura 3.4: ToS e Diffserv/ECN	71
Figura 3.5: Funcionamento do DiffServ	72
Figura 3.6: Modelo de classificação e condicionamento de pacotes	73
Figura 3.7: Componente de Controlo e de Transporte	79
Figura 3.8: MPLS Shim Header	80
Figura 3.9: Exemplo de uma rede MPLS com três níveis.....	81
Figura 3.10: Rede MPLS DiffServ.....	84
Figura 3.11: Mapeamento entre um cabeçalho IP e um cabeçalho MPLS SHIM para E-LSP	85
Figura 3.12: Mapeamento entre um cabeçalho IP e um cabeçalho MPLS SHIM para L-LSP	85
Figura 3.13: Fluxo de pacotes em MPLS sem DiffServ	86

Figura 3.14: Fluxo de pacotes em MPLS com DiffServ	87
Figura 4.1: Arquitectura do NS	92
Figura 4.2: Aspecto geral do ponto de vista de um utilizador do NS	93
Figura 4.3: Exemplo da representação de nós em NS.....	94
Figura 4.4: Objectos presentes numa ligação unidireccional	95
Figura 4.5: Cenário de uma simulação com agentes.....	96
Figura 4.6: Exemplo de valores extraídos do <i>Queue Monitor</i>	98
Figura 4.7: NAM	99
Figura 4.8: Exemplo de gráfico criado pelo Gnuplot.....	100
Figura 5.1: Ideia base do empréstimo de largura de banda no algoritmo a implementar	111
Figura 6.1: Modelo inicial apenas com dois tipos de tráfego EF e BE	130
Figura 6.2: Geração de tráfego AF.....	137
Figura 6.3: Tráfego EF gerado	138
Figura 6.4: Tráfego AF gerado.....	139
Figura 6.5: Tráfego BE gerado.....	139
Figura 6.6: Alterações nos valores das filas reais e virtuais do tráfego BE com diferentes valores de b num caso de saturação.....	146
Figura 6.7: Alterações nos valores das filas reais e virtuais do tráfego AF com diferentes valores de b num caso de saturação.....	148

Índice Tabelas

Tabela 3.1: <i>Codepoints</i> DiffServ AF recomendados	76
Tabela 3.2: Tabela comparativa entre os mecanismos DiffServ, MPLS-TE e DiffServ-TE	88
Tabela 5.1: Algoritmo proposto	126
Tabela 5.2: Simplificação do algoritmo proposto	127
Tabela 6.1: Resultados da simulação parte 1	133
Tabela 6.2: Tabela comparativa no caso de saturação usando WFQ	142
Tabela 6.3: Tabela comparativa no caso de próximo da saturação usando WFQ.....	143
Tabela 6.4: Tabela comparativa em ambos os cenários usando PQ	143
Tabela 6.5: Tabela comparativa em ambos os cenários usando SF	144
Tabela 6.6: Tabela comparativa em ambos os casos usando SF com diferentes valores de b	145
Tabela 6.7: Tabela comparativa usando SF com diferentes valores de b	149

1. INTRODUÇÃO

1.1. ENQUADRAMENTO

Nos anos 70 do século XX, quando foi enviado o primeiro e-mail e depois nos anos seguintes com a utilização massiva deste e outros serviços (por exemplo, transferência remota de ficheiros), percebeu-se que mais cedo ou mais tarde o volume de tráfego iria provocar uma grande alteração no modo como as redes eram geridas, não só para suportar esse aumento mas igualmente para o fazer de forma eficiente e económica.

Para o efeito desenvolveu-se a técnica de comutação de pacotes, comunicação de dados em que pacotes (unidade de transferência de informação) são individualmente comutados e encaminhados entre nós da rede através de ligações tipicamente partilhadas por múltiplos fluxos de dados. Isto contrasta com o paradigma da comutação de circuitos, que se baseia no estabelecimento de canais dedicados entre nós de forma a constituir circuitos de comunicação, de uso exclusivo, entre pontos periféricos da rede.

A comutação por pacotes pode efectuar-se:

- **Com ligação (circuito virtual):** é estabelecido um caminho fixo (mas sem atribuição estática de recursos, como na comutação de circuitos) e todos os pacotes

associados a um circuito virtual seguirão o mesmo caminho. Uma grande vantagem é que oferece uma elevada garantia de entrega dos pacotes, com preservação da respectiva ordem dos pacotes. Como exemplo temos o ATM (comutação de células), o Frame Relay e o X.25;

- **Sem ligação (datagrama):** os pacotes são encaminhados independentemente, oferecendo flexibilidade e robustez superiores, já que a rede pode adaptar-se a possíveis quebras de ligações ou nós. As decisões de encaminhamento baseiam-se no endereço de destino presente no cabeçalho dos pacotes (por exemplo, endereço IP). As redes IP baseiam-se neste modelo e o protocolo IP tornou-se a referência (plataforma universal) para interligação de redes de diferentes tecnologias.

No princípio, as redes IP existiam sem qualquer mecanismo de qualidade de serviço. A pilha protocolar TCP/IP base não foi planeada de raiz para suportar tráfego de voz ou outros serviços que têm requisitos estritos de largura de banda¹, atrasos e *jitter*. As redes IP foram desenvolvidas tendo como uma das suas premissas básicas poderem ser utilizadas com diversos meios físicos e tecnologias (“IP sobre tudo”), de forma a viabilizar a comunicação entre aplicações em rede (tipicamente segundo o modelo cliente-servidor).

Neste contexto, as decisões arquitectónicas tomadas na concepção do protocolo IP foram, na sua maioria, no sentido da simplificação, visando responder ao cenário imaginado na época. Sendo assim, o protocolo IP oferecia um serviço de comutação de datagramas (*connectionless*), não fiável (por outras palavras, e usando a terminologia habitual, um serviço *best effort* [Best Effort]).

O cenário das redes IP apresentado anteriormente mudou. O novo cenário baseia-se na premissa de que um grande número de aplicações suportadas em redes IP devem beneficiar de qualidade de serviço. Ou seja, a situação actual é no sentido de “Tudo sobre IP”.

¹ A expressão “largura de banda” será usada no decorrer desta dissertação também com o significado de capacidade do canal

Sendo assim, e visto que o paradigma mudou, a questão que se colocou foi a de identificar as eventuais limitações do IP e perceber os procedimentos necessários para adequá-lo à nova realidade. A qualidade de serviço em redes IP não é necessariamente resolvida com um único protocolo ou algoritmo. Na maioria dos casos e dependendo das necessidades das aplicações, um conjunto de novos algoritmos deve ser utilizado.

Os desafios na utilização do IP como plataforma de suporte para aplicações em redes são em resumo os seguintes:

- O IP, como protocolo, não oferece praticamente nenhuma garantia de qualidade de serviço.
- A base instalada do IP é muito grande, o que torna a mudança do protocolo uma ideia pouco viável.

O primeiro desafio é de carácter técnico e diz respeito ao paradigma inicialmente previsto para o protocolo que enfatizava a simplicidade de concepção. Por exemplo, o IP não oferece qualquer garantia de largura de banda para uma aplicação em particular. Além disso, uma aplicação não pode obter do IP garantia de entrega dos seus próprios pacotes, que podem ser descartados pela rede, sem que seja tomada qualquer acção correctiva (se necessário, a garantia de entrega é remetida para a camada de Transporte com recurso ao protocolo TCP). Não existe igualmente nenhuma garantia de tempo de entrega dos pacotes, o que é agravado com retransmissões efectuadas pela camada de Transporte, caso as aplicações corram sobre TCP.

O segundo desafio é uma questão de como se adequar ao novo paradigma sem efectivamente mudar o protocolo já instalado. O IPv4 deverá em breve ser substituído pelo IPv6 mas, mesmo neste caso, mantém-se o paradigma da simplicidade inicial. O IPv6 aborda outra questão de implementação do protocolo (endereçoamento, segurança, ...) e não apresenta nenhuma solução completa para os desafios citados.

A forma de abordar as “deficiências” do protocolo IP consiste então em propor novos protocolos, algoritmos e mecanismos que tratem das limitações arquitectónicas

intrínsecas ao protocolo e permitam o suporte efectivo de qualquer tipo de aplicação sobre redes IP.

Sendo assim, os fornecedores de serviços IP (ISP – *Internet Service Providers*) foram obrigados a investir em soluções que se baseiam em mecanismos de controlo de tráfego e de Qualidade de Serviço (QoS – *Quality of Service*), pois existem inúmeras razões para tal. **Primeiro**, com o crescimento uniforme e gradual do volume de tráfego nas suas redes, os ISPs descobriram que é difícil combater o congestionamento apenas com provisão de mais largura de banda. Alterações nas distribuições de tráfego e falhas nos nós/ligações podem resultar em congestionamentos imprevisíveis, e claro está que sobredimensionar recursos (*overprovisioning*) pode tornar-se extremamente dispendioso. **Segundo**, sendo o crescimento das redes uma realidade, o aumento do investimento no melhoramento e na instalação de novos equipamentos físicos não é suficiente para dar resposta aos requisitos das novas aplicações, pelo que têm de apostar igualmente em melhorar o desempenho da infra-estrutura actual. **Terceiro**, e muito importante, a indústria tem apostado em novos modelos de negócio por forma a responder melhor às expectativas dos seus clientes, por exemplo diferenciação no tipo de serviços. No futuro, e mesmo a curto prazo, é expectável que as redes venham a oferecer serviços de voz, dados e vídeo de uma forma que seja fácil de operar e de gerir, suportando entre outras as seguintes aplicações:

- Telefonia e Fax sobre IP (VoIP)
- Comércio Electrónico
- Vídeo sobre IP
- Educação à Distância (*E-Learning*)
- Vídeo-conferência
- Aplicações Colaborativas e de Grupo
- Aplicações Multimédia
- Aplicações de Tempo Real
- Outras

Genericamente, a maioria das aplicações citadas são aplicações multimédia na medida em que envolvem a transferência de múltiplos tipos de informação (dados, voz, vídeo, gráficos, ...) com requisitos de tempo real e até de sincronização entre componentes (por exemplo, áudio e vídeo) e de preservação da relação temporal entre unidades de dados de um mesmo fluxo (isto é, eliminação do *jitter*) para operarem com qualidade.

As redes IP constituem hoje em dia a infra-estrutura global de comunicações. O desenvolvimento de um crescente número de aplicações que tiram partido das redes actuais de comunicações coloca novos desafios na gestão e na atribuição de recursos, de forma a satisfazer um conjunto diversificado de requisitos e garantias de desempenho. Algumas aplicações, como as de áudio, têm exigências rígidas quanto ao atraso extremo-a-extremo, mas podem tolerar perda de pacotes (desde que a respectiva taxa seja inferior a um limiar crítico), enquanto outras, como transferência de ficheiros, são sensíveis a este último aspecto, mas as exigências quanto ao atraso são pouco severas (o que permite o recurso a um serviço de transporte fiável baseado no TCP).

No entanto, paralelamente a este crescimento têm sido efectuados inúmeros esforços para desenvolver novas soluções arquitectónicas e tecnologias de suporte, de modo a melhorar a qualidade de serviço, permitindo assim que os fornecedores de serviço pudessem responder aos seus clientes em quantidade, qualidade e com a máxima fiabilidade. Desenvolveram-se vários [mecanismos de suporte a QoS](#) que podem ser considerados como componentes (*Building Blocks*) de arquitecturas de QoS. Com este objectivo, o IETF definiu vários [modelos](#) que têm como objectivo melhorar o desempenho e a qualidade de serviço combinando os mecanismos de suporte do modo mais eficiente, de salientar o *Integrated Services* (IntServ) e o *Differentiated Services* (DiffServ). Este trabalho adopta como ponto de partida o modelo DiffServ. Para além disso, existem técnicas de [engenharia de tráfego](#), suportadas por exemplo por *Multi Protocol Label Switching* (MPLS) que, isoladamente ou em conjunto com os modelos anteriores, permitem distribuir tráfego na rede de modo a otimizar a utilização de recursos e evitar congestionamento.

Mas, mais em concreto, o que se entende por “qualidade de serviço”?

1.1.1. QUALIDADE DE SERVIÇO, QoS

Quando a rede dispõe de um conjunto de mecanismos que lhe permite satisfazer diferentes requisitos de tráfego e de serviço de elementos de rede, podemos afirmar que essa rede suporta Qualidade de Serviço. Isto só é possível se houver interacção entre várias camadas protocolares e cooperação entre equipamentos da rede. As necessidades dos utilizadores e das aplicações ao nível da qualidade devem ser reflectidas em valores de atributos de serviços de rede. Esses valores devem ser acordados aquando da negociação entre os ISPs e os utilizadores.

Assim sendo, numa primeira abordagem podemos afirmar que o termo “Qualidade de Serviço” pode ser entendido da seguinte forma:

QoS está associada a um requisito das aplicações para a qual se exige que determinados atributos (atrasos, perdas, largura de banda,...) estejam dentro de limites bem definidos.

A qualidade de serviço negociada deve ser garantida pela rede. Do ponto de vista dos programas de aplicação, a QoS é tipicamente expressa e solicitada em termos de um “Pedido de Serviço” ou “Contrato de Serviço”. O contrato de serviço é denominado tipicamente por *Service Level Agreement*, SLA. O SLA deve definir claramente quais os requisitos que devem ser garantidos para que as aplicações possam executar com a qualidade esperada, de acordo com o contrato. Assim sendo, um SLA pode incluir, entre outros, os seguintes parâmetros:

- Largura de banda e taxa de transferência (*throughput*)

A largura de banda é o parâmetro mais básico da QoS e é necessário para a operação adequada de qualquer aplicação. Este parâmetro indica a quantidade de informação

que nominalmente pode ser transferida de um nó para outro num determinado período de tempo. Ou seja, quando um cliente contrata um débito (máximo, médio, etc.), a rede, tendo em atenção a política de gestão de recursos e a natureza das garantias, disponibiliza uma certa largura de banda para o efeito, capaz de suportar o débito pedido pelo cliente. A quantidade de dados reservada é contabilizada para efeito de controlo de admissão.

- *Atraso*

O atraso é um parâmetro também importante para garantir QoS. Do ponto de vista da aplicação, o atraso é o tempo necessário para a entrega de um pacote, desde a emissão até à recepção. De uma maneira geral, o atraso da rede pode ser entendido como o somatório dos atrasos impostos pela rede. Os parâmetros mais importantes para o valor do atraso são: atraso de propagação, atraso de transmissão, tempo de processamento nos equipamentos (incluindo o processo de comutação) e o tempo (variável) que os pacotes permanecem nas filas de espera.

O atraso pode ser expresso por meio de parâmetros como o *one-way delay* (OWD) [Almes] ou como o *round-trip delay* ou *round-trip time* (RTT).

Round-trip delay é importante em aplicações interactivas, como navegação na Web ou conversacionais, como o serviço de voz; por outro lado, o RTT tem impacto directo no desempenho de protocolos de transporte fiáveis, como o TCP (que requer, na sua operação, uma estimativa do RTT).

- *Jitter*

É a variação do atraso na entrega de pacotes, causado pelo atravessamento de filas de espera nos nós, que pode ser agravado devido a congestionamento na rede, por alterações nas rotas, etc.

Para calcular o *Jitter*, pode-se utilizar o *IP Packet Delay Variation Metric* (ipdv) que mede a diferença entre o *one-way-delay* de um par de pacotes.

Fonte:[Semeria&Steward]

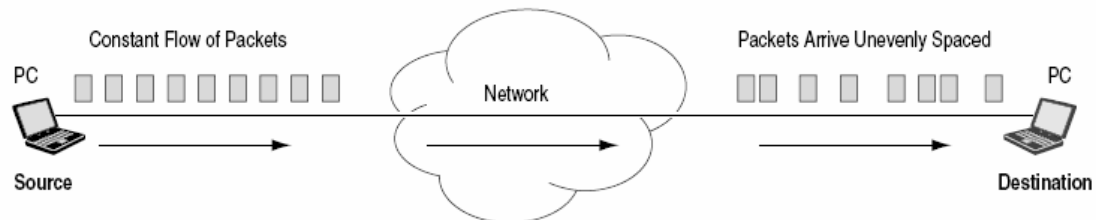


Figura 1.1: Exemplo do *jitter* para as aplicações

- Perdas

As perdas de pacotes ocorrem principalmente em função de factores tais como: descarte de pacotes nos *routers* e *switches* e perdas de pacotes devido a erros ocorridos no processo de transmissão e não recuperados na camada de ligação de dados (que, em muitos casos, se limita a detectar e a eliminar tramas com erros). Este é um problema sério para alguns tipos de aplicações, como por exemplo, voz sobre IP, desde que a taxa de ocorrência ultrapasse um valor crítico, uma vez que não é desejável recorrer a mecanismos de retransmissão na camada de Transporte (TCP). Do ponto de vista da qualidade de serviço da rede a preocupação é normalmente no sentido de especificar e garantir limites razoáveis consoante a aplicação em causa; o problema não é crítico para aplicações tolerantes ao atraso que podem beneficiar dos mecanismos de retransmissão do TCP, embora um número excessivo de retransmissões contribua para uma redução significativa da eficiência da rede e do débito útil das aplicações.

- Disponibilidade

A disponibilidade é um aspecto da qualidade de serviço abordada normalmente na fase de planeamento/projecto da rede. Em termos práticos, este parâmetro é uma garantia de execução da aplicação ao longo do tempo e depende de factores, tais como:

disponibilidade dos equipamentos, disponibilidade da rede física e da infra-estrutura física dos ISPs. As empresas dependem cada vez mais das redes para a viabilização dos seus negócios e, neste sentido, a disponibilidade é um requisito bastante rígido.

Assim sendo, podemos ver que diferentes aplicações dependem mais de uns parâmetros do que de outros. Por exemplo, aplicações de voz e multimédia são muito sensíveis ao atraso e ao *jitter*, admitindo a ocorrência de perdas ocasionais, enquanto outras aplicações de dados não toleram perdas, o que obriga à construção de um serviço de transporte fiável sobre um serviço de rede não fiável.

Em resumo, estamos a falar de QoS como um serviço necessário a várias aplicações. Mecanismos de qualidade de serviço e mecanismos de controlo de rede, nomeadamente funções de gestão de recursos, permitem a uma rede satisfazer parâmetros de qualidade. Para além disso, é importante perceber que o objectivo de uma arquitectura de QoS não é providenciar mais largura de banda, mas sim gerir a largura de banda disponível tendo em atenção os tipos de fluxos de tráfego presentes em cada momento na rede.

Tendo o conceito de QoS definido, olharemos agora para os requisitos fundamentais que devem ser considerados para garantir este conceito.

De forma a fornecer qualidade de serviço ao maior tipo de aplicações, a rede deve satisfazer duas condições:

- a primeira condição é que qualquer aplicação deve ter garantida a largura de banda acordada em qualquer circunstância (no limite, a garantia pode ser nula).
- a segunda condição é que o fluxo de tráfego de uma aplicação na rede deve receber um tratamento apropriado à sua classe de tráfego, incluindo escalonamento e descarte de pacotes.

Temos que pensar neste duas condições como ortogonais. Um fluxo deve ter largura de banda suficiente para ser transmitido, mas não deverá receber um tratamento diferente daquilo que é esperado para a sua classe de tráfego, ou seja, sofrer perdas

nem ficar muito tempo à espera de ser escalonado (a primeira condição é conhecida logo à entrada da rede, a segunda condição não). Alternativamente, um fluxo pode ser servido na maioria dos nós da rede mas pode eventualmente ser eliminado ou atrasado em algum nó por causa de uma ocasional escassez de largura de banda (a segunda condição é conhecida mas a primeira não). Apesar disso, é necessário satisfazer ambas as condições de forma a obter todas as garantias de qualidade de serviço requeridas tanto pelos fornecedores de serviços como pelos clientes.

1.2. OBJECTIVOS / CARACTERIZAÇÃO SUCINTA DO PROBLEMA

O principal objectivo desta dissertação foi idealizar um [novo algoritmo de escalonamento](#) que tirasse partido da combinação de mecanismos de Engenharia de Tráfego suportados por MPLS e do modelo de QoS DiffServ, de forma a ultrapassar as limitações de soluções existentes, conforme será discutido e analisado mais à frente.

Para atingir este objectivo será necessário analisar e testar através de simulação a entrega de pacotes numa rede com as características apresentadas, tendo em mente os seguintes objectivos intermédios:

- Verificar através da simulação o comportamento dinâmico dos níveis de qualidade de diferentes classes de tráfego;
- Verificar a capacidade de isolamento de tráfego e atribuição de largura de banda entre diversas classes de serviços tipificadas no modelo DiffServ;
- Verificar o efeito do tráfego *best effort*, ou ausência desse, sobre o atraso, variação do atraso e taxa de perda de pacotes nos fluxos de tráfego que necessitam de garantias (rígidas ou não);
- Verificar a adaptabilidade dos mecanismos de escalonamento para as necessidades dos clientes.

Atingidos estes objectivos intermédios, o objectivo final é a implementação, a configuração e a análise do novo algoritmo idealizado.

1.3. ORGANIZAÇÃO DA DISSERTAÇÃO

Esta dissertação está organizada em 8 capítulos.

O **Capítulo 2** apresenta os mecanismos de suporte à qualidade de serviço, tais como controlo de admissão, classificação, policiamento, *shaping*, algoritmos de escalonamento, descarte e marcação. Estes mecanismos são transversais a qualquer modelo, podendo ser combinados.

No **Capítulo 3** são apresentados dois modelos de Qualidade de serviço, o modelo de serviços integrados e o modelo de serviços diferenciados, para além de uma técnica de engenharia de tráfego, o *Multi Protocol Switching Label*.

O **Capítulo 4** apresenta o simulador NS-2, sendo explorados os conceitos básicos do NS necessários à criação, simulação e análise de uma ligação ponto a ponto.

No **Capítulo 5** são apresentados os pressupostos e é realizada uma caracterização detalhada do novo algoritmo proposto, bem como alguns detalhes de implementação.

O **Capítulo 6** inclui os cenários de simulação, os resultados e respectiva análise e a validação do modelo.

No **Capítulo 7** são apresentadas as conclusões genéricas da dissertação, bem como as linhas orientadoras do trabalho futuro.

No **Capítulo 8** são apresentadas todas as referências consultadas no decorrer deste trabalho.

2. MECANISMOS DE SUPORTE À QoS

Neste capítulo serão apresentados vários mecanismos de suporte à qualidade de serviço.

O controlo de tráfego é o processo de gestão, priorização, controlo ou redução do tráfego na rede, particularmente da largura de banda, para reduzir congestionamento, latência e perdas. O controlo de tráfego envolve controlo de admissão, policiamento, classificação (*classifying*), conformação (*shaping*), escalonamento (*scheduling*), descarte (*dropping*) e a marcação do tráfego (*marking*).

2.1. CONTROLO DE ADMISSÃO

Acções executadas pela rede na sequência do pedido de estabelecimento de uma nova conexão e que levam a um processo de decisão que consiste na aceitação ou rejeição do pedido. Uma quantidade de tráfego só é aceite na rede se, face aos recursos disponíveis e às características do tráfego, for previsível que é possível cumprir o contrato, sem comprometer (degradar) os contratos estabelecidos, isto é, a qualidade de serviço das conexões em curso deverá manter-se dentro dos limites negociados.

2.2. CLASSIFICAÇÃO

Este mecanismo realiza a classificação de pacotes com base no conteúdo do respectivo cabeçalho, de acordo com regras definidas, com o objectivo de permitir tratamento diferenciado de pacotes por parte de outros mecanismos. Em particular, permite separar fluxos distintos de dados e associá-los a diferentes filas de espera. As filas são baseadas no tipo de serviço. O algoritmo que serve as diferentes filas determina a taxa de serviço com que o tráfego deve deixar a fila.

É possível distinguir o tráfego em pelo menos três tipos:

- tráfego que pode ser transportado sem garantias de entrega, atrasos ou outras características associadas ao desempenho;
- tráfego que tem requisitos muito apertados relativamente à entrega e aos atrasos;
- tráfego em que é necessário garantir a entrega do mesmo mas não tendo garantias tão apertadas relativamente aos atrasos.

2.3. MARCAÇÃO

Marcar significa associar um pacote a uma classe ou tipo de serviço, através de um valor num campo do cabeçalho. Os pacotes são normalmente marcados (ou remarcados) nos nós de entrada de uma rede ou domínio (*edge nodes*). A marcação permite dar um tratamento diferenciado aos pacotes (quer nos nós de entrada, quer nos nós internos), nomeadamente no que se refere a *shaping*, descarte ou escalonamento.

2.4. POLICIAMENTO E SHAPING

Policimento e *shaping* são ferramentas utilizadas para identificar violações de quantidade de tráfego numa interface durante um intervalo de tempo. Ambos utilizam métodos semelhantes na classificação e na medição do tráfego classificando os pacotes respectivos como fora do perfil (*out-of-profile*) acordado com a rede, ou de acordo

com o perfil (*in-profile*). Os mecanismos diferem no modo como reagem a essas variações:

- A configuração de *shaping* tipicamente atrasa o tráfego excessivo utilizando um *buffer* para armazenar os pacotes e modelar o fluxo quando a transmissão de uma determinada origem é maior do que o esperado. O objectivo é garantir que o tráfego gerado esteja de acordo com o declarado e que portanto seja aceite pela rede, depois de policiado.
- A configuração de policiamento normalmente descarta o tráfego excedente e é utilizado para forçar que o tráfego recebido de um utilizador não ultrapasse o valor contratado em cada intervalo de tempo.

Para garantir o policiamento e o *shaping* são usados os mecanismos *Leaky Bucket* e o *Token Bucket*.

2.4.1. LEAKY BUCKET

O *Leaky Bucket* permite monitorizar os pacotes que entram na rede e verificar se os contratos de tráfego negociados no estabelecimento das ligações correspondentes, estão a ser respeitados. Este mecanismo tem como objectivo transformar tráfego *bursty* num fluxo em que o débito máximo instantâneo é controlado (r – *leak rate*). O *Leaky Bucket* aceita *bursts* na entrada, mas impede que ocorram na saída. Para suportar esta funcionalidade o *Leaky Bucket* possui um *buffer* de tamanho B , que limita o tamanho máximo dos *bursts* admitidos.

O funcionamento do *Leaky Bucket* é o seguinte:

- quando o *buffer* estiver cheio, novos pacotes são descartados;
- enquanto o *buffer* tiver pacotes, estes são espaçadas de $\frac{1}{r}$;
- o número máximo de pacotes que podem ser aceites sem perdas num *burst* submetido com débito $R > r$ é $N_{\max} = \frac{R \times B}{(R - r)}$;

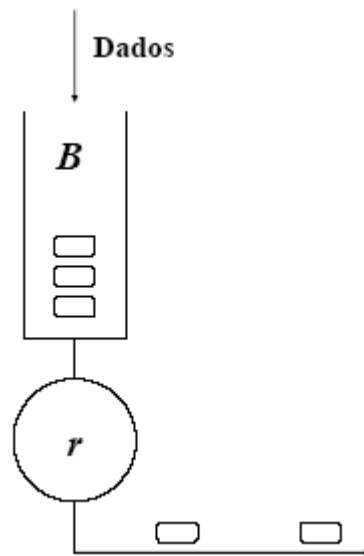


Figura 2.1: Leaky Bucket

No entanto este algoritmo tem um problema, pois não aproveita da melhor forma os recursos da rede. Isto acontece devido à sua taxa de transmissão constante.

2.4.2. TOKEN BUCKET

O objectivo deste mecanismo, *Token Bucket*, é controlar o débito médio de tráfego *bursty*, permitindo a transmissão de *bursts* com um débito instantâneo superior ao médio, mas limitando a sua duração, isto é, o *Token Bucket* permite *burstiness*, mas limita-a.

Sendo assim, o funcionamento do *Token Bucket* é o seguinte:

- Um *token* é adicionado ao *bucket* a cada $\frac{1}{r}$ segundos;
- O *bucket* pode conter no máximo b *tokens*. Se for gerado um *token* quando o *bucket* está cheio, é descartado;
- Quando um pacote de n *bytes* chega, são removidos n *tokens* do *bucket* e o pacote é escalonado para ser enviado;
- Se não houver n *tokens* disponíveis, nenhum *token* é removido do *bucket* e nenhum pacote é escalonado;

O algoritmo permite *bursts* de b bytes (de facto mais, pois à medida que *tokens* são consumidos, outros são gerados), mas tem que respeitar a taxa r ao longo do tempo. Pacotes não conformes podem ser tratados de vários modos:

- Podem ser descartados;
- Podem ser colocados na fila para serem transmitidos mais tarde, quando existe *tokens* suficientes no *bucket*;
- Podem ser transmitidos, mas marcados como tráfego não conforme, podendo ser descartados em nós a jusante se a rede estiver em sobrecarga;

Fonte: [Ruela_QoS_ATM]

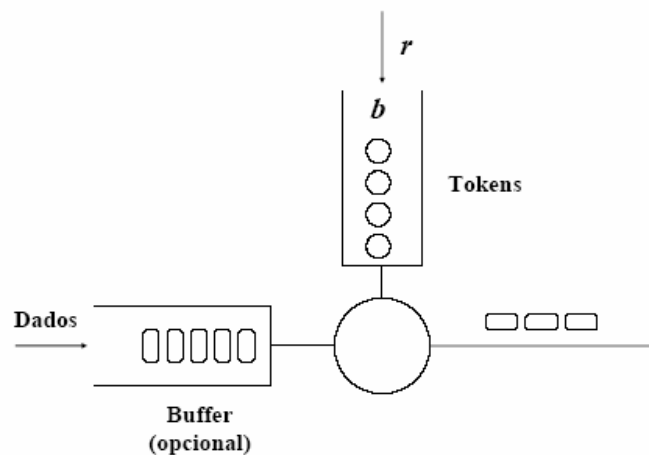


Figura 2.2: *Token Bucket*

2.5. ESCALONAMENTO

Um algoritmo de escalonamento decide qual o próximo pacote que será servido na fila de espera, permitindo assim distribuir a largura de banda da ligação pelos diferentes fluxos (atribuindo a cada fluxo a largura de banda que foi pedida e aceite).

A partilha de recursos em redes de comunicação e, em particular, a adopção de estratégias de multiplexagem estatística, originam situações de contenção, devido à competição pela utilização de recursos. Em redes que suportam integração de serviços torna-se necessário fornecer QoS diferenciada por categorias de serviço (ou classes de tráfego), oferecendo garantias de desempenho a aplicações críticas e ao mesmo tempo permitindo uma partilha de recursos de acordo com critérios de equidade (*fairness*).

Para se atingir estes objectivos é necessário recorrer a algoritmos de escalonamento (*scheduling algorithms*) pois estes permitem associar as disciplinas de serviços a fluxos aceites. O algoritmo de escalonamento decide a fila de espera a servir – e portanto o pacote à cabeça dessa fila. Este algoritmo é um dos mecanismos responsáveis por distribuir a largura de banda da ligação pelos diferentes fluxos (atribuindo a cada fluxo a largura de banda que foi pedida e aceite). Mas não é a única condição a satisfazer. Há também objectivos de atraso e portanto um compromisso.

Um algoritmo de escalonamento pode ser do tipo *work-conserving* ou *non-workconserving*. No primeiro caso, o servidor “trabalha” sempre, isto é, havendo pacotes em espera, eles serão sempre transmitidos. No segundo caso, um nó só pode transmitir um pacote quando este se torna elegível, isto é, quando o tempo necessário para ele se manter em espera termina. Se no nó apenas se encontrarem pacotes não elegíveis em espera, então o servidor manter-se-á inactivo. Este tipo de algoritmos de escalonamento foi projectado para aplicações que não toleram variações no atraso de transmissão. A desvantagem óbvia destes algoritmos é o desperdício de largura de banda durante os períodos em que apenas existem pacotes não elegíveis em espera.

A classificação destes algoritmos de escalonamento pode ser efectuada de acordo com os princípios que definem a ordem de envio dos pacotes: por ordem de chegada, de uma forma estrita, de uma forma rotativa, por aproximação ao modelo de fluidos, e em tempos predefinidos.

De seguida serão apresentados alguns dos principais algoritmos de escalonamento.

2.5.1. ALGORITMO FIRST COME FIRST SERVE, FCFS OU FIRST IN FIRST OUT, FIFO

O escalonador FCFS ou FIFO (são usadas ambas as designações) usa a técnica “primeiro a entrar, primeiro a sair”. Neste algoritmo os pacotes são servidos por ordem de chegada, não sendo possível diferenciar serviços porque o fluxo de tráfego é tratado pela fila de espera como um todo.

FIFO é em muitos casos o algoritmo de escalonamento padrão, porém possui muitas deficiências:

- não toma nenhuma decisão sobre a prioridade do pacote;
- a ordem de chegada determina a largura de banda que será obtida, a prioridade e a atribuição de *buffers*;
- não possui protecção contra aplicações ou fontes de tráfego com comportamento malicioso [Ferguson].

Fonte:[Semeria]

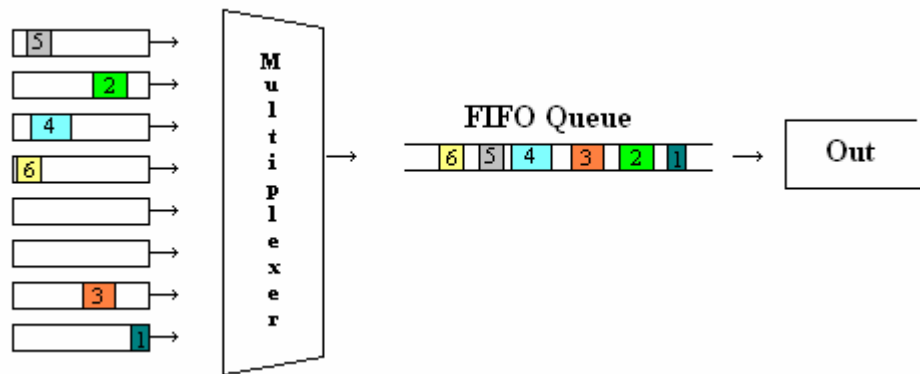


Figura 2.3: Escalonador FIFO

A Figura 2.3 mostra que à medida que os pacotes chegam são colocados na fila de saída pela mesma ordem que foram recebidos. Fluxos que geram mais tráfego são mais vezes servidos, ou seja, este algoritmo não protege a rede de fluxos que a monopolizam. Além disso, os fluxos com pacotes maiores recebem mais serviço. Trata-se de um algoritmo muito utilizado devido à sua simplicidade de implementação. Se forem organizadas filas por classe de tráfego, a ordem de serviço em cada fila é tipicamente FIFO.

Vantagens e Desvantagens

Quando a rede opera com suficiente nível de capacidade de transmissão e comutação, os *buffers* são necessários somente para assegurar que tráfego em rajadas de curta duração não cause descarte de pacotes. O escalonamento FIFO é altamente eficiente, pois, na medida em que o tamanho da fila permanece pequeno, a média de tempo de espera de pacotes na fila é uma fracção de tempo demasiado pequena relativamente ao tempo de transmissão *end-to-end*. Entretanto, quando a carga na rede aumenta, o tráfego em rajada causa significativos atrasos de escalonamento em relação ao tempo de transmissão total e, quando a fila está totalmente cheia, todos os pacotes subsequentes são descartados. Aqui o problema é que este algoritmo não isola os fluxos de diferentes classes, se for usada apenas uma fila.

2.5.2. ALGORITMO PRIORITY QUEUEING, PQ

O mecanismo de escalonamento por prioridade foi projectado para dar prioridade rígida ao tráfego importante [Cisco]. Este mecanismo é baseado no conceito de que certos tipos de tráfego podem ser identificados e colocados como tráfego prioritário e desta forma transmitidos primeiro que outro tráfego.

Fonte:[Semeria]

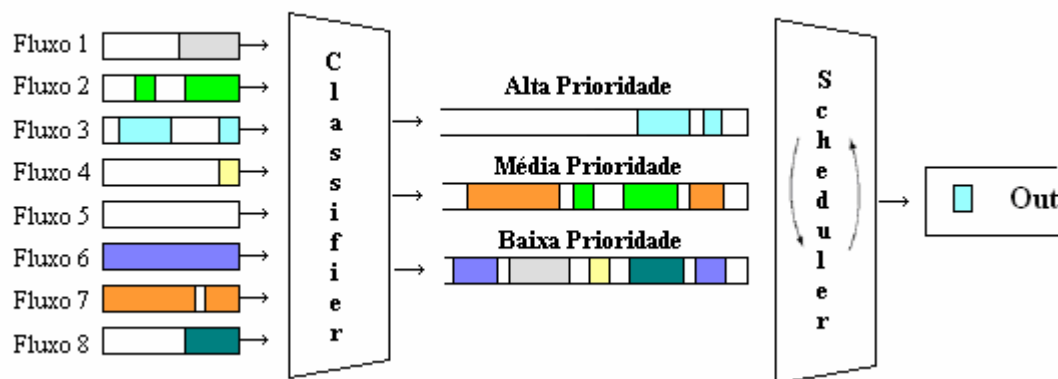


Figura 2.4: Escalonador PQ

Este mecanismo de escalonamento tem no entanto vulnerabilidades [Ferguson]. O *router* tem de analisar em detalhe cada pacote para saber como este deve ser escalonado, sobrecarregando desta forma o processador; isto acontece para todos os

algoritmos de prioridade não FIFO. Em ligações de baixa velocidade, o *router* tem mais tempo para examinar e manipular os pacotes. Entretanto, à medida que a velocidade do canal de comunicação aumenta, o impacto no desempenho torna-se mais visível. A Figura 2.4 mostra como os pacotes são recebidos na interface de entrada e reordenados, com base em critérios definidos pelo utilizador, até serem colocados na fila de saída. Os pacotes de mais alta prioridade são colocados na fila de saída antes dos pacotes com prioridade mais baixa. É importante lembrar que uma fila só é servida se todas as filas com prioridade mais alta estiverem vazias, isto pode levar a problema de negação de serviço (*starvation*) a menos que sejam impostos limites à largura de banda usada por cada classe (e portanto controlo de admissão).

Vantagens e Desvantagens

Diversos níveis de prioridade são possíveis, tais como: alto, médio, normal e baixo. Também a forma como o tráfego pode ser classificado na fila de saída é bastante flexível. Serviços específicos dentro de uma família de protocolos podem ser classificados da seguinte forma: o tráfego TCP pode ser escalonado com uma prioridade diferente relativamente ao UDP ou serviços *Telnet* relativamente a serviços FTP.

2.5.3. ALGORITMO GENERALIZED PROCESSOR SHARING, GPS

O GPS é um algoritmo ideal não implementável, que tem como objectivo a atribuição de recursos de acordo com o critério *max-min fairness*.

Este algoritmo possui uma fila por ligação (fluxo). O servidor vai servindo cada uma das N filas (não vazias) com um débito igual a $1/N$ da capacidade da ligação física.

De forma a limitar o grau de injustiça (*unfairness*) é possível associar pesos às ligações permitindo assim que estas recebam serviços de forma proporcional aos seus pesos sempre que a fila não esteja vazia.

A emulação mais simples do GPS é a disciplina *Round-Robin* (RR), que admite uma variante com pesos associados às ligações, *Weighted Round-Robin* (WRR). As disciplinas *Weighted Fair Queueing* (WFQ), também designada por *Packetized Generalized Processor Sharing* (PGPS) e *Worst-case Fair Weighted Fair Queueing* (WF²Q) são emulações mais elaboradas do GPS. A complexidade computacional de WFQ e WF²Q motivou a proposta de uma disciplina de serviço mais simples, designada *Self-Clocked Fair Queueing* (SCFQ).

Mais em pormenor, no GPS considera-se que os pacotes podem ser fragmentados arbitrariamente, o que é chamado de “Aproximação de Fluido”. Nesta política, a capacidade de processamento do *router* e a largura de banda disponível são distribuídas entre os fluxos concorrentes. Cada um desses fluxos recebe (pelo menos) uma parte β_i (equivalente aos pesos referidos anteriormente) desses recursos, sendo que o somatório de todos os β_i de todos os fluxos é 1.

Esta disciplina pode ser visualizada imaginando um fluxo contínuo de dados, separado por classes de prioridades, sendo cada classe i servida de acordo com o peso β_i [Bennet].

A Figura 2.5 representa o modelo GPS [Hersent]. Nessa figura pode-se observar a recepção de três pacotes de três fluxos distintos (A, B e C) com os seus respectivos pesos (25%, 25% e 50%). Três situações distintas podem acontecer:

- Enquanto o fluxo A não está a partilhar o canal de saída com nenhum dos outros fluxos (não existem os outros fluxos), este utiliza todo o canal (100%).
- Quando apenas os fluxos A e B partilham o canal, eles utilizam todo o canal na proporção dos seus pesos. Visto que são iguais, cada um utiliza 50% do canal.
- Quando o canal é partilhado pelos 3 fluxos (A, B e C), estes partilham-no na proporção dos seus pesos. De acordo com o modelo GPS, o pacote C, apesar de ser o último a chegar, seria o primeiro a completar a transmissão. Isto verifica-se porque o modelo GPS utiliza a aproximação de fluido e desta forma consegue ser um algoritmo mais justo na atribuição da largura de banda.

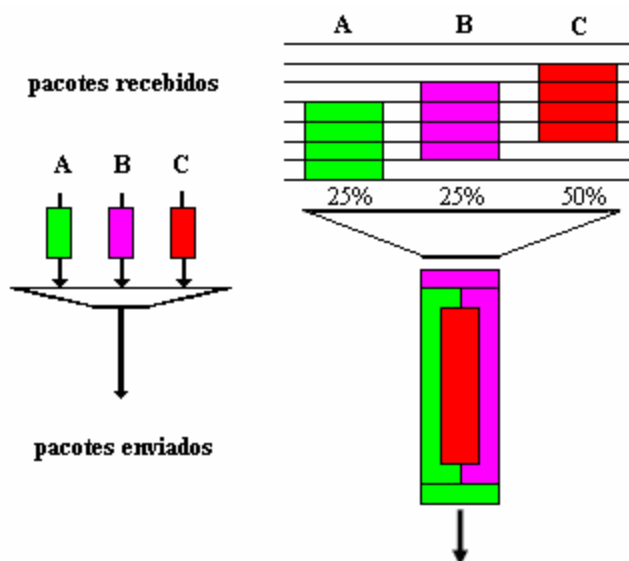


Figura 2.5: Escalonador GPS

Cada sessão i pode usar no mínimo a capacidade $\beta_i * R$, onde R é a capacidade total da interface de saída considerada. As sessões activas partilham os recursos das sessões inactivas proporcionalmente aos seus β_i .

Num sistema GPS, todos os fluxos que utilizam uma porção da largura de banda do canal de saída menor que a parte que lhes é atribuída são servidos de imediato, e a largura de banda não utilizada por estes fluxos é partilhada pelos demais na proporção dos seus pesos garantindo alguma justiça no escalonamento.

Na prática não se podem utilizar modelos de fluidos, pois os dados de cada fluxo não podem ser fragmentados arbitrariamente. O modelo de fluído que está por detrás deste modelo não é válido para redes no mundo real, nas quais os pacotes são recebidos como entidades inteiras e são inseridos na fila de saída como blocos de bits contíguos.

2.5.4. ALGORITMO ROUND ROBIN, RR

O algoritmo de escalonamento Round-Robin é um dos algoritmos mais antigos e mais simples, além de ser totalmente imune a problemas de *starvation*. É usado em

projectos de sistemas operacionais multi-tarefa, e foi projectado especialmente para sistemas *time-sharing*.

O RR serve rotativamente um pacote de cada fila não vazia (em vez de realizar serviço infinitesimal, como em GPS).

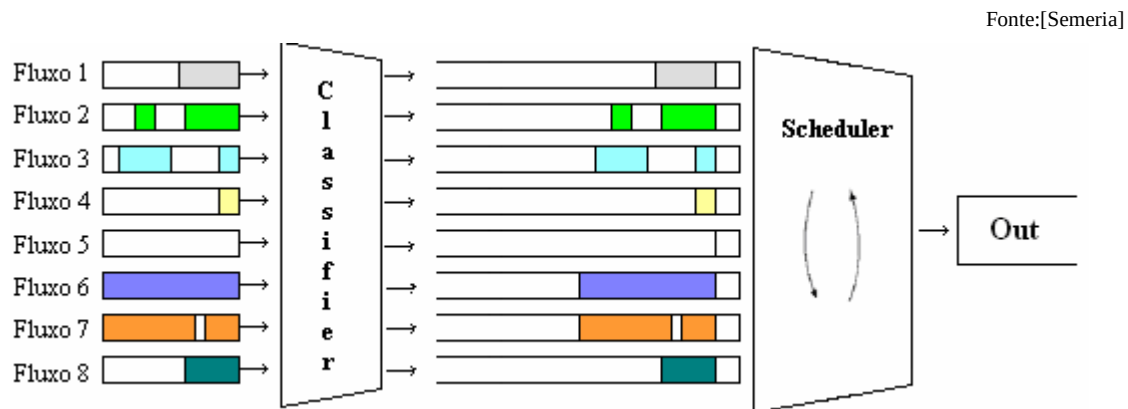


Figura 2.6: Round Robin

Vantagens e Desvantagens

A vantagem principal do RR é que os fluxos *bursty* não afectam o desempenho total. Se um fluxo for demasiado *bursty*, somente a sua fila será afectada, sem nenhuma interferência com os outros fluxos.

Os inconvenientes do RR são demonstrados a seguir. O escalonador utiliza um pacote de cada vez e de cada fila não obstante o comprimento do pacote. Por exemplo, se uma fila contiver pacotes maiores do que outras filas, essa fila usará uma parcela maior da largura de banda total da rede e será servido durante mais tempo. Além disso, o algoritmo do RR é mais complicado do que o FIFO e o PQ. No geral, este algoritmo é bom para partilhar a mesma porção da largura de banda entre muitos fluxos, mas não pode ser usado para garantir requisitos de largura de banda diferentes. Além disso, não há nenhuma maneira fácil de fornecer serviços com requisitos de tempo real.

No algoritmo RR, filas vazias são saltadas, e para um certo número de conexões activas, a largura de banda é igualmente dividida entre elas [Syed].

2.5.5. ALGORITMO WEIGHTED ROUND ROBIN, WRR

WRR é uma variante do algoritmo anterior mas que admite pesos associados às filas. Ou seja, WRR serve as ligações proporcionalmente aos pesos atribuídos.

Para uma distribuição de recursos de acordo com o critério *max-min fairness* é necessário conhecer o tamanho médio dos pacotes gerados por cada fonte.

Deficit Round Robin (DRR) é uma modificação do WRR. DRR [Shreedhar&Varghese] foi proposto em 1995 por M. Shreedhar and G. Varghese. Este algoritmo trabalha com pacotes de tamanho variável sem precisar de saber o seu tamanho médio. O tamanho máximo do pacote é subtraído ao comprimento do pacote, e o excedente é atrasado até a visita seguinte do escalonador.

2.5.6. ALGORITMO WEIGHTED FAIR QUEUEING, WFQ

O algoritmo WFQ não assume o tamanho infinitesimal dos pacotes (como no GPS) e não necessita de conhecer o tamanho médio dos pacotes (como WRR). Em WFQ, o servidor escolhe o pacote que completaria primeiro o seu serviço num servidor GPS de referência.

O WFQ tenta emular o GPS, calculando o tempo de saída do pacote num sistema GPS correspondente, e utilizando este tempo (*timestamp*) para ordenar os pacotes. O tempo calculado para o sistema GPS não representará o tempo real de partida do pacote no WFQ, mas será associado à ordem de saída do pacote. Por exemplo, se o pacote A tem a sua transmissão finalizada antes do pacote B num sistema GPS, o pacote A será seleccionado para ser transmitido antes do pacote B no WFQ.

No mecanismo WFQ define-se uma função de sistema chamada *tempo virtual*. O *tempo virtual*, $v(t)$, está associado ao sistema de referência GPS e mede o progresso do trabalho neste sistema e é dependente da carga do sistema. O *tempo virtual* é utilizado no WFQ para definir a ordem pela qual os pacotes são servidos no algoritmo. A

função *tempo virtual* $v(t)$ tem uma taxa de crescimento no tempo igual ao serviço normalizado recebido por qualquer fluxo armazenado no sistema de referência GPS.

Dessa forma, o mecanismo WFQ tem que identificar cada pacote que chega com o seu *tempo virtual final* e servi-los por ordem crescente destas identificações. O cálculo do *tempo virtual final* para cada pacote é uma operação que requer o conhecimento do tamanho do pacote, do *tempo virtual final* do pacote anterior, e do *tempo virtual do sistema* quando o pacote chega.

O cálculo do *tempo virtual* requer que o mecanismo WFQ mantenha os parâmetros necessários: as coordenadas da última mudança de inclinação, a inclinação actual e o número de sessões atendidas. A partir do momento em que a chegada e a partida de pacotes alteram estes valores, e que pacotes podem chegar simultaneamente, este é um grande desafio computacional.

Fonte:[Semeria]

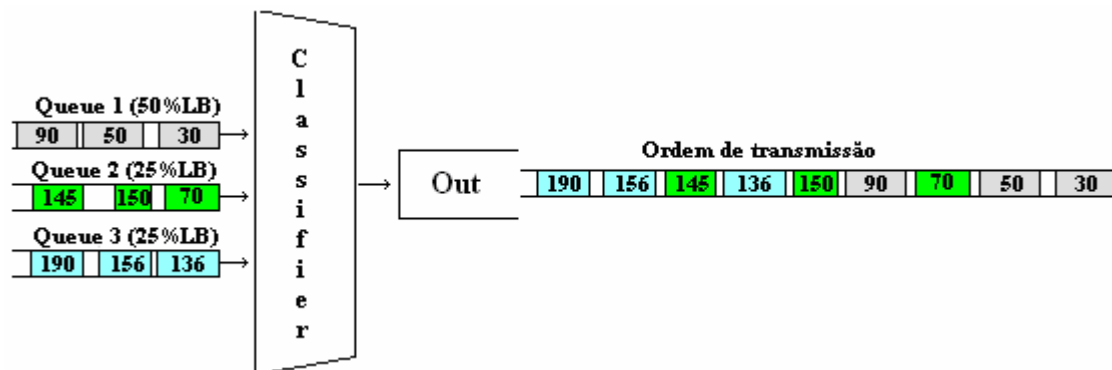


Figura 2.7: Escalonador WFQ

O termo “*weighted*” utilizado no nome deste algoritmo advém do facto deste algoritmos utilizar “pesos” para determinar a ordem de escalonamento dos pacotes na transmissão. Para realizar esta ordenação é preciso que cada pacote seja diferenciado quanto ao seu nível de prioridade. Portanto, o WFQ analisa o campo prioridade (*Precedence*) do cabeçalho do pacote IP a fim de atribuir um determinado peso a esse pacote.

É um mecanismo eficiente na medida em que pode utilizar toda a largura de banda disponível para transmitir o tráfego de baixa prioridade se não existir tráfego de mais alta prioridade.

Vantagens e Desvantagens

WFQ possui algumas das características dos mecanismos de escalonamento por prioridade (PQ) e de escalonamento baseado em classes (CBQ) e apresenta, pelos mesmos motivos, problemas de escalabilidade. Um outro problema está na falta de granularidade no controlo do mecanismo utilizado para favorecer alguns fluxos sobre os outros. WFQ protege fluxos de baixo volume de tráfego no esforço de fornecer equidade para todos (*max-min fairness*). A utilização de pesos é atractiva do ponto de vista de justiça entre classes. Contudo, não são fornecidas formas de avaliar ou ajustar estes parâmetros para alterar o comportamento, pois o método de preferir alguns fluxos sobre outros é estaticamente definido na implementação específica de cada fabricante e/ou ISP. O algoritmo WFQ permite majorar o atraso máximo, independentemente do número de conexões, o que constitui uma propriedade interessante [Guérin].

2.5.7. ALGORITMO SELF-CLOCKING FAIR QUEUING, SCFQ

Uma das maiores desvantagens do WFQ está na complexidade envolvida na implementação computacional da função *tempo virtual*. O SCFQ foi proposto em [Golestani] como um modelo mais simples do algoritmo WFQ [Guérin]. O SCFQ também é baseado na noção de *tempo virtual*, contudo é referenciado no escalonamento actual do sistema, em vez de um sistema hipotético.

O SCFQ propõe uma aproximação simples para calcular o instante de partida (tempo final de serviço) no correspondente sistema GPS. Quando um pacote chega a uma fila de espera, o SCFQ usa como $v(t)$ o *tempo virtual final* do pacote que está nesse momento em serviço. Quando nenhum pacote é encontrado em qualquer fila, o algoritmo atribui o valor zero para $v(t)$.

2.5.8. ALGORITMO START-TIME FAIR QUEUING, SFQ

O algoritmo *Start-Time Fair Queuing*, SFQ, foi proposto em [Goyal] como uma outra variante do mecanismo *Fair Queuing*. É semelhante ao SCFQ com a principal diferença de escolher o pacote com o menor *tempo virtual* de início de serviço (em oposição ao menor tempo virtual de final de serviço) para transmissão no canal de saída [Guérin].

No SFQ, dois identificadores, um identificador de início e um de fim, são associados a cada pacote. Contudo, ao contrário do WFQ e do SCFQ, os pacotes são servidos pela ordem crescente do identificador de início dos pacotes.

Inicialmente o *tempo virtual* do servidor é 0. Durante um período de ocupação, o *tempo virtual* no instante t , $v(t)$, é definido como sendo igual ao identificador de partida do pacote, servido no instante t . No final do período de ocupação, $v(t)$ assume o valor máximo do identificador de fim dos pacotes que já tenham sido servidos. Os pacotes são servidos na ordem crescente dos identificadores de partida.

Como evidenciado na definição, o custo computacional da implementação do SFQ é baixo e equivalente ao SCFQ, uma vez que envolve apenas a manipulação dos identificadores de partida dos pacotes.

2.5.9. ALGORITMO WORST-CASE FAIR WEIGHTED FAIR QUEUEING, WF²Q

Proposto por Bennet e Zhang em [Bennet], WF²Q é a aproximação mais fiel do conceito de GPS, sendo o seu serviço quase idêntico a um serviço oferecido por um escalonador que utilize o conceito de GPS. Ao escolher o próximo pacote a ser processado, ao invés de escolher um dos pacotes que estão à espera de ser escalonados no servidor real, como no WFQ, o WF²Q faz a escolha entre os pacotes que já tenham começado a ser servidos no sistema GPS correspondente. Entre estes pacotes o que será processado será aquele que terminar o serviço em primeiro lugar [Bennet].

Visto que os algoritmos GPS, WFQ e WF²Q são muito similares aqui fica um exemplo comparativo entre eles.

Um dos problemas do WFQ é que há alguns casos em que não se aproxima do GPS. No seguinte exemplo todos os pacotes são de tamanho 1 e a taxa de dados é 1 pacote por unidade de tempo. Há seis fluxos com os seguintes pesos: o fluxo F1 tem um peso atribuído de 50% da ligação da saída total; os fluxos F2-F6 têm um peso atribuído de 10%. Um *stream* de seis pacotes (A a F) chega no F1 ao mesmo tempo que os pacotes individuais chegam nos fluxos F2-F6.

O GPS serve os pacotes de F2-F6 de t igual a zero até t igual a 10. Igualmente serve o primeiro pacote do F1 (A) de t igual a zero até t igual a 2, depois o pacote B de t igual a dois até t igual a 4, e assim por diante.

WFQ serve os pacotes segundo as indicações da Figura 2.8. Observe-se a falta do pacote F logo depois do pacote E e a falta de tráfego do fluxo 1 durante 5s, pois foram servidos os primeiros 5 pacotes de F1 (A até E) e depois foi feita a rotação pelos restantes fluxos, servindo os pacotes em espera desses fluxos.

WF²Q é apresentado também, e pode verificar-se que entre pacote de maior prioridade é transmitido um pacote de menor prioridade.

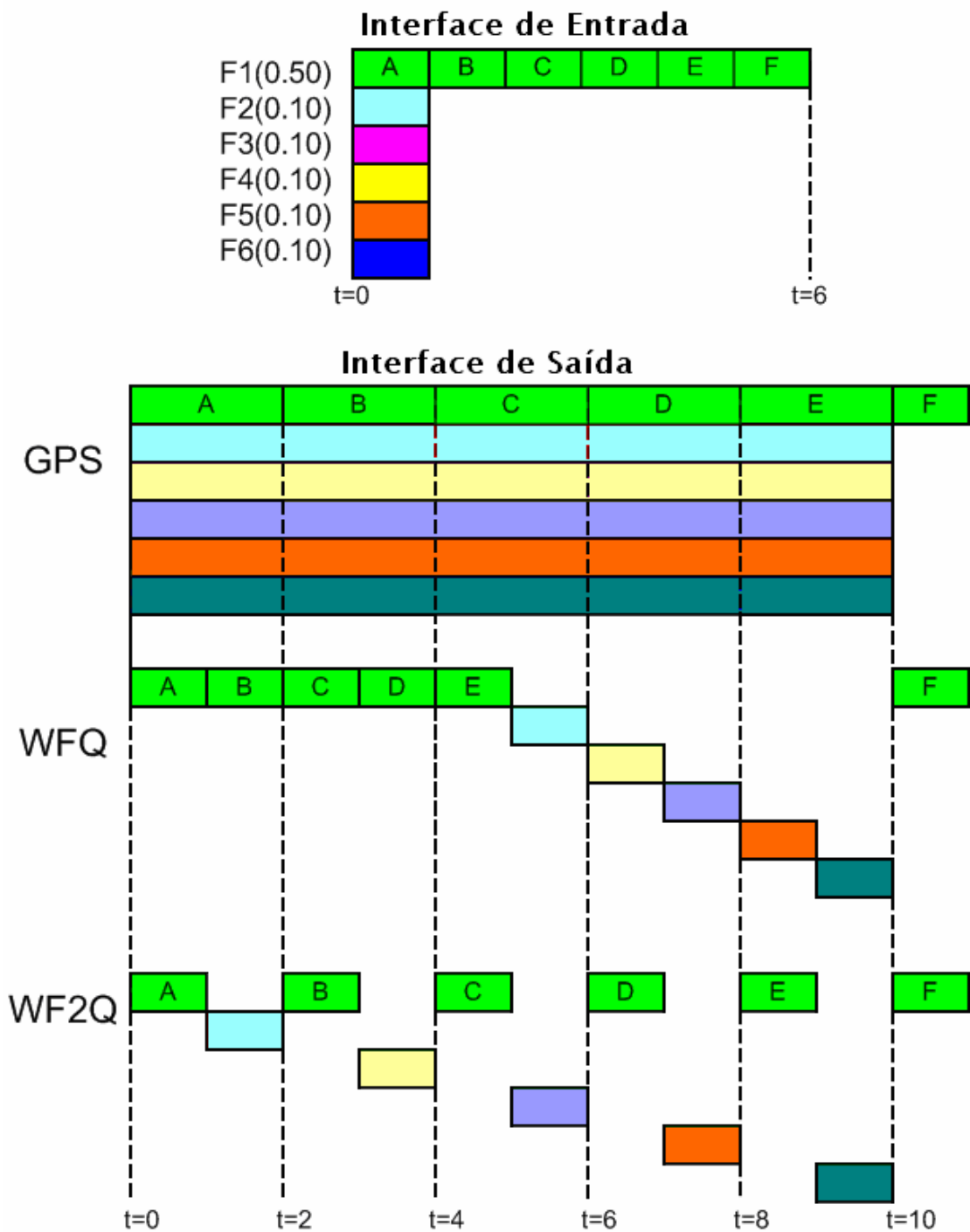


Figura 2.8: Comparação entre os mecanismos de escalonamento GPS, WFQ, WF²Q

2.5.10. ALGORITMO CLASS BASED QUEUEING, CBQ

O mecanismo de escalonamento CBQ, também designado *Custom Queuing* (CQ) foi projectado para permitir que várias aplicações, com especificações de largura de banda mínimas ou exigências de latência controlada, partilhem a rede [Cisco]. Este mecanismo é uma variação do escalonamento por prioridades, onde várias filas de saída podem ser definidas. Pode-se também definir a preferência com que cada fila será servida e a quantidade de tráfego escalonado que deve ser escoada de cada fila em cada passagem na rotação do serviço [Ferguson]. Ou seja, este algoritmo pretende distribuir a largura de banda disponível entre diversas classes de tráfego, tendo em conta a prioridade e a percentagem de largura de banda requerida por cada classe.

Fonte:[Semeria]

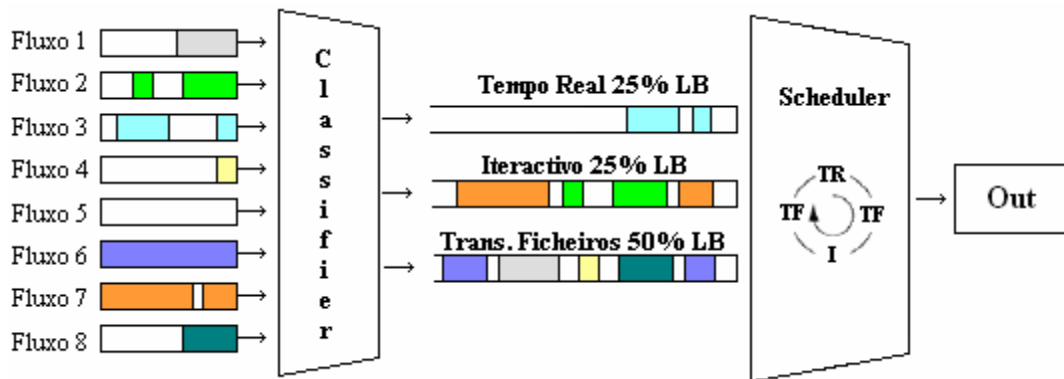


Figura 2.9: Escalonador CBQ

No exemplo da Figura 2.9 foram criados 3 *buffers*: alta, média e baixa prioridade. Por exemplo, o *router* pode ser configurado para servir 200 bytes na fila de alta prioridade, 150 bytes na fila de prioridade média e 100 bytes na fila de baixa prioridade em cada rotação. Após o tráfego em cada fila ser processado, os pacotes continuam a ser servidos até que o contador exceda o limite configurado para essa fila ou a fila esteja vazia. Desta forma, o tráfego que foi categorizado e classificado para ser escalonado em diversas filas tem boas possibilidades de ser transmitido sem que uma latência significativa seja induzida, permitindo ao sistema evitar escassez de *buffer*.

O CBQ divide-se em dois módulos:

- um algoritmo de escalamento tradicional que decide qual o próximo pacote a ser transmitido;
- e um algoritmo de escalonamento *link-sharing* que autoriza ou não o envio do pacote escolhido e garante que cada classe recebe a sua largura de banda predefinida;

No modo tradicional, a cada classe é associada uma fila de espera, as filas com a mesma prioridade são servidas com um algoritmo de escalonamento *Round Robin*, RR, ou *Weighted Round Robin*, WRR. A prioridade definida para cada classe é estrita, ou seja, pacotes de maior prioridade são servidos primeiro que os pacotes de prioridade inferior. A Figura 2.10 ilustra uma configuração possível para CBQ, onde são definidas 3 classes (voz, vídeo e dados), 2 prioridades (voz e vídeo com maior prioridade) e cada classe requer uma percentagem de largura de banda disponível (voz 3%, vídeo 32% e dados 65%).

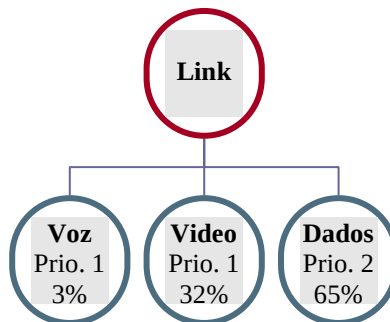


Figura 2.10: Distribuição da capacidade da ligação

Noutro modelo, apresentado na Figura 2.11, é possível criar uma estrutura hierárquica para partilha dos recursos da ligação, entre duas organizações X e Y. À organização X é garantida 70% da capacidade da ligação e os restantes 30% à outra organização. Em cada organização são definidas 2 classes de tráfego (voz e dados), cada uma com uma determinada prioridade e uma percentagem de largura de banda requerida.

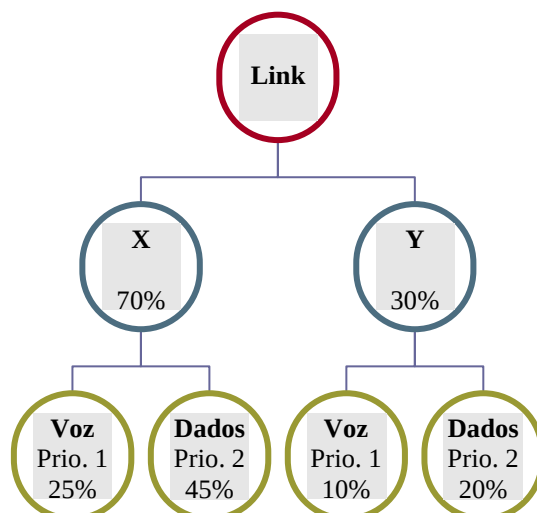


Figura 2.11: Distribuição da capacidade da ligação com estrutura hierárquica

O algoritmo de escalonamento *link-sharing* garante a largura de banda predefinida para cada classe e redistribui a largura de banda excedente através de um método controlado. No caso da Figura 2.10, se não houver dados para transmitir, os 65% da LB são distribuídos entre a voz e o vídeo de acordo com os pesos de cada um. No entanto, se não houver vídeo, os 32% da largura de banda configurada para o vídeo são distribuídos apenas para a voz que tem prioridade superior aos dados. No caso da Figura 2.11, se a organização X não tiver voz para transmitir, os 25% da largura de banda configurados para o efeito são distribuídos para os dados da organização X e não para a voz da organização Y.

Vantagens e Desvantagens

O mecanismo de priorização baseado em classes é geralmente tido como um método de atribuição de porções dedicadas de largura de banda para tipos específicos de tráfego. Mas, na realidade, ele fornece um mecanismo onde o modelo absoluto de serviço para a fila de alta prioridade e total falta de recursos para as outras filas do modelo PQ (*priority queuing*) é substituído por um modelo mais equitativo com um aumento de recursos para a fila de mais alta prioridade e uma diminuição relativa para as filas de mais baixa prioridade. A questão fundamental colocada é que a falta absoluta de recursos é de longe pior que a redução de recursos [Ferguson].

CBQ tem sido considerado um método razoável para implementação de tecnologias que fornecem partilha de canais de comunicação para classes de serviços e um método eficiente para gerir os recursos de filas.

2.5.11. ALGORITMO HIERARCHICAL TOKEN BUCKET, HTB

O HTB é um mecanismo de escalonamento que foi criado por Martin Devera nos finais de 2001, sendo actualmente o sucessor do *Class Based Queueing* (CBQ). HTB tem algumas vantagens relativamente a outras técnicas de QoS e especialmente quando comparado com CBQ: é mais simples e mais intuitivo, é mais preciso na implementação da partilha de tráfego e permite técnicas de empréstimo. Assegura que pelo menos uma quantidade mínima de tráfego para uma classe é fornecida; quando essa mesma classe não usa os recursos atribuídos, a largura de banda que não está a ser usada é temporariamente distribuída pelas outras classes.

No intuito de implementar um mecanismo que permitisse especificar regras que optimizem a utilização de um *link*, a estrutura de partilha de recursos especifica uma divisão da largura de banda para um *link* em particular de uma forma estática ou dinâmica. Cada classe deverá receber a sua largura de banda atribuída [Floyd]. Uma partilha de *link* hierárquica pode contemplar múltiplos *links* tal como indica a Figura 2.12. Um *link* pode ser partilhado por 3 organizações (Org A, B, C), o tráfego de todas as organizações é escalonado segundo os protocolos (P1, P2, P3) e a hierarquia pode descer ainda para *links* individuais (L1, L2) com classes de serviço.

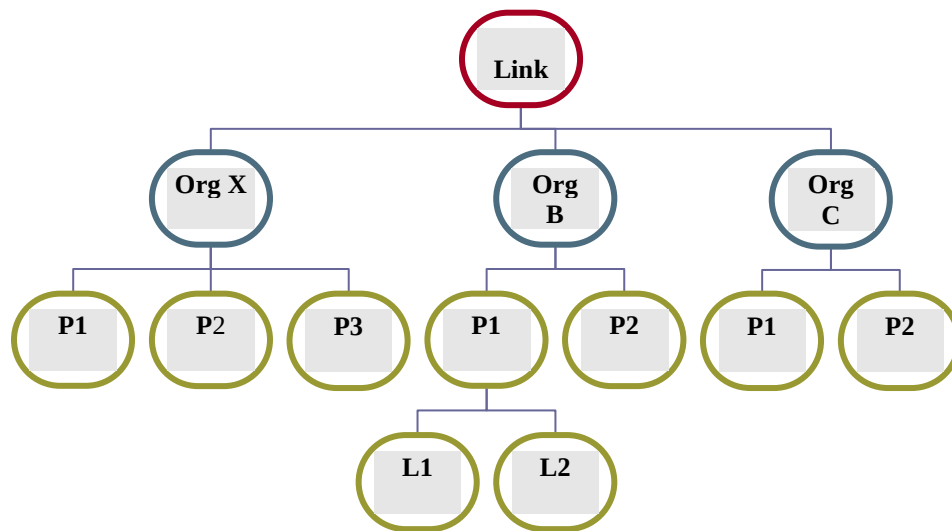


Figura 2.12: Partilha hierárquica de recursos

Como se pode ver a Org B recebe uma percentagem de largura de banda total do *link* para os protocolos P1 e P2; se P1 não tiver dados para transmitir a largura de banda não utilizada pode ser utilizada pela subclasse P2 da mesma organização.

HTB assume que a árvore está completa e o tráfego é dividido em fluxos. O algoritmo usado para escalonar pacotes é o seguinte: **primeiro**, escolhe todos as “folhas” da árvore cuja taxa máxima ainda não foi atingida e envia os pacotes dessas “folhas” começando pelos pacotes de mais alta prioridade e continuando com as “folhas” de menor prioridade. Para “folhas” com a mesma prioridade usa-se DRR. Quando as taxas para todas as “folhas” são excedidas, todo o ciclo é repetido mas para cada “folha” é realizado um teste para verificar se é possível obter empréstimo dos seus pais sem utilizar o seu próprio débito; quando não faltar mais nenhuma classe, o ciclo é repetido começando pelos primeiros nós da árvore.

Ainda relativamente à Figura 2.12, quando a classe Org A tem maior prioridade que a Org B e ambas vão receber largura de banda emprestada pela Org C, então Org A serve-se o máximo que pode e o restante é dado a Org B; se Org B tem a mesma prioridade que Org A então o empréstimo de largura de banda por parte de Org C deve ser dividido entre as outras duas organizações na mesma proporção que as suas próprias taxas iniciais.

O HTB é mais lento mas mais preciso que CBQ. O CBQ mantém uma variável e um ciclo de alto nível somente sobre as folhas e em cada “folha” tenta emprestar para os seus antecessores até ao nível superior. A taxa é medida pelo intervalo de tempo entre ocorrência de pacotes. Os ciclos do HTB sobre as folhas ocorrem mais vezes tornando o processo mais lento mas mais preciso.

São usados os seguintes termos:

- *Ceil* – a largura de banda máxima que uma classe pode usar
- *Burst* – a quantidade de dados que pode ser enviada com a máxima velocidade para uma classe sem servir outra classe. O valor para este parâmetro nunca deve ser maior para a classe filha do que para os seus pais.

Um exemplo pode ser visto na figura seguinte:

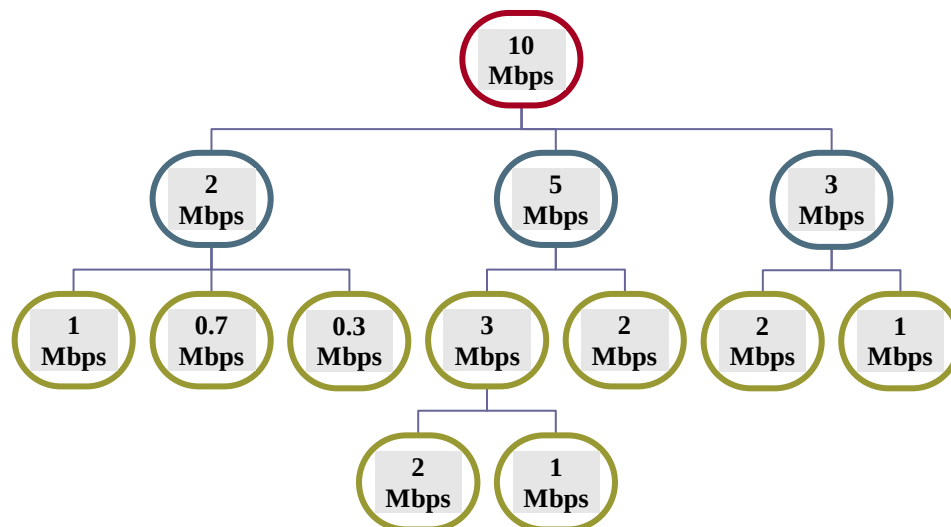


Figura 2.13: Exemplo prático de uma árvore HTB

Por exemplo, num ISP, em que cada utilizador paga pela sua ligação, basta colocar a taxa com o mesmo valor que o *ceil* na configuração do HTB.

Entre as inúmeras vantagens que advêm do uso do HTB, destaca-se o facto de ter a mesma capacidade de "traffic shaper" que o *Token Bucket Filter*, TBF. A configuração e o uso de classes hierárquicas são fáceis, permitindo assim a divisão de um *link* entre fluxos heterogéneos. No entanto a implementação é bastante complexa.

2.6. DESCARTE

As técnicas de descarte definem quais os pacotes, numa fila de espera de um *router*, que devem ser descartados em determinado instante. Esse descarte pode ser efectuado numa situação de congestionamento ou com o objectivo de evitar essa situação [Keshav]. No entanto, no caso mais simples, os pacotes podem ser descartados mesmo antes de serem colocados na fila de espera do *router* (*tail drop*).

As técnicas de descarte distinguem-se pela posição na fila de espera do pacote a descartar e pela condição que define o descarte dos pacotes. A situação mais comum e mais simples de implementar é o descarte do último pacote recebido. Neste caso, quando um pacote chega ao sistema, é verificado se:

- a fila de espera tem espaço disponível para receber o mesmo,
- ou, mesmo que a fila não esteja cheia, se pode ser aceite de acordo com a estratégia adoptada (e.g., *Random Early Detection* - RED),

se estes dois pontos não se verificarem o pacote é descartado.

O descarte do primeiro pacote da fila é mais complexo porque implica alterar o ponteiro que indica qual o próximo pacote a ser transmitido. Para além disso, no caso dos recursos da fila de espera serem geridos em bytes, é necessário verificar se o tamanho do pacote a descartar é igual ou superior ao tamanho do pacote que acabou de chegar; caso isto não se verifique, é necessário descartar um número n de pacotes para que a fila tenha espaço para receber o pacote que acaba de chegar. Escolher um pacote aleatório para descarte ainda é mais complexo, pois requer a existência de um ponteiro para cada pacote, o que torna mais complexa a gestão da fila de espera. Na prática, devido à sua complexidade, esta técnica não é muito utilizada.

As técnicas de descarte analisadas neste capítulo serão a *DropTail* e a *Random Early Detection*, RED. A primeira descarta pacotes da sua fila de espera sempre que não tem espaço disponível para receber o pacote que acaba de chegar. No caso da técnica de descarte RED, os pacotes são descartados aleatoriamente, com uma determinada probabilidade dependendo do tamanho da fila de espera e do número de pacotes que

chegam, com o objectivo de manter o tamanho médio da fila de espera entre os valores configurados.

Várias variantes do RED foram já propostas e adoptadas pela comunidade IP. Um estudo comparativo sobre algumas variantes do RED, incluindo WRED, GRED, RIO-C e RIO-DC, é apresentado em [Makkar].

3. MODELOS DE QoS / ENGENHARIA DE TRÁFEGO

Tal como já foi referido anteriormente, as redes IP têm baseado o seu funcionamento num modelo de serviços *best effort*, não oferecendo garantias quanto à entrega ou ao atraso na entrega de pacotes. Com o objectivo de garantir diferentes níveis de QoS e a capacidade de gerir a atribuição de recursos por fluxos ou classes de tráfego, foram definidos pelo IETF dois modelos de QoS IP:

- O modelo de **Serviços Integrados** (IntServ - *Integrated Services*), que providencia QoS por fluxo (aplicações individuais).
- O modelo de **Serviços Diferenciados** (DiffServ - *Differentiated Services*), que providencia QoS a classes de serviço ou fluxos de tráfego agregados.

3.1. SERVIÇOS INTEGRADOS, INTSERV

O modelo IntServ é orientado para suportar QoS extremo-a-extremo a fluxos individuais de pacotes (*flows*²). Para tal é necessário que os *routers* que estão no

² Um fluxo é uma sequência de pacotes relacionados e que recebem idêntico tratamento em cada nó, sendo normalmente identificado pelo conjunto de endereços IP (origem e destino), portas (origem e destino) e tipo de protocolo.

percurso dos dados sejam capazes de reservar recursos. A arquitectura deste modelo assenta num princípio básico de concepção,

**A qualidade de serviço na arquitectura IntServ é garantida
através de mecanismos de reserva de recursos na rede.**

Para que os *routers* sejam capazes de reservar recursos é necessário um protocolo de sinalização para o efeito e que os nós sejam capazes de manter informação de estado por fluxo. O protocolo *Resource Reservation Protocol*, RSVP [Braden], é o mais utilizado. No entanto, pode ser utilizado outro mecanismo de sinalização que não o RSVP.

Este modelo IntServ propõe três classes de serviço: ***Best Effort***, ***Guaranteed Service***, para aplicações que requerem garantia de largura de banda e que o atraso dos pacotes não exceda um valor predefinido, e ***Controlled Load Service***, para aplicações que são tolerantes a atrasos e se adaptam a perdas ocasionais de pacotes (incluindo as resultantes de atrasos superiores a um valor limite aceitável).

- ***Best effort***

Best-effort trata-se de um serviço standard em redes IP. Tal como foi dito anteriormente não garante a entrega do pacote, atrasos, ou outras características associadas ao desempenho.

- ***Controlled Load***

Esta classe tem como objectivo oferecer a fluxos de dados um serviço com características idênticas ao do serviço *best-effort* numa rede moderadamente carregada. Isto significa que uma alta percentagem de pacotes transmitidos são recebidos com sucesso e com um atraso que não excede de forma significativa o atraso mínimo (esta caracterização é qualitativa e intencionalmente imprecisa). Este serviço fornece alta qualidade de entrega sem garantir a latência mínima [Wroclawski].

- **Guaranteed Service**

A classe *Guaranteed Service* fornece alta qualidade na entrega de pacotes, garantindo que o atraso é majorado. Trata-se de um serviço pesado para a rede e apenas é utilizado em situações em que o tráfego não se adapta facilmente às alterações [Guérin&Partridge&Shenker].

Para poder suportar QoS, controlar o congestionamento e partilhar largura de banda por várias classes de tráfego o IntServ necessita de um conjunto de funções. Estas funções estão organizadas em componentes que constituem a base da arquitectura de *routers* IntServ. As funções em causa foram apresentadas no [Capítulo 2](#), nomeadamente, classificação, escalonamento, descarte, policiamento, controlo de admissão.

Fonte: [Braden]

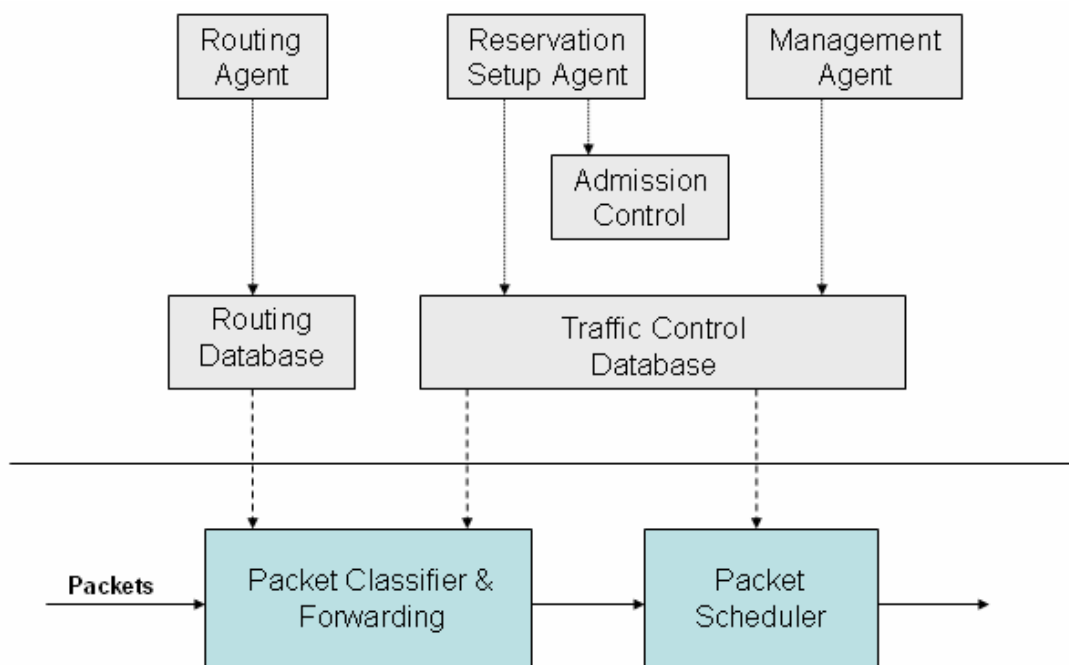


Figura 3.1: Arquitectura IntServ

O protocolo RSVP [Braden] foi proposto em 1993 e adoptado como protocolo de reserva de recursos na arquitectura de Serviços Integrados. Este protocolo permite a reserva de recursos em cada nó da rede mas não realiza funções de encaminhamento, controlo de admissão ou escalonamento de pacotes, implementados por outros

componentes da arquitectura; os pedidos de reserva de recursos têm de ser validados face aos recursos disponíveis (Controlo de Admissão) e permissões de carácter administrativo (*Policy Control*). Sendo o RSVP um protocolo de sinalização pode necessitar de um protocolo de encaminhamento que descubra rotas sujeitas a restrições de QoS. A reserva de recursos é da iniciativa dos receptores (*receiver initiated*), sendo a reserva realizada por fluxo. A sinalização é realizada para fluxos unidireccionais (*simplex*) e transporta parâmetros relativos a QoS e políticas. O modelo de reserva é do tipo *soft state*, ou seja, as reservas são mantidas temporariamente, sendo eliminadas se não forem actualizadas regularmente pelos receptores (por exemplo, a intervalos de 30 s).

A sinalização é realizada em dois passos:

- Os emissores enviam periodicamente mensagens PATH, com endereços *unicast* ou *multicast*. As mensagens PATH incluem um TSpec e um *Sender Template* para fornecer aos receptores informação sobre as características do tráfego e do(s) emissor(es), possibilitando assim reservas compatíveis com a natureza do emissor e os requisitos a satisfazer.
- Os receptores solicitam a reserva de recursos em mensagens RESV, enviadas pelo percurso inverso do das mensagens PATH. As mensagens RESV incluem um *Flow Descriptor* (*FlowSpec* e *Filter Spec*), o *FlowSpec* é constituído por um TSpec, idêntico ao do emissor, e por um RSpec. O *FilterSpec* (idêntico ao *Sender Template*) inclui informação para configurar os classificadores de pacotes.

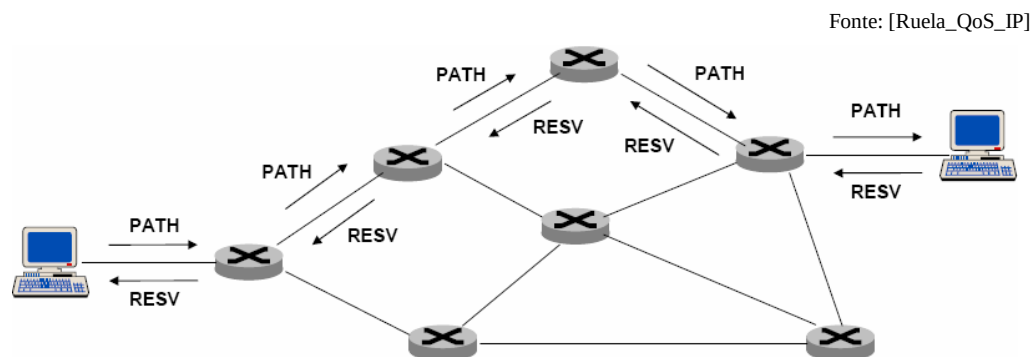


Figura 3.2: Troca de mensagens PATH e RESV

Na exposição anterior, foram apresentados alguns parâmetros de tráfego e QoS, que são agora revistos. O *FlowSpec* contém informação que caracteriza o tráfego a submeter à rede (*TSpec*) e o serviço pretendido com QoS associada (*RSpec*) e é usado para reservar recursos e parameterizar os escalonadores.

O *TSpec* inclui os seguintes parâmetros: *peak rate* (p), *token bucket rate* (r), *bucket size* (b), *maximum datagram size* (M) e *minimum policed unit* (m).

O *RSpec* é apenas especificado para o serviço garantido, *Guaranteed Service*, e inclui um *service rate* (R) (deve ser superior ou igual a r) e um *delay slack* (S); este representa o atraso adicional aceitável, relativamente ao que seria obtido com reserva igual a R . $S = 0$ significa que deve ser reservada largura de banda igual a R , enquanto $S > 0$ significa que pode ser reservada uma largura de banda inferior a R que cumpra o atraso tolerado.

As principais vantagens deste modelo IntServ relativamente ao QoS ATM, são a robustez (*soft state*) e a escalabilidade do mecanismo de reserva de recursos (iniciada pelos receptores). As reservas iniciadas pelo(s) receptor(es) têm algumas limitações pois requerem instalação e manutenção de informação sobre rotas nos *routers*, e trata-se de reservas unidireccionais, o que obriga a duplicar os procedimentos no caso de serviços com tráfego bidireccional, e requerem dois passos.

Para além disso, o modelo tem limitações de escalabilidade, que põem em causa a sua viabilidade em redes de grande dimensão, devido ao facto da reserva de recursos ser temporária e orientada a fluxos individuais pois exige um elevado número de mensagens de sinalização e manutenção da informação por fluxo em cada *router*; para além de que o tráfego de sinalização também consome recursos. A partir do reconhecimento que a proposta do IntServ não seria escalável para grandes redes *backbone*, o grupo criado em 1998 propôs de Serviços Diferenciados (DiffServ) [Black], onde os fluxos individuais seriam classificados e recursos atribuídos a classes, ou seja, a agregados de fluxos.

3.2. SERVIÇOS DIFERENCIADOS, DIFFSERV

O modelo DiffServ tem por base um conjunto de mecanismos relativamente simples. À entrada de uma rede o tráfego é classificado, condicionado e integrado numa de diferentes classes de tráfego, sendo cada classe caracterizada pelo respectivo *Differentiated Service Code Point* (DSCP). Nesta secção será realizada uma breve abordagem ao modelo DiffServ, realçando os aspectos que se julgam mais necessários para a discussão da proposta. Para uma abordagem mais aprofundada deverão ser consultados os RFC 2474 [Nichols] e RFC 2475 [Black].

O modelo DiffServ tem como principal objectivo providenciar QoS diferenciada e de forma escalável.

A Figura abaixo representa uma infra-estrutura de comunicações DiffServ, com dois domínios diferentes (DS1 e DS2). Na figura são ilustrados os diversos elementos que compõem um domínio DiffServ, tais como: os *Edge Routers* (ER), os *Core Routers* (CR), os *Border Routers* (BR), os *End Devices* (ED) e os sistemas de gestão e de implementação de políticas.

Fonte: [Rabadão]

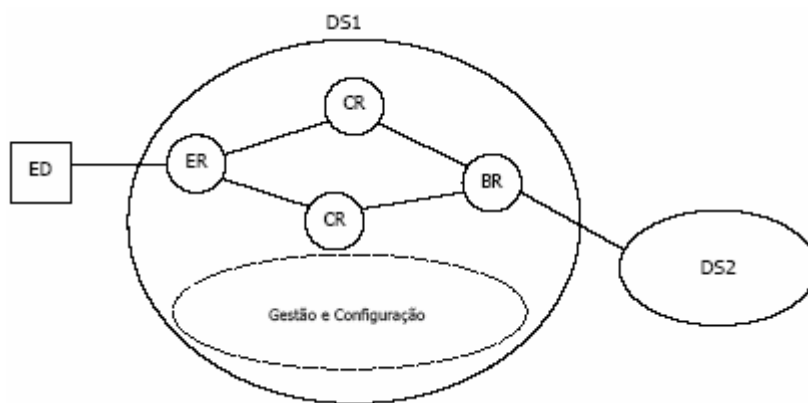


Figura 3.3: Elementos de uma infra-estrutura de comunicações DiffServ.

No modelo DiffServ os serviços são oferecidos a fluxos agregados (ou classes) e não por fluxo, as funções complexas são remetidas para a periferia da rede (*edge*), onde é feita a classificação, policiamento e agregação de tráfego, os nós internos (*core*) são simplificados, uma vez que não é necessário manter informação de estado e executar

procedimentos de sinalização por fluxo. A diferenciação permite satisfazer requisitos heterogéneos das aplicações e diferentes expectativas dos utilizadores, e aplicar tarifas por serviço.

Num domínio DiffServ não existe a necessidade de mecanismos de sinalização e os recursos são atribuídos, em cada nó, a fluxos agregados de tráfego que recebem o mesmo tratamento; a estes fluxos dá-se o nome de *Behavior Aggregate* (BA) (um BA corresponde a uma classe de tráfego). Os nós deverão tratar de forma diferenciada pacotes de diferentes BAs. Cada pacote tem de ser marcado (identificado) de acordo com o BA a que pertence, o que permite que pacotes de um mesmo BA tenham o mesmo tratamento e que portanto evidenciem um comportamento idêntico.

Em DiffServ são definidos comportamentos esperados e externamente observáveis designados por *Per-Hop Behaviors* (PHB) e não a forma como os nós operam de forma a produzir esses comportamentos. Ou seja, cada PHB define o comportamento individual de cada nó e não o comportamento extremo-a-extremo (os nós operam de forma independente).

A diferenciação dos serviços é realizada com base num campo DS (*Differentiated Services*) presente no cabeçalho de pacotes IP; o campo DS foi definido de forma a ser compatível com os octetos *Type of Service* (ToS) em IPv4 [Nichols] e *Traffic Class* em IPv6 e é constituído pelo *Differentiated services codepoint* (DSCP) – 6 bits e pelo *Explicit Congestion Notification* (ECN) – 2 bits.

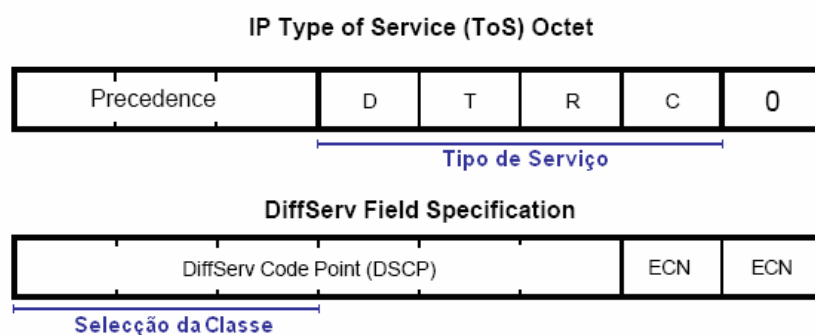


Figura 3.4: ToS e Diffserv/ECN

Nota: No caso de IPv4 o campo DSCP é representado nos seis bits mais significativos do byte ToS e no caso de IPv6 no byte Traffic Class.

A arquitetura *DiffServ* é composta por um conjunto de elementos funcionais implementados nos nós da rede, incluindo um conjunto reduzido de comportamentos adoptados nos nós para despacho (*forwarding*) de pacotes, ou seja, *Per-Hop Behaviors* (PHB), funções de classificação de pacotes e funções de condicionamento de tráfego (*traffic conditioning*), apresentadas no [Capítulo 2](#). Os PHBs são implementados nos nós por meio de mecanismos de gestão de *buffers* e de escalonamento de pacotes (disciplinas de serviço) e PHB constitui o meio de um nó atribuir recursos a um BA.

Em DiffServ, conforme indicado em [Nortel Networks 98], há três fases distintas no fluxo de cada pacote através de um determinado domínio: a fase de entrada, de encaminhamento e de saída.

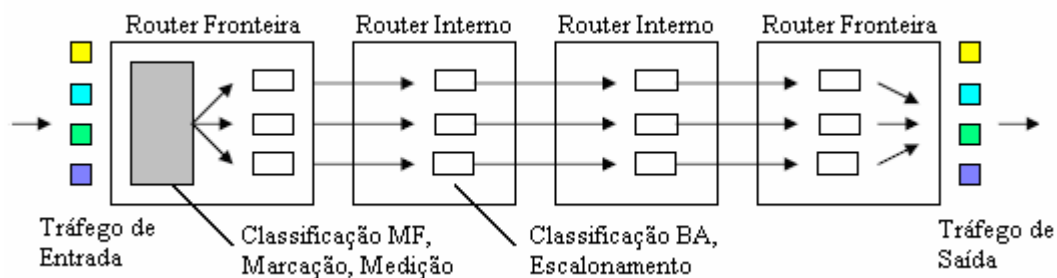


Figura 3.5: Funcionamento do DiffServ

O modelo de classificação e condicionamento de pacotes usado em DiffServ pode ser analisado na figura a seguir,

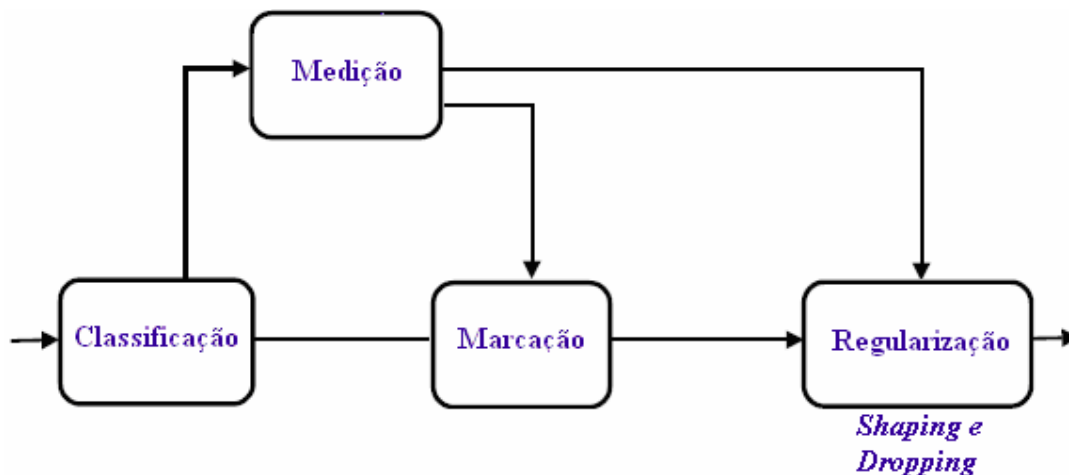


Figura 3.6: Modelo de classificação e condicionamento de pacotes

Estas funções de controlo têm o objectivo de garantir a observação das regras especificadas num *Traffic Conditioning Agreement* (TCA), isto é, garantir a conformidade de um fluxo de tráfego com o respectivo perfil. São usadas para garantir que são respeitados acordos entre domínios e para permitir que tráfego receba serviço diferenciado, através da marcação do *codepoint* apropriado e monitorização e alteração das suas características temporais, se necessário (isto é, pode incluir re-marcação, *descarte* ou *shaping*).

A gestão de recursos (num domínio e entre domínios DS), configuração de nós, realização de controlo de admissão de fluxos e negociação de SLAs entre domínios pode ser baseada em entidades designadas por *Bandwidth Brokers*. O *Bandwidth Broker* é uma entidade central que num domínio de Serviços Diferenciados realiza o controlo de admissão através da gestão e supervisão global dos recursos disponíveis, podendo também controlar e regular o tráfego aí existente.

As funções típicas de um *Bandwidth Broker* são as seguintes:

- Automatizar o processo de negociação de SLAs baseado em políticas configuradas de fornecimento de serviços (*service provisioning*)
- Realizar controlo de admissão
- Estabelecer e manter acordos (SLAs) com domínios vizinhos.

- Configuração dos equipamentos da rede de forma a suportar os níveis de QoS negociados

Foram definidos catorze PHBs, incluindo um PHB para *Expedited Forwarding* (EF), doze PHB para *Assured Forwarding* e um PHB para *Best Effort*. Os doze AF PHBs estão divididos em quatro *PHB Scheduler Classes* (PSCs), e cada um deles em três sub-comportamentos relacionados com tratamentos diferentes de descarte de pacotes.

- **Expedited Forwarding PHB**

O objectivo deste EF PHB [Jacobson] é a construção de serviços com pequena taxa de perda de pacotes, pequena latência e *jitter* reduzido. Tal como qualquer PHB, o EF PHB define apenas o comportamento de cada nó. O comportamento de um conjunto de nós deverá ser especificado através de um *Per-Domain Behavior* [Carpenter&Nichols].

Este serviço pode ser usado em aplicações como a videoconferência, voz sobre IP (*VoIP*), ou emulação de uma linha dedicada, que requerem alta qualidade de serviço e a especificação de um *peak bit rate*.

Genericamente, as filas de espera relativas a este tipo de tráfego (EF) devem estar vazias ou conter um pequeno número de pacotes, reduzindo assim o atraso, o *jitter*, e consequentemente a perda de pacotes. A formação de fila de espera ocorre se o débito de chegada de tráfego (em intervalos de curta duração) exceder o débito de saída. Torna-se assim necessário, ao especificar o EF PHB, definir com rigor as condições que permitem assegurar que um agregado EF é servido com um determinado débito (*service rate*) configurado.

No caso de se utilizar EF PHB com mecanismos de prioridade estrita, deve-se utilizar em cada nó um *token bucket* (com parâmetros configurados pelo administrador da rede), para compensar o possível prejuízo causado ao tráfego não EF, sendo o tráfego EF em excesso descartado.

O valor do DCSP recomendado para o PHB EF é o 101110.

Nota: Os três primeiros bits definem a classe, os dois bits seguintes especificam o descarte de pacotes e o último bit é sempre 0.

- **Assured Forwarding PHB**

[Heinanen] define um grupo de PHBs designado por *Assured Forwarding PHB Group* (AF), que é composto por quatro classes e três níveis de precedência para descarte em cada classe, a que correspondem doze DS *codepoints*, isto é, doze níveis diferentes de garantia de entrega de pacotes. Este serviço oferece garantias qualitativas e relativas (com base em prioridades). Este grupo de PHBs pode ser usado, por exemplo, para implementar um serviço com três classes (bronze, prata e ouro), enquanto a quarta classe poderá ser usada para suportar um serviço idêntico a *best effort*.

Os ISPs devem providenciar recursos a cada classe de acordo com os débitos subscritos (SLA) e de acordo com o comportamento (PHB) esperado. Os pacotes IP são atribuídos (pelo cliente ou pelo fornecedor) a uma ou mais classes AF, de acordo com os serviços contratados; dentro de cada classe os pacotes são marcados (pelo cliente ou pelo fornecedor) com um nível de precedência de descarte.

As classes AF são tratadas de forma independente. Não existe agregação de tráfego entre diferentes classes e o nível de precedência de descarte é diferente de classe para classe. Independentemente do nível de precedência de descarte, os nós DS não devem reordenar os pacotes que pertençam à mesma classe AF. A cada uma das classes é atribuída em cada nó do domínio DS uma certa quantidade mínima de recursos (*buffers* e largura de banda).

Este serviço possui uma elevada probabilidade de entrega (melhor que *best effort*), desde que o tráfego não exceda o débito contratado. Em situação de congestionamento, tráfego conforme (*in-profile*) deve receber um serviço idêntico

ao esperado numa rede moderadamente carregada, enquanto que o tráfego em excesso terá uma menor probabilidade de entrega. Em cada nó o nível de garantia de envio de um pacote IP depende do nível de tráfego, dos recursos atribuídos à classe a que o pacote pertence e do nível de precedência (no caso de congestionamento na classe).

Na tabela seguinte, são apresentados os *codepoints* recomendados.

Drop Precedences	Class 1	Class 2	Class 3	Class 4
Low	001010 AF 11 DSCP 10	010010 AF 21 DSCP 18	011010 AF 31 DSCP 26	100010 AF 41 DSCP 34
Medium	001100 AF 12 DSCP 12	010100 AF 22 DSCP 20	011100 AF 32 DSCP 28	100100 AF 42 DSCP 36
High	001110 AF 13 DSCP 14	010110 AF 23 DSCP 22	011110 AF 33 DSCP 30	100110 AF 43 DSCP 38

Tabela 3.1: *Codepoints* DiffServ AF recomendados

O mapeamento dos *codepoints* AF recomendados em cima não interfere com outros espaços de mapeamento usados localmente.

- **Best Effort PHB**

Cada nó deve proporcionar um BE PHB e o *codepoint* recomendado é 00.

Uma implementação sensata deste PHB pode basear-se numa disciplina de serviço que envia pacotes deste agregado sempre que o *link* de saída não está a ser utilizado a satisfazer outro PHB. Uma política para construir estes serviços deve garantir que o agregado nunca será completamente privado de serviço. Isto pode ser conseguido com um mecanismo em cada nó que reserve recursos mínimos, utilizando para isso *buffers* e largura de banda para este tipo de agregados.

No entanto para se poder garantir QoS é necessário a combinação de DiffServ com os mecanismos de Engenharia de Tráfego do MPLS, pois o modelo DiffServ por si não controla as rotas seguidas pelos pacotes e em caso de congestionamento não garante largura de banda. Por outro lado, o MPLS apenas encaminha tráfego através de rotas específicas mas não especifica tratamento diferenciado de classes de fluxos pelos nós. Assim sendo é necessário combinar os dois mecanismos.

3.3. MULTIPROTOCOL LABEL SWITCHING

Nesta secção será apresentada uma breve descrição sobre os conceitos e terminologia adoptada em MPLS e descritas também sucintamente as tecnologias MPLS-TE e RVSP-TE. Para um estudo mais exaustivo ficam aqui algumas referências [Rosen], [MPLSRC] e [Osbone].

Em virtude do crescimento das redes nos últimos anos, e das necessidades inerentes a esse crescimento, os ISPs têm a necessidade de disponibilizar redes (*backbone*) de elevada capacidade, pequena latência, flexíveis, escaláveis, e que permitam oferecer qualidade de serviço diferenciada. As redes têm evoluído para uma organização baseada num **núcleo** (*core*) de alta velocidade que interliga sistemas na **periferia** (*edge*), onde é realizado o processamento mais complexo.

A arquitectura MPLS foi inspirada em arquitecturas multi-camada que tinham o objectivo de integração de IP com ATM. No entanto, o MPLS tem um carácter mais genérico e permite suportar múltiplos protocolos de rede (para além do IP) sobre qualquer tecnologia de comutação de etiquetas (não necessariamente ATM).

As arquitecturas de comutação multi-camada (*multilayer switching*), de que o MPLS se constitui como modelo, têm como ideia base combinar:

- técnicas simples e robustas de encaminhamento na camada de rede (de que o paradigma é o IP e protocolos associados) – *Layer 3 Routing / Control*

- técnicas de comutação rápida, eficientes e escaláveis, na camada de ligação de dados (de que o ATM é a principal referência) – *Layer 2 Forwarding / Switching*

Nesta arquitectura as **funções de Controlo** e as **funções de Transporte de Dados** encontram-se separadas; as primeiras, realizadas através de *software*, baseiam-se em protocolos de encaminhamento e em protocolos de sinalização. As segundas, funções de Transporte (*forwarding / switching*), são realizadas em *hardware* e baseiam-se em técnicas de comutação de etiquetas.

O objectivo da componente de transporte num domínio MPLS é comutar pacotes de forma rápida e eficiente, com base numa etiqueta transportada num cabeçalho adicional.

O objectivo da componente de controlo é manter a informação topológica da rede necessária à construção e manutenção das tabelas de encaminhamento e de comutação. Para tal é necessário recorrer a protocolos de encaminhamento e sinalização.

A informação sobre os percursos possíveis mantida numa tabela de encaminhamento (*routing table*) é usada para construir a tabela de comutação (*forwarding table*) que associa um par “(porta, etiqueta)” na entrada a outro par na saída, realizando-se deste modo o mapeamento entre rotas e etiquetas (*label binding* – mapeamento entre a informação de nível 2 e de nível 3). Para manter a informação das tabelas de comutação dos nós correcta, é necessário um protocolo de distribuição de etiquetas chamado *Label Distribution Protocol* (LDP).

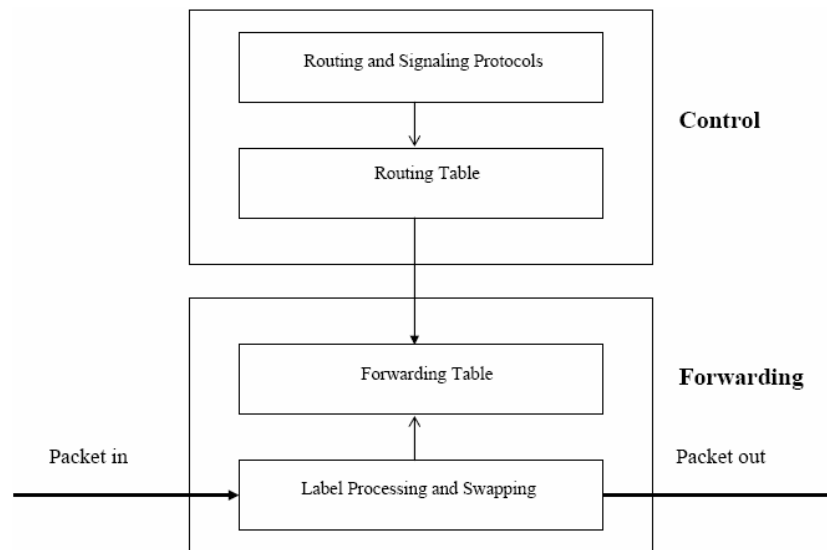


Figura 3.7: Componente de Controle e de Transporte

Os nós de um domínio MPLS designam-se genericamente por *Label Switching Routers* (LSR). É, no entanto, habitual designar os nós de entrada e de saída no domínio por *Label Edge Routers* (LER). Todos os nós participam nos protocolos de encaminhamento, de distribuição de etiquetas e outros protocolos de controlo. Um percurso (rota) é criado pela concatenação de *Label Switched Hops* (salto entre dois nós MPLS adjacentes entre os quais a transferência de pacotes é baseada numa mesma etiqueta). Um *Label Switched Path* (LSP) é um percurso entre nós de entrada e saída num domínio MPLS ao qual está associada uma sequência ordenada de etiquetas, o que permite transporte de pacotes entre nós MPLS pela simples troca de etiquetas.

Foi também definido o conceito de *Forwarding Equivalence Class* (FEC) para grupos de pacotes que podem ser tratados de maneira equivalente, do ponto de vista do transporte na rede (pelo mesmo percurso, sujeitos ao mesmo tratamento), pelo que podem ser mapeados na mesma etiqueta.

Os passos realizados na comutação de etiquetas são os seguintes:

- O LER na entrada do domínio MPLS adiciona uma etiqueta ao pacote IP.
- Os LSR ao longo do LSP realizam comutação com base na etiqueta (*label swapping*)

- O LER de saída remove a etiqueta, recuperando o pacote inicial.

O estabelecimento de *Label Switched Paths*, condição para se poder realizar o transporte de pacotes num domínio MPLS, exige um conjunto de passos, que podem ser realizados de acordo com várias estratégias:

- Descoberta e selecção de rotas
- Criação de etiquetas
- Associação de etiquetas a cada FEC (*label binding*) ao longo da rota (LSP)
- Distribuição da informação sobre a associação etiqueta / FEC pelos LSR que fazem parte do LSP a estabelecer, *Label Distribution Protocol* (LDP)

Os pacotes de um mesmo *Forwarding Equivalence Class* (FEC) são transportados no mesmo LSP. O LSP é definido pela sequência de etiquetas associadas ao FEC em cada nó, ao longo de um percurso na rede.

A fim de executar a distribuição das etiquetas foi proposto em [Callon] um novo protocolo específico denominado *Label Distribution Protocol* (LDP) responsável por distribuir a informação que permite criar e manter as tabelas de comutação nos LSRs. As especificações existentes no IETF não obrigam o uso do LDP como protocolo de distribuição de etiquetas, podendo ser utilizados outros mecanismos, como o RSVP estendido.

Nas tecnologias que não usam etiquetas, como a *Ethernet*, é inserido um pequeno campo adicional ao cabeçalho do pacote, entre os cabeçalhos das camada de ligação e de rede, denominado “*shim header*”, cujo formato é apresentado abaixo.

Fonte: [Zhang]

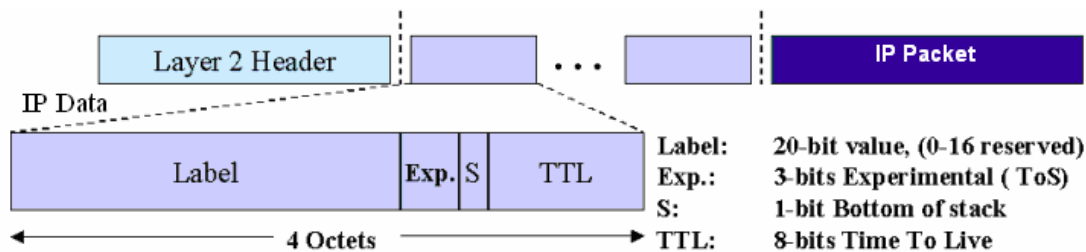


Figura 3.8: MPLS Shim Header

Conceptualmente um LSP é um túnel MPLS. Um pacote IP é encapsulado com uma etiqueta (transportada num cabeçalho de nível 2 ou num *shim header* entre cabeçalhos de nível 2 e 3) e viaja inalterado no interior do domínio MPLS. Em MPLS é possível usar vários níveis de encapsulamento, recorrendo a uma pilha de etiquetas (*label stack*), isto é, é possível criar túneis MPLS (LSPs) dentro de um LSP de nível superior, o que permite fazer encaminhamento hierárquico.

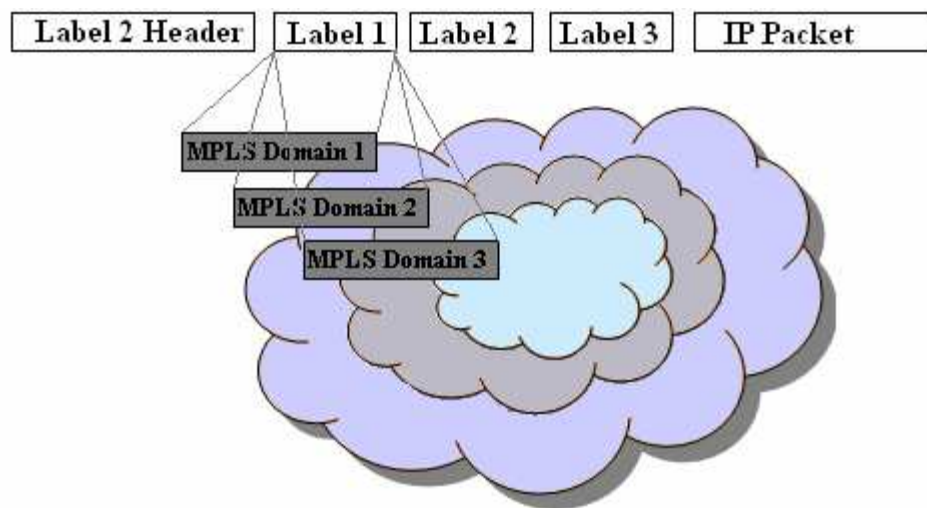


Figura 3.9: Exemplo de uma rede MPLS com três níveis

Concluindo, o MPLS proporciona uma melhoria significativa no processo de encaminhamento de um pacote devido à sua simplicidade, evitando a necessidade de realizar análise do cabeçalho IP ao longo do caminho, e criando um ambiente de suporte controlado de QoS. O MPLS permite a integração do IP com ATM e diversas outras tecnologias de camada 2 e camada 3; suporta a convergência de serviços (voz, dados e vídeo); oferece novas oportunidades à Engenharia de Tráfego.

3.3.1. ENGENHARIA DE TRÁFEGO NO MPLS

O paradigma de encaminhamento de pacotes proposto pela arquitectura MPLS possibilita a inclusão de uma série de recursos para atender a questões relacionadas com a diferenciação de serviços, nos quais a Engenharia de Tráfego (TE - *Traffic Engineering*) é uma das mais significativas [Awduche]. É importante destacar que TE

tem aplicabilidade em qualquer rede que use etiquetas para marcar os seus pacotes e onde existam pelo menos dois caminhos entre dois nós.

Os objectivos de desempenho associados a TE podem ser classificados em:

- **Objectivos Orientados ao Tráfego** são os objectivos que incluem os aspectos relacionados com a melhoria dos fluxos de tráfego com QoS, tais como: minimização da perda de pacotes, minimização do atraso, maximização do escoamento das filas de espera e melhoramentos do SLA.
- **Objectivos Orientados aos Recursos** são os objectivos que incluem os aspectos relativos à optimização da utilização dos recursos de rede, através da gestão eficiente desses recursos, como exemplo, o caso da largura de banda. Uma vez que a largura de banda é um recurso essencial nas redes, é desejável garantir que algumas partes da rede não sejam sobre-utilizadas e fiquem congestionadas, enquanto outras partes permaneçam sub-utilizadas.

Vale a pena salientar que a minimização da probabilidade de congestionamento é um objectivo de desempenho primário, tanto para o tráfego quanto para os recursos de rede. Visando alcançar o objectivo principal da TE, o MPLS-TE deve apresentar as seguintes características:

- Capacidade para calcular na fonte um caminho tendo em conta todas as restrições. Para fazer isto, a fonte deve possuir toda a informação necessária, seja ela disponível localmente ou obtida através dos *routers* da rede.
- Capacidade para distribuir, através da rede, a informação sobre a topologia da rede e os atributos associados às ligações. Uma vez que o caminho foi computado, o MPLS precisa de um modo para providenciar o encaminhamento dos pacotes de um mesmo FEC ao longo do referido caminho.
- Capacidade para reservar recursos da rede, bem como modificar os atributos da ligação, sempre que novos fluxos de dados entrem na rede, fazendo uso de certas rotas e consumindo recursos.

A TE fornecida pelo MPLS pode também incluir um processo de monitorização, em que a informação pode ser encaminhada através da rede, de acordo com o ponto de vista da gestão global dos recursos disponíveis na rede, além das cargas de tráfego esperadas ou actuais [Extreme].

Outra alternativa para se conseguir TE no MPLS pode ser através da configuração explícita de múltiplos LSP, usando o protocolo RSVP-TE [Awduche&Berger]. Este protocolo permite a reserva de recursos numa rede IP. As aplicações podem usar RSVP para indicar a outros nós a natureza (largura de banda, *jitter*, *burst* máximo, entre outros) dos pacotes que desejam receber.

A TE fornecida pelo MPLS também pode ser usada para suportar funções de balanceamento de carga, através do encaminhamento de tráfego IP sobre múltiplos LSP configurados explicitamente e que são tratados num dado FEC. A possibilidade do MPLS definir de modo flexível LSPs explícitos também pode ser usada, juntamente com os recursos disponibilizados pelo *router* e/ou *switch*, para dar suporte a objectivos de gestão de QoS, pela associação de classes específicas de serviço aos LSP, garantindo a largura de banda predeterminada. As questões de escalonamento são “transparentes” para o MPLS-TE, ou seja, cada *router* e *switch* implementa-as à sua maneira (por exemplo, com base em DiffServ). O MPLS-TE só transporta os pedidos de reserva.

3.3.2. MPLS SUPPORT OF DIFFSERV

Hoje, ambas as tecnologias estão à disposição da comunidade e podem ser combinadas para garantir QoS. Recordando, DiffServ fornece tratamentos QoS a fluxos de tráfego agregados, é escalável e operacionalmente simples, uma vez que não requer sinalização por fluxo. No entanto, não permite garantir largura de banda, porque não influencia o caminho do pacote, e por conseguinte, durante congestionamento ou falhas, pacotes de alta prioridade não têm garantia de largura de banda.

MPLS, por outro lado, pode forçar um caminho para um pacote e em combinação com encaminhamento sujeito a restrições pode-se garantir largura de banda para os FECs. Mas na sua forma básica MPLS não permite diferenciação no tratamento de fluxos, do ponto de vista de QoS (não determina o escalonamento de pacotes, gestão de buffers, políticas de descarte, etc.).

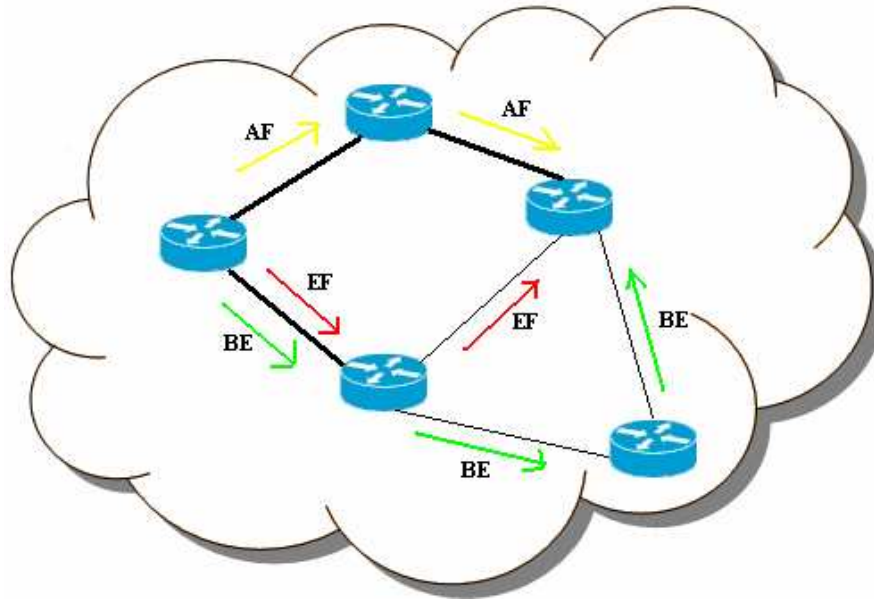


Figura 3.10: Rede MPLS DiffServ

Ou seja, DiffServ e MPLS não podem ser confundidos: um usa os DSCP para seleccionar um comportamento (escalonamento), enquanto o outro usa uma etiqueta para seleccionar a porta de saída (comutação).

Combinando a definição do DiffServ base e PHBs com MPLS baseado em TE é possível providenciar de facto qualidade de serviço em *backbones*. A isto podemos chamar “*MPLS support of DiffServ*” [Fauncher].

Na definição são apresentados dois tipos de LSPs: os E-LSPs e os L-LSPs. Num E-LSP, a etiqueta é usada como uma indicação de FEC, e o campo “*Exp*” é usado como indicador da classe do fluxo para seleccionar o PHB, incluindo quer o escalonamento quer a prioridade de descarte. Note-se que o DiffServ usa 6 bits para definir os BAs e os correspondentes PHBs. De salientar que o E-LSP apenas usa 3 bits para esta funcionalidade. Nos L-LSP, uma etiqueta é usada como indicação para o FEC e para a

prioridade de escalonamento. O campo “Exp” num L-LSP é apenas usado para indicar a prioridade de descarte.

Como se pode ver nas figuras seguintes, o significado para “5-tuple” é referente aos cinco campos de um cabeçalho IP (endereços IP da fonte e do destino, portas TCP ou UDP da fonte e do destino, e protocolo) que podem ser usados para definir o FEC.

Fonte: [Fineberg]

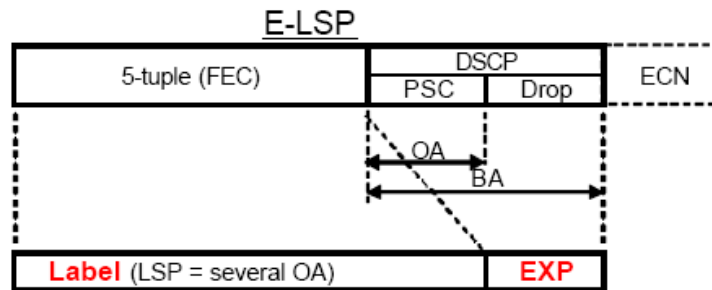


Figura 3.11: Mapeamento entre um cabeçalho IP e um cabeçalho MPLS SHIM para E-LSP

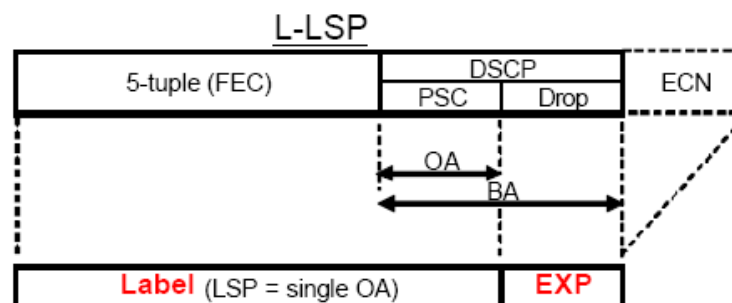


Figura 3.12: Mapeamento entre um cabeçalho IP e um cabeçalho MPLS SHIM para L-LSP

Nota: estas imagens são exemplificativas e não representam a estrutura total de um cabeçalho.

Cada tipo de LSP tem vantagens e desvantagens. O E-LSP é fácil de utilizar e é escalável porque preserva as etiquetas e usa o campo *Exp* para as características do DiffServ. Mas considerando que a sinalização de MPLS reserva a largura de banda por LSP, a largura de banda é reservada para um LSP sem a granularidade PSC, e pode haver escassez de largura de banda nas filas que servem alguns PSCs.

L-LSP, por outro lado, é mais difícil de planejar, porque são necessárias mais etiquetas para rotular todos os PSCs de todos os FECs. Mas, como as etiquetas transportam a informação de escalonamento, quando a largura de banda é reservada para um dado L-LSP, é associado com uma fila de prioridade para cada um dos LSP.

As duas figuras seguintes ilustram a forma como se distribui o tráfego e os parâmetros de QoS que melhoram o desempenho da rede usando MPLS e depois *DiffServ Support of MPLS* [Fineberg].

Fonte: [Fineberg]

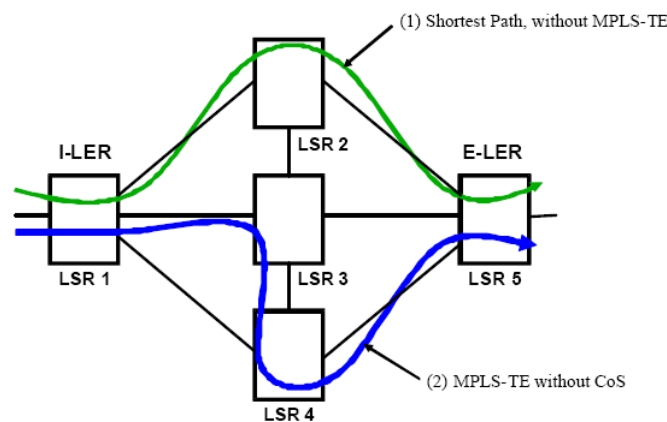


Figura 3.13: Fluxo de pacotes em MPLS sem DiffServ

Esta figura ilustra a diferença entre um caminho tomado pelos pacotes que são encaminhados com base no caminho mais curto (1) e encaminhados com base em engenharia de tráfego (2). Este segundo percurso poderia ter sido escolhido porque tem largura de banda suficiente para servir um dado FEC, mas esta largura de banda não está associada a nenhuma *class of service*, CoS, e assim o tráfego prioritário (por exemplo, VoIP) não tem largura de banda suficiente para esta fila em particular.

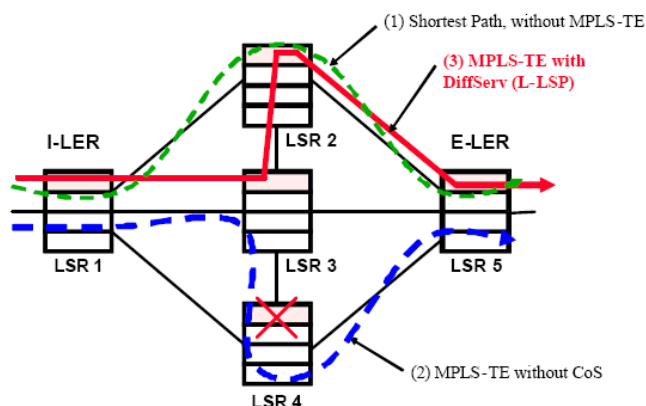


Figura 3.14: Fluxo de pacotes em MPLS com DiffServ

Nota: Os caminhos (1 e 2) da figura anterior estão representados nesta figura em linha não contínua para referência.

Esta figura ilustra um melhoramento na arquitetura. Nesta arquitetura, a tecnologia *MPLS support of DiffServ* é usada, e reservas de largura de banda podem ser realizadas respeitando a prioridade específica das filas. Vamos assumir que tráfego VoIP usa a fila-0, que é fila de topo em todos os LSR.

LSR-4 deve ter largura de banda suficiente em todas as suas filas, no entanto por algum motivo não tem largura de banda suficiente na fila-0, consequentemente, o caminho (2) não fornece QoS, que seria necessário para este tipo de tráfego, VoIP. Esta é a razão para a existência da cruz no LSR-4. Mas se um L-LSP é usado com reservas da largura de banda na fila-0, o tráfego pode assim ser encaminhado ao longo do caminho (3), através dos LSR-3 e LSR-2, permitindo que o tráfego VoIP possa ser entregue com garantia de QoS.

Resumindo, esta tecnologia *MPLS Support of DiffServ* satisfaz ambas as condições para o QoS: garante largura de banda e fornece tratamento diferenciado por fila. MPLS satisfaz a primeira condição, ou seja, garante que fluxos aplicativos passem em caminhos com LB garantida; e ao longo desses caminhos, DiffServ satisfaz a segunda condição providenciando serviço diferenciado por fila.

De salientar que este mecanismo é ainda mais simples e mais escalável que o IntServ com o standard RSVP. IntServ requer sinalização por microfluxo e manutenção de informação de estado de cada microfluxo em cada *router*. Em contraste, LSPs podem eles próprios transportar agregados de muitos microfluxos e assim requerer menos sinalização. Adicionalmente, os *routers* não guardam informação de estado por microfluxos. Apesar disso, os LSRs mantêm informação agregada da largura de banda disponível para todos os LSPs ou para cada uma das filas de prioridade.

Resumindo temos que:

Mecanismo	Reserva de recursos	Âmbito
DiffServ	Por classe (fila) num nó	Ponto a ponto
MPLS-TE	Largura de banda RSVP	Extremo a extremo
DiffServ-TE	Largura de banda RSVP por classe IP	Extremo a extremo

Tabela 3.2: Tabela comparativa entre os mecanismos DiffServ, MPLS-TE e DiffServ-TE

4. O AMBIENTE DE SIMULAÇÃO

Neste capítulo será apresentado de uma forma sucinta o porquê da simulação, o porquê da escolha de um simulador, a arquitectura do simulador escolhido e alguns aspectos relativamente a detalhes do mesmo simulador.

4.1. A NECESSIDADE DE UM SIMULADOR

As principais vantagens que um utilizador pode obter numa simulação são facilmente perceptíveis, nomeadamente:

- Um modelo, uma vez criado, pode ser utilizado inúmeras vezes para avaliar projectos e políticas propostas;
- A metodologia de análise utilizada pela simulação permite a avaliação de um sistema proposto, mesmo que os dados de entrada estejam, ainda, na forma de esquemas ou rascunhos;
- A simulação é, geralmente, mais fácil de aplicar do que métodos analíticos;
- Enquanto modelos analíticos requerem um número muito grande de simplificações para torná-los matematicamente tratáveis, os modelos de simulação, baseados na execução de algoritmos, não apresentam tais restrições (embora em alguns casos as simplificações sejam necessárias, de modo a reter apenas os elementos essenciais dos algoritmos, no que se refere a aspectos funcionais ou de

desempenho, conforme o objectivo da simulação). Além disso, nos modelos analíticos, as análises recaem apenas sobre um número limitado de medidas de desempenho. De maneira contrária, as informações geradas pelos modelos de simulação permitem a análise de, praticamente, qualquer medida computacionalmente aceitável;

- Uma vez que os modelos de simulação podem ser quase tão detalhados quanto os sistemas reais, novas políticas e procedimentos operacionais, regras de decisão, fluxos de informações etc. podem ser avaliados sem que o sistema real seja perturbado;
- Hipóteses sobre como ou porquê certos fenómenos acontecem podem ser testadas para confirmação. Permite também de uma forma metódica separar problemas/componentes que são inerentes à execução em tempo real;
- O tempo pode ser controlado. Pode ser comprimido ou expandido, permitindo reproduzir os fenómenos de maneira lenta ou acelerada, para que se possa melhor estudá-los;
- Podem-se compreender melhor quais são as variáveis mais importantes em relação ao desempenho e como as mesmas interagem entre si e com os outros elementos dos sistemas;
- A identificação de *bottleneck*, a maior preocupação na administração operacional de inúmeros sistemas, tais como fluxos de materiais, de informações e de produtos, pode ser obtida de forma facilitada, principalmente com ajuda visual;
- Um estudo de simulação costuma mostrar como realmente um sistema opera, em oposição à maneira como se pensa que ele opera;

Como se pode confirmar pela lista anterior a importância da simulação é um factor que não se pode dissociar de um trabalho com estas características.

4.2. NETWORK SIMULATOR (NS)

Depois de perceber a necessidade e a importância de utilizar um simulador, foi necessário analisar o mercado e analisar quais as ferramentas existentes e quais as que nos permitiriam atingir os objectivos propostos [SimComp].

Com base nestes pressupostos, surgiu-nos a hipótese do Network Simulator [NS] que é um simulador de eventos discretos para redes de comutação de pacotes. NS começou como uma variante do [REAL network simulator](#) em 1989 e nos últimos tempos tem evoluído substancialmente. Hoje em dia, o desenvolvimento do NS é suportado pela DARPA e pela NSF, incluindo outros centros de investigação como [ACIRI](#).

Esta ferramenta permite-nos uma customização total por se tratar de uma ferramenta *Open-Source*, escrita numa linguagem de acessível compreensão, baseada no paradigma *object oriented* e, sendo uma ferramenta bastante universal no mundo académico, possui bastantes módulos ao nível de redes, úteis para os fins do nosso trabalho. Nomeadamente,

- módulos para redes com e sem fios (incluindo ligações via satélite)
- módulos para protocolos de família TCP/IP
 - o IP, IP multicast, TCP, UDP,
 - o protocolos unicast e multicast,
 - o Aplicações: FTP, Telnet, Web, etc...
- Módulos de *Differentiated services*, *Multi Protocol Label Switching*
- Módulos de controlo de tráfego
- Módulos de geração de gráficos e estatísticas
- Módulos de disponibilizam ligações, nós, tipo de escalonadores, tipos de tráfego,
- entre outros...

4.3. ARQUITECTURA DO NS

Tal como já foi referido o Network Simulator é um simulador de eventos discretos baseado na metodologia *object oriented* visto basear-se essencialmente em duas linguagens também elas *object oriented*, C++ e TCL.

A arquitectura do NS é composta por 5 partes, mais a camada de base:

- Event scheduler
- Network components
- TclCL
- OTcl library
- Tcl 8.0 script language
- C/C++ (camada de base)

A figura seguinte mostra a arquitectura do simulador em causa.

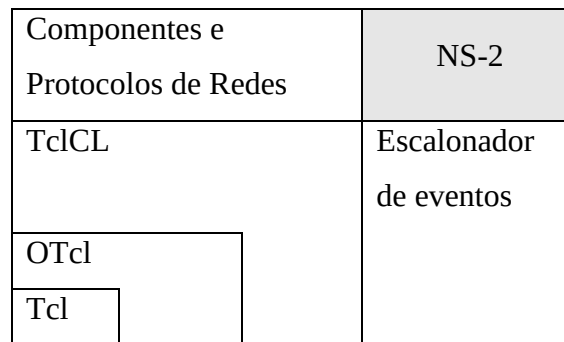


Figura 4.1: Arquitectura do NS

De acordo com [Chung], um utilizador pode começar a desenhar o seu modelo pelo canto inferior esquerdo, escrevendo, desenhando e correndo simulações em Tcl usando o simulador de objectos presente na biblioteca OTcl. A camada TclCL serve para partilha de métodos e variáveis entre as linguagens. Por fim, temos o gerador de eventos e os componentes de redes e de aplicação que estão implementados em C++ devido a questões de eficiência.

Na prática, o utilizador usa o NS da seguinte forma:

Fonte: [Karl]

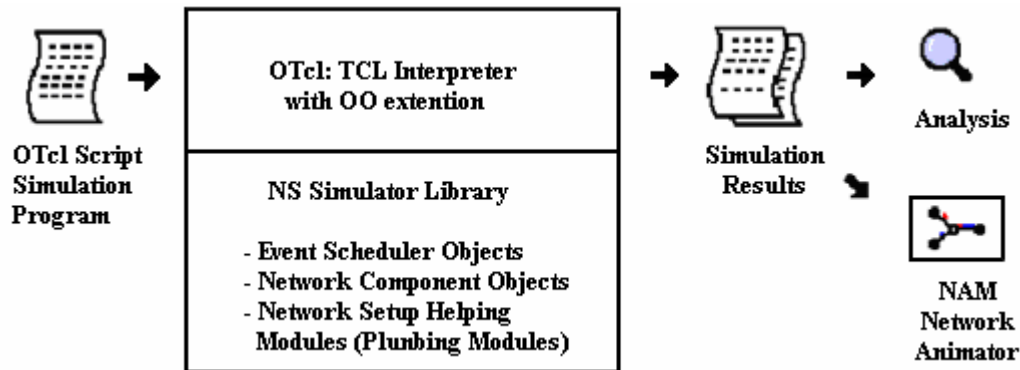


Figura 4.2: Aspecto geral do ponto de vista de um utilizador do NS

Ou seja, implementa-se um *script* em OTcl, o mesmo é interpretado pelo *core* do NS, que produz como *output* ficheiros com resultados que podem ser analisados e/ou colocados num outro simulador chamado *Network Animator* [NAM], que permite visualizar de uma forma gráfica o trabalho que foi desenvolvido pelo *core*; nomeadamente, pode observar-se a estrutura da rede desenhada no *script* OTcl e o tráfego que circula na mesma. Por forma a realizar uma análise mais visual usam-se muitas vezes ferramentas gráficas que permitem analisar os resultados na forma gráfica, salientando o uso do *Gnuplot* [Gnuplot].

4.4. OBJECTOS PRESENTES NO NS

A simulação de uma ligação ponto-a-ponto é iniciada pela construção da parte física da rede a simular, constituída por nós ligados entre si [Cunha]. Nos extremos da ligação, designados por nó de origem e nó de destino, são colocadas entidades responsáveis por definir o protocolo de transporte, UDP ou TCP, que se designam por agentes. A transmissão de dados é efectuada através de fontes de tráfego e/ou aplicações ligadas aos respectivos agentes, por exemplo CBR sobre UDP e FTP sobre TCP.

O início e o fim da simulação são definidos na agenda de eventos discretos, bem como os instantes de começo e da finalização da transmissão de dados.

O objectivo da simulação é estimar métricas de desempenho do sistema.

Será apresentada uma breve descrição dos objectos presentes no NS:

- **Nó**

Representa um emissor, um receptor ou um comutador da rede. Quando um objecto do tipo nó é criado, são também criados dois outros objectos, um classificador de endereços e um classificador de portas. A função destes classificadores é distribuir correctamente os pacotes que chegam aos agentes e os que partem para as ligações, respectivamente.

O nó criado é identificado numericamente de uma forma sequencial. Cada nó possui uma lista dos nós vizinhos e outra dos agentes ligados a si. Possui também um módulo de encaminhamento, onde são colocados os classificadores específicos de cada protocolo de encaminhamento.

Graficamente costumam ser representados da seguinte forma:



Figura 4.3: Exemplo da representação de nós em NS

- **Ligação**

A ligação entre dois nós pode ser unidireccional ou bidireccional. Ao criar a ligação é necessário especificar a largura de banda, o algoritmo de escalonamento e/ou técnica de descarte da fila de espera, e o atraso de propagação.

A Figura 4.4 apresenta todos os elementos (objectos) que constituem uma ligação unidireccional: *Queue*, *Delay*, *Agent/Null* e *Time To Live(TTL)*. Estes elementos também existem no sentido oposto no caso da ligação ser bidireccional.

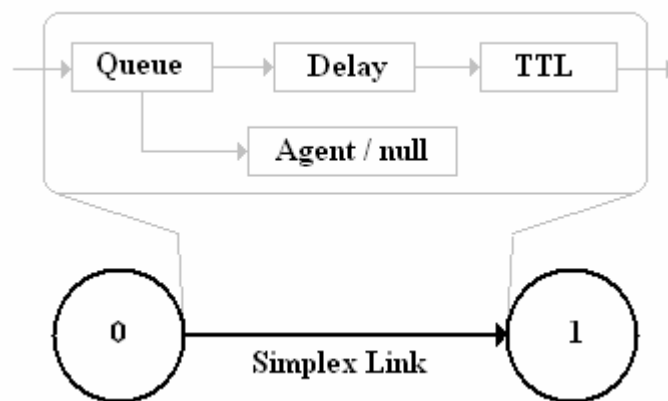


Figura 4.4: Objectos presentes numa ligação unidireccional

No NS, a fila de espera, *Queue*, não é implementada no nó, mas sim na ligação. Os algoritmos de escalonamento e as técnicas de descarte possíveis de configurar foram descritos no [Capítulo 2](#).

O objecto *Agent/Null* é utilizado quando existe a necessidade de descartar os pacotes que chegam à fila. O atraso de propagação é simulado através do objecto *Delay*. Por último, o objecto *TTL* actualiza no cabeçalho de cada pacote o valor de *TTL*.

- **Agente**

Os agentes definem o protocolo de transporte a utilizar, TCP ou UDP, e indicam qual o nó de origem e o nó de destino do tráfego. Os agentes são criados aos pares, ou seja, para cada agente de origem é necessário um agente de destino.

O UDP é um protocolo de transporte não orientado à ligação. Neste protocolo não existe o conceito de sessão. Os pacotes podem chegar ao destino fora de ordem e não existe garantia de entrega dos mesmos. O TCP é um protocolo de transporte orientado à ligação. Neste protocolo existe o conceito de sessão sendo as ligações estabelecidas através de um processo designado por *Three-way-handshaking*. O TCP garante a entrega dos pacotes pela ordem que foram enviados e implementa mecanismos de controlo de fluxo.

Os agentes de destino no caso UDP, aquele que foi utilizado na elaboração deste trabalho, não têm qualquer papel na implementação do protocolo, mas podem ser utilizados para obter informações sobre a rede como por exemplo número de pacotes recebidos e/ou perdidos.

Nesta ferramenta de simulação os agentes de origem e destino, depois de criados, são associados aos seus respectivos nós e são ligados entre si. A Figura 4.5 apresenta agentes de origem, *scr0* e *scr1*, e agentes de destino, *sink0* e *sink1*, a partilhar a ligação entre o nó 0 e o nó 1.

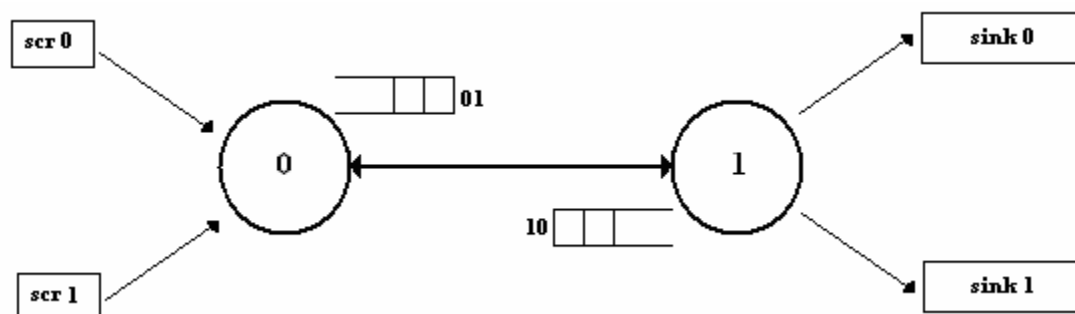


Figura 4.5: Cenário de uma simulação com agentes

- **Gerador de tráfego e aplicação**

O envio de pacotes entre o nó de origem e o nó de destino no NS pode ser efectuado através de várias fontes de tráfego e aplicações presentes no simulador. Foram utilizadas dois tipos de fontes de tráfego CBR e Exponencial.

A fonte de tráfego **CBR**, gera tráfego com uma taxa de transmissão fixa; já a **Exponencial**, gera tráfego com períodos ON/OFF de acordo com uma distribuição exponencial, os pacotes são enviados com uma taxa de transmissão fixa durante o período ON. Nenhum pacote é enviado durante o período OFF.

No NS existe também a possibilidade de usar aplicações, para simular Telnet ou FTP.

- **Agendar eventos discretos**

A agenda de eventos discretos é criada aquando da criação do objecto *Simulator*.

No *script* TCL define-se o início e o fim da simulação, que correspondem ao primeiro e ao último evento da agenda. Durante a simulação novos eventos podem ser agendados pelos objectos, por exemplo início e o fim da transmissão de dados. Os eventos são colocados e ordenados na agenda de eventos por ordem cronológica. O simulador começa a ler a agenda quando recebe o evento de início de simulação, lê o primeiro evento, executa o evento e passa ao próximo evento. Repete estes passos até receber o evento que indica o fim da simulação.

De salientar que o tempo no NS não corresponde ao tempo real. Não existe qualquer relação entre a duração da simulação e o instante de tempo agendado para o evento que indica o fim da simulação. A duração da simulação depende do número de eventos que são executados durante o decorrer da simulação.

- **Monitorização da fila de espera**

O NS permite monitorizar qualquer fila de espera, de modo a calcular os seus parâmetros de desempenho, como por exemplo calcular o número de pacotes ou *bytes* que chegam e que partem da fila de espera ou saber qual a atraso médio dos pacotes na fila de espera. O objecto responsável pela monitorização é o *QueueMonitor*, que por omissão monitoriza a fila com um período de 0,1s (valor configurável).

CP	TotPkts	TxPkts	ldrops	edrops
----	-----	-----	-----	-----
0	23270	100.00%	0.00%	0.00%
10	20500	100.00%	0.00%	0.00%
46	14575	100.00%	0.00%	0.00%
48	27	0.00%	0.00%	100.00%
----	-----	-----	-----	-----
All	58372	99.95%	0.00%	0.05%

Figura 4.6: Exemplo de valores extraídos do *Queue Monitor*

- **Ferramentas de visualização**

O NS disponibiliza um programa para visualizar a topologia da rede assim como os pacotes transferidos, algo mais interactivo, chamado *Network Animator* (NAM) e outro para efectuar gráficos, *Xgraph*. No entanto, o *Gnuplot*, programa disponível no Unix para gerar gráficos, também pode ser utilizado.

O **NAM** é baseado num ficheiro de recolha da actividade da rede, criado e activado no *script* TCL. Correndo então o ficheiro criado obtemos o resultado apresentado na Figura 4.7.

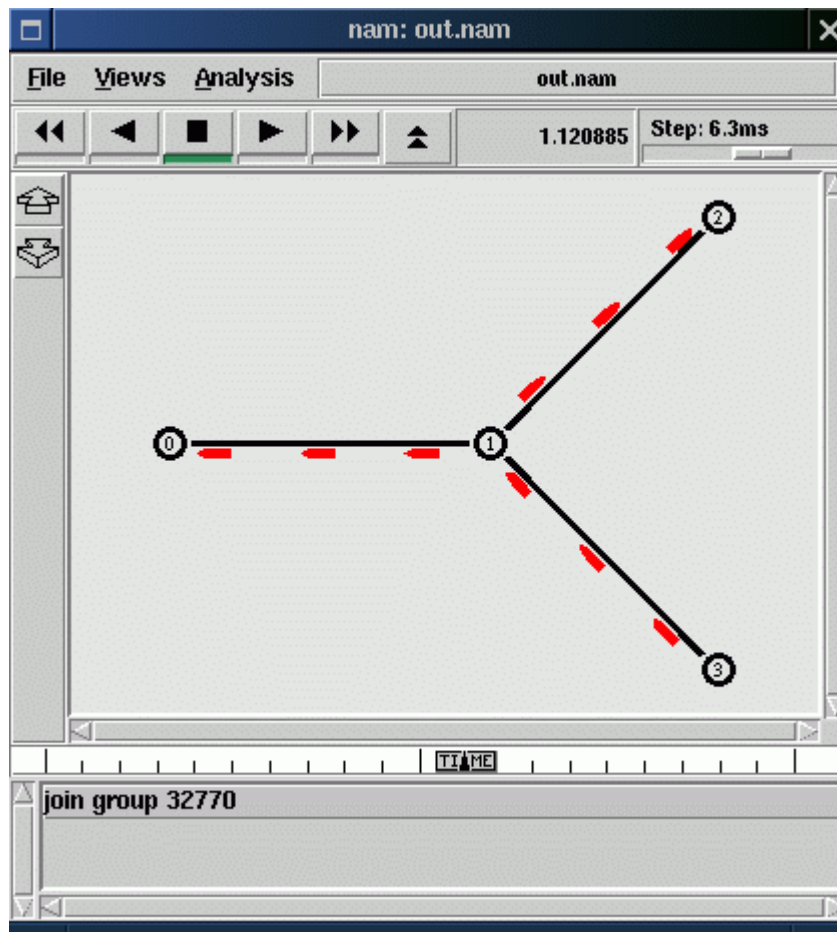


Figura 4.7: NAM

O NAM permite diferenciar por cores todos os elementos da topologia, por exemplo atribuir cores a cada nó e a cada ligação, bastante interessante para se poder ver o que se passa aquando de uma simulação.

O **Xgraph** é um programa para criar gráficos a duas dimensões. É possível gravar os mesmos nos seguintes formatos: *postscript*, *Tgif*, *HPGL* e *ldraw*. Os ficheiros interpretados por esta aplicação devem conter duas colunas, uma com os dados da coordenada X e outra com o valor da coordenada Y.

O **Gnuplot** é um programa presente no Unix/Linux que permite desenhar gráficos a duas ou a três dimensões. Esta aplicação é mais versátil que a anterior pois: permite distinguir as curvas não apenas pelas cores, o que é muito útil quando se imprime documentos a preto e branco; permite escolher num ficheiro quais as

colunas a utilizar para efectuar os gráficos, o que permite ter os dados para vários gráficos num único ficheiro; permite um grande número de possíveis extensões para gravar em ficheiro.

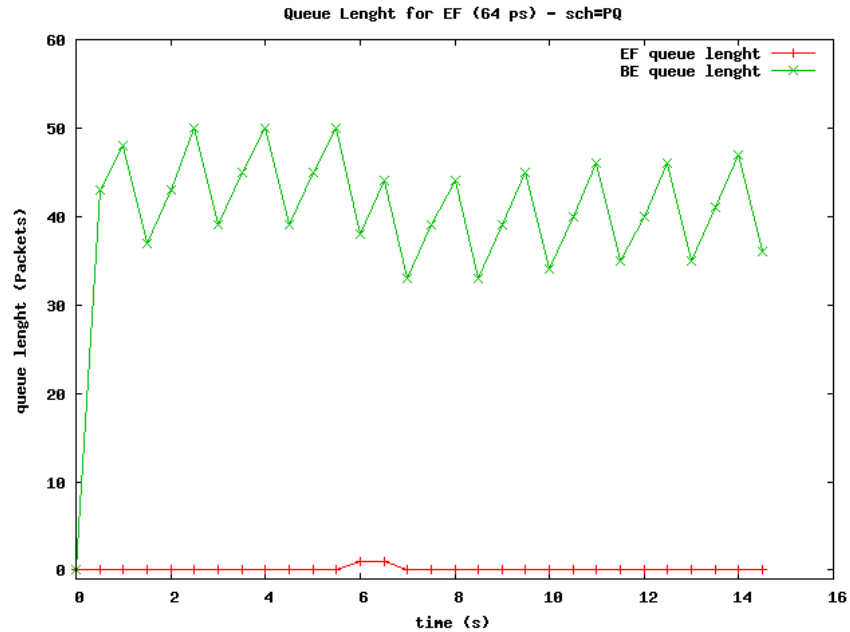


Figura 4.8: Exemplo de gráfico criado pelo GnuPlot

- **Algoritmos de escalonamento e técnicas de descarte**

Em termos genéricos, no NS não é feita a distinção entre algoritmos de escalonamento e técnicas de descarte. Por esta razão, nesta ferramenta de simulação o algoritmo de escalonamento do tipo FIFO é designado por *DropTail*. O nome do algoritmo refere a técnica de descarte presente, que neste caso tanto pode ser *DropTail* como *DropFront*.

4.5. MÓDULO DIFFSERV

Tal como foi dito anteriormente uma das grandes vantagens do simulador em questão é que se trata de uma ferramenta *freeware* muito utilizada a nível académico, o que permite disponibilizar um conjunto de módulos que oferecem um conjunto interessante de funções básicas e não só. Existem também *upgrades* a módulos que

embora disponíveis de raiz pelo NS não fazem o esperado ou não respondem a todas as situações.

O módulo de *Diffserv* foi desenvolvido pela *Nortel Networks* e adicionado ao NS em 2 de Novembro de 2000; no entanto, o suporte por parte desta empresa foi descontinuado, o que fez com que outros utilizadores desta ferramenta de simulação tivessem desenvolvido o *Diffserv4NS* [NS Diffserv] que adicionou novas funcionalidades [Andreozzi] ao módulo base. As mais utilizadas no decorrer deste projecto foram:

- Possibilidade de definir regras baseadas no nó de entrada e no nó de destino, no tipo de protocolo e no tipo de aplicação.
- Adicionados novos escalonadores: WFQ, WF²Q+, SCFQ, SFQ, LLQ, ...
- Possibilidades de novos métodos de monitorização:
 - o Para tráfego UDP
 - One Way Delay, OWD
Mede o atraso que um pacote sofre no percurso *end-to-end*
 - IP Packet Delay Variation, IPDV
Permite medir a variação do atraso num nó de destino para pacotes de um mesmo microfluxo.

Para além disso, na prática, este módulo simula também dois grupos de PHB normalizados pelo IETF, o AF e o EF. No caso do AF é possível configurar quatro classes e três níveis de *Drop Precedence*, DP, por classe, o que permite diferenciação de serviço dentro da mesma classe. Os AF PHB, definidos na tabela, são mapeados directamente numa fila de espera física (que define qual a classe do pacote) e numa fila de espera virtual (que define qual a DP). Os DP são simulados utilizando para cada classe (por cada fila física), três filas de espera virtuais configuradas com a técnica de descarte RED, onde os parâmetros do RED definem o nível baixo, médio ou alto, da probabilidade de descarte. Cada DSCP configurado é mapeado para uma fila de espera virtual de uma determinada fila de espera física. O EF PHB é configurado utilizando *Priority Queuing*, PQ, como algoritmo de escalonamento, onde

é possível configurar qual a taxa máxima dedicada à fila, de modo a que outras filas possam ser servidas.

4.6. TIPOS DE FONTES DE TRÁFEGO

Nesta secção serão apresentadas as fontes de tráfego usadas no decorrer deste projecto. Existem inúmeras soluções que podem ser usadas para simular a variedade de tráfego que temos hoje em dia. No entanto, só serão apresentadas as seguintes: *Constant Bit Rate* e Exponencial.

4.6.1. TRÁFEGO CONSTANT BIT RATE, CBR

No modelo de tráfego CBR, os pacotes são gerados com uma taxa fixa, isto é, o pacote e o intervalo entre pacotes são constantes. Primeiro, é necessário especificar a taxa da injeção dos dados. Depois, é necessário especificar o tamanho do pacote ou o intervalo entre pacotes. Aplicações como voz digital não comprimida (amostras de 8 bits transmitidas em 64 kbps), áudio e vídeo não comprimido são exemplos típicos do tráfego CBR. A facilidade de implementação é a vantagem deste modelo do tráfego. O inconveniente principal deste modelo é o facto que as aplicações reais variam normalmente as suas taxas da injeção dos dados.

Podem ser configurados os seguintes parâmetros:

PacketSize_ tamanho constante dos pacotes gerados.

rate_ taxa de saída.

interval_ (opcional) intervalo entre pacotes.

maxpkts_ número máximo de pacotes enviados.

4.6.2. TRÁFEGO EXPONENCIAL

Tráfego exponencial gera tráfego On/Off exponencial. Durante os períodos *on*, os pacotes são gerados com uma taxa constante. Durante os períodos *off*, não existe geração de tráfego. Os valores dos parâmetros *burst time* e *idle time* são calculados

segundo uma distribuição exponencial. Neste tipo de tráfego são configurados os seguintes parâmetros:

PacketSize_ tamanho dos pacotes gerados.

burst_time_ tempo médio dos períodos *on*.

idle_time_ tempo médio dos períodos *off*

rate_ taxa de geração de tráfego nos períodos *on*.

5. PROPOSTA DE UM NOVO ALGORITMO DE ESCALONAMENTO

Neste capítulo serão apresentados as principais motivações, princípios e pressupostos que levaram à idealização de um novo modelo de escalonamento, bem como uma descrição do novo algoritmo proposto.

5.1. INTRODUÇÃO

Alguns modelos de escalonamento descritos na literatura [Zeng] e concebidos para tratar de forma diferenciada classes de tráfego (exemplo: DiffServ) não são os mais adequados no que se refere à possibilidade de controlar parâmetros de desempenho, conforme se discutirá de seguida. (esta constatação será demonstrada no [Capítulo 6](#) com base em resultados de simulação).

Na análise consideramos três classes de tráfego com características e requisitos que podem ser associados às classes (PHB) DiffServ EF, AF e BE que, por isso serão usadas como referência. Considera-se apenas um PHB AF, mas o modelo pode ser generalizado pela inclusão de várias classes AF. Não será objecto de discussão a existência de níveis de precedência de descarte em AF – uma vez que tal não tem impacto directo nas decisões de escalonamento, sendo objecto de outros mecanismos

(policiamento e marcação por um lado, e descarte, por exemplo com base em RED, por outro).

Na análise preliminar de possíveis algoritmos de escalonamento podemos considerar num extremo um **algoritmo de Prioridade estrita (PQ)**, servindo (por ordem de prioridade) as classes EF, AF e BE.

Servir tráfego EF com base em prioridade estrita é aceitável (e usual, sendo também adoptado no algoritmo proposto), mas será altamente discutível no escalonamento de tráfego AF e BE. De facto conceder prioridade a AF sobre BE significaria continuar a servir AF em detrimento de BE aquando de picos de tráfego AF (isto é, *bursts* de pacotes com débito instantâneo acima do débito médio) – o tráfego AF receberia um serviço com uma qualidade superior ou mesmo muito superior à contratada (débito médio garantido), em detrimento de BE. Uma vez que a aceitação de fluxos EF e AF é controlada por um mecanismo de Controlo de Admissão e que os fluxos aceites são policiados, é certo que o tráfego BE tem um débito médio garantido, mas seria claramente e desnecessariamente prejudicado (nomeadamente com eventuais perdas adicionais) em escalas temporais da ordem da duração dos *bursts* de tráfego AF, se AF fosse tratado com prioridade estrita.

Num outro extremo poderíamos ter um **algoritmo do tipo Weight Fair Queueing WFQ** (ou semelhante). WFQ pode ser adequado para escalonar fluxos de uma mesma classe mas, mesmo neste caso, poderá não garantir equidade no caso de fluxos com diferentes graus de *burstiness*. Considerando fluxos agregados e diferentes classes de tráfego, a adopção de WFQ pressupõe a atribuição de pesos às diferentes classes. No entanto, a escolha de pesos com base em (proporcional aos) débitos médios não é aconselhável (como é fácil de concluir e se evidenciará no [Capítulo 6](#), por meio de simulação). Por um lado não seria possível oferecer as garantias críticas requeridas pela classe EF (débito instantâneo e atraso). Por outro, não trataria de forma adequada situações de tráfego *bursty* em que os débitos instantâneos podem ter variações significativas em relação à média. Em termos gerais não seria possível controlar de forma apertada o atraso do tráfego EF (o que é crítico) nem o do tráfego AF, embora

neste caso o controlo pudesse ser mais relaxado (o que poderia constituir um critério interessante na diferenciação entre AF e BE). Naturalmente que seria possível atribuir pesos que, em primeiro lugar, favorecessem EF em relação às outras classes e que igualmente favorecessem AF em relação a BE mas, como se verá por meio de simulação, não é fácil dimensionar de forma rigorosa estes pesos face a objectivos de desempenho, em particular se as condições de tráfego variarem quer no contexto dos fluxos activos quer por força da alteração do número de fluxos aceites. O ajuste dos pesos teria sempre um carácter empírico, pois seria difícil relacioná-los de forma objectiva e mensurável com parâmetros de desempenho; mesmo para os perfis de tráfego negociados, as variações inerentes à natureza *bursty* do tráfego não permitiriam uma afinação óptima válida para diferentes situações de tráfego. O problema agrava-se no caso de haver necessidade de reconfigurar parâmetros gerais do sistema devido a políticas administrativas ou alteração de parâmetros de algoritmos de controlo, devido à necessidade de alterar a respectiva estratégia. Para além disso, é reconhecida a complexidade de implementação do algoritmo WFQ e suas variantes.

Entre os dois extremos apresentados, seria ainda possível considerar um mecanismo de prioridade para EF e tratar AF e BE com base em WFQ. No entanto temos de novo o problema de determinar os pesos de forma a controlar de forma apertada a atribuição de recursos a AF, tendo em atenção a variabilidade deste tráfego em diferentes escalas temporais. Por um lado temos de considerar a possibilidade de ocorrência de *bursts* transmitidos com um débito instantâneo superior ao débito médio. Por outro lado a ocorrência de um *burst* máximo pode originar atrasos para os quais é possível estimar um valor alvo de referência, admitindo que (na pior das hipóteses) o *burst* seria servido com um débito igual ao débito médio negociado. A escolha dos pesos (com eventual favorecimento de AF, por força das garantias que lhe são devidas) teria igualmente consequências na forma como AF e BE repartiriam eventuais recursos não utilizados por EF. De notar que a utilização destes recursos por parte de AF permite melhorar marginalmente o respectivo desempenho (uma vez que AF tem já garantido um débito médio), mas pode melhorar significativamente o desempenho de BE o que justifica a necessidade de algum controlo sobre este processo em escalas temporais relativamente curtas, como as atrás referidas (duração de *bursts* de tráfego AF e tempo

de atraso de referência dos pacotes AF). Claramente a manipulação de pesos WFQ não oferece uma resposta geral para o problema, que se adapte à dinâmica do sistema.

Pelas razões expostas não subscrevemos nenhuma das soluções referidas, o que justificou a proposta de um novo algoritmo de escalonamento, baseado num conjunto de princípios e num modelo genérico a seguir apresentados.

5.2. OBJECTIVOS

O modelo proposto foi concebido com o objectivo de dar resposta às questões (e às limitações) anteriormente levantadas e discutidas.

O novo modelo baseia-se num conjunto de pressupostos relacionados com a adopção de técnicas de Engenharia de Tráfego (por exemplo suportadas por MPLS) associadas a DiffServ e com o facto de os fluxos EF e AF serem objecto de controlo de admissão, *shaping* e policiamento. Isto permite assumir que a rede dispõe de recursos compatíveis com o tráfego submetido e as garantias negociadas por estas classes de tráfego aquando da respectiva aceitação, sendo da competência do algoritmo de escalonamento a decisão de seleccionar a classe de tráfego a servir, tendo em atenção os recursos atribuídos a cada classe, o tráfego gerado por cada classe e os respectivos objectivos de desempenho.

O modelo usa ou baseia-se em parâmetros usados por outros mecanismos e atribui-lhes significado específico no contexto do escalonamento, o que lhe confere uma grande simplicidade e permite uma visão coerente e integrada do processo global de controlo de tráfego.

Um dos problemas identificados em alguns algoritmos de escalonamento é a impossibilidade de desacoplar o controlo do débito dos fluxos do respectivo atraso (possivelmente relevante no caso de tráfego AF), situação a que igualmente se pretende dar resposta com o presente algoritmo.

5.3. PRESSUPOSTOS E RELAÇÕES IMPORTANTES

O cenário que se considera é o de uma rede DiffServ com mecanismos de Engenharia de Tráfego idênticos aos que seriam disponibilizados por MPLS (pelo que se considera este caso como referência).

Assume-se que o tráfego de cada classe DiffServ (EF, AF e BE) é associado a LSPs distintos. Assume-se ainda que, com base nos mecanismos de Engenharia de Tráfego, é associada largura de banda a cada LSP estabelecido na rede (compatível com a repartição da largura de banda disponível para as várias classes de tráfego determinada, por exemplo, por políticas administrativas).

Isto tem duas consequências (vantagens) importantes, que são exploradas pelo algoritmo de escalonamento:

- em primeiro lugar, o mecanismo de Controlo de Admissão pode fazer uso da informação relativa à largura de banda total atribuída ao LSP que transportará o fluxo objecto de negociação, bem como do nível actual de tráfego aceite pelo LSP (com base nos parâmetros negociados pelos fluxos já admitidos ou em medidas efectivas do nível real de tráfego) e das características e requisitos do fluxo a admitir.
- em segundo lugar é possível em cada nó e em cada interface de saída associar largura de banda a cada classe DiffServ e que é simplesmente a soma das larguras de banda atribuídas a todos os LSPs que atravessam essa interface e que transportam tráfego dessa classe.

O algoritmo de escalonamento proposto e cujos princípios e fundamentos serão apresentados a seguir será discutido nesta perspectiva.

Consideram-se três classes de tráfego, que passaremos a designar por EF, AF e BE. Cada classe tem associada uma certa largura de banda (ou capacidade), que será designada, respectivamente, por C_{EF} , C_{AF} e C_{BE} , sendo a respectiva soma igual à capacidade C da ligação física.

$$C_{EF} + C_{AF} + C_{BE} = C$$

Pressupõe-se desta forma que a classe BE tem, implicitamente, associada uma largura de banda mínima garantida que corresponde à largura de banda não atribuída a EF e AF.

Admitimos ainda que o tráfego EF e o tráfego AF são objecto de controlo de admissão, sendo definidos limiares de aceitação tipicamente inferiores às capacidades associadas aos respectivos LSPs (o que corresponde a um certo nível de *overprovisioning*). Acresce ainda que o nível de tráfego efectivamente negociado e aceite pode ser inferior a esses limiares e que o tráfego efectivamente gerado é tipicamente inferior ao contratado (no limite poderia ser igual, uma vez que a função de policiamento impede que seja superior).

Isto significa que em cada nó e em cada interface os níveis médios de tráfego EF e AF são, em princípio, inferiores às respectivas capacidades associadas. A capacidade nominalmente atribuída a cada classe pode, eventualmente, ser (muito) superior à necessária para escoar o tráfego real, o que em primeiro lugar permite oferecer a essa um desempenho (muito) superior ao esperado e, em segundo lugar, disponibilizar a outras classes os recursos não utilizados. O escalonador tem como referência a capacidade atribuída a cada classe e não (no caso de EF e AF) o nível de tráfego de facto aceite pelo processo de Controlo de Admissão.

A cada classe é associado um débito de serviço de referência que é nominalmente igual à respectiva capacidade, isto é, $R_{EF} = C_{EF}$, $R_{AF} = C_{AF}$ e $R_{BE} = C_{BE}$, sendo portanto

$$R_{EF} + R_{AF} + R_{BE} = C$$

Admitimos que no caso de tráfego AF o critério de Controlo de Admissão será baseado no débito médio dos fluxos, enquanto que no caso de EF poderá ser usado um

critério de débito médio ou de débito de pico. No último caso o nível efectivo de utilização de recursos será menor do que no primeiro caso (menor número de fluxos admitidos), pelo que o nível de reutilização por outras classes (em particular por BE) será superior.

Designamos por r_{AF} o valor médio negociado dos fluxos AF aceites pelo processo de Controlo de Admissão e assumimos que $r_{AF} \leq R_{AF}$.

No caso de tráfego EF, representamos por p_{EF} o débito de pico (que será na pior das hipóteses a soma dos débitos de pico dos fluxos aceites) e por r_{EF} o valor do débito médio total dos fluxos aceites. Dependendo do modelo de controlo de admissão, poderá eventualmente ser conhecido (negociado) apenas um dos valores. Conforme os casos será $r_{EF} \leq R_{EF}$ ou $p_{EF} \leq R_{EF}$, mas uma vez que $r_{EF} \leq p_{EF}$ será sempre, em qualquer caso,

$$r_{EF} + r_{AF} < R_{EF} + R_{AF}$$

No entanto, se for usado um critério de débito médio para a aceitação de tráfego EF, deveria ainda garantir-se

$$p_{EF} + r_{AF} < C$$

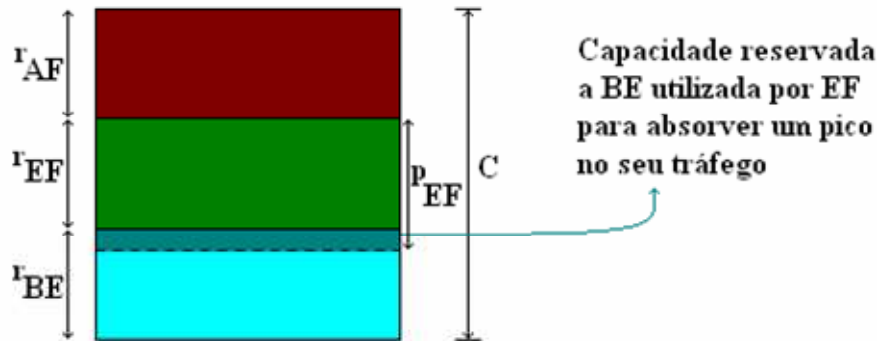


Figura 5.1: Ideia base do empréstimo de largura de banda no algoritmo a implementar

A não verificação desta condição (que é externa e portanto não controlada pelo escalonador) pode induzir uma penalização temporária para o tráfego AF. No entanto, conforme se verá, o escalonador está preparado para lidar com esta situação, tentando minimizar os problemas que lhe estão associados.

Nota: Os problemas de Engenharia de Tráfego e Controlo de Admissão não são obviamente tratados no âmbito deste trabalho. No entanto o algoritmo de escalonamento tem impacto no desempenho global do sistema, pelo que informação relacionada com o desempenho e os recursos utilizados pode ser usada para:

- alteração dinâmica de limiares do mecanismo de Controlo de Admissão (o que não tem qualquer impacto nos parâmetros do algoritmo de escalonamento, mas que pode afectar o volume de tráfego aceite por classe e portanto o respectivo desempenho);
- alteração da distribuição de largura de banda pelas classes, o que tem como consequência a alteração de parâmetros do escalonador (mas o mesmo aconteceria por exemplo em WFQ com a necessidade de alteração de pesos). Estas alterações são, em princípio pouco frequentes.

5.4. PRINCÍPIOS DO MODELO

O modelo proposto baseia-se, conforme atrás referido, na atribuição de prioridade estrita ao tráfego EF (uma possível variante, com impacto reduzido nos princípios adoptados, seria conceder prioridade estrita ao tráfego EF após passagem por um *Leaky Bucket* que controlasse o respectivo débito de pico). Por outro lado, a decisão de escalonamento de tráfego AF e BE baseia-se na comparação dos débitos reais de serviço com os respectivos débitos de referência e numa política de empréstimos que tem também em conta a interferência de tráfego EF (que tem prioridade no acesso a recursos e posterior devolução, ou que pode simplesmente não estar a consumir a totalidade de recursos que lhe são atribuídos).

Qualquer classe pode estar a consumir recursos abaixo da sua quota (isto é, em relação ao respectivo débito de serviço de referência), momentaneamente ou de forma continuada, o que pode dever-se a várias razões (que podem coexistir):

- nível de tráfego aceite pelo processo de Controlo de Admissão (caso de EF e AF) inferior ao limiar de aceitação definido;
- tráfego médio gerado inferior ao negociado (EF e AF) ou ao valor de referência (BE);
- tráfego instantâneo abaixo do respectivo valor médio negociado (EF e AF) ou de referência (BE);

Isto significa que é possível adoptar uma política de empréstimos entre classes que deve ter regras muito precisas com base na comparação dos valores com o serviço de referência.

5.4.1. POLÍTICA DE EMPRÉSTIMOS

As regras para o empréstimo podem ser descritas pelos três tipos de classes de tráfego, apresentadas anteriormente.

- **Tráfego EF**

Uma vez que tem prioridade estrita, todo o tráfego EF é servido à velocidade da ligação física. Isto significa que monopoliza os recursos de transmissão – mas fá-lo apenas momentaneamente, isto é, de seguida “liberta” os seus recursos para compensar os que usou em excesso, uma vez que o tráfego EF é limitado pelo processo de Controlo de Admissão (sendo neste contexto irrelevante se a aceitação de fluxos em relação ao respectivo limiar de admissão foi feita com base num critério de débito máximo ou de débito médio). Para além desta troca (empréstimo e devolução), recursos eventualmente não utilizados por EF (por qualquer das razões acima referidas) ficam disponíveis para AF e BE.

- **Tráfego AF**

Os recursos reservados para este tráfego (e que são tidos em conta pelo processo de Controlo de Admissão) garantem o débito médio negociado. Por outro lado, uma vez que a duração dos *bursts* de um fluxo é superiormente limitada (policiamento por meio de *token bucket*, cujos parâmetros são objecto de negociação e aceitação) é possível dimensionar o sistema de forma a evitar perdas e controlar o atraso, na condição de ser garantido o débito médio. A situação real pode, no entanto, afastar-se momentaneamente deste padrão. Por um lado, o tráfego EF pode temporariamente monopolizar os recursos de transmissão e, sendo o tráfego AF de débito variável, podem ocorrer simultaneamente picos de tráfego EF e AF. Por outro lado, os picos de tráfego AF podem beneficiar de recursos que não estejam a ser usados por tráfego EF (mas neste caso o seu uso pode ter de ser partilhado com tráfego BE). Finalmente, se (ou quando) o débito instantâneo do tráfego AF for inferior à média, os recursos correspondentes podem ser usados por tráfego BE, com posterior restituição (se necessário).

- **Tráfego BE**

Este tipo de tráfego terá um débito mínimo garantido, mas uma vez que o tráfego BE não é objecto de Controlo de Admissão nem de policiamento, pode exceder largamente a respectiva quota, com a consequente ausência de garantias. Em termos simples, o tráfego BE usa os recursos que estejam momentaneamente disponíveis. Mas uma vez que tem uma quota mínima garantida e existe uma política de empréstimos, a afirmação anterior deve ser entendida no contexto de um mecanismo de controlo de empréstimos. Por exemplo, se o tráfego BE não estiver a usar a sua quota garantida de recursos (de que pode beneficiar o tráfego AF para escoar os respectivos picos), não poderá vir a reclamar esse facto posteriormente (“*use it or loose it*”). No entanto se estiver a ser momentaneamente privado desses recursos, terá “direito” a reclamá-los posteriormente. Da mesma forma que se durante um período prolongado estiver a usar recursos acima da sua quota (recursos não utilizados por EF e/ou AF), não deverá vir a ser penalizado por esse facto durante um período de tempo prolongado – isto é, a

“memória” relativa à utilização de recursos acima da quota garantida deverá ser de curta duração (a discutir adiante).

Uma vez que o tráfego EF é tratado com prioridade estrita, pode dizer-se que o algoritmo de escalonamento se reduz a definir em que condições deverá ser escalonado tráfego AF ou BE. No entanto, convém não esquecer que EF desempenha um papel muito importante neste processo – quer porque vai usar temporariamente recursos de BE e eventualmente de AF, com a consequente posterior devolução; quer porque os recursos que lhe estão reservados e que este não use podem ser utilizados pelas outras classes. O algoritmo proposto tem assim de lidar com estas situações de forma coerente, tendo em atenção critérios bem definidos e que devem ser relativamente simples.

O algoritmo proposto baseia-se em primeiro lugar na largura de banda de referência atribuída a cada classe (independentemente do nível de tráfego EF e AF realmente admitido em relação aos respectivos limiares de aceitação ou do nível real de tráfego aceite que está a ser gerado em cada classe) e num parâmetro adicional, cuja variação pode permitir diferentes objectivos de controlo. Para além disso, embora seja exposto um critério para decidir o escalonamento de tráfego AF e BE, são possíveis outras variantes sem que sejam alterados os princípios básicos do modelo.

De notar que as larguras de banda de referência poderiam ser usadas em WFQ como critério para a definição inicial de pesos, sujeita a uma posterior redefinição para polarizar o algoritmo primeiro em favor de EF depois em favor de AF, de acordo com a discussão anterior.

Como ponto de partida assumimos que o tráfego BE tem uma largura de banda mínima garantida (no limite pode ser zero, mas normalmente será diferente de zero, para evitar situações em que o tráfego BE possa ficar totalmente privado de recursos). Essa garantia é entendida como um valor médio da largura de banda remanescente (não atribuída a EF e AF), uma vez que a largura de banda usada por estas classes de

tráfego é naturalmente limitada, quer pelo processo de Controlo de Admissão, quer por *Shaping* e Policiamento dos fluxos aceites.

Quanto ao tráfego AF, sendo possível garantir um débito médio (excepto possivelmente em intervalos muito curtos correspondentes a *bursts* de tráfego EF), o problema que se coloca é naturalmente como eventualmente melhorar o seu desempenho durante picos de tráfego face aos recursos disponíveis e possíveis empréstimos. Tal será possível atribuindo recursos adicionais para escoar esses picos, com a contrapartida de que haverá intervalos de tempo em que o débito instantâneo será inferior ao débito médio, o que permitirá então libertar os respectivos recursos (exclui-se da discussão o caso em que o nível médio de tráfego AF seja de tal forma baixo face à largura de banda atribuída à classe que os picos de tráfego possam ser escoados com os recursos disponíveis para a classe, sem necessidade de empréstimo). Está aqui implícito um processo de troca entre AF e BE e é exactamente este processo de troca em escalas temporais curtas que pretendemos controlar.

Este processo pode ser conceptualizado considerando apenas o tráfego AF e BE mas, na verdade, o processo sofre a interferência de EF (como já referido). No limite podíamos considerar que atribuiríamos todos os recursos necessários para escoar os picos de tráfego AF (a menos dos que estejam a ser usados por EF), com a certeza de que os recursos atribuídos em excesso (e tomados por empréstimo a BE) são seguramente devolvidos a seguir (por força da limitação do débito médio do tráfego AF). Esta solução corresponde a dar prioridade estrita a AF, o que vimos não ser desejável. Uma solução alternativa seria garantir incondicionalmente a BE (a menos dos recursos reclamados por EF) um serviço com o seu débito de referência, em intervalos de tempo de curta duração, o que poderia significar (na pior das hipóteses) que o tráfego AF seria servido com um débito igual ao seu débito de referência e portanto com acumulação momentânea de pacotes e consequente atraso suplementar durante os respectivos picos de tráfego. O algoritmo proposto considera de facto situações intermédias, que permitem não apenas um controlo do débito mas também, e de forma independente, do atraso do tráfego AF, e que tem como casos particulares e extremos os dois que acabam de ser referidos.

5.4.2. COMPARAÇÃO COM SERVIÇO DE REFERÊNCIA

Para cada classe designamos por g o débito médio de chegada de pacotes ao sistema (tráfego gerado), e por s e R os correspondente débito de serviço real e de referência. Enquanto g e s variam no tempo, R é constante.

O algoritmo proposto baseia-se na comparação de s com R quer para AF quer para BE (uma variante, discutida no fim, apenas necessita de fazer a comparação para o tráfego AF).

A comparação dos débitos pode ser feita com base na comparação do nível de ocupação (em bits) de duas filas de espera – uma real, isto é, escoada com o débito real de serviço decidido pelo escalonador, e outra virtual, servida com o débito de referência, designadas respectivamente por Q e Q_v . A fila virtual é de facto um simples contador.

As filas de espera são ambas alimentadas com o tráfego real cujo débito pode ser superior, igual ou inferior quer ao débito de serviço real quer ao débito de serviço de referência. Isto significa que qualquer das filas pode crescer, manter o nível ou diminuir, o que acontece de forma independente.

A afirmação de que, por exemplo, o débito de serviço é superior ao débito de referência (e que poderíamos representar de forma simplista por $s > R$) deve ser interpretada no sentido em que $\int s \, dt > \int R \, dt$, num intervalo em que a comparação seja relevante ou faça sentido, tendo como consequência que $Q < Q_v$, isto é, a fila real é escoada mais rapidamente que a fila virtual (e inversamente se $s < R$).

Podemos então usar com rigor a seguinte notação (tomando como exemplo o tráfego AF):

$$Q_{AF} = \int (g_{AF} - s_{AF}) dt$$

$$Q_{vAF} = \int (g_{AF} - R_{AF}) dt$$

E portanto

$$Q_{vAF} - Q_{AF} = \int (s_{AF} - R_{AF}) dt$$

Em rigor devemos assumir que

$$Q_{vAF} = \max\left(\int (g_{AF} - R_{AF}) dt, 0\right)$$

ou que $R_{AF} = 0$ quando a fila virtual estiver vazia.

Temos assim um critério muito simples para decidir se um fluxo está a ser servido acima ou abaixo do seu débito de referência. No primeiro caso diremos que está “avanzado” e no segundo que está “atrasado” em relação à referência utilizada. Caso esteja “atrasado” deverá ser promovida a respectiva recuperação. Caso esteja “avanzado” poderá eventualmente ter de restituir recursos. De agora em diante “avanzado” e “atrasado” serão usados nesta acepção.

Se considerarmos um cenário simples com apenas duas classes de tráfego, ambas a gerar tráfego com débito superior ao respectivo débito de referência (e com a soma destes igual à capacidade total a partilhar) e portanto a competir pela totalidade dos recursos, podemos dizer que o avanço de uma corresponderia ao atraso da outra e o processo de troca seria directo, de acordo com as regras que se entendesse definir.

No presente caso o processo de troca entre AF e BE é mais complexo, pois não só a existência de tráfego EF introduz uma perturbação adicional (com redução ou aumento da largura de banda disponível para AF e BE), como é possível que em certos períodos o tráfego instantâneo de uma classe seja inferior ao débito de referência, traduzido no facto de a fila virtual ficar vazia (isso pode acontecer com o tráfego AF pois o débito médio dos fluxos aceites é inferior ao débito de referência, a menos que o processo de Controlo de Admissão aceitasse tráfego acima desse limiar – *underprovisioning*).

Consideremos agora a análise detalhada das filas real e virtual para cada classe de tráfego.

- **Tráfego AF**

Assumimos que o tráfego AF (g_{AF}) é controlado por um *token bucket*, com parâmetros (r_{AF}, b_{AF}) de acordo com o contrato negociado. Tipicamente $r_{AF} \leq R_{AF}$, mas na análise das situações mais desfavoráveis (isto é, para obtermos condições limite) assumimos que $r_{AF} = R_{AF}$.

Durante *bursts* de tráfego transmitidos com um débito instantâneo superior ao débito de referência R_{AF} , a fila virtual cresce até um valor que no máximo pode igualar a profundidade do *token bucket* (b_{AF}).

Quanto à fila real, assumimos que em condições normais o débito de serviço é pelo menos igual ao débito de referência (inerente à garantia de um débito médio). Em caso de igualdade (leia-se $\int s_{AF} dt = \int R_{AF} dt$), $Q_{AF} = Q_{vAF}$. No caso de ser superior (leia-se $\int s_{AF} dt > \int R_{AF} dt$), então $Q_{AF} < Q_{vAF}$, podendo eventualmente Q_{AF} tender para zero, uma vez que g_{AF} não pode manter-se persistentemente acima de r_{AF} e o débito de serviço s_{AF} (em termos médios) poderá ser pelo menos igual a R_{AF} ; para além disso, se neste caso o débito de chegada g_{AF} for inferior ao débito de referência (leia-se $\int g_{AF} dt < \int R_{AF} dt$), então Q_{vAF} tende para zero, enquanto que o *token bucket* tenderá a ficar cheio (Q_{AF} também tende para zero uma vez que é inferior a Q_{vAF}).

Caso momentaneamente o débito de serviço seja inferior ao débito de referência (devido a monopolização de recursos por EF), $Q_{AF} > Q_{vAF}$, mas esta situação deve ser rapidamente corrigida (isto é, logo que existam recursos disponíveis, devem ser afectados a AF, de forma a garantir $Q_{AF} \leq Q_{vAF}$, objectivo essencial a atingir). Um caso particular será o de $Q_{vAF} = 0$ (*token bucket* cheio), sendo neste caso $Q_{AF} = 0$ o

objectivo a atingir (os recursos necessários para escoar o tráfego são inferiores aos nominalmente afectados ao tráfego AF). $Q_{vAF} - Q_{AF}$ é assim uma medida do “avanço” em relação ao serviço de referência.

O facto de o limite inferior de Q_{vAF} ser zero e não um valor negativo significa que não é contabilizada a não utilização de recursos abaixo da quota para posterior reivindicação – aliás é esse exactamente o comportamento de um *token bucket* quando cheio, situação em que novos tokens gerados são descartados.

Ou seja $Q_{vAF} = Q_{AF} = 0$ pode ter um duplo significado – serviço realizado com um débito igual ao débito de referência ou tráfego com débito inferior ao débito de referência e que portanto, em condições normais, deveria ser servido imediatamente.

$Q_{vAF} = b_{AF}$ significa que o *token bucket* está vazio, o que quer dizer que, mantendo-se esta situação, não será recebido tráfego adicional, pois isso é impedido pelo facto de não haver tokens disponíveis.

Podemos concluir que, no caso do tráfego AF, o comportamento da fila virtual está naturalmente relacionado com o mecanismo de *token bucket*. Por outro lado, as garantias que são devidas ao tráfego AF impõem que se deve garantir $Q_{AF} \leq Q_{vAF}$ (forçando a que isso aconteça o mais rapidamente possível quando $Q_{AF} > Q_{vAF}$) e que em condições normais $Q_{vAF} - Q_{AF} > 0$, sendo o respectivo valor uma medida do avanço de AF em relação à respectiva referência.

- **Tráfego BE**

A situação do tráfego BE é totalmente diferente, pois uma vez que pode exceder largamente o valor da respectiva quota, o valor de Q_{vBE} pode crescer sem controlo, tal como o valor de Q_{BE} .

Em condições normais é expectável que $\int s_{BE} dt > \int R_{BE} dt$ (pois os recursos de EF e AF não são utilizados na totalidade, pelas várias razões apontadas). Uma vez que Q_{BE} pode também crescer sem controlo e que o tráfego BE disporá de um *buffer* limitado B , tráfego recebido enquanto o *buffer* estiver cheio será descartado.

O tráfego BE pode encontrar-se momentaneamente “atrasado” (abaixo da sua quota) e este facto deve ser contabilizado (tal facto é indicado por $Q_{BE} - Q_{vBE} > 0$). No entanto, uma vez que o tráfego BE pode eventualmente estar “avanzado” ($Q_{vBE} - Q_{BE} > 0$), deve prever-se um tamanho para o buffer virtual B_v superior a B , de forma a acomodar na fila virtual pacotes aceites no sistema.

Um valor $Q_{vBE} - Q_{BE}$ elevado significaria que o tráfego BE beneficiou durante um intervalo de tempo longo de recursos de outras classes, estando a ser servido acima do seu débito de referência. É assim candidato a devolver recursos, nomeadamente em benefício de AF. No limite (e porque BE é o tráfego menos “prioritário”), poderíamos ser tentados a reduzir ou até interromper o serviço BE enquanto se verificasse $Q_{vBE} - Q_{BE} > 0$ (eventualmente até se atingir o valor nulo, altura em que o débito médio do tráfego BE igualaria o débito de referência). Isto seria uma penalização excessiva para o tráfego BE, que poderia simplesmente ter beneficiado de um nível reduzido de tráfego EF e/ou AF, não relacionado com a política de empréstimos.

Para limitar o tempo de memorização do avanço de BE deve impor-se uma condição do tipo $Q_{vBE} - Q_{BE} < X$ (pelo que deve ser $B_v = B + X$). X está de facto relacionado com o tempo durante o qual se memoriza o avanço de BE em relação ao débito de referência (memória de curta duração).

Pode atribuir-se a X o seguinte significado: a condição $Q_{vBE} = Q_{BE}$ é atingida ao fim de um intervalo de tempo igual a $\frac{X}{R_{BE}}$ durante o qual o tráfego BE não fosse servido.

Um possível critério para dimensionar X será discutido adiante.

As regras que se aplicam à fila real e à fila virtual BE são logicamente as seguintes:

- se a chegada de um pacote encontrar a fila real cheia, o pacote é descartado e também não é colocado na fila virtual (pois este pacote não será de facto escalonado);
- um pacote só é inserido na fila virtual se não violar a condição $Q_{vBE} - Q_{BE} < X$ (impondo esta condição e uma vez que $B_v = B + X$, a fila virtual nunca atinge o valor máximo).

5.4.3. ALGORITMO DE ESCALONAMENTO

Passamos então a discutir o problema do escalonamento de AF e BE (que só se coloca quando a fila EF estiver vazia e as filas AF e BE tiverem pacotes para escalonar).

Tendo maneira de decidir se o tráfego AF ou BE está avançado ou atrasado, um primeiro algoritmo de escalonamento muito simples poderia ser o seguinte

- se AF estiver atrasado é escalonado (independentemente do estado de BE)
- se BE estiver atrasado e AF avançado, deve ser escalonado BE (independentemente da causa do atraso)
- caso AF e BE estejam ambos avançados, poder-se-ia encontrar argumentos em favor de escalonar pacotes de uma ou outra classe com base apenas nesta informação. No entanto, seria possível considerar um processo de decisão mais complexo, baseado na quantificação do grau de avanço de AF e/ou BE (por exemplo em relação a um limiar); por exemplo o limiar de avanço de AF poderia ser relacionado (hipótese a verificar) com o controlo do atraso do tráfego AF. Foi exactamente neste sentido que evoluiu a análise do problema.

Como se discutiu, o principal mecanismo de empréstimo de largura de banda verifica-se entre AF e BE (pois EF apenas intervém indirectamente, uma vez que tem prioridade na utilização de recursos).

Para perceber como funciona o mecanismo de empréstimo entre AF e BE começou por se considerar o caso ideal em que não existe interferência de EF – isto é, trata-se de permuta directa entre AF e BE.

Começamos por considerar um caso em que o débito de chegada de AF e BE são ambos superiores ao respectivos débitos de referência e que a soma destes é igual à capacidade a partilhar. Se ambos os fluxos forem servidos com esses débitos, $Q_v = Q$ em qualquer caso. No entanto se, por exemplo, o tráfego AF for servido acima do seu débito de referência ($R_{AF} + \delta R$), o tráfego BE será servido abaixo do respectivo débito de referência ($R_{BE} - \delta R$) e o avanço de AF (medido por $Q_{vAF} - Q_{AF} = \delta Q$) será igual ao atraso de BE (medido por $Q_{BE} - Q_{vBE} = \delta Q$).

Podem no entanto ocorrer situações particulares – nomeadamente o caso em que BE adquiriu avanço pelo facto de AF estar momentaneamente a transmitir abaixo da sua quota (a que se poderá seguir um *burst* acima do seu débito médio negociado). Neste caso não há verdadeiramente atraso de AF, uma vez que o tráfego pode ser imediatamente servido, visto ser inferior ao débito médio garantido (como se viu, esta situação pode traduzir-se por $Q_{vAF} = Q_{AF} = 0$). Pode, no entanto, dizer-se que o avanço de BE se deve à não utilização de recursos por parte de AF, pelo que se justificaria a sua devolução (pelo menos parcial, tendo em atenção que o avanço de BE é apenas memorizado até um certo limite).

Pode então argumentar-se que, na ocorrência de um *burst*, será aceitável conceder um “avanço” a AF, que é compensado com uma diminuição do “avanço” anterior de BE. No limite, o avanço máximo possível de AF será igual a b_{AF} (correspondendo a $Q_{vAF} = b_{AF}$ e $Q_{AF} = 0$), a que corresponde a diminuição máxima do avanço de BE. Se memorizarmos um avanço máximo de BE igual a b_{AF} , significa que no limite do processo de devolução o avanço de BE seria na pior das hipóteses anulado. Isto significa que não necessitamos de memorizar em BE um avanço superior a b_{AF} . Podemos, no entanto, decidir memorizar um avanço menor, o que significa que nesse

caso BE perderia o avanço antes de se conseguir transmitir um *burst* máximo de AF – o que determinaria a elegibilidade de BE, com base no critério de estar atrasado, para um eventual escalonamento mais cedo e portanto mais favorável. Quanto menor for o valor máximo do atraso memorizado de BE, mais favorecido é o tráfego BE na política de empréstimo (no limite podia não ser memorizado qualquer avanço, isto é, $X = 0$). O avanço máximo de BE que é memorizado está naturalmente relacionado com a quantidade máxima de recursos a devolver (ou o tempo durante o qual a devolução se processa).

A situação que acaba de se descrever corresponde a ambos os fluxos estarem avançados, devido a uma condição particular inicial de AF – no entanto com AF a aumentar o seu avanço e BE a perdê-lo, por troca. Podemos naturalmente impor uma condição em que a troca não se faça por completo, uma vez que não é necessário servir o *burst* AF com atraso nulo (ou muito pequeno). Aliás, se a condição inicial fosse um *burst* AF, o raciocínio seria o oposto – para servir AF acima da média, seria necessário atrasar BE, e este processo deveria ser também controlado. Numa situação de empréstimo directo a situação é simples de entender – há pacotes que são servidos em detrimento de outros, pelo que a ocupação global do sistema é a mesma. Para que o tráfego BE não sofra perdas adicionais neste processo de empréstimo, basta que disponha dos *buffers* que seriam usados pelo tráfego AF caso a troca não tivesse lugar.

Uma conclusão desta discussão é que a decisão mais crítica corresponde ao processo de compensação do avanço de AF com o atraso (ou a diminuição do avanço) de BE, o que pode ser feito através de um limiar de decisão.

Inicialmente considerou-se que seria indiferente definir o limiar de decisão quer na perspectiva do atraso de BE quer na do avanço de AF. No entanto da análise anterior concluiu-se que pode ocorrer uma situação de avanço simultâneo, pelo que o mecanismo de troca implica uma redução do avanço de BE (sem que isso implique necessariamente um atraso).

Sendo assim optou-se por definir o limiar de decisão relativamente ao avanço de AF, comparando-se o valor do avanço $Q_{vAF} - Q_{AF}$ com um limiar b_1 , e definindo comportamentos diferentes do algoritmo conforme o resultado (positivo ou negativo) da comparação. Esta opção tem uma justificação adicional, pois deste modo é possível (através de b_1) controlar o atraso dos fluxos AF (tomando como referência o atraso máximo $\frac{b_{AF}}{R_{AF}}$ associado a um serviço realizado com o débito nominal R_{AF} e considerando que o tráfego é policiado por um *token bucket* (r_{AF} , b_{AF}), e que no caso mais desfavorável r_{AF} pode igualar R_{AF}).

Como foi referido a memória de avanço de BE é de curto prazo – isto significa que limitamos o tempo durante o qual BE devolve recursos até que o avanço acumulado seja anulado. Deste ponto de vista podemos começar por admitir que se BE estiver avançado poderá ceder recursos a AF (qualquer que seja o estado deste) pelo menos até que o avanço de BE se anule (se tal vier a acontecer). A justificação reside no facto de podermos controlar a janela temporal durante a qual contabilizamos o avanço (não penalizando BE por recursos usados para lá dessa janela no passado). Por outro lado assumimos que AF deve ser escalonado pelo menos até que o avanço atinja o limiar b_1 , como forma adicional de controlar o respectivo atraso através deste parâmetro.

Então o critério adoptado é o seguinte:

- AF é escalonado se estiver atrasado ou se BE estiver avançado (o que inclui o caso de ambos estarem avançados, conforme análise anterior);
- no caso de BE estar atrasado e AF avançado, a decisão é tomada com base no limiar b_1 , isto é, BE é escalonado apenas se o avanço de AF for superior a b_1 ($Q_{vAF} - Q_{AF} > b_1$); caso contrário é escalonado AF.

	BE Avanço	BE Atraso	
		$Q_{vAF} - Q_{AF} > b_1$	$Q_{vAF} - Q_{AF} \leq b_1$
AF Avanço	AF	BE	AF
AF Atraso	AF	AF	

Tabela 5.1: Algoritmo proposto

Se for possível convergir para a situação em que $Q_{vAF} - Q_{AF} = b_1$, então pode afirmar-se que se consegue reduzir o atraso dos pacotes AF de $\frac{b_1}{R_{AF}}$ em relação ao valor alvo de $\frac{b_{AF}}{R_{AF}}$. Manter constante $Q_{vAF} - Q_{AF} = b_1$ significa que, após atingir o estado de avanço correspondente, o tráfego AF passa a ser servido com um débito igual a R_{AF} , pelo que, do seu ponto de vista, o mecanismo de troca com BE termina.

É fácil então de concluir que, se fixarmos $b_1 = 0$ estamos numa situação em que estritamente se garante ao tráfego AF um serviço com o débito de referência. Em contrapartida se fixarmos $b_1 > b_{AF}$, ao tráfego AF é dada prioridade sobre BE, uma vez que teoricamente o avanço de AF (medido por $Q_{vAF} - Q_{AF}$) não excede b_{AF} .

Poder-se-ia aumentar a complexidade do algoritmo (tendo em atenção a interferência de tráfego EF, que introduz perturbações na relação entre o avanço de AF e o atraso de BE), introduzindo um limiar nos atrasos de BE e considerando as possíveis combinações. Essa complexidade adicional (dois parâmetros e respectivas afinações) não parece justificar-se. Poderia ainda considerar-se o dimensionamento de X , usando um valor $0 < X < b_{AF}$.

Por outro lado pode ainda considerar-se uma alternativa ao caso em que BE está avançado – poderia optar-se por escalonar BE no caso de o avanço de AF ser superior ao limiar b_1 . O algoritmo seria simplificado para:

- Se AF estiver avançado acima do limiar b_1 , escalonar BE;
- Caso contrário, escalonar AF.

AF Avanço	(Acima de b_1)	BE
	(Abaixo de b_1)	AF
AF Atraso		AF

Tabela 5.2: Simplificação do algoritmo proposto

A conclusão óbvia neste caso é que não precisamos de contabilizar nem o avanço nem o atraso de BE, dispensando assim a respectiva fila virtual. Em relação ao anterior, este algoritmo favorece BE, pois não tem em conta o respectivo grau de avanço como elemento de decisão em favor de AF.

Quanto à fila virtual de AF foi apenas utilizada para efeitos de simulação. A informação que é obtida a partir da fila virtual pode ser obtida a partir do *token bucket* e do nível de ocupação dos *buffers* AF – o avanço mede-se por $b_{AF} - (z + Q_{AF}) = (b_{AF} - z) - Q_{AF}$, em que z é o número de *tokens* disponíveis no *bucket*, ou seja $b_{AF} - z = Q_{vAF}$ (isto é, o número de *tokens* capturados pelos pacotes recebidos e que na fila virtual são servidos com débito R_{AF}).

6. SIMULAÇÃO E AVALIAÇÃO DO DESEMPENHO

O processo de simulação foi um trabalho que foi evoluindo com a experiência e com o conhecimento adquirido. Na prática, a fase de simulação foi dividida em duas partes; **a primeira parte**, onde o objectivo era perceber a ferramenta de simulação, os modelos já desenvolvidos e as limitações associadas a esses modelos e **a segunda parte**, a simulação do algoritmo proposto.

Sendo assim será apresentada de uma forma não tão detalhada a primeira parte como um processo de aprendizagem e depois os resultados e respectiva análise na segunda parte.

6.1. ANÁLISE E VALIDAÇÃO

Os objectivos desta primeira fase eram estudar o simulador NS e o efeito da implementação da QoS num determinado cenário, utilizando a arquitectura DiffServ, para além de se estudar os escalonadores PQ e WFQ. Neste cenário, foram apenas analisados dois tipos de tráfego EF e BE, de forma a poder perceber de uma forma mais simples quais os “parâmetros em jogo”.

O cenário implementado foi o apresentado na figura seguinte.

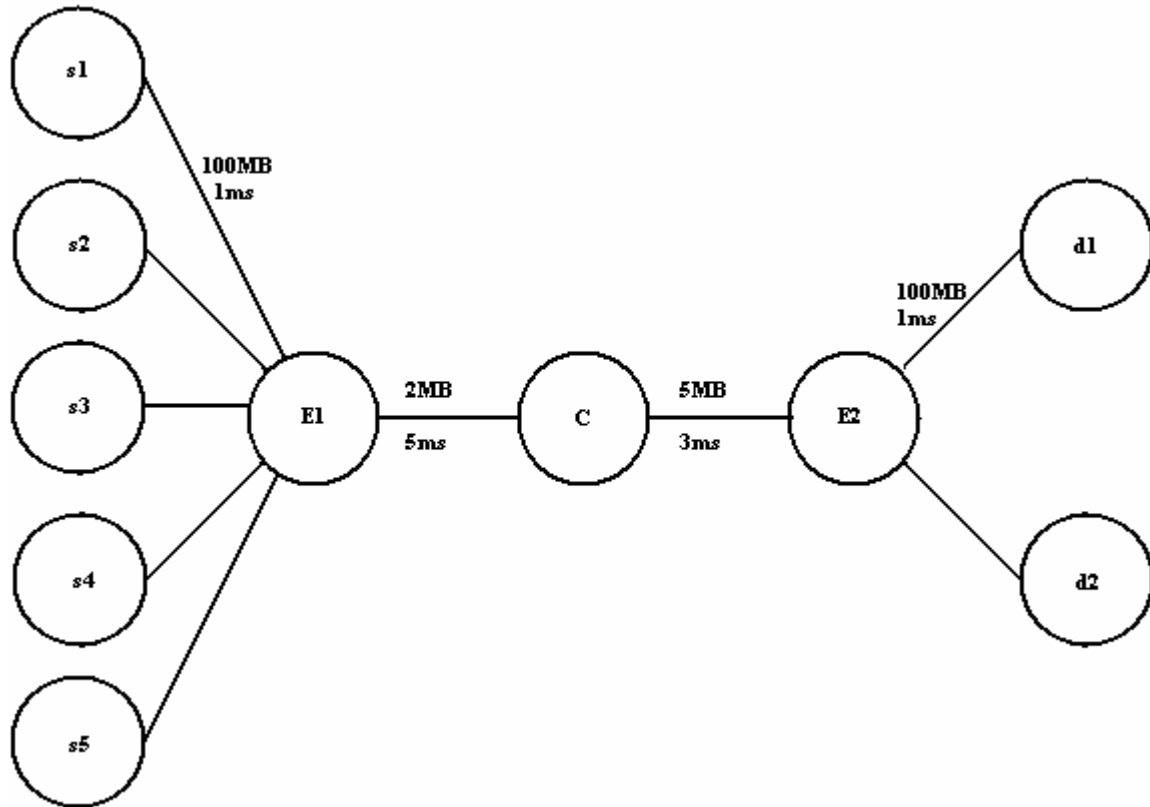


Figura 6.1: Modelo inicial apenas com dois tipos de tráfego EF e BE

As especificações deste cenário são as seguintes:

- 5 Nós clientes (s1, s2, s3, s4 e s5)
- 2 Nós destinos (d1 e d2)
- 2 Nós de fronteira (E1 e E2)
- 1 Nó interior (C)
- As ligações Fontes/NóFronteira e a ligação NóFronteira/Destinos são ligações unidireccionais de capacidade 100Mbit/s com 1ms de atraso de propagação e no caso de descarte usam *DropTail*.
- A ligação NóFronteira/NóInterior é uma ligação bidireccional e tem capacidade de 2Mbit/s e um atraso de propagação de 5ms. A técnica de descarte é RED.
- A ligação NóInterior/NóFronteira é uma ligação unidireccional de 5Mbit/s, com 3ms de atraso de propagação e *DropTail*.

- A capacidade da fila de espera para o tráfego EF no nó C é de 30 pacotes enquanto para o tráfego BE é de 50 pacotes.
- O *codepoint* utilizado para marcar o tráfego EF é 46 e para o tráfego BE é 0. No entanto, o tráfego EF pode também ser marcado com o *codepoint* 48 se for ultrapassado o limite do *token bucket*.
- Foi implementada 1 fonte de tráfego CBR que gera 300kbit/s de tráfego EF entre s5 e d0
- Foram implementadas 20 fontes de tráfego, também CBR, para gerar tráfego BE entre s0, s1, s2, s3 e s4 e o nó destino d1, sendo aleatório o nó fonte utilizado. Cada fonte gera 100kbit/s.
- O tamanho dos pacotes é de 64bytes.

O protocolo de transporte escolhido foi o UDP e todas as fontes foram iniciadas em simultâneo com o início da simulação. Cada simulação pretendeu simular o que acontece durante 15 segundos reais. O tráfego injectado na rede foi 300kbit/s de EF e 2Mbit/s de BE o que originou um congestionamento na ligação E₁C (2Mbit/s).

Todos os pacotes que entraram na rede foram classificados e marcados por tipo de tráfego (DiffServ), com base no seu endereço de destino (etiqueta MLPS). Para além disso, todo o tráfego do tipo EF foi policiado através de um *token bucket*, sendo marcado com o *codepoint* 46 todo o tráfego conforme. Os parâmetros deste *token bucket* são os seguintes: CIR de 300kbit/s e o CBS igual ao tamanho dos pacotes mais 1, ou seja, 65 bytes. No entanto, se o tráfego na rede fosse superior ao CIR, os pacotes eram marcados com o *codepoint* 48 e eram tratados de forma idêntica aos pacotes do tráfego BE. Para o tipo de tráfego BE existiu um “policiador” que apenas media os pacotes que atravessavam a rede, de nome *Dumb*.

Todas estas configurações foram implementadas no ficheiro *tcl* e podem ser vistas no [Anexo 1](#).

Usando o cenário anteriormente apresentado, foram realizados vários testes alterando apenas alguns parâmetros, tais como:

- tempo da simulação
- as quantidades de tráfego gerado,
- os valores para os limites das filas de espera,
- o comprimento dos pacotes,
- a sincronização das fontes,
- o tipo de geradores de tráfego (para além de CBR foi também gerado tráfego exponencial).

no entanto, os resultados destes testes não serão apresentados pois os resultados obtidos estavam de acordo com os valores teóricos e porque não influenciavam os resultados para este trabalho.

Os resultados observados, com base nos pressupostos apresentados inicialmente, são os seguintes:

Algoritmo	Pesos EF BE	Codepoint	TotPackets	TxPackets %	Ldrops %	Edrops %	OWD EF ms
PQ		0	58200	85.06	14.94	0.00	10.63
		46	8730	100.00	0.00	0.00	
		Total	66930	87.01	12.99	0.00	
WFQ	1 10	0	58200	90.97	9.03	0.00	95.02
		46	8730	60.94	39.06	0.00	
		Total	66930	87.05	12.95	0.00	
WFQ	1 9	0	58200	90.06	9.94	0.00	86.92
		46	8730	67.00	33.00	0.00	
		Total	66930	87.01	12.99	0.00	
WFQ	3 17	0	58200	85.06	14.94	0.00	11.82
		46	8730	100.00	0.00	0.00	
		Total	66930	87.01	12.99	0.00	
WFQ	1 5	0	58200	85.06	14.94	0.00	11.65
		46	8730	100.00	0.00	0.00	
		Total	66930	87.01	12.99	0.00	
WFQ	10 1	0	58200	85.06	14.94	0.00	10.63
		46	8730	100.00	0.00	0.00	
		Total	66930	87.01	12.99	0.00	
WFQ	1 1	0	58200	85.06	14.94	0.00	10.63
		46	8730	100.00	0.00	0.00	
		Total	66930	87.01	12.99	0.00	

Tabela 6.1: Resultados da simulação parte 1

TotPackets → número total de pacotes que foram recebidos pela *queue* no nó C, ligação E₁C.

TxPackets → número total de pacotes enviados pelo nó C, ligação E₁C, em percentagem

Ldrops → os pacotes que são descartados devido ao excesso de tráfego no *link*, em %

Edrops → *RED early dropping*, em %

Nota: O não aparecimento do *codepoint* 48 nesta tabela indica que o tráfego EF que está a ser gerado não ultrapassa o contratado entre o cliente e o ISP não atingindo o CIR.

Uma das primeiras dificuldades encontradas na utilização do escalonador WFQ foi definir o melhor “peso” para as classes de tráfego por forma a dar garantias ao tráfego EF. O valor para o peso pode ser calculado teoricamente mas trata-se de um valor estático; como nos casos reais o tráfego é normalmente dinâmico, os valores do peso também deveriam ser dinâmicos para se poder adaptar os valores dos pesos ao tráfego instantâneo. A tabela anterior demonstra que dependendo do valor dos pesos, os atrasos e a quantidade de tráfego de cada tipo de tráfego *versus* perdas variam. O valor teórico para os pesos, nestas condições de tráfego, são $p_{EF} = 3$ e $p_{BE} = 17$. Isto porque

$$p_{EF} = \frac{300000}{2000000} = \frac{3}{20}$$
$$p_{BE} = \frac{1700000}{2000000} = \frac{17}{20} \text{ (é igual à largura de banda restante)}$$

Ou seja, com um escalonador WFQ, os pesos 3|17 significa que em cada 20 pacotes serão transmitidos 3 pacotes de EF e 17 de BE, isto porque temos 300kbit/s de EF para transmitir num canal de 2Mbit/s, sobrando os 1700kbit/s para BE. Para valores como 1|9 e 1|10 o tráfego BE é favorecido mas prejudica e cria perdas no tráfego EF, situação de todo a evitar. Por outro lado, 1|5, 1|1 e 10|1 estamos na situação em que não existe perdas no tráfego EF, a escolha dos pesos apenas influencia o valor máximo do *one way delay* (OWD), quanto mais os pesos beneficiam o tráfego EF menor é o OWD (tendendo para o valor obtido com PQ). Outra conclusão importante é a comparação entre os dois algoritmos presentes na Tabela 6.1, dependendo das configurações dos pesos no WFQ é possível atingir o mesmo valor relativamente às perdas de tráfego, usando PQ ou WFQ. No entanto, relativamente ao valor do *One Way Delay* é facilmente perceptível que o melhor algoritmo é o PQ, pois o WFQ apenas se aproxima dos menores valores de WFQ quando os pesos beneficiam o tráfego EF.

Estando agora já familiarizado com os termos e com a ferramenta NS foi possível chegar à primeira iteração do algoritmo proposto no [Capítulo 5](#). Ao longo do tempo e com base nos resultados intermédios o algoritmo foi evoluindo, percebendo-se e corrigindo-se erros que foram aparecendo, erros esses muitas vezes descobertos por comparação dos resultados obtidos com a teoria que lhes estava associada.

Com base no cenário anteriormente apresentado, ver Figura 6.1, foram realizados dois tipos de testes independentemente do escalonador utilizado. O primeiro teste, numa situação perto da saturação da ligação E_1C , (foi injectado tráfego BE por forma a não saturar a ligação em questão) e o segundo teste já no estado de saturação. Ou seja, no primeiro teste foi injectado tráfego na rede perto de 2Mbit/s e no segundo teste foi injectado tráfego acima dos 2Mbit/s, variando apenas o volume de tráfego BE.

No entanto, foram introduzidas alterações ao cenário inicial. A maior diferença verificada foi a inserção de mais uma classe de tráfego, AF, suprimida no primeiro caso com o intuito de simplificar e tornar os resultados mais claros, houve também alteração nas fontes de tráfego por forma a simular variação na distribuição de tráfego.

Sendo assim, este novo cenário de simulação foi o seguinte:

- 9 Nós clientes (s1, s2, s3, s4, s5, s6, s7, s8 e s9)
- 3 Nós destinos (d1, d2 e d3)
- 2 Nós de fronteira (E1 e E2)
- 1 Nó interior (C)
- As ligações Fontes/NóFronteira e NóFronteira/Destinos são ligações unidireccionais com capacidade 100Mbit/s com 1ms de atraso de propagação e no caso de descarte usam *DropTail*.
- A ligação NóFronteira/NóInterior é uma ligação bidireccional e tem capacidade 2Mbit/s e um atraso de propagação de 5ms. A técnica de descarte é RED.
- A ligação NóInterior/NóFronteira é uma ligação unidireccional de 5Mbit/s, com 3ms de atraso de propagação e *DropTail*.

- A capacidade da fila de espera para o tráfego EF no nó C, ligação E₁C é de 30 pacotes, para o tráfego AF é de 1000 pacotes e para o tráfego BE é de 2500 pacotes.
- O *codepoint* utilizado para marcar o tráfego EF foi o 46, para o tráfego AF foi utilizado o *codepoint* 10 e para o tráfego BE o 0.
- Geradores de tráfego EF: foram implementadas 4 fontes de tráfego CBR que geram um total de 400kbit/s e uma fonte também CBR mas com períodos *on-off* de 0,5s que gera em média 100kbit/s de tráfego EF entre s₁, s₂ e s₃ (aleatoriamente) e d₀.
- Para o tráfego AF foram implementadas 5 fontes, no entanto com uma distribuição de tráfego um pouco diferentes das anteriores de forma a poder criar alguma variação na distribuição de tráfego. O tráfego AF é gerado a partir dos nós s₄, s₅ e s₆ com o destino final em d₁ com uma distribuição semelhante à da figura seguinte. Cada fonte é constituída por duas mini-fontes.

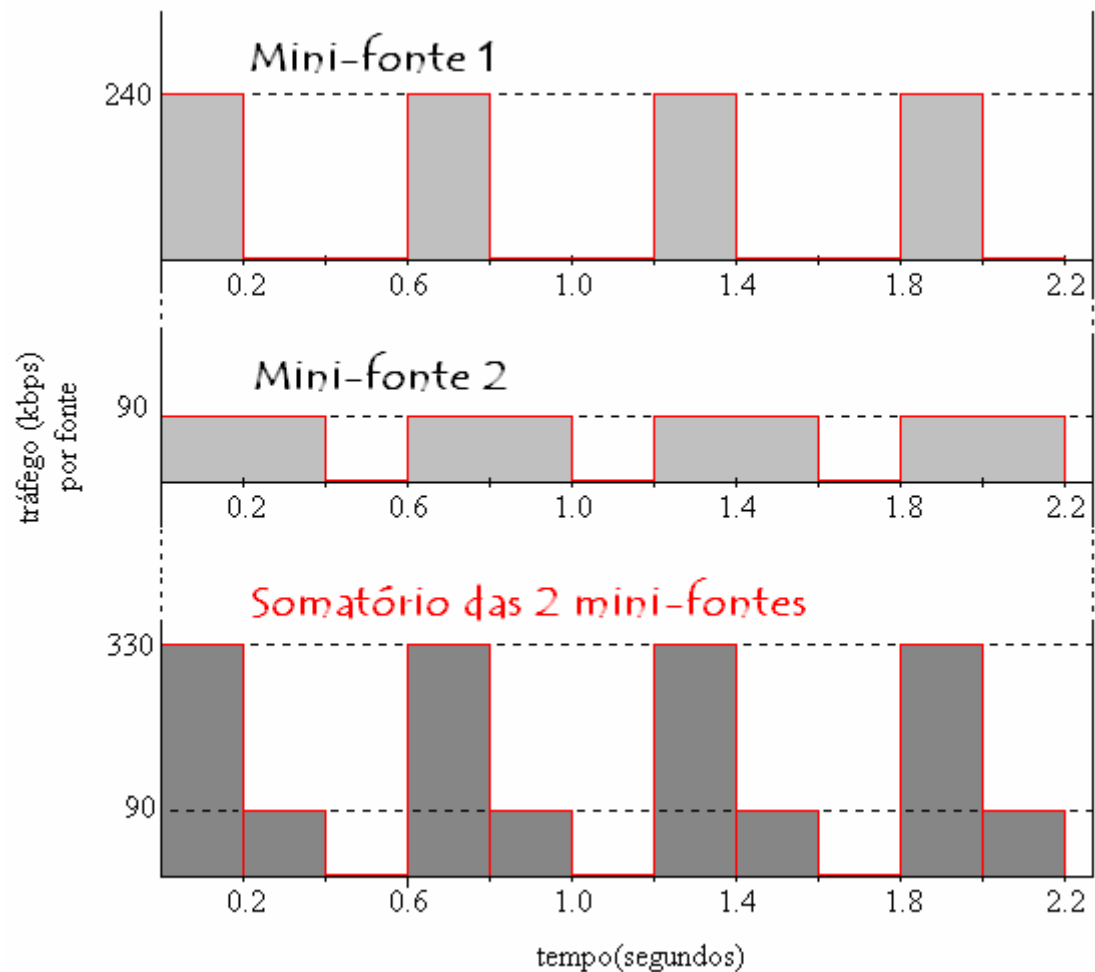


Figura 6.2: Geração de tráfego AF

- Foram implementadas 10 fontes de tráfego Exponencial para gerar tráfego BE entre s6, s7 e s8 e o nó destino d2, sendo aleatório o nó fonte utilizado. Cada fonte gera 120kbps na situação de saturação e 80kbps na situação próxima da saturação, ou seja, um total de 1200kbps ou de 800kbps, respectivamente.
- O tamanho dos pacotes é de 64 bytes.

A justificação/motivação para se ter escolhido os valores apresentados anteriormente para a geração de tráfego é apresentada de seguida:

- **tráfego EF** débito médio igual a 500kbit/s, com valores instantâneos de 400kbits e 600kbit/s, alternando em períodos de 0.5 s (período de repetição igual a 1 s);

- **tráfego AF** débito médio igual a 700kbit/s, com padrão que se repete a intervalos de 0.6s, durante os quais se pretende simular um comportamento com elevado grau de *burstiness* por meio de três níveis de tráfego em intervalos de 0.2 s;

Deste modo a combinação de tráfego EF e AF permite criar situações em que existem máximos ou mínimos simultâneos ou em que o máximo de uma classe coincide com um mínimo da outra; naturalmente que o padrão conjunto se repete com um período de 3 s.

A geração de tráfego pode ser vista nos gráficos a seguir:

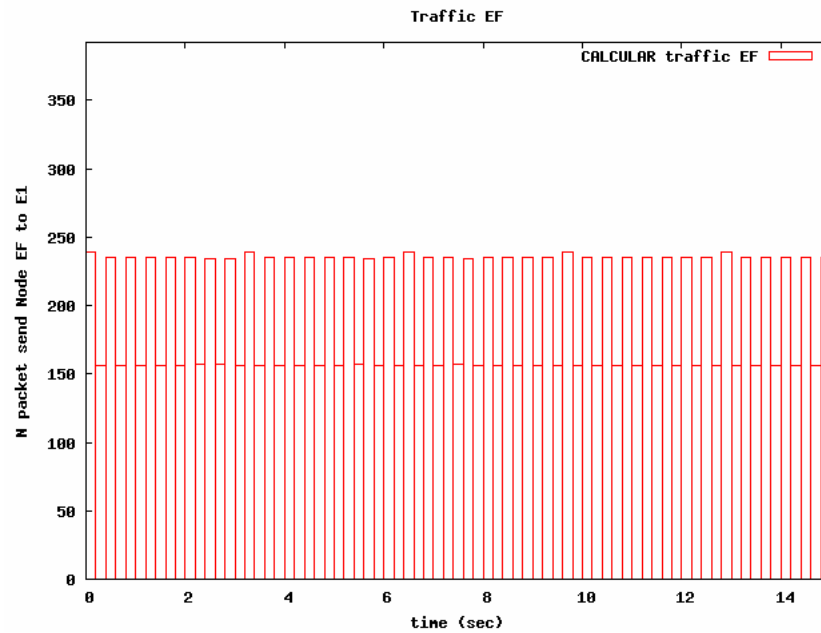


Figura 6.3: Tráfego EF gerado

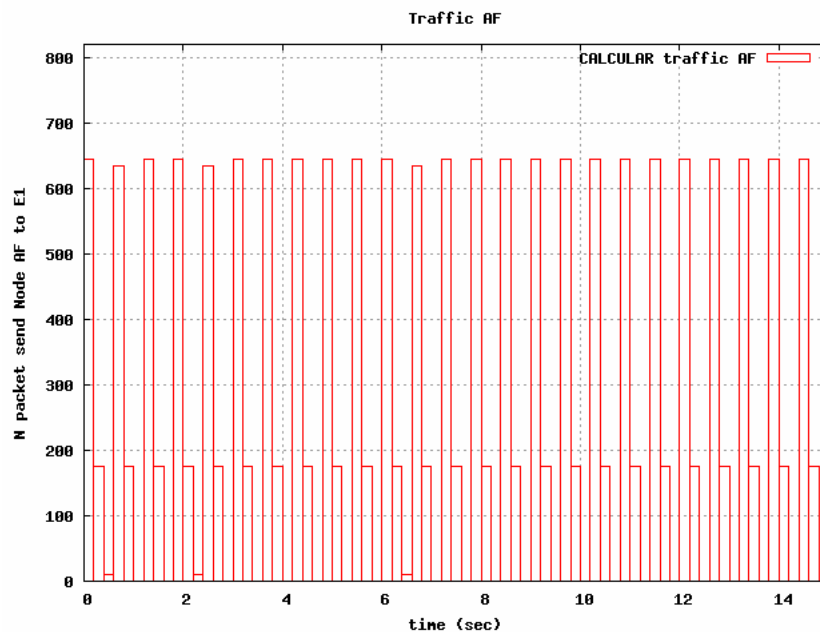


Figura 6.4: Tráfego AF gerado

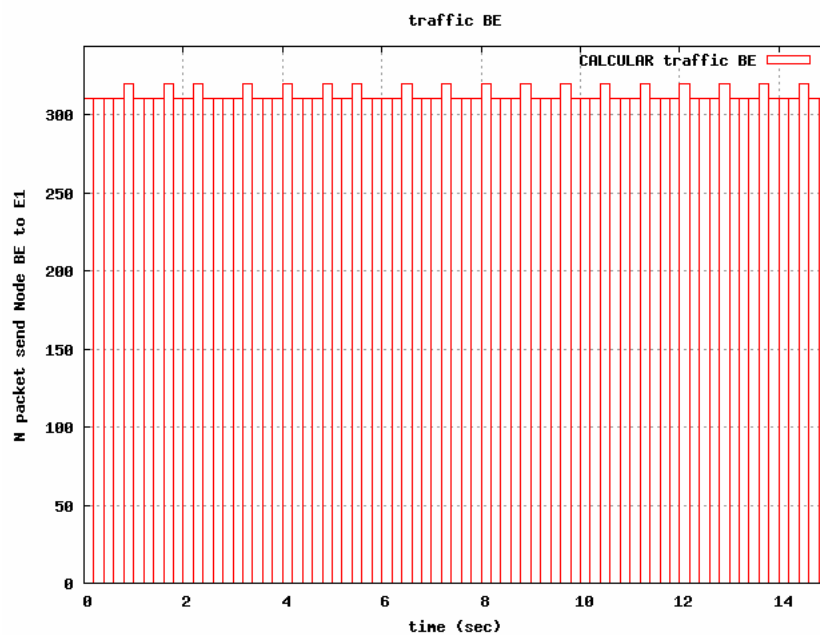


Figura 6.5: Tráfego BE gerado

O mecanismo de classificação, marcação e policiamento para o tráfego EF é o mesmo que no primeiro caso, variando apenas os valores do CIR e do CBS que passaram a ser, 500kbit/s e 3125bytes. No entanto, quer para o tráfego AF quer para o tráfego BE foram criados dois novos mecanismos baseados no *token bucket*, designados por

“*Token Bucket Special*” para o tráfego AF e “*Dumb Special*” para o tráfego BE (ver as alterações ao módulo DiffServ do NS no [Anexo 2](#)).

O *TokenBucketSpecial* possui os seguintes parâmetros, um CIR de 700kbit/s e um CBS de 4375 bytes. Neste novo mecanismo foi implementado uma variável que monitoriza o tamanho da fila virtual, incrementada e decrementada aquando da chegada ou da saída de um pacote, respectivamente. O valor desta variável, juntamente com o valor real do tamanho da fila real, irá permitir perceber se o tráfego AF se encontra atrasado ou adiantado em relação a um escalonamento ideal feito com base no débito médio contratado, tal como explicado anteriormente ([Capítulo 5](#)).

O *DumbSpecial* ao contrário do mecanismo *Dumb*, que apenas guardava valores, é um mecanismo que deriva do método *Token Bucket* e como tal possui parâmetros de configuração como CIR e CBS, tendo os valores 800kbit/s e 5000bytes, respectivamente. Este mecanismo foi desenvolvido para saber a diferença entre o tráfego BE que deveria ter sido servido e o que na realidade foi, tal como para AF. Da mesma forma que para o tráfego AF, foi implementada uma variável que monitoriza o tamanho da fila virtual. No entanto, existe uma ligeira diferença relativamente a este mecanismo e o apresentado anteriormente; a variável utilizado neste mecanismo possui um limite máximo. Sempre que chega um pacote e o tamanho da fila real é menor que o seu tamanho máximo é incrementado o valor da variável, somando ao valor da variável os bytes de um pacote. Este valor poderá ser maior ou menor consoante o histórico que se pretende implementar no sistema. Quanto maior for o valor de limite, mais o sistema se recordará do passado e tomará decisões com base nesse histórico. Tal como foi dito no [Capítulo 5](#) o sistema deverá ter memória de curta duração. No entanto, quando um pacote é servido o valor da variável é decrementando de n bytes, sendo n o número de bytes de um pacote. Ou seja,

$$\begin{aligned}
& Se(FilaRealBE < tamanho_max_FilaRealBE) \\
& \quad FilaVirtualBE = FilaVirtualBE + tamanho_pacote \\
& \quad Se(FilaVirtualBE > B \times tamanho_pacote) \\
& \quad \quad FilaVirtualBE = B \times tamanho_pacote
\end{aligned}$$

$$\begin{aligned}
& Se(FilaVirtualBE < 0) \\
& \quad FilaVirtualBE = 0
\end{aligned}$$

onde B é igual ao limite máximo da fila de espera virtual. Estas configurações podem ser vistas no [Anexo 2](#).

Tendo implementado os mecanismos de controlo das filas reais e das filas virtuais quer para AF quer para BE, faltava apenas implementar a “inteligência” do algoritmo, ou seja, o mecanismo de escalonamento. Este escalonador toma decisões com base no tipo de tráfego e nos valores das filas reais e virtuais. Tendo estes mecanismos implementados estávamos perante o novo algoritmo proposto, o *Scheduler Fair, SF*.

Sendo assim, se o pacote que chega ao SF for um pacote do tipo EF é escalonado de imediato, no caso de ser AF ou BE então temos que analisar alguns parâmetros e com base neles tomar a decisão de qual o tipo de tráfego a escalonar. Ou seja, o código implementado foi baseado na Tabela 5.1, em pseudo-código, temos que

$$\begin{aligned}
& Se(FilaVirtualBE < FilaRealBE \quad e \quad (FilaVirtualAF - FilaRealAF > b) \quad ou \\
& \quad (FilaRealBE > 0 \quad e \quad FilaRealAF == 0)) \\
& \quad Escalona \quad pacotes \quad BE \\
& \quad caso \quad contrário \\
& \quad \quad Escalona \quad pacotes \quad AF
\end{aligned}$$

Outras versões deste algoritmo foram testadas, com o objectivo de melhorar os resultados; no entanto esta versão apresentada parece ser a que melhor serve os objectivos iniciais. De salientar que foi introduzido um novo parâmetro b que pode variar entre 0 e um valor máximo. Quando se aproxima do valor máximo estamos perante um escalonador semelhante a PQ; no caso oposto são usados os respectivos débitos de referência; não sendo “igual” a WFQ produz resultados semelhantes no

caso de os pesos de WFQ serem definidos com base nos débitos de referência (ver [Anexo 3](#)).

Então quais as situações que temos em jogo?

As situações que foram analisadas, tendo sempre na ideia a melhoria do serviço por parte do ISP, foram com a finalidade de controlar o atraso de AF e reduzir as perdas de BE de forma integrada e maximizar a utilização da largura de banda:

- PQ vs WFQ
- PQ vs SF
- SF com diferentes valores de b

Os resultados gráficos de todas as situações anteriores podem ser observados no [Anexo 4](#).

Para o primeiro caso, e com um escalonador do tipo WFQ, obtivemos os seguintes resultados:

Pesos			Saturação						
			Atraso máximo (milissegundos)			Perdas de pacotes			
			EF	AF	BE	Total (pacotes)	EF (%)	AF (%)	BE (%)
5	7	8	41	279	1585	9257	24.9	0	3.08
100	7	8	12	292	1611	9264	0	0	26.20
100	4	1	12	95	1691	9361	0	0	26.74
100	5	1	12	84	1694	9361	0	0	26.74
100	10	1	12	64	1694	9361	0	0	26.74

Tabela 6.2: Tabela comparativa no caso de saturação usando WFQ

			Próximo da Saturação						
Pesos			Atraso máximo (milissegundos)			Perdas de pacotes			
EF	AF	BE	EF	AF	BE	Total (pacotes)	EF (%)	AF (%)	BE (%)
5	7	8	42	272	17	9257	2.94	0	0
100	7	8	12	291	31	27	0	0	0
100	4	1	12	95	234	27	0	0	0
100	5	1	12	85	234	27	0	0	0
100	10	1	12	64	246	27	0	0	0

Tabela 6.3: Tabela comparativa no caso de próximo da saturação usando WFQ

Comparando estas duas tabelas verifica-se que os atrasos para o tráfego EF e para o tráfego AF, em ambas as situações, são similares o que significa que os atrasos não são afectados no caso de estarmos na situação de saturação ou não. Relativamente ao tráfego BE no caso de proximidade da saturação os atrasos são menores pois existe menos tráfego na rede e nas filas de espera e como todo o tráfego gerado é servido, não existem perdas.

O mesmo estudo foi feito mas utilizando o algoritmo de escalonamento PQ.

Teste/Caso	Atraso máximo (milissegundos)			Perdas de pacotes	
	EF	AF	BE	Total (Pacotes)	BE (%)
Saturação	12	44	1694	9361	26.74
Próximo Saturação	12	44	260	27	0,00

Tabela 6.4: Tabela comparativa em ambos os cenários usando PQ

Neste cenário, verifica-se que apenas existe diferença nos valores dos atrasos e das perdas relativamente ao tráfego BE, pois do caso de saturação para o caso próximo da saturação o tráfego na rede diminui.

Comparando os algoritmos de escalonamento PQ e WFQ verifica-se que o PQ introduz menores atrasos mantendo os valores do pacotes perdidos. No entanto, em ambos os cenários o problema mantém-se, o tráfego AF é sempre beneficiado em relação a BE, mesmo quando existem picos de tráfego, o que não é aceitável face ao tipo de contrato AF. Deve haver um compromisso entre os atrasos de tráfego AF e as perdas de tráfego BE. Foi com base nestes resultados que surgiu a hipótese de se desenvolver um novo escalonador baseado no controlo de débitos e atrasos.

Implementando o novo algoritmo SF os resultados obtidos, para $b = 500$, foram os seguintes:

Teste/Caso	Atraso máximo (milissegundos)			Perdas de pacotes	
	EF	AF	BE	Total (Pacotes)	BE (%)
Saturação	12	44	1691	9361	26.74
Próximo Saturação	12	44	260	27	0.00

Tabela 6.5: Tabela comparativa em ambos os cenários usando SF

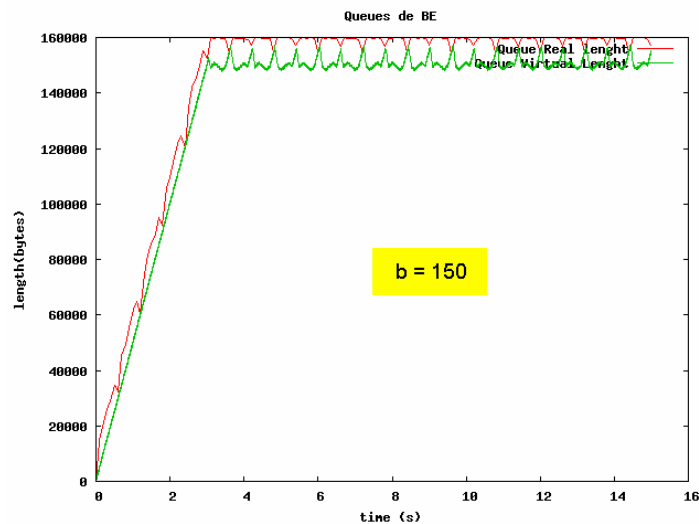
Com $b = 500$, estamos numa situação em que o SF se comporta de forma semelhante ao PQ. Ou seja, serve-se o tráfego com base no tipo de tráfego, primeiro EF depois AF e por fim BE.

Por fim, podemos analisar em que sentido a escolha do valor de b influencia (piora ou melhora) os parâmetros em causa. A tabela seguinte mostra os valores encontrados para diferentes valores de b .

Teste/Caso	B	Atraso máximo (milissegundos)			Perdas de pacotes	
		EF	AF	BE	Total (Pacotes)	BE (%)
Saturação	150	12	177	1677	9326	26.64
Próximo Saturação	150	12	177	134	27	0.00
Saturação	250	12	102	1691	9355	26.72
Próximo Saturação	250	12	105	201	27	0.00
Saturação	500	12	44	1691	9361	26.74
Próximo Saturação	500	12	44	260	27	0.00

Tabela 6.6: Tabela comparativa em ambos os casos usando SF com diferentes valores de b

Para além dos valores apresentados na tabela anterior, podemos observar também os seguintes gráficos, que nos mostram os valores das filas reais e das filas virtuais para o tipo de tráfego BE, no caso de saturação,



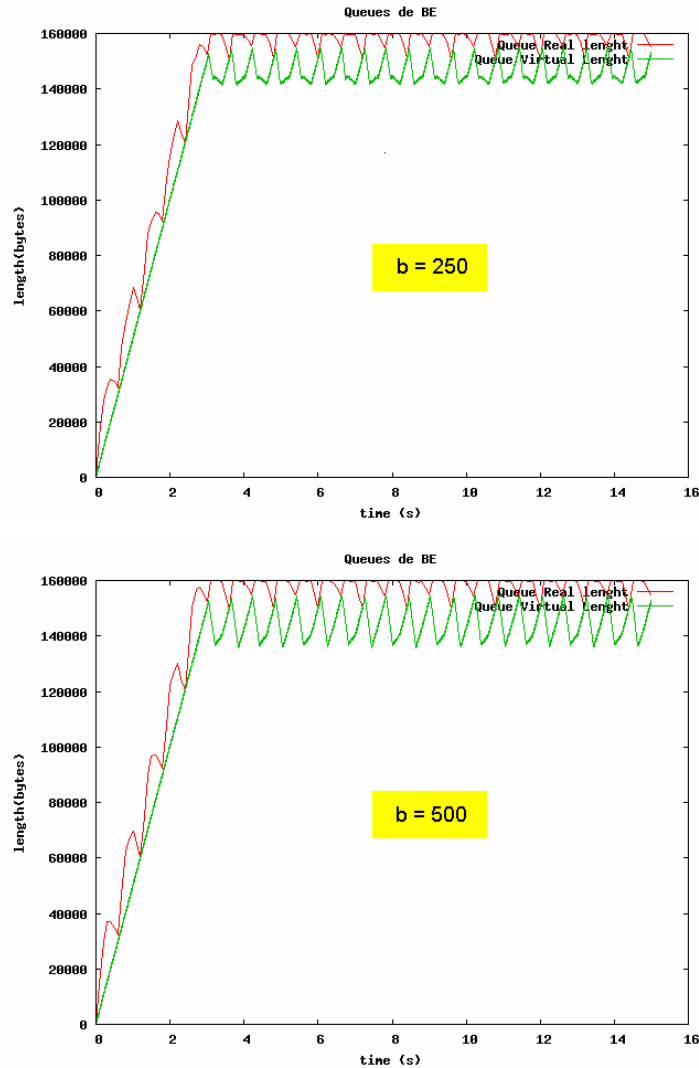


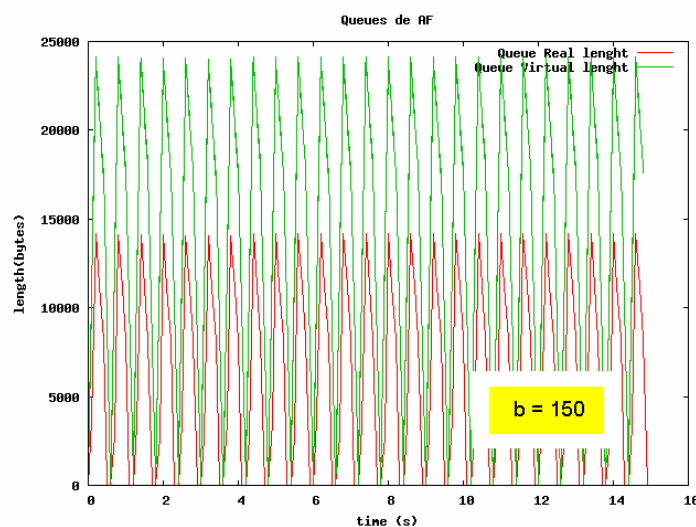
Figura 6.6: Alterações nos valores das filas reais e virtuais do tráfego BE com diferentes valores de b num caso de saturação

Graficamente, quanto mais pequeno for o valor de b , mais perto se encontram os valores da fila virtual relativamente aos valores da fila real, o que corresponde a um controlo mais apertado do débito médio do tráfego BE. Tal é visível antes de se atingir a saturação dos *buffers* BE com o afastamento e posterior convergência da ocupação da fila real para a da fila virtual (o afastamento máximo aumenta com o valor de b). Algo de semelhante ocorre após se atingir o limite de ocupação do *buffer*, mas a partir de então o comportamento é determinado pelo descarte de pacotes BE (devido ao facto de o débito de chegada exceder o débito de serviço). É importante lembrar que um dos

pontos principais do algoritmo proposto é tomar decisões por forma a ter a fila real a convergir para a fila virtual com base num parâmetro.

Para além disso, pode verificar-se que a saturação do *buffer* BE é atingida num instante entre os 2 e os 4 segundos. Até esse ponto verifica-se que existe uma diferença maior entre as duas filas (reais e virtuais) quando b é maior. O mesmo se pode observar depois de ser atingida a saturação, no entanto os valores da fila virtual deixam de crescer e ficam em torno do limite máximo. Outra conclusão que se pode tirar é que o débito instantâneo de saída se aproxima tanto mais do débito médio quanto menor for o valor de b , o que significa que o escalonador reage mais rapidamente aos desvios do tráfego submetido relativamente ao débito médio garantido.

Analisando agora as filas reais e virtuais do tipo de tráfego AF, podemos verificar que quanto maior o valor de b menor é o limite máximo atingido pela fila real, isto significa, que o escalonador se aproxima do escalonador PQ, sendo o tráfego AF “mais” favorecido em relação ao tráfego BE.



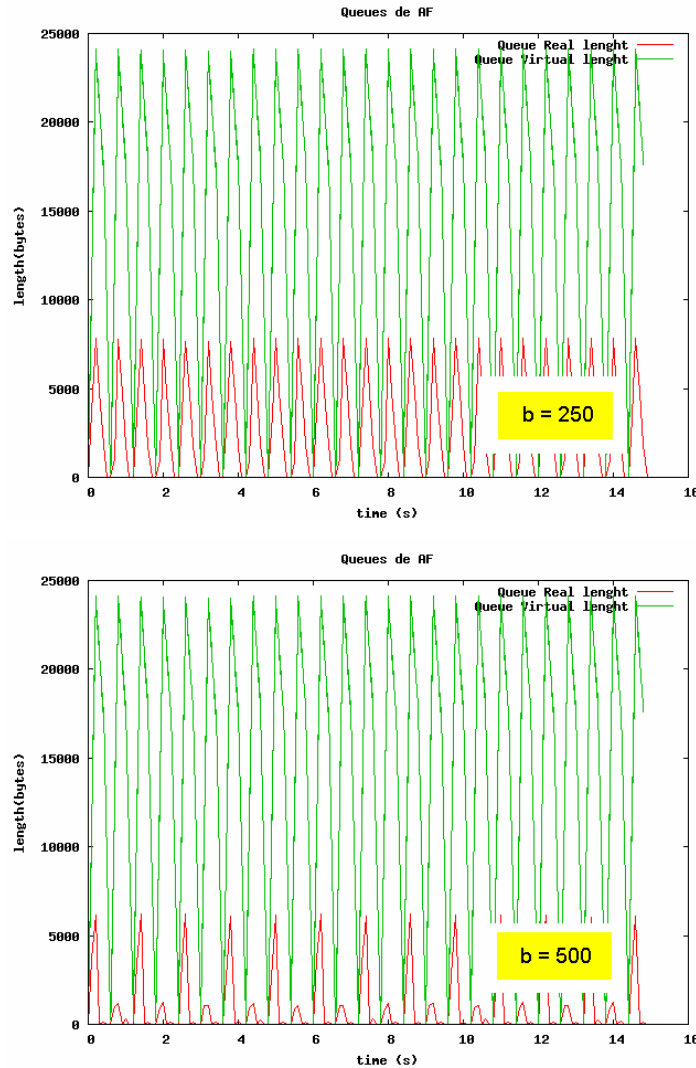


Figura 6.7: Alterações nos valores das filas reais e virtuais do tráfego AF com diferentes valores de b num caso de saturação

Para além das conclusões já apresentadas, falta ainda reflectir sobre o desacoplamento entre a largura de banda e o atraso. Tal como se pode verificar, analisando a Tabela 6.7, verifica-se que alterando um único parâmetro, b , e mantendo o valor da largura de banda, conseguimos obter variações no valor do atraso do tráfego AF. Verifica-se então que é possível controlar separadamente (desacoplar) os valores da largura de banda e os valores do atraso, o que não é possível em alguns algoritmos de escalonamento. Baixando o valor de b , as perdas de pacotes BE diminuem, tal como o atraso deste tráfego, comprometendo moderadamente (e de forma controlada) o tráfego do tipo AF. Este facto não é muito grave, pois o valor médio do débito

contratado é garantido e este tipo de tráfego não requer limites rígidos no que respeita a atrasos.

Teste/Caso	b	Atraso máximo (milissegundos)			Perdas de pacotes	
		EF	AF	BE	Total (Pacotes)	BE (%)
Saturação	150	12	177	1677	9326	26.64
Saturação	250	12	102	1691	9355	26.72
Saturação	500	12	44	1691	9361	26.74

Tabela 6.7: Tabela comparativa usando SF com diferentes valores de b

Na situação de saturação, a ocorrência de perdas de pacotes BE é inevitável. As vantagens do algoritmo proposto são evidentes no caso em que o tráfego BE se aproxima da saturação, observando-se claramente o *trade-off* entre AF e BE no que respeita aos atrasos. Reduzir os atrasos do tráfego BE significa reduzir a probabilidade de perdas, para o mesmo tamanho de *buffers* (ou para a mesma probabilidade de perdas poder utilizar *buffers* mais pequenos). Estas vantagens são importantes próximo da saturação ou para saturação de curta duração, naturalmente não resolvendo o problema de saturações prolongadas.

7. CONCLUSÕES

Este capítulo apresenta as conclusões gerais deste trabalho. Os objectivos definidos para esta dissertação foram na sua generalidade atingidos. Através da utilização do NS foi possível estudar vários aspectos relativos ao desempenho de redes de comunicação dotadas de mecanismos de suporte a Qualidade de Serviço.

Depois de um estudo das tecnologias existentes no mercado e de técnicas associadas a este tema, *Mecanismos de Qualidade de Serviço e de Engenharia de Tráfego em Redes DiffServ/MPLS*, foi possível identificar as vantagens e desvantagens de diferentes soluções. Concluiu-se, em particular, que os algoritmos de escalonamento estudados, concebidos para tratar de forma diferenciada classes de tráfego, não são os mais adequados no que se refere à possibilidade de controlar, de forma independente, os parâmetros de desempenho mais significativos.

Com base nesta premissa, idealizou-se com sucesso um algoritmo que permite fazer a distribuição do tráfego de diferentes classes, em diferentes situações de carga na rede, com diversos perfis de tráfego e de acordo com objectivos de desempenho específicos de cada classe. O algoritmo permite igualmente controlar de forma independente a largura de banda e o atraso.

Outra conclusão muito importante desta dissertação é relativa à utilização do NS. Trata-se de uma ferramenta que requer um período inicial considerável para compreensão das funcionalidades e mecanismos básicos. Possui um conjunto de módulos/*patches* que permitem estender o NS base a outros temas nesta área. Sendo assim, a utilização do *patch* DiffServ poupou muito trabalho e ajudou na implementação dos pressupostos iniciais deste trabalho (no entanto, a instalação deste *patch* extra não foi fácil devido a vários conflitos de versões não só do NS como dos compiladores necessários à compilação da ferramenta). Embora exista alguma informação e alguns foruns na *web* o apoio e a documentação são deficitários. Apesar das dificuldades apresentadas, a utilização desta ferramenta é essencial e uma mais valia. Com o decorrer do tempo é fácil perceber o porquê de ser das ferramentas mais utilizadas no “mundo” da simulação das redes. Ponderando as vantagens e desvantagens o saldo é claramente positivo.

7.1. TRABALHO FUTURO

O algoritmo idealizado e implementado poderá ser refinado, relaxando a prioridade estrita do tráfego EF em certas condições que deverão ser estudadas (por exemplo considerando prioridade estrita apenas se o débito for inferior a um determinado valor do *peak rate*), considerando mais que um PHB AF, e estudando outras condições de elegibilidade de escalonamento de tráfego AF e BE.

Também poderão ser realizados mais testes com outros tipos de tráfegos, em outros cenários mais complexos usando outro tipo de métricas.

8. REFERÊNCIAS

[Almes] Almes, G. Kalidindi, S. Zekauskas, M; “A One-way Delay Metric for IPPM”, Advanced Network & Services, Setembro 1999

<ftp://ftp.rfc-editor.org/in-notes/rfc2679.txt>

[Andreozzi] Andreozzi, Sergio; Diffserv

<http://www.cnaf.infn.it/~andreozzi/research/network/index.html>

[Awduche] Awduche, D.; et al. “Requirements for Traffic Engineering over MPLS”. IETF RFC 2702. Setembro 1999.

<http://www.ietf.org/rfc/rfc2702.txt>

[Awduche&Berger] Awduche, D; Berger, L; et al. “RSVP-TE: Extensions to RSVP for LSP Tunnels”. IETF RFC 3209. Dezembro 2001.

<http://www.ietf.org/rfc/rfc3209.txt>

[Baldi] Baldi, Mario; Risso, Fulvio. “Efficiency of Packet Voice with Deterministic Delay, in IEEE Communications Magazine, vol. 28 nº 5, página 170-177”, Maio 2000.

[Bennet] Bennet, J.R., Zhang, H. “WF2Q: Worst-case Fair Weighted Fair Queueing”. INFOCOM 96. San Francisco, California, Março 1996.

[Bernet] Bernet, Y. “Networking Quality of Service and Windows® Operating Systems”, New Riders, 2001.

[Best Effort] Best Effort,
http://en.wikipedia.org/wiki/Best_effort_delivery

[Black] Blake, S.; Black, D.; Carlson, M.; Davies, E.; Zhang, Z.; Weiss, W; “An Architecture for Differentiated Services”. IETF RFC 2475. 1998
<http://www.ietf.org/rfc/rfc2475.txt>

[Braden] Braden, R. “Resource Reservation Protocol (RSVP)”. IETF RFC 2205. Setembro 1997.
<http://www.ietf.org/rfc/rfc2205.txt>

[Callon] Callon, Ross; Rosen, Eric C.. Viswanathan “Multiprotocol Label Switching Architecture”. Internet Draft. Agosto de 1999.
<http://www.ietf.org/internetdrafts/draft-ietf-mpls-arch-06.txt>

[Carpenter&Nichols] Carpenter, B. Nichols, K. “Definition of Differentiated Services Per Domain Behaviors and Rules for their Specification”. IETF RFC 3086. Abril 2001
<http://www.ietf.org/rfc/rfc3086.txt>

[Cisco] Cisco System Inc. “Internetworking Technology Overview”, Chapter 46.1999
<http://www.cisco.com/>

[Chung] Chung, J.; Claypool, M. “NS by Example”. Worcester Polytechnic Institute (WPI)
<http://ce.sharif.edu/~mohebbi/ns/nsbyExamples.pdf>

[Cunha] Cunha, Filipe. “Simulação de redes de comunicações em NS”. Universidade de Aveiro. 2006

[Davie] Davie, B. et al. “An Expedited Forwarding PHB”. Draft-ietf-diffserv-rfc2598bis-02.txt. Setembro 2001.

<http://www.ietf.org/internet-drafts/draft-ietf-diffserv-rfc2598bis-02.txt>

[Davie B] Davie, Bruce. Rekhter, Yakov. (2000). MPLS. Technology and Applications. Morgan Kaufmann Publishers. 2000.

[Ethan] Ethan, Robbins;Yong, Sim. “Algoritms study”. Laboratory of Networking and Information Systems (NISLAB).

<http://nislabs.bu.edu/sc546/sc441Spring2003/wfq/intro.htm>

[Extreme] “Leveraging MPLS to enhance network transport capabilities - In service provider and enterprise environments”. Extreme Networks. White Paper.

<http://apps.extremenetworks.com/apps/whitepaper?wpurl=/common/asp/frameHandler.asp?go=/libraries/whitepapers/technology/MPLSWhitePaper.pdf>

[Faucher] Faucher; F. et al. “MPLS Support of Differentiated Services”. IETF RFC 3270. Maio 2007.

<http://www.ietf.org/rfc/rfc3270.txt>

[Ferguson] Ferguson, P.; Huston, G.. “Quality of Service: Delivering QoS on the Internet and in Corporate Networks”. Wiley Computer Publishing.1999.

[Fineberg] Fineberg, Victoria. “QoS Support in MPLS Networks”. Frame Relay Forum. Maio 2003

[Floyd] Floyd, Sally; Jacobson, Van; “Link Sharing and resource management models do packet networks”, IEEE/ACM Transactions on Networking, Vol. 3, Nº4 (Agosto 1995), pp. 365-386

[Gnuplot] Gnuplot

<http://www.gnuplot.info/>

[Golestani] Golestani, S. Jamaloddin, “A Self-Clocked Fair Queueing Scheme for Broadband Applications” - IEEE INFOCOM'94 - Toronto, Canada - June 1994. PP. 636-646.

[Goyal] Pawan Goyal, Harrick M. Vin, and Haichen Cheng, “Start-Time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks”, IEEE / ACM Transactions on Networking, October 1997. Vol. 5, No 5, PP. 690-704.

[Guérin] R. Guérin and V. Peris, “Quality-of-service in packet networks: basic mechanisms and directions”, Computer Networks, 31:3, PP. 169-189, February 1999.

[Guérin&Partridge&Shenker] Guerin, R. Partridge, C. Shenker, S. “Specification of Guaranteed Quality of Service”. IETF RFC 2212. Setembro 1997

<http://www.ietf.org/rfc/rfc2212.txt>

[Heinanen] J. Heinanen, et al, “An Assured Forwarding PHB Group”. IETF RFC 2597. Junho 1999.

<http://www.ietf.org/rfc/rfc2597.txt>

[Hersent] Oliver Hersent, David Guide & Jean-Pierre Petit, “IP Telephony: Packet-based Multimedia Communications Systems”, Publicação Addison Wesley, Dezembro 1999.

[Huitema] Huitema, C., “Routing in the Internet”, Prentice Hall, 2000.

[Legout] Legout, A; Biersack, E.W. “PLM: Fast Convergence for Cumulative Layered Multicast Transmission Schemes”. ACM SIGMETRICS 2000, Santa Clara, California, USA. Junho 2000.

[Jacobson] Jacobson, V.; Nichols, K. ; Poduri, K. “An Expedited Forwarding PHB”. IETF RFC 2598. Junho 1999.

<http://www.ietf.org/rfc/rfc2598.txt>

[Karl] Karl, Michael. “A comparison of the architecture of network simulator NS2 and TOSSIM”. Studienprojekt Cubus. Alemanha. 31 Janeiro 2005

[Keshav] Keshav, S. “An Engineering Approach to Computer Networking”. Addison-Wesley. 1997

[Makkar] Makkar,R.; Salim, J.; Seddigh, N.; Nandy,B.; “Empirical Study of Buffer Management Scheme for Diffserv Assured Forwarding PHB”, ICCCN2000.

[MPLSRC] White Papers related to MPLS. MPLS Resource Center.

<http://www.mplsrc.com>

[MPLS RSVP-TE] Berger, L. “Generalized MPLS signaling – RSVP-TE Extensions”. IETF. Janeiro 2003.

<http://www.ietf.org/internet-drafts/draft-ietf-mpls-generalized-rsvp-te-09.txt>

[Nagle] Nagle, J. “On packet switches with infinite storage”. IEEE Transactions on Communications”. Abril 1987.

[NAM] Network Animator, NAM

<http://www.isi.edu/nsnam/nam/>

[Nichols] Nichols, K., Blake, S., Baker, F., Black, D. “Definition of the Differentiated Services Field (DS Field) in the Ipv4 and Ipv6 Headers”. Internet RFC 2474.

Dezembro 1998.

<http://www.ietf.org/rfc/rfc2474.txt>

[Nortel Networks 98] IP QoS – A Bold New Network. White Paper. September 1999. Nortel Networks.

<http://www.nortel.com>

[NS] The Network Simulator, NS-2

<http://nsnam.isi.edu/nsnam.index.php>

[NS Diffserv] DiffServ4NS

<https://sourceforge.net/projects/diffserv4ns/>

[Osbone] Osbone, Eric; Simha, Ajay, “Traffic Engineering with MPLS”, Cisco Press, Julho 2002.

[Parekh volume1] Parekh, A. K.; Gallager, R. G. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-node Case, IEEE/ACM Transactions on Networking, vol.1,no. 3, pp. 344–357, Junho 1993.

[Parekh volume2] A. K. Parekh and R. G. Gallager, A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Multiple Node Case, IEEE/ACM Transactions on Networking, vol. 2, no. 2, página. 137–150, Abril. 1994.

[Rabadão] Rabadão, Carlos; Monteiro, Edmundo. “Segurança e QoS no Modelo DiffServ”. Escola Superior de Tecnologia e Gestão e Laboratório de Comunicações e Telemática.

[Risso] Risso, Fulvio. “Decoupling Bandwidth and Delay Properties in Class Based Queuing”. Dipartimento di Automatica e Informatica – Politecnico di Torino Corso Duca degli Abruzzi.

<http://netgroup.polito.it/pubs/pdf/2001/iscc01-dcbq.pdf>

[Rosen] E. Rosen, A. Viswanathan, R. Callon, “Multiprotocol label Switching Architecture”. IETF 3031. Janeiro 2002.

<http://www.ietf.org/rfc/rfc3031.txt>

[Ruela] Ruela, José. “Apontamentos sobre Serviços Integrados na Internet”. Apontamentos para a disciplina de Redes de Banda Larga FEUP. 2002/2003

http://paginas.fe.up.pt/~mricardo/02_03/amsr/acetatos/intServ_v1.pdf

[Ruela_DiffServ] Ruela, José. “Serviços Diferenciados na Internet”. Apontamentos para a disciplina de Redes de Banda Larga, FEUP. 2002/2003.

[Ruela_MPLS] Ruela, José. “Apontamentos sobre MPLS Multiprotocol Label Switching”. Apontamentos para a disciplina de Redes de Banda Larga, FEUP. 2005/2006.

[Ruela_QoS_ATM] Ruela, José. “Apontamentos sobre Qualidade de Serviços em Redes ATM”. Apontamentos para a disciplina de Redes de Banda Larga, FEUP. 2005/2006.

[Ruela_QoS_IP] Ruela, José. “Apontamentos sobre Qualidade de Serviços em Redes IP”. Apontamentos para a disciplina de Redes de Banda Larga, FEUP. 2005/2006.

[Semeria] Semeria, Chuck. “Supporting Differentiated Service Classes: Queue Scheduling Disciplines” – Juniper Networks White Paper (Part Number:200020-001) – December 2001.

https://www2.juniper.net/solutions/literature/white_papers/200020.pdf

[Semeria&Steward]Semeria, Chuck; Stewart III, John W.. “Supporting Differentiated Service Classes in Large IP Networks” – Juniper Networks White Paper (Part Number: 200019-001) – December 2001.

http://www.juniper.net/solutions/literature/white_papers/200019.pdf

[Shreedhar&Varghese] Shreedhar, M.; Varghese, G. (October 1995). "Efficient fair queuing using deficit round robin". *ACM SIGCOMM Computer Communication Review* 25 (4).

[SimComp] Comparison of Network Simulator

<http://ti.tuwien.ac.at/ecs/people/albeseder/simcomp/simcomp/>

[Syed] Syed Ljlal Ali Shah, “Bringing Comprehensive Quality of Service Capabilities to Next-Generation Networks”. Motorola QOSM-WP/D. October 2001.

[Wroclawski] Wroclawski, J. “Specification of the Controlled-Load Network Element Service”. IETF RFC 2211. September 1997

<http://www.ietf.org/rfc/rfc2211.txt>

[Zhang] Zhang, Wiliam. “MPLS Technology”. The School of Engineering Science.

<http://www.ensc.sfu.ca/~ljilja/cnl/presentations/william/mpls/index.htm>

[Zeng] Zeng, Ximming; Lung Chung-Horong; Huang, Changcheng. “A bandwidth-efficient Scheduler for MPLS DiffServ Networks”. Department of System and Computer Engineering, Carleton University.

ANEXOS

ANEXO 1

Ficheiro TCL Base

```
# FEUP #
# Mestrado 2008/2009 #
# Daniel Ramos #
# ficheiro de simulação – caso novo algoritmo #

##### ARGUMENTS #####
set alg [lindex $argv 0]
set testTime [lindex $argv 1]
set EFPacketSize [lindex $argv 2]
set quiet false
#-----#

##### DECLARATIONS #####
set ns [new Simulator]
set EFRateOnOff 20b ; # 1 fonts - 200kbps * 1/2 = 100kbps
set EFRate 10b ; # 4 fonts - 100kbps * 4 = 400kbps
set AFRate 14b ; # 5 fonts - 140kbps * 5 = 700kbps
set BERate 12b ; # 10 fonts - 120kbps * 10 = 1200kbps
set cirEF 50 ; # parameters for Token Bucket Policer (bits/s)
set cirAF 70 ; # parameters for Token Bucket Policer (bits/s)
set cirBE 80 ; # parameters for Token Bucket Policer (bits/s)
set cbsEF [expr $cirEF*0.05/8] ; # committed burst size in bits
set cbsAF [expr $cirAF*0.05/8] ; # committed burst size in bits
set cbsBE [expr $cirBE*0.05/8] ; # committed burst size in bits
set AFPacketSize 64 ; # upd packet size (in bytes)
set BEPacketSize 64 ; # upd packet size (in bytes)
set rndStartTime [new RNG]
$rndStartTime seed 0 ; # seeds the RNG heuristically
set rndSourceNode [new RNG]
$rndSourceNode seed 0
set txTime [expr $EFPacketSize*8.0/2000]; # in millisec
set sumEFQueueLen 0
set sumAFQueueLen 0
set sumBEQueueLen 0
set samplesNum 0
#-----#

##### FILES TO SAVE DATA #####
if {$quiet == "false"} {
    set ServiceRate [open ServiceRate.tr w]
    set EFQueueLen [open EFQueueLen.tr w]
    set AFQueueLen [open AFQueueLen.tr w]
    set BEQueueLen [open BEQueueLen.tr w]
    set EFQueueLenBytes [open EFQueueLenBytes.tr w]
    set AFQueueLenBytes [open AFQueueLenBytes.tr w]
    set BEQueueLenBytes [open BEQueueLenBytes.tr w]
    set PELoss [open PELoss.tr w]
    set PLLoss [open PLLoss.tr w]
    set OWD [open OWD.tr w]
    set IPDV [open IPDV.tr w]
```

```

    set OWDAF          [open OWDAF.tr w]
    set IPDVAF          [open IPDVAF.tr w]
    set OWDBE           [open OWDBE.tr w]
    set IPDVBE          [open IPDVBE.tr w]
}
set f [open test1.out w]
$ns trace-all $f
set nf [open test1.nam w]
$ns namtrace-all $nf
#-----#

##### TOPOLOGY #####
puts "-----create topology ...."
set s(0) [$ns node]
set s(1) [$ns node]
set s(2) [$ns node]
set s(3) [$ns node]
set s(4) [$ns node]
set s(5) [$ns node]
set s(6) [$ns node]
set s(7) [$ns node]
set s(8) [$ns node]
set e1 [$ns node]
set core [$ns node]
set e2 [$ns node]
set dest(0) [$ns node]
set dest(1) [$ns node]
set dest(2) [$ns node]
set dest(3) [$ns node]
set dest(4) [$ns node]
#-----#

##### LINKS #####
puts "----- links FUNCTION-----"
$ns duplex-link $s(0) $e1 100Mb 1ms DropTail
$ns duplex-link $s(1) $e1 100Mb 1ms DropTail
$ns duplex-link $s(2) $e1 100Mb 1ms DropTail
$ns duplex-link $s(3) $e1 100Mb 1ms DropTail
$ns duplex-link $s(4) $e1 100Mb 1ms DropTail
$ns duplex-link $s(5) $e1 100Mb 1ms DropTail
$ns duplex-link $s(6) $e1 100Mb 1ms DropTail
$ns duplex-link $s(7) $e1 100Mb 1ms DropTail
$ns duplex-link $s(8) $e1 100Mb 1ms DropTail
$ns simplex-link $e1 $core 2Mb 5ms dsRED/edge
$ns simplex-link $core $e1 2Mb 5ms dsRED/core
$ns duplex-link $core $e2 5Mb 3ms DropTail
$ns duplex-link $e2 $dest(0) 100Mb 1ms DropTail
$ns duplex-link $e2 $dest(1) 100Mb 1ms DropTail
$ns duplex-link $e2 $dest(2) 100Mb 1ms DropTail
$ns duplex-link $e2 $dest(3) 100Mb 1ms DropTail
$ns duplex-link $e2 $dest(4) 100Mb 1ms DropTail
$ns duplex-link-op $s(0) $e1 orient right-up
$ns duplex-link-op $s(1) $e1 orient right-up
$ns duplex-link-op $s(2) $e1 orient right-up
$ns duplex-link-op $s(3) $e1 orient right-down
$ns duplex-link-op $s(4) $e1 orient right-down
$ns duplex-link-op $s(5) $e1 orient right-down

```

```

$ns simplex-link-op $e1 $score orient right
$ns simplex-link-op $score $e1 orient right
$ns duplex-link-op $e2 $dest(0) orient left-up
$ns duplex-link-op $e2 $dest(1) orient left
$ns duplex-link-op $e2 $dest(2) orient left
$ns duplex-link-op $e2 $dest(3) orient left
$ns duplex-link-op $e2 $dest(4) orient left-down
#-----#

##### QUEUES #####
puts "-----queues FUNCTION-----"
set qE1C [$ns link $e1 $score] queue
set qCE1 [$ns link $score $e1] queue
# Set DS RED parameters from Edge1 to Core:
$qE1C set numQueues_ 3 ; # number of physical queues
if {($alg=="PQ")} {
    $qE1C setSchedulerMode PRI
}
if {($alg=="PQSpecial")} {
    $qE1C setSchedulerMode MySM
}
if {($alg=="WFQ")} {
    $qE1C setSchedulerMode WFQ
    $qE1C addQueueWeight 0 4
    $qE1C addQueueWeight 1 15
}
if {($alg=="SCFQ")} {
    $qE1C setSchedulerMode SCFQ
    $qE1C addQueueWeight 0 3
    $qE1C addQueueWeight 1 17
}
if {($alg=="SFQ")} {
    $qE1C setSchedulerMode SFQ
    $qE1C addQueueWeight 0 3
    $qE1C addQueueWeight 1 17
}
if {($alg=="WF2Qp")} {
    $qE1C setSchedulerMode WF2Qp
    $qE1C addQueueWeight 0 3
    $qE1C addQueueWeight 1 17
}
#-----#

##### POLICIES #####
puts "-----policies FUNCTION-----"
puts "----- EF -----"
# queue and precedence levels settings
$qE1C setQSize 0 30
$qE1C setNumPrec 0 2 ; # queue 0, two levels of precedence -> sets the number of virtual
                        # queues within one physical queue.

#classifying and marking
$qE1C addMarkRule 46 -1 [$dest(0) id] any any ; #EF recommend codepoint
#metering
$qE1C addPolicyEntry 46 TokenBucket $cirEF $cbsEF ; # depending on SLS cir in bit/s, cbs in bytes
$qE1C addPolicyEntry 48 DumbSpecial
$qE1C addPolicerEntry TokenBucket 46 48
$qE1C addPHBEntry 46 0 0

```

```

$qE1C addPHBEntry 48 0 1
#shaping/dropping
$qE1C setMREDMode DROP 0
$qE1C configQ 0 0 30 ; # max in-profile packets in queue 0
$qE1C configQ 0 1 -1 ; # drop all out-of-profile packets
puts "----- AF -----"
# queue and precedence levels settings
$qE1C setQSize 1 30
$qE1C setNumPrec 1 1 ; # Gold Service queue 1, one levels of
                        #precedence, AF11

#classifying and marking
$qE1C addMarkRule 10 -1 [$dest(1) id] any any ;# AF recommend codepoint
$qE1C addPolicyEntry 11 DumbSpecial
#metering
$qE1C addPolicyEntry 10 TokenBucketSpecial $cirAF $cbsAF ; # depending on SLS cir in bit/s,
                        # cbs in bytes

$qE1C addPolicerEntry TokenBucketSpecial 10 10
$qE1C addPHBEntry 10 1 0
$qE1C addPHBEntry 11 1 1

#shaping/dropping
$qE1C setMREDMode DROP 1
$qE1C configQ 1 0 1000 ; # other value used: 200100
$qE1C configQ 1 1 -1 ; # drop all out-of-profile packets
puts "----- BE -----"
# queue and precedence levels settings
$qE1C setQSize 2 80
$qE1C setQSize 0 30
$qE1C setQSize 1 1000 ; #200100
#$qE1C setNumPrec 2 1 ;
#classifying and marking
$qE1C addMarkRule 0 -1 [$dest(2) id] any any ; #BE recommend codepoint
#metering
$qE1C addPolicyEntry 0 DumbSpecial $cirBE $cbsBE
$qE1C addPolicerEntry DumbSpecial 0 0
$qE1C addPHBEntry 0 2 0
#shaping/dropping
$qE1C setMREDMode DROP 2
$qE1C configQ 2 0 2500 ; # other value used: 200100
$qE1C setQSize 2 2500 ; # other value used: 200100
#-----

## --- CORE -> EDGE1 --- ##
# Set DS RED parameters from Core to Edge1:
$qCE1 setMREDMode DROP
$qCE1 set numQueues_ 1
$qCE1 setNumPrec 1
$qCE1 addPHBEntry 10 0 0
$qCE1 configQ 0 1 50
#insert new module
if {($alg=="PQSpecial")} {
    $qE1C setSchedulerMode MySM $qE1C
    #$qE1C printPolicyTable
}
#-----#

##### TRAFFIC #####

```

```

puts "-----traffic FUNCTION-----"
source traffic-generator-cenario3.tcl
#source traffic-generator.tcl
puts "---EF traffic data ---"
for {set i 0} {$i < 4} {incr i} {
    set startTime 0
    set sourceNodeID [$rndSourceNode integer 3]
    #puts "o random gerado foi: $sourceNodeID"
    cbr_connection $i 0 $s($sourceNodeID) $dest(0) 0 $EFPacketSize $EFRate $startTime $testTime
    if {($quiet == "false")} {
        puts "EF : s($sourceNodeID)->d(0) - Traffic: CBR - PktSize: $EFPacketSize - Rate: $EFRate - Start
$startTime"
    }
}
#generate another source
set sourceNodeIDa [$rndSourceNode integer 3]
set onoff 0.2
cbr_connection_onoff 4 0 $s($sourceNodeIDa) $dest(0) 0 $EFPacketSize $EFRateOnOff 0 $testTime
$onoff
if {($quiet == "false")} {
    puts "EF : s($sourceNodeIDa)->d(0) - Traffic: CBR OnOff - PktSize: $EFPacketSize - Rate:
$EFRateOnOff - Start 0"
}
$Sink_(0) FrequencyDistribution 0.001 0.001 $EFPacketSize\_FD.tr
puts "--- BE traffic ---"
for {set i 0} {$i < 10} {incr i} {
    set startTime 0.0
    set sourceNodeID [expr 6+[$rndSourceNode integer 3]] expoo_connection [expr $i+40] 2
    $s($sourceNodeID) $dest(2) 2 $BEPacketSize $BERate $startTime $testTime
    puts "BE: s($sourceNodeID)->d(2) - Traffic: Pareto - PktSize: $BEPacketSize - Rate: $BERate - Start
$startTime"
    set BEPacketSize [expr $BEPacketSize]
}
$Sink_(2) FrequencyDistribution 0.001 0.001 $BEPacketSize\_BE\_FD.tr
puts "----- AF traffic -----"
for {set i 0} {$i < 5} {incr i} {
    set startTime 0.0
    set sourceNodeID [expr 3+[$rndSourceNode integer 3]]
    puts "o random gerado foi: $sourceNodeID"
    cbr_connectionAF_onoff0 [expr $i+80] 1 $s($sourceNodeID) $dest(1) 1 $AFPacketSize $AFRate
    $startTime $testTime
    #generate another source
    set sourceNodeID [expr 3 + [$rndSourceNode integer 3]]
    cbr_connectionAF_onoff1 [expr $i+85] 1 $s($sourceNodeID) $dest(1) 1 $AFPacketSize $AFRate
    $startTime $testTime
    if {($quiet == "false")} {
        puts "AF : s($sourceNodeID)->d(1) - Traffic: Expoo - PktSize: $AFPacketSize - Rate: $AFRate -
Start $startTime"
    }
}
$Sink_(1) FrequencyDistribution 0.001 0.001 $AFPacketSize\_AF\_FD.tr
#-----#

#####record_departure_rate#####
puts "----- record_departure_rate Function -----"
proc record_departure_rate {} {
    global qE1C ServiceRate ServiceRateBytes

```

```

#Get an instance of the simulator
set ns [Simulator instance]
#Get the current time
set now [$ns now]
#Set the time after which the procedure should be called again
set time 0.01
set EFRate      [expr [$qE1C getDepartureRate 0 ]/1000 ]
set AF1Rate     [expr [$qE1C getDepartureRate 1 ]/1000 ]
#set AF2Rate    [expr [$qE1C getDepartureRate 2 ]/1000 ]
#set AF3Rate    [expr [$qE1C getDepartureRate 3 ]/1000 ]
set BERate      [expr [$qE1C getDepartureRate 2 ]/1000 ]
#bits por second
puts $ServiceRate "$now $EFRate $BERate $AF1Rate"
#Re-schedule the procedure
$ns at [expr $now+$time] "record_departure_rate"
}
#-----#

##### record_delay #####
puts "----- record_delay Function -----"
proc record_delay {} {
    global Sink_ OWD IPDV txTime OWDAF IPDVAF OWDBE IPDVBE
    #Get an instance of the simulator
    set ns [Simulator instance]
    #Set the time after which the procedure should be called again
    set time 0.5
    set sumOWD [$Sink_(0) set sumOwd_]
    set sumIPDV [$Sink_(0) set sumIpdv_]
    set pkts [$Sink_(0) set npktsFlowid_]
    set sumOWDAF [$Sink_(1) set sumOwd_]
    set sumIPDVAF [$Sink_(1) set sumIpdv_]
    set pktsAF [$Sink_(1) set npktsFlowid_]
    set sumOWDBE [$Sink_(2) set sumOwd_]
    set sumIPDVBE [$Sink_(2) set sumIpdv_]
    set pktsBE [$Sink_(2) set npktsFlowid_]
    #Get the current time
    set now [$ns now]
    if {($pkts<2)} {
        puts $OWD "$now 0"
        puts $OWDAF "$now 0"
        puts $OWDBE "$now 0"
    } else {
        puts "qual é o valor: $now"
        puts $OWD "$now [expr $sumOWD*1000/$pkts-$txTime]"

        if {($sumOWDAF != 0 && $pktsAF != 0) } {
            puts $OWDAF "$now [expr $sumOWDAF*1000/$pktsAF-$txTime]"
        } else {
            puts $OWDAF "$now 0"
        }
        if {($sumOWDBE != 0 && $pktsBE != 0) } {
            puts $OWDBE "$now [expr $sumOWDBE*1000/$pktsBE-$txTime]"
        } else {
            puts $OWDBE "$now 0"
        }
    }
}
puts $IPDV "$now [expr $sumIPDV*1000/($pkts-1)]"

```

```

puts $IPDVAF "$now [expr $sumIPDVAF*1000/($pktsAF-1)]"

if {($sumIPDVBE != 0 && $pktsBE != 0) } {
    puts $IPDVBE "$now [expr $sumIPDVBE*1000/($pktsBE-1)]"
} else {
    puts $IPDVBE "$now 0"
}
#Re-schedule the procedure
$ns at [expr $now+$time] "record_delay"
}
#-----#

##### record_queue_len #####
puts "----- record_queue_len Function -----"

proc record_queue_len {} {
    global EFQueueLen AFQueueLen BEQueueLen EFQueueLenBytes AFQueueLenBytes
    BEQueueLenBytes qE1C samplesNum sumEFQueueLen sumAFQueueLen sumBEQueueLen quiet
    EFPacketSize AFPacketSize BEPacketSize
    #Get an instance of the simulator
    set ns [Simulator instance]
    #Set the time after which the procedure should be called again
    set time 0.1
    set queue0Len [$qE1C getQueueLen 0]
    set queue1Len [$qE1C getQueueLen 1]
    set queue2Len [$qE1C getQueueLen 2]
    set sumEFQueueLen [expr $sumEFQueueLen + $queue0Len ]
    set sumAFQueueLen [expr $sumAFQueueLen + $queue1Len ]
    set sumBEQueueLen [expr $sumBEQueueLen + $queue2Len ]
    set samplesNum [expr $samplesNum+1]
    #Get the current time
    set now [$ns now]
    if {($quiet == "false")} {
        puts $EFQueueLen "$now $queue0Len"
        puts $EFQueueLenBytes "$now [expr $queue0Len * $EFPacketSize]"
        puts $AFQueueLen "$now $queue1Len"
        puts $AFQueueLenBytes "$now [expr $queue1Len * $AFPacketSize]"
        puts $BEQueueLen "$now $queue2Len"
        puts $BEQueueLenBytes "$now [expr $queue2Len * $BEPacketSize]"
    }
    #Re-schedule the procedure
    $ns at [expr $now+$time] "record_queue_len"
}
#-----#

##### record_packet_loss #####
puts "----- record_packet_loss Function -----"
proc record_packet_loss {} {
    global PELoss PLLoss qE1C
    #Get an instance of the simulator
    set ns [Simulator instance]
    #Set the time after which the procedure should be called again
    set time 0.1
    set edropsEF 0.0
    set edropsAF 0.0
    set edropsBE 0.0
    set dropsEF 0.0

```

```

set dropsAF 0.0
set dropsBE 0.0
#Get the current time
set now [$ns now]
if {[expr [$qE1C getStat edrops 48]] > 0.0 && [$qE1C getStat pkts 48] > 0.0} {
    set edropsEF [expr [$qE1C getStat edrops 48]*100.0/([$qE1C getStat pkts 48] + [$qE1C getStat
    pkts 46])]
}
if {[expr [$qE1C getStat edrops 11]] > 0.0 && [$qE1C getStat pkts 11] > 0.0} {
    set edropsAF [expr [$qE1C getStat edrops 11]*100.0/([$qE1C getStat pkts 11] + [$qE1C getStat
    pkts 10])]
}
if {[expr [$qE1C getStat edrops 0]] > 0.0 && [$qE1C getStat pkts 0] > 0.0} {
    set edropsBE [expr [$qE1C getStat edrops 0]*100.0/[$qE1C getStat pkts 0]]
}
puts $PELoss "$now $edropsEF $edropsAF $edropsBE"
if {[expr [$qE1C getStat drops 48]] > 0.0 && [$qE1C getStat pkts 48] > 0.0} {
    set dropsEF [expr [$qE1C getStat drops 48]*100.0/([$qE1C getStat pkts 48] + [$qE1C getStat
    pkts 46])]
}
if {[expr [$qE1C getStat drops 11]] > 0.0 && [$qE1C getStat pkts 11] > 0.0} {
    set dropsAF [expr [$qE1C getStat drops 11]*100.0/([$qE1C getStat pkts 11] + [$qE1C getStat
    pkts 10])]
}
if {[expr [$qE1C getStat drops 0]] > 0.0 && [$qE1C getStat pkts 0] > 0.0} {
    set dropsBE [expr [$qE1C getStat drops 0]*100.0/[$qE1C getStat pkts 0]]
}
puts $PLLoss "$now $dropsEF $dropsAF $dropsBE"
#Re-schedule the procedure
$ns at [expr $now+$time] "record_packet_loss"
}
#-----#

##### finish #####

proc finish {} {
    global ns Sink_ OWD IPDV OWDAF IPDVAF OWDBE IPDVBE ServiceRate quiet EFPacketSize
    AFPacketSize BEPacketSize alg EFQueueLen AFQueueLen BEQueueLen EFQueueLenBytes
    AFQueueLenBytes BEQueueLenBytes sumEFQueueLen samplesNum qE1C testTime PELoss PLLoss

    $ns flush-trace
    $Sink_(0) flushFD
    $Sink_(1) flushFD
    $Sink_(2) flushFD

    if {$quiet == "false"} {
        close $OWD
        close $IPDV
        close $OWDAF
        close $IPDVAF
        close $OWDBE
        close $IPDVBE
        close $ServiceRate
        close $EFQueueLen
        close $AFQueueLen
        close $BEQueueLen
        close $EFQueueLenBytes
        close $AFQueueLenBytes
    }
}

```

```

close $BQueueLenBytes
close $PELoss
close $PLLoss

source gnuplot-x-cenario3.tcl
exec gnuplot bw_x.p
exec gnuplot owd_x.p
exec gnuplot ipdv_x.p
exec gnuplot owdAF_x.p
exec gnuplot ipdvAF_x.p
exec gnuplot owdBE_x.p
exec gnuplot ipdvBE_x.p
exec gnuplot queue_x.p
exec gnuplot queuebytes_x.p
exec gnuplot Queue.p
exec gnuplot QueueAF.p
exec gnuplot QueueBE.p

exec gnuplot owdFD_x.p
exec gnuplot ipdvFD_x.p
exec gnuplot owdAFFD_x.p
exec gnuplot ipdvAFFD_x.p
exec gnuplot owdBEFD_x.p
exec gnuplot ipdvBEFD_x.p
exec gnuplot Debito.p
exec gnuplot Contador.p
exec gnuplot losses_drops.p
exec gnuplot losses_edrops.p
exec gnuplot QueueVirtual.p
#call a script to calculate initial traffic
exec tclsh calcular_testar.tcl $EFPacketSize $BEPacketSize $AFPacketSize $testTime
}

set gIPDV [open "$alg\_IPDV.tr" a]
set gOWD [open "$alg\_OWD.tr" a]
set gQueueLen [open "$alg\_QueueLen.tr" a]
set gPktLoss [open "$alg\_PktLoss.tr" a]
set gEFPktLoss [open "$alg\_EFPktLoss.tr" a]
set gEF_MBS [open "$alg\_EF_MBS.tr" a]

set sumOWD [$Sink_(0) set sumOwd_]
set sumIPDV [$Sink_(0) set sumIpdv_]
set npkts [$Sink_(0) set npktsFlowid_]
set sumOWDAF [$Sink_(1) set sumOwd_]
set sumIPDVAF [$Sink_(1) set sumIpdv_]
set npktsAF [$Sink_(1) set npktsFlowid_]
set sumOWDBE [$Sink_(2) set sumOwd_]
set sumIPDVBE [$Sink_(2) set sumIpdv_]
set npktsBE [$Sink_(2) set npktsFlowid_]

puts $gOWD "EF $EFPacketSize [expr $sumOWD*1000/$npkts]"
puts $gIPDV "EF $EFPacketSize [expr $sumIPDV*1000/($npkts-1)]"
puts $gOWD "AF $AFPacketSize [expr $sumOWDAF*1000/$npktsAF]"
puts $gIPDV "AF $AFPacketSize [expr $sumIPDVAF*1000/($npktsAF- 1)]"
puts $gOWD "BE $BEPacketSize [expr $sumOWDBE*1000/$npktsBE]"
puts $gIPDV "BE $BEPacketSize [expr $sumIPDVBE*1000/($npktsBE- 1)]"
puts $gQueueLen "EF $EFPacketSize [expr $sumEFQueueLen/$samplesNum]"

```

```

set EF_MBS [$qE1C set MBS0_ ]
puts $gEF_MBS "$EFPacketSize $EF_MBS"

set Pkt    [$qE1C getStat pkts ]
set dropPkt [$qE1C getStat drops ]
set edropPkt [$qE1C getStat edrops ]
puts $gPktLoss "$EFPacketSize [expr ($Pkt-$dropPkt- $edropPkt)*100.0/$Pkt] [expr
$dropPkt*100.0/$Pkt] [expr $edropPkt*100.0/$Pkt]"

set Pkt    [$qE1C getStat pkts 46 ]
set dropPkt [$qE1C getStat drops 46 ]
set edropPkt [$qE1C getStat edrops 46 ]
puts $gEFPktLoss "$EFPacketSize [expr ($Pkt-$dropPkt-$edropPkt)*100.0/$Pkt] [expr
$dropPkt*100.0/$Pkt] [expr $edropPkt*100.0/$Pkt]"

close $gOWD
close $gIPDV
close $gQueueLen
close $gPktLoss
close $gEFPktLoss
close $gEF_MBS
exec tclsh calcular_testar.tcl $EFPacketSize $BEPacketSize $AFPacketSize $testTime
exit 0
}
puts "EF packet size: $EFPacketSize"
# CP -> codepoint
# Totpkts -> packets received
# Txpkts -> Packets sent
# Idrops -> packets are dropped due to link overflow - os pacotes sao descartados devido ao excesso de
trafego no link
# edrops -> RED early dropping -
$qE1C printPolicyTable
$qE1C printPolicerTable
$qE1C printPHBTable
$ns at 0.0 "record_departure_rate"
$ns at 0.0 "record_delay"
$ns at 0.0 "record_packet_loss"
puts "[expr $testTime/2]"

$ns at [expr $testTime/2] "$qE1C printStats"
puts "[expr $testTime]"

$ns at [expr $testTime - 0.1] "$qE1C printStats"
$ns at 0.0 "record_queue_len"
$ns at $testTime "finish"

$ns run

```

Ficheiro anexo ao TCL Base

```

# SourceId    : source agent index
# SinkId      : sink  agent index
# src         : source node
# dst         : destination node
# flowId      : flow Id used by LossMonitor to compute parameters as OWD and IPDV
# PacketSize  : UDP packet size (in bytes)

```

```

# Rate   :
# Start  :
# Stop   :
proc cbr_connection_onoff {SourceId SinkId src dst flowId PacketSize Rate start testtime onoff} {
    global ns Sink_
    set udp_($SourceId) [new Agent/UDP]
    $udp_($SourceId) set class_ $flowId
    $ns attach-agent $src $udp_($SourceId)
    set cbr_($SourceId) [new Application/Traffic/CBR]
    $cbr_($SourceId) attach-agent $udp_($SourceId)
    $cbr_($SourceId) set packet_size_ $PacketSize
    $udp_($SourceId) set packetSize_ $PacketSize
    $cbr_($SourceId) set rate_ $Rate

    if {($flowId == 0) || TRUE} {
        set Sink_($SinkId) [new Agent/LossMonitor]
        $Sink_($SinkId) set flowid_ $flowId
        puts "Sink_($SinkId) $flowId"
    } else {
        set Sink_($SinkId) [new Agent/Null]
    }

    $ns attach-agent $dst $Sink_($SinkId)
    $ns connect $udp_($SourceId) $Sink_($SinkId)

    set stop 0
    while { $start < [expr $testtime - $onoff] } {
        set stop [expr $start + $onoff]
        $ns at $start "$cbr_($SourceId) start"
        $ns at $stop "$cbr_($SourceId) stop"
        set start [expr $stop + $onoff]
    }
}

proc cbr_connectionAF_onoff0 {SourceId SinkId src dst flowId PacketSize Rate start testtime} {
    global ns Sink_
    set udp_($SourceId) [new Agent/UDP]
    $udp_($SourceId) set class_ $flowId
    $ns attach-agent $src $udp_($SourceId)
    set cbr_($SourceId) [new Application/Traffic/CBR]
    $cbr_($SourceId) attach-agent $udp_($SourceId)
    $cbr_($SourceId) set packet_size_ $PacketSize
    $udp_($SourceId) set packetSize_ $PacketSize
    $cbr_($SourceId) set rate_ 238000b

    if {($flowId == 0) || TRUE} {
        set Sink_($SinkId) [new Agent/LossMonitor]
        $Sink_($SinkId) set flowid_ $flowId
        puts "Sink_($SinkId) $flowId"
    } else {
        set Sink_($SinkId) [new Agent/Null]
    }

    $ns attach-agent $dst $Sink_($SinkId)
    $ns connect $udp_($SourceId) $Sink_($SinkId)

    set stop 0
    while { $start < [expr $testtime - 0.2] } {

```

```

        set stop [expr $start + 0.2]
        puts "start $start"
        puts "stop $stop"
        $ns at $start "$cbr_($SourceId) start"
        $ns at $stop "$cbr_($SourceId) stop"
        set start [expr $stop + 0.4]
    }
}

proc cbr_connectionAF_onoff1 {SourceId SinkId src dst flowId PacketSize Rate start testtime} {
    global ns Sink_
    set udp_($SourceId) [new Agent/UDP]
    $udp_($SourceId) set class_ $flowId
    $ns attach-agent $src $udp_($SourceId)
    set cbr_($SourceId) [new Application/Traffic/CBR]
    $cbr_($SourceId) attach-agent $udp_($SourceId)
    $cbr_($SourceId) set packet_size_ $PacketSize
    $udp_($SourceId) set packetSize_ $PacketSize
    $cbr_($SourceId) set rate_ 90000b ; #deveria ser 90000b

    if {($flowId == 0) || TRUE} {
        set Sink_($SinkId) [new Agent/LossMonitor]
        $Sink_($SinkId) set flowid_ $flowId
        puts "Sink_($SinkId) $flowId"
    } else {
        set Sink_($SinkId) [new Agent/Null]
    }

    $ns attach-agent $dst $Sink_($SinkId)
    $ns connect $udp_($SourceId) $Sink_($SinkId)

    set stop 0
    while { $start < [expr $testtime - 0.2] } {
        set stop [expr $start + 0.4]
        puts "start AF1 $start"
        puts "stop AF1 $stop"
        if {($start < 0.3)} {
            $cbr_($SourceId) set rate_ 60000b
        } else {
            $cbr_($SourceId) set rate_ 90000b
        }
        $ns at $start "$cbr_($SourceId) start"
        $ns at $stop "$cbr_($SourceId) stop"
        set start [expr $stop + 0.2]
    }
}

proc cbr_connection {SourceId SinkId src dst flowId PacketSize Rate start testtime} {
    global ns Sink_
    set udp_($SourceId) [new Agent/UDP]
    $udp_($SourceId) set class_ $flowId
    $ns attach-agent $src $udp_($SourceId)
    set cbr_($SourceId) [new Application/Traffic/CBR]
    $cbr_($SourceId) attach-agent $udp_($SourceId)
    $cbr_($SourceId) set packet_size_ $PacketSize
    $udp_($SourceId) set packetSize_ $PacketSize
    $cbr_($SourceId) set rate_ $Rate

```

```

if {($flowId == 0) || TRUE} {
    set Sink_($SinkId) [new Agent/LossMonitor]
    $Sink_($SinkId) set flowid_ $flowId
    puts "Sink_($SinkId) $flowId"
} else {
    set Sink_($SinkId) [new Agent/Null]
}

$ns attach-agent $dst $Sink_($SinkId)
$ns connect $udp_($SourceId) $Sink_($SinkId)

$ns at $start "$cbr_($SourceId) start"
$ns at $testtime "$cbr_($SourceId) stop"
}

proc pareto_connection {SourceId SinkId src dst flowId PacketSize Rate start stop} {
    global ns Sink_
    set udp_($SourceId) [new Agent/UDP]
    $udp_($SourceId) set class_ $flowId
    $ns attach-agent $src $udp_($SourceId)
    set pareto_($SourceId) [new Application/Traffic/Pareto]
    $pareto_($SourceId) attach-agent $udp_($SourceId)
    $pareto_($SourceId) set packetSize_ $PacketSize

    $pareto_($SourceId) set packetSize_ $PacketSize
    $pareto_($SourceId) set burst_time_ 200ms
    $pareto_($SourceId) set idle_time_ 200ms
    $pareto_($SourceId) set rate_ $Rate
    $pareto_($SourceId) set shape_ 1

    $udp_($SourceId) set packetSize_ $PacketSize

    if {($flowId == 0) || TRUE} {
        set Sink_($SinkId) [new Agent/LossMonitor]
        $Sink_($SinkId) set flowid_ $flowId
    } else {
        set Sink_($SinkId) [new Agent/Null]
    }

    $ns attach-agent $dst $Sink_($SinkId)
    $ns connect $udp_($SourceId) $Sink_($SinkId)
    $ns at $start "$pareto_($SourceId) start"
    $ns at $stop "$pareto_($SourceId) stop"
}

proc expoo_connection {SourceId SinkId src dst flowId PacketSize Rate start stop} {
    global ns Sink_
    set udp_($SourceId) [new Agent/UDP]
    $udp_($SourceId) set class_ $flowId
    $ns attach-agent $src $udp_($SourceId)
    set expoo_($SourceId) [new Application/Traffic/Exponential]
    $expoo_($SourceId) attach-agent $udp_($SourceId)
    $expoo_($SourceId) set packetSize_ $PacketSize

    $expoo_($SourceId) set packetSize_ $PacketSize
    $expoo_($SourceId) set burst_time_ 500ms
    $expoo_($SourceId) set idle_time_ 0ms

```

```

$expoo_($SourceId) set rate_ $Rate

$udp_($SourceId) set packetSize_ $PacketSize

if {($flowId == 0)|| TRUE} {
    set Sink_($SinkId) [new Agent/LossMonitor]
    $Sink_($SinkId) set flowid_ $flowId
    puts "Sink_($SinkId) $flowId"
} else {
    set Sink_($SinkId) [new Agent/Null]
}

$ns attach-agent $dst $Sink_($SinkId)
$ns connect $udp_($SourceId) $Sink_($SinkId)
$ns at $start "$expoo_($SourceId) start"
$ns at $stop "$expoo_($SourceId) stop"
}

proc expoo_connectionEF {SourceId SinkId src dst flowId PacketSize Rate start stop} {
    global ns Sink_
    set udp_($SourceId) [new Agent/UDP]
    $udp_($SourceId) set class_ $flowId
    $ns attach-agent $src $udp_($SourceId)
    set expoo_($SourceId) [new Application/Traffic/Exponential]
    $expoo_($SourceId) attach-agent $udp_($SourceId)
    $expoo_($SourceId) set packetSize_ $PacketSize

    $expoo_($SourceId) set packetSize_ $PacketSize
    $expoo_($SourceId) set burst_time_ 200ms
    $expoo_($SourceId) set idle_time_ 200ms
    $expoo_($SourceId) set rate_ $Rate

    $udp_($SourceId) set packetSize_ $PacketSize

    if {($flowId == 0)|| TRUE} {
        set Sink_($SinkId) [new Agent/LossMonitor]
        $Sink_($SinkId) set flowid_ $flowId
        puts "Sink_($SinkId) $flowId"
    } else {
        set Sink_($SinkId) [new Agent/Null]
    }

    $ns attach-agent $dst $Sink_($SinkId)
    $ns connect $udp_($SourceId) $Sink_($SinkId)
    $ns at $start "$expoo_($SourceId) start"
    $ns at $stop "$expoo_($SourceId) stop"
}

proc cbr_connectionEF {SourceId SinkId src dst flowId PacketSize Rate start stop} {
    global ns Sink_
    set udp_($SourceId) [new Agent/UDP]
    $udp_($SourceId) set class_ $flowId
    $ns attach-agent $src $udp_($SourceId)
    set cbr_($SourceId) [new Application/Traffic/CBR]
    $cbr_($SourceId) attach-agent $udp_($SourceId)

```

```
$cbr_($SourceId) set packet_size_ $PacketSize
$udp_($SourceId) set packetSize_ $PacketSize
$cbr_($SourceId) set rate_ $Rate

if {($flowId == 0) || TRUE} {
    set Sink_($SinkId) [new Agent/LossMonitor]
    $Sink_($SinkId) set flowid_ $flowId
    puts "Sink_($SinkId) $flowId"
} else {
    set Sink_($SinkId) [new Agent/Null]
}

$ns attach-agent $dst $Sink_($SinkId)
$ns connect $udp_($SourceId) $Sink_($SinkId)
$ns at $start "$cbr_($SourceId) start"
$ns at $stop "$cbr_($SourceId) stop"
}
```

ANEXO 2

Alterações aos ficheiros “dsPolicy.cc” e “dsEdge.cc” disponível no módulo DiffServ do NS.

A2.1 dsPolicy.cc

```
#include <fstream>
#include <string>
#include "dsPolicy.h"
#include "packet.h"
#include "tcp.h"
#include "random.h"
#include "dsEdge.h"
#define TimerT 0.004
//The definition of class PolicyClassifier.
//Constructor.
PolicyClassifier::PolicyClassifier() {
    int i;
    policyTableSize = 0;
    policerTableSize = 0;
    for (i = 0; i < MAX_POLICIES; i++)
        policy_pool[i] = NULL;
}
/*-----*/
void addPolicyEntry()
    Adds an entry to policyTable according to the arguments in argv. A source and destination node ID must
    be specified in argv, followed by a policy type and policy-specific parameters. Supported policies and their
    parameters are:
    TSW2CM InitialCodePoint CIR
    TSW3CM InitialCodePoint CIR PIR
    TokenBucket InitialCodePoint CIR CBS
    srTCM InitialCodePoint CIR CBS EBS
    trTCM InitialCodePoint CIR CBS PIR PBS
    No error-checking is performed on the parameters. CIR and PIR should be specified in bits per second;
    CBS, EBS, and PBS should be specified in bytes. If the Policy Table is full, this method prints an error
    message.
    -----*/
void PolicyClassifier::addPolicyEntry(int argc, const char*const* argv, edgeQueue *eq) {
    if (policyTableSize == MAX_POLICIES)
        printf("ERROR: Policy Table size limit exceeded.\n");
    else {
        policyTable[policyTableSize].codePt = (int)strtod(argv[2],NULL);
        policyTable[policyTableSize].arrivalTime = 0;
        policyTable[policyTableSize].winLen = 1.0;
        /* daniel ini */
        policyTable[policyTableSize].timeref = 0;
        /* daniel fim */
        if (strcmp(argv[3], "Dumb") == 0) {
            if(!policy_pool[DUMB])
```

```

        policy_pool[DUMB] = new DumbPolicy;
        policyTable[policyTableSize].policy_index = DUMB;
        policyTable[policyTableSize].policer = dumbPolicer;
        policyTable[policyTableSize].meter = dumbMeter;
    }
    /* daniel ini */
    else if (strcmp(argv[3], "DumbSpecial") == 0) {
        /*if(!policy_pool[DUMB])
        policy_pool[DUMB] = (Policy *) new DumbSpecialPolicy;
        policyTable[policyTableSize].policy_index = DUMB;
        policyTable[policyTableSize].policer = dumbPolicer;
        policyTable[policyTableSize].meter = dumbMeter;
        policyTable[policyTableSize].cir = policyTable[policyTableSize].avgRate = (double)
        atof(argv[4])/8.0;
        policyTable[policyTableSize].cbs = policyTable[policyTableSize].cBucket = (double)
        atof(argv[5]);
        policyTable[policyTableSize].qmother = eq;
        } else if (strcmp(argv[3], "TSW2CM") == 0) {
            if(!policy_pool[TSW2CM])
                policy_pool[TSW2CM] = new TSW2CMPolicy;
            policyTable[policyTableSize].policy_index = TSW2CM;
            policyTable[policyTableSize].policer = TSW2CMPolicer;
            policyTable[policyTableSize].meter = tswTagger;
            policyTable[policyTableSize].cir = policyTable[policyTableSize].avgRate = (double)
            atof(argv[4]) / 8.0;
            if (argc == 8)
                policyTable[policyTableSize].winLen = (double) atof(argv[5]);/* mb */
        } else if (strcmp(argv[3], "TSW3CM") == 0) {
            if(!policy_pool[TSW3CM])
                policy_pool[TSW3CM] = new TSW3CMPolicy;
            policyTable[policyTableSize].policy_index = TSW3CM;
            policyTable[policyTableSize].policer = TSW3CMPolicer;
            policyTable[policyTableSize].meter = tswTagger;
            policyTable[policyTableSize].cir = policyTable[policyTableSize].avgRate = (double)
            atof(argv[4]) / 8.0;
            policyTable[policyTableSize].pir = (double) atof(argv[5]) / 8.0;
        } else if (strcmp(argv[3], "TokenBucketSpecial") == 0) {
            /*if(!policy_pool[TBSpecial])
            policy_pool[TBSpecial] = (Policy *) new TBSpecialPolicy;
            policyTable[policyTableSize].policy_index = TBSpecial;
            policyTable[policyTableSize].policer = tokenBucketSpecialPolicer;
            policyTable[policyTableSize].meter = tokenBucketSpecialMeter;
            policyTable[policyTableSize].cir =
            policyTable[policyTableSize].avgRate = (double) atof(argv[4])/8.0;
            policyTable[policyTableSize].cbs = policyTable[policyTableSize].cBucket = (double)
            atof(argv[5]);
            policyTable[policyTableSize].qmother = eq;
            } else if (strcmp(argv[3], "TokenBucket") == 0) {
                if(!policy_pool[TB])
                    policy_pool[TB] = (Policy *) new TBPolicer;
                policyTable[policyTableSize].policy_index = TB;
                policyTable[policyTableSize].policer = tokenBucketPolicer;
                policyTable[policyTableSize].meter = tokenBucketMeter;
                policyTable[policyTableSize].cir = policyTable[policyTableSize].avgRate = (double)
                atof(argv[4])/8.0;
                policyTable[policyTableSize].cbs = policyTable[policyTableSize].cBucket = (double)
                atof(argv[5]);
            }
        }
    }
}

```

```

    policyTable[policyTableSize].qmother = eq;
} else if (strcmp(argv[3], "srTCM") == 0) {
    if(!policy_pool[SRTCM])
        policy_pool[SRTCM] = new SRTCMPolicy;
    policyTable[policyTableSize].policy_index = SRTCM;
    policyTable[policyTableSize].policer = srTCMPolicer;
    policyTable[policyTableSize].meter = srTCMMeter;
    policyTable[policyTableSize].cir = policyTable[policyTableSize].avgRate = (double)
atof(argv[4]) / 8.0;
    policyTable[policyTableSize].cbs = policyTable[policyTableSize].cBucket = (double)
atof(argv[5]);
    policyTable[policyTableSize].ebs = policyTable[policyTableSize].eBucket = (double)
atof(argv[6]);
} else if (strcmp(argv[3], "trTCM") == 0) {
    if(!policy_pool[TRTCM])
        policy_pool[TRTCM] = new TRTCMPolicy;
    policyTable[policyTableSize].policy_index = TRTCM;
    policyTable[policyTableSize].policer = trTCMPolicer;
    policyTable[policyTableSize].meter = trTCMMeter;
    policyTable[policyTableSize].cir = policyTable[policyTableSize].avgRate = (double)
atof(argv[4]) / 8.0;
    policyTable[policyTableSize].cbs = policyTable[policyTableSize].cBucket = (double)
atof(argv[5]);
    policyTable[policyTableSize].pir = (double) atof(argv[6]) / 8.0;
    policyTable[policyTableSize].pbs = policyTable[policyTableSize].pBucket = (double)
atof(argv[7]);
} else if (strcmp(argv[3], "FW") == 0) {
    if(!policy_pool[FW])
        policy_pool[FW] = new FWPolicy;
    policyTable[policyTableSize].policy_index = FW;
    policyTable[policyTableSize].policer = FWPolicer;
    policyTable[policyTableSize].meter = fwTagger;
// Use cir as the transmission size threshold for the moment.
    policyTable[policyTableSize].cir = (int)strtod(argv[4], NULL);
} else {
    printf("No applicable policy specified, exit!!!\n");
    exit(-1);
}
    policyTableSize++;
}
}

/*-----*/
policyTableEntry* PolicyClassifier::getPolicyTableEntry(long source, long dest)
Pre: policyTable holds exactly one entry for the specified source-dest pair.
Post: Finds the policyTable array that matches the specified source-dest pair.
Returns: On success, returns a pointer to the corresponding policyTableEntry; on failure, returns NULL.
Note: the source-destination pair could be one-any or any-any (xuanc)
-----*/
policyTableEntry* PolicyClassifier::getPolicyTableEntry(int codePt) {
    for (int i = 0; i <= policyTableSize; i++)
        if (policyTable[i].codePt==codePt)
            return(&policyTable[i]);
    printf("ERROR: No Policy Table entry found for DSCP %d\n", codePt);
    //printPolicyTable();
    return(NULL);
}

```

```

policyTableEntry* PolicyClassifier::setPolicyTableEntry(int codePt, double value) {
    for (int i = 0; i <= policyTableSize; i++)
        if (policyTable[i].codePt==codePt)
            policyTable[i].cbs = value;
    return(NULL);
}

/*-----*/
void addPolicerEntry(int argc, const char*const* argv)
Pre: argv contains a valid command line for adding a policer entry.
Post: Adds an entry to policerTable according to the arguments in argv. No error-checking is done on the
arguments. A policer type should be specified, consisting of one of the names {TSW2CM, TSW3CM,
TokenBucket, srTCM, trTCM}, followed by an initial code point. Next should be an out-of-profile code
point for policers with two-rate markers; or a yellow and a red code point for policers with three drop
precedences. If policerTable is full, an error message is printed.
/*-----*/
void PolicyClassifier::addPolicerEntry(int argc, const char*const* argv) {
    if (policerTableSize == MAX_CP)
        printf("ERROR: Policer Table size limit exceeded.\n");
    else {
        if (strcmp(argv[2], "Dumb") == 0) {
            if (!policy_pool[DUMB])
                policy_pool[DUMB] = new DumbPolicy;
            policerTable[policerTableSize].policer = dumbPolicer;
            policerTable[policerTableSize].policy_index = DUMB;
        } else if (strcmp(argv[2], "DumbSpecial") == 0) {
            if (!policy_pool[DUMB])
                policy_pool[DUMB] = new DumbSpecialPolicy;
            policerTable[policerTableSize].policer = dumbPolicer;
            policerTable[policerTableSize].policy_index = DUMB;
        } else if (strcmp(argv[2], "TSW2CM") == 0) {
            if (!policy_pool[TSW2CM])
                policy_pool[TSW2CM] = new TSW2CMPolicy;
            policerTable[policerTableSize].policer = TSW2CMPolicer;
            policerTable[policerTableSize].policy_index = TSW2CM;
        } else if (strcmp(argv[2], "TSW3CM") == 0) {
            if (!policy_pool[TSW3CM])
                policy_pool[TSW3CM] = new TSW3CMPolicy;
            policerTable[policerTableSize].policer = TSW3CMPolicer;
            policerTable[policerTableSize].policy_index = TSW3CM;
        } else if (strcmp(argv[2], "TokenBucket") == 0) {
            if (!policy_pool[TB])
                policy_pool[TB] = new TBPolicer;
            policerTable[policerTableSize].policer = tokenBucketPolicer;
            policerTable[policerTableSize].policy_index = TB;
        } else if (strcmp(argv[2], "TokenBucketSpecial") == 0) {
            if (!policy_pool[TBSpecial])
                policy_pool[TBSpecial] = new TBSpecialPolicy;
            policerTable[policerTableSize].policer = tokenBucketSpecialPolicer;
            policerTable[policerTableSize].policy_index = TBSpecial;
        } else if (strcmp(argv[2], "srTCM") == 0) {
            if (!policy_pool[SRTCMT])
                policy_pool[SRTCMT] = new SRTCMPolicy;
            policerTable[policerTableSize].policer = srTCMPolicer;
            policerTable[policerTableSize].policy_index = SRTCMT;
        } else if (strcmp(argv[2], "trTCM") == 0){
            if (!policy_pool[TRTCMT])
                policy_pool[TRTCMT] = new TRTCMPolicy;
        }
    }
}

```

```

        policerTable[policerTableSize].policer = trTCMPolicer;
        policerTable[policerTableSize].policy_index = TRTCM;
    } else if (strcmp(argv[2], "FW") == 0) {
        if(!policy_pool[FW])
            policy_pool[FW] = new FWPolicy;
        policerTable[policerTableSize].policer = FWPolicy;
        policerTable[policerTableSize].policy_index = FW;
    } else {
        printf("No applicable policer specified, exit!!!\n");
        exit(-1);
    }
};
policerTable[policerTableSize].initialCodePt = (int)strtod(argv[3], NULL);
//printf("Policer: %s %s \n", argv[2], argv[3]);
if (argc == 5)
    policerTable[policerTableSize].downgrade1 = (int)strtod(argv[4], NULL);
if (argc == 6)
    policerTable[policerTableSize].downgrade2 = (int)strtod(argv[5], NULL);
policerTableSize++;
}

// Return the entry of Policer table with policerType and initCodePoint matched
policerTableEntry* PolicyClassifier::getPolicerTableEntry(int policy_index, int oldCodePt) {
    for (int i = 0; i < policerTableSize; i++)
        if ((policerTable[i].policy_index == policy_index) && (policerTable[i].initialCodePt ==
            oldCodePt))
            return(&policerTable[i]);
    printf("ERROR: No Policer Table entry found for initial code point %d.\n", oldCodePt);
    //printPolicerTable();
    return(NULL);
}

/*-----*/
int mark(Packet *pkt)
Pre: The source-destination pair taken from pkt matches a valid entry in policyTable.
Post: pkt is marked with an appropriate code point.
/*-----*/

int PolicyClassifier::mark(Packet *pkt) {
    policyTableEntry *policy;
    policerTableEntry *policer;
    int policy_index, codePt, fid;
    hdr_ip* iph;
    iph = hdr_ip::access(pkt);
    fid = iph->flowid();
    policy = getPolicyTableEntry(iph->prio());
    codePt = policy->codePt;
    policy_index = policy->policy_index;
    policer = getPolicerTableEntry(policy_index, codePt);
    if (policy_pool[policy_index]) {
        policy_pool[policy_index]->applyMeter(policy, pkt, true);
        codePt = policy_pool[policy_index]->applyPolicer(policy, policer, pkt);
    } else {
        printf("The policy object doesn't exist, ERROR!!!\n");
        exit(-1);
    }
    iph->prio_ = codePt;
    return(codePt);
}

// Prints the policyTable, one entry per line.

```

```

void PolicyClassifier::printPolicyTable() {
    for (int i = 0; i < policyTableSize; i++){
        switch (policyTable[i].policer) {
            case dumbPolicer:
                printf("Traffic marked with DSCP %2d : DUMB policer\n", policyTable[i].codePt);
                break;
            case TSW2CMPolicer:
                printf("Traffic marked with DSCP %2d : TSW2CM policer, ", policyTable[i].codePt);
                printf("CIR %.1f bps.\n", policyTable[i].cir * 8);
                break;
            case TSW3CMPolicer:
                printf("Traffic marked with DSCP %2d : TSW3CM policer, ", policyTable[i].codePt);
                printf("CIR %.1f bps, PIR %.1f bytes.\n", policyTable[i].cir * 8, policyTable[i].pir * 8);
                break;
            case tokenBucketPolicer:
                printf("Traffic marked with DSCP %2d : Token Bucket policer, ", policyTable[i].codePt);
                printf("CIR %.1f bps, CBS %.1f bytes.\n", policyTable[i].cir * 8, policyTable[i].cbs);
                break;
            case tokenBucketSpecialPolicer:
                printf("Traffic marked with DSCP %2d : Token Bucket Special policer, ",
policyTable[i].codePt);
                printf("CIR %.1f bps, CBS %.1f bytes.\n", policyTable[i].cir * 8, policyTable[i].cbs);
                break;
            case srTCMPolicer:
                printf("Traffic marked with DSCP %2d : srTCM policer, ", policyTable[i].codePt);
                printf("CIR %.1f bps, CBS %.1f bytes, EBS %.1f bytes.\n", policyTable[i].cir * 8,
policyTable[i].cbs, policyTable[i].ebs);
                break;
            case trTCMPolicer:
                printf("Traffic marked with DSCP %2d : trTCM policer, ", policyTable[i].codePt);
                printf("CIR %.1f bps, CBS %.1f bytes, PIR %.1f bps, ", policyTable[i].cir * 8,
policyTable[i].cbs, policyTable[i].pir * 8);
                printf("PBS %.1f bytes.\n", policyTable[i].pbs);
                break;
            case FWPolicer:
                printf("Traffic marked with DSCP %2d : FW policer, ", policyTable[i].codePt);
                printf("TH %d bytes.\n", (int)policyTable[i].cir);
                break;
            default:
                printf("ERROR: Unknown policer type in Policy Table.\n");
        }
    }
}

```

// Prints the policerTable, one entry per line.

```

void PolicyClassifier::printPolicerTable() {
    bool threeColor, dumb;
    printf("Policer Table:\n");
    for (int i = 0; i < policerTableSize; i++) {
        threeColor = false; dumb=false;
        switch (policerTable[i].policer) {
            case dumbPolicer:
                printf("Dumb ");
                dumb=true;
                break;
            case TSW2CMPolicer:
                printf("TSW2CM ");

```

```

        break;
    case TSW3CMPolicer:
        printf("TSW3CM ");
        threeColor = true;
        break;
    case tokenBucketPolicer:
        printf("Token Bucket ");
        break;
    case tokenBucketSpecialPolicer:
        printf("Token Bucket Special");
        break;
    case srTCMPolicer:
        printf("srTCM ");
        threeColor = true;
        break;
    case trTCMPolicer:
        printf("trTCM ");
        threeColor = true;
        break;
    case FWPolicer:
        printf("FW ");
        break;
    default:
        printf("ERROR: Unknown policer type in Policer Table.");
    }
    if (dumb)
        printf("policer: DSCP %2d is not policed\n", policerTable[i].initialCodePt);
    else if (threeColor) {
        printf("policer: DSCP %2d is policed to yellow ", policerTable[i].initialCodePt);
        printf("DSCP %2d and red DSCP %2d.\n", policerTable[i].downgrade1,
policerTable[i].downgrade2);
    } else
        printf("policer: DSCP %2d is policed to DSCP %2d.\n", policerTable[i].initialCodePt,
policerTable[i].downgrade1);
    }
}

// The beginning of the definition of DumbPolicy
// DumbPolicy will do nothing, but is a good example to show how to add new policy.
/*-----*/
void DumbPolicy::applyMeter(policyTableEntry *policy, Packet *pkt)
Do nothing
-----*/
void DumbPolicy::applyMeter(policyTableEntry *policy, Packet *pkt, bool a) {
    policy->arrivalTime = Scheduler::instance().clock();
}
/*-----*/
int DumbPolicy::applyPolicer(policyTableEntry *policy, int initialCodePt, Packet *pkt)
Always return the initial codepoint.
-----*/
int DumbPolicy::applyPolicer(policyTableEntry *policy, policerTableEntry *policer, Packet *pkt) {
    return(policer->initialCodePt);
}

// The end of DumbPolicy
// The beginning of the definition of SpecialPolicy
// DumbPolicy will do nothing, but is used to control traffic
DumbSpecialPolicy::DumbSpecialPolicy() : Policy(){
    this->timer=new UpdateDumbSpecialVirtualQueue(this);

```

```

}
/*-----
void DumbSpecialPolicy::applyMeter(policyTableEntry *policy, Packet *pkt)
Do nothing
// "flag" indica se vem do update ou vem do applymeter mesmo
// flag = false -> timer
// flag = true -> applymeter
-----*/

void DumbSpecialPolicy::applyMeter(policyTableEntry *policy, Packet *pkt, bool flag) {
    //policy->arrivalTime = Scheduler::instance().clock();
    edgeQueue *qm = policy->qmother;
    double now = Scheduler::instance().clock();
    double bytesVirtual;
    hdr_cmn* hdr;
    ofstream fileFD;
    int size=0;
    if (pkt!=NULL) {
        hdr=hdr_cmn::access(pkt);
        size=hdr->size();
    } else
        size=64;
    fileFD.open("QueueVirtual_.tr", ios::app);
    //update Timer
    if (timer!=NULL && pkt!=NULL && !timer->init) {
        timer->setPolicyHandle(policy);
        timer->resched(TimerT);
    }
    if (!flag) {
        //printf("LOL CIR = %f\n", (double) policy->cir);
        bytesVirtual = (double) policy->cir * TimerT;
        if (policy->QVirtual_ - bytesVirtual > 0)
            policy->QVirtual_ -= bytesVirtual;
        else
            policy->QVirtual_ = 0.0;
        return;
    }
    /*printf("now %f Qreal%d %d\n",now, policy->codePt, qm->redq_[1].getRealLength());
    printf("now %f Qreal%d %d\n",now, policy->codePt, qm->redq_[1].getRealLength() * 64);
    printf("now dsPolicy Qreal %f Qvirtual %f\n",now, policy->codePt, qm->redq_[2].qlen*1.0);
    */
    if (qm->numQueues_>=2 && qm->redq_[2].qlen < 2500 && flag) {
        policy->QVirtual_ += (double) size;
        if (policy->QVirtual_ > 3000 * (double) size)
            policy->QVirtual_ = 3000 * (double) size;
        fileFD << now << " " << policy-> QVirtual_ << endl;
    }
    policy->arrivalTime = now;
    policy->timeref = now;
    fileFD.close();
}
/*-----

int DumbSpecialPolicy::applyPolicer(policyTableEntry *policy, int initialCodePt, Packet *pkt)
Always return the initial codepoint.
-----*/

int DumbSpecialPolicy::applyPolicer(policyTableEntry *policy, policerTableEntry *policer, Packet *pkt) {
    return (policer->initialCodePt);
}

```

```

// The end of DumbSpecialPolicy
(...)
// Begin of Token Bucket.
/*-----*/
void TBPoly::applyMeter(policyTableEntry *policy, Packet *pkt)
Pre: policy's variables cBucket, cir, cbs, and arrivalTime hold valid values.
Post: Increments policy's Token Bucket state variable cBucket according to the elapsed time since the last
packet arrival. cBucket is filled at a rate equal to CIR, capped at an upper bound of CBS. This method also
sets arrivalTime equal to the current simulator time.
-----*/
void TBPoly::applyMeter(policyTableEntry *policy, Packet *pkt, bool a) {
    double now = Scheduler::instance().clock();
    double tokenBytes;
    tokenBytes = (double) policy->cir * (now - policy->arrivalTime);
    if (policy->cBucket + tokenBytes <= policy->cbs)
        policy->cBucket += tokenBytes;
    else
        policy->cBucket = policy->cbs;
    policy->arrivalTime = now;
}
/*-----*/
int TBPoly::applyPolicer(policyTableEntry *polimarcy, int initialCodePt,
    Packet* pkt)
Pre: policy points to a policytableEntry that is using the Token Bucket policer and whose state variable
(cBucket) is up to date. pkt points to a newly-arrived packet.
Post: If policy's cBucket is at least as large as pkt's size, cBucket is decremented by that size and the initial
code point is retained. Otherwise, the code point is downgraded.
Returns: A code point to apply to the current packet.
Uses: Method downgradeOne().
-----*/
int TBPoly::applyPolicer(policyTableEntry *policy, policerTableEntry *policer, Packet* pkt) {
    hdr_cmh* hdr = hdr_cmh::access(pkt);
    double size = (double) hdr->size();
    if (!policer)
        printf("policer=NULL\n");
    if ((policy->cBucket - size) >= 0) {
        policy->cBucket -= size;
        return (policer->initialCodePt);
    } else
        return (policer->downgrade1);
}
//Token bucket Special
TBSpecialPolicy::TBSpecialPolicy() : Policy() {
    this->timer=new UpdateVirtualQueue(this);
}
//"flag" indica se vem do update ou vem do applymeter mesmo
//flag = false -> timer
//flag = true -> applymeter
void TBSpecialPolicy::applyMeter(policyTableEntry *policy, Packet *pkt, bool flag) {
    edgeQueue *qm = policy->qmother;
    double now = Scheduler::instance().clock();
    double tokenBytes;
    ofstream fileFD;
    hdr_cmh* hdr;
    int size=0;
    if (pkt!=NULL) {
        hdr=hdr_cmh::access(pkt);

```

```

        size=hdr->size();
    }
    else
        size=64; // default tamanho dos pacotes 64
    fileFD.open("QueueVirtualTB_tr", ios::app);
    //update Timer
    if (timer!=NULL && pkt!=NULL && !timer->init) {
        timer->setPolicyHandle(policy);
        timer->resched(TimerT);
    }
    if (!flag) {
        tokenBytes = (double) policy->cir * TimerT;
        if(policy->QVirtual_ - tokenBytes > 0)
            policy->QVirtual_ -= tokenBytes;
        else
            policy->QVirtual_ = 0.0;
        return;
    }
    if (qm->numQueues_>=2 && flag) {
        policy->QVirtual_ += (double) size;
        fileFD << now << " " << policy-> QVirtual_ << endl;
        //printf("now dsPolicy = QVirtual_ %f \n", policy->QVirtual_);
    }
    if (policy->cBucket + tokenBytes <= policy->cbs){
        policy->cBucket += tokenBytes;
    }
    else
        policy->cBucket = policy->cbs;
    //printf("Atualizei %f\n", now);
    policy->timeref = now;
    policy->arrivalTime = now;
    fileFD.close();
}
// TBSpecialPolicy
int TBSpecialPolicy::applyPolicer(policyTableEntry *policy, policerTableEntry *policer, Packet* pkt) {
    hdr_cmh* hdr = hdr_cmh::access(pkt);
    double size = (double) hdr->size();
    if(!policer)
        printf("policer=NULL\n");
    if ((policy->cBucket - size) >= 0) {
        policy->cBucket -= size;
        return(policer->initialCodePt);
    } else
        return(policer->downgrade1);
}
// End of Tocken Bucket.
(...)
void UpdateVirtualQueue::resched(double delay) {
    //printf("ola!\n");
    TimerHandler::resched(delay);
}
void UpdateVirtualQueue::expire(Event *e) {
    if (policy!=NULL) {
        //printf("%f - vou executar timer TBSpecial!\n",Scheduler::instance().clock());
        obj->applyMeter(policy,NULL, false);
        this->resched(TimerT);
    }
}

```

```

}
void UpdateDumbSpecialVirtualQueue::resched(double delay) {
    //printf("ola!\n");
    TimerHandler::resched(delay);
}
void UpdateDumbSpecialVirtualQueue::expire(Event *e) {
    if (policy!=NULL) {
        //printf("%f - vou executar timer DumbSpecial !\n",Scheduler::instance().clock());
        obj_->applyMeter(policy,NULL, false);
        this->resched(TimerT);
    }
}
// End of FW

```

A2.2 dsEdge.cc

```

#include <fstream>
#include <string>

#include "dsEdge.h"
#include "dsPolicy.h"
#include "packet.h"
#include "tcp.h"
#include "random.h"
#include "dsQueueManager.h"

/*-----
class edgeClass
-----*/
static class edgeClass : public TclClass {
public:
    edgeClass() : TclClass("Queue/dsRED/edge") {}
    TclObject* create(int, const char*const*) {
        return (new edgeQueue);
    }
} class_edge;

/*-----
edgeQueue() Constructor.
-----*/
edgeQueue::edgeQueue() {
    NumMarkRules=0;
}

/*-----
void enqueue(Packet* pkt)
Post: The incoming packet pointed to by pkt is marked with an appropriate code
point (as handled by the marking method) and enqueued in the physical and
virtual queue corresponding to that code point (as specified in the PHB
Table).
Uses: Methods Policy::mark(), lookupPHBTable(), and redQueue::enqueue().
-----*/
void edgeQueue::enqueue(Packet* pkt) {

```

```

int codePt;

// Mark the packet with the specified priority:
mark(pkt);
codePt = policy.mark(pkt);
dsREDQueue::enqueue(pkt);
hdr_cmn* hdr = hdr_cmn::access(pkt);
qm.p[qm.PolicyCount] = policy.getPolicyTableEntry(codePt);

for(int i = 0; i < qm.PolicyCount; i++)
{
    if(qm.p[i]->codePt == codePt)
    {
        qm.p[i]->byts = qm.p[i]->byts + (double) hdr->size();
    }
}
}
/*daniel ini*/
/*-----
Packet deque()
*-----*/
Packet *edgeQueue::deque() {
    Packet* p = dsREDQueue::deque();

    if(p != NULL)
    {
        //printf("Tou na dsEDGE.cc a fazer deque\n");
        hdr_cmn* hdr = hdr_cmn::access(p);
        hdr_ip* iph = hdr_ip::access(p);

        int codePt = iph->prio();

        for(int i = 0; i < qm.PolicyCount; i++)
        {
            if(qm.p[i]->codePt == codePt)
            {
                qm.p[i]->byts = qm.p[i]->byts - (double) hdr->size();
            }
        }
    }
    return p;
}
/*daniel fini*/
/*-----
void addMarkRule()
*-----*/
void edgeQueue::addMarkRule(int argc, const char*const* argv) {

    if (NumMarkRules == MAX_MARK_RULES)
        printf("ERROR: Max number of mark rules limit exceeded\n");
    else {
        if (argc==7) {
            MarkRulesTable[NumMarkRules].DSCP = (int)strtod(argv[2], NULL);
            MarkRulesTable[NumMarkRules].saddr = (int)strtod(argv[3], NULL);
            MarkRulesTable[NumMarkRules].daddr = (int)strtod(argv[4], NULL);

```

```

const char* PacketType=argv[5];
if (strcmp(PacketType, "tcp") == 0)
    MarkRulesTable[NumMarkRules].pType = PT_TCP;
else if (strcmp(PacketType, "ack") == 0)
    MarkRulesTable[NumMarkRules].pType = PT_ACK;
else if (strcmp(PacketType, "udp") == 0)
    MarkRulesTable[NumMarkRules].pType = PT_UDP;
else if (strcmp(PacketType, "any") == 0)
    MarkRulesTable[NumMarkRules].pType = PT_NTTYPE; //we use as every packet meaning
else {
    printf("in addMarkRule Application Packet type %s doesn't match\n",PacketType);
    MarkRulesTable[NumMarkRules].pType = PT_NTTYPE;
}

const char* AppType=argv[6];
if (strcmp(AppType, "telnet") == 0)
    MarkRulesTable[NumMarkRules].appType = PT_TELNET;
else if (strcmp(AppType, "ftp") == 0)
    MarkRulesTable[NumMarkRules].appType = PT_FTP;
else if (strcmp(AppType, "http") == 0)
    MarkRulesTable[NumMarkRules].appType = PT_HTTP;
else if (strcmp(AppType, "audio") == 0)
    MarkRulesTable[NumMarkRules].appType = PT_AUDIO;
else if (strcmp(AppType, "realaudio") == 0)
    MarkRulesTable[NumMarkRules].appType = PT_REALAUDIO;
else if (strcmp(AppType, "cbr") == 0)
    MarkRulesTable[NumMarkRules].appType = PT_CBR;
else if (strcmp(AppType, "any") == 0)
    MarkRulesTable[NumMarkRules].appType = PT_NTTYPE; //we use as every packet meaning
else { printf("in addMarkRule packet type %s doesn't match\n",AppType);
    MarkRulesTable[NumMarkRules].appType = PT_NTTYPE;
}

    NumMarkRules++;
}
else printf("ERROR: wrong number of parameters in addMarkRule\n");
}
}

/*-----
void mark(Packet *pkt)
-----*/

void edgeQueue::mark(Packet *pkt) {
    hdr_ip* iph=hdr_ip::access(pkt);
    hdr_cmh* cmh=hdr_cmh::access(pkt);
    int marked=0;
    int MarkRule;

    for (MarkRule=0; MarkRule<NumMarkRules; MarkRule++)
        if (( (MarkRulesTable[MarkRule].saddr==iph->saddr()) || ( MarkRulesTable[MarkRule].saddr==-1) )
        &&((MarkRulesTable[MarkRule].daddr==iph->daddr()) || ( MarkRulesTable[MarkRule].daddr==-1) )
        &&((MarkRulesTable[MarkRule].appType==cmh->app_type()) ||(MarkRulesTable[MarkRule].appType==
        PT_NTTYPE))
        &&((MarkRulesTable[MarkRule].pType==cmh->pType()) || (MarkRulesTable[MarkRule].pType==
        PT_NTTYPE)))
        {
            iph->prio_ = MarkRulesTable[MarkRule].DSCP;

```

```

        marked=1; break;
    }
    //if (iph->prio_==1)
    // printf(" marked %d tabType %d pcktype %d tab App %d apptype %d\n", iph->prio_,
MarkRulesTable[MarkRule].pType,cmn->ptype(),MarkRulesTable[MarkRule].appType,cmn-
>app_type());

    if (!marked)
        iph->prio_ = 0; // Default PHB; Always define a queue for this aggregate
}

/*-----
int command(int argc, const char*const* argv)
    Commands from the ns file are interpreted through this interface.
-----*/
int edgeQueue::command(int argc, const char*const* argv) {
    if (strcmp(argv[1], "addPolicyEntry") == 0) {
        // Note: the definition of policy has changed.
        policy.addPolicyEntry(argc, argv, this);
        if (strcmp(argv[3], "TokenBucket") == 0) {
            qm.p[qm.PolicyCount] = policy.getPolicyTableEntry((int)strtod(argv[2],NULL));
            qm.PolicyCount++;
        }
        /* daniel ini */
        if (strcmp(argv[3], "TokenBucketSpecial") == 0) {
            qm.p[qm.PolicyCount] = policy.getPolicyTableEntry((int)strtod(argv[2],NULL));
            qm.PolicyCount++;
        }
        if (strcmp(argv[3], "DumbSpecial") == 0) {
            qm.p[qm.PolicyCount] = policy.getPolicyTableEntry((int)strtod(argv[2],NULL));
            qm.PolicyCount++;
        }
        /* daniel fini */
    }
    return(TCL_OK);
};

if (strcmp(argv[1], "addPolicerEntry") == 0) {
    // Note: the definition of policy has changed.
    /*daniel ini*/
    //policy.addPolicerEntry(argc, argv);
    policy.addPolicerEntry(argc, argv); //this->redq_
    /*daniel fim*/
    return(TCL_OK);
};

if (strcmp(argv[1], "addMarkRule") == 0) {
    addMarkRule(argc, argv);
    return(TCL_OK);
};

if (strcmp(argv[1], "printPolicyTable") == 0) {
    policy.printPolicyTable();
    return(TCL_OK);
}

if (strcmp(argv[1], "printPolicerTable") == 0) {
    policy.printPolicerTable();
}

```

```
        return(TCL_OK);
    }

    return(dsREDQueue::command(argc, argv));
};

/*-----
QueueManager() Constructor.
-----*/
QueueManager::QueueManager() {
    PolicyCount = 0;
}
```

ANEXO 3

```
#include <fstream>
#include <string>

#include "dsconsts.h"
#include "dsscheduler.h"
#include "dsPolicy.h"
//#include "dsEdge.h"

/* Scheduler class
*/

/* helpful functions */
#define max(x,y) ((x>y)?x:y)
#define min(x,y) ((x>y)?y:x)
#define MAXDOUBLE 100000000.0

dsScheduler::dsScheduler(int NQ)
{
    printf("int NQ\n");
    winLen=1;
    initMeter();
}

dsScheduler::dsScheduler(int NQ, double LBw)
{
    printf("int NQ, double LBw\n");
    winLen=1;
    initMeter();
}

dsScheduler::dsScheduler(int NQ, double LBw, const char* PFQSchedType)
{
    printf("int NQ, double LBw, const char* PFQSchedType\n");
    winLen=1;
    initMeter();
}

void dsScheduler::initMeter()
{
    for (int i=0; i<MAX_QUEUES; i++) {
        queueAvgRate[i]=0;
        for (int j=0; j<MAX_PREC; j++)
            qpAvgRate[i][j]=0;
    }
}

void dsScheduler::applyTSWMeter(int PSize, double *AvgRate, double *ArrTime)
{
    winLen = 1;
    double now, bytesInTSW, newBytes;
    bytesInTSW = *AvgRate * winLen;
```

```

newBytes = bytesInTSW+PSize;
now = Scheduler::instance().clock();
/*printf("PSize %d\n", PSize);

printf("newbytes %f\n", newBytes);
printf("now %f\n", now);
printf("ArrTime %f\n", ArrTime);
printf("winLen %f\n", winLen);*/
*AvgRate = newBytes/ (now - *ArrTime + winLen); // in bytes
//printf("AvgRate %f\n", *AvgRate);
*ArrTime = now;
}

void dsScheduler::UpdateDepartureRate(int Queue, int Prec, int PSize)
{
    for (int i=0; i<NumQueues; i++) {
        applyTSWMeter((i==Queue) ? PSize : 0, &queueAvgRate[i], &queueArrTime[i]);
        for (int j=0; j<MAX_PREC; j++)
            applyTSWMeter( ((i==Queue)&&(j==Prec)) ? PSize : 0, &qpAvgRate[i][j], &qpArrTime[i][j]);
    }
}

double dsScheduler::GetDepartureRate(int Queue, int Prec)
{
    //printf("daniel %d\n",Queue);
    if (Prec==-1)
    {
        double aux = queueAvgRate[Queue]*8;
        //printf("queueAvgRate[%d]*8 = %f\n",Queue, queueAvgRate[Queue]*8);
        return aux;
    }
    else
    {
        return(qpAvgRate[Queue][Prec]*8); // in bit per seconds
    }
}

dsRR::dsRR(int NQ):dsScheduler(NQ)
{
    NumQueues=NQ;
    for (int i=0; i<MAX_QUEUES; i++) queueLen[i]=0;
    qToDq=-1;
    Reset();
}

void dsRR::Reset()
{
}

int dsRR::DequeEvent()
{
    int i=0;
    qToDq = ((qToDq + 1) % NumQueues);
    while ((i < NumQueues) && (queueLen[qToDq] == 0)) {
        qToDq = ((qToDq + 1) % NumQueues);
        i++;
    }
}

```

```

    if (i==NumQueues) return(-1);
    else {
        queueLen[qToDq]--;
        return qToDq;
    }
}

void dsRR::EnqueEvent(Packet* pkt, int Queue)
{
    queueLen[Queue]++;
}

dsWRR::dsWRR(int NQ):dsScheduler(NQ)
{
    NumQueues=NQ;
    for (int i=0; i<MAX_QUEUES; i++) {queueLen[i]=0; queueWeight[i]=1;}
    qToDq=0;
    Reset();
}

void dsWRR::Reset()
{
    for (int i=0; i<MAX_QUEUES; i++) {wirrTemp[i]=0;}
}

int dsWRR::DequeEvent()
{
    int i=0;
    if (wirrTemp[qToDq]<=0){
        qToDq = ((qToDq + 1) % NumQueues);
        wirrTemp[qToDq] = queueWeight[qToDq] - 1;
    } else wirrTemp[qToDq] = wirrTemp[qToDq] -1;

    while ((i < NumQueues) && (queueLen[qToDq] == 0)) {
        wirrTemp[qToDq] = 0;
        qToDq = ((qToDq + 1) % NumQueues);
        wirrTemp[qToDq] = queueWeight[qToDq] - 1;
        i++;
    }
    if (i==NumQueues)
        return(-1);
    else {
        queueLen[qToDq]--;
        return qToDq;
    }
}

void dsWRR::EnqueEvent(Packet* pkt, int Queue)
{
    queueLen[Queue]++;
}

dsWIRR::dsWIRR(int NQ):dsScheduler(NQ)
{
    NumQueues=NQ;
    queuesDone = MAX_QUEUES;
    for (int i=0; i<MAX_QUEUES; i++) {queueLen[i]=0; queueWeight[i]=1;}
}

```

```

    qToDq=0;
    Reset();
}

void dsWIRR::Reset()
{
    for(int i=0;i<MAX_QUEUES;i++){
        sliceCount[i]=0;
        wirrTemp[i]=0;
        wirrqDone[i]=0;
    }
}

int dsWIRR::DequeEvent()
{
    int i=0;
    qToDq = ((qToDq + 1) % NumQueues);
    while ((i<NumQueues) && ((queueLen[qToDq]==0) || (wirrqDone[qToDq]))) {
        if (!wirrqDone[qToDq]) {
            queuesDone++;
            wirrqDone[qToDq]=1;
        }
        qToDq = ((qToDq + 1) % NumQueues);
        i++;
    }
    if (wirrTemp[qToDq] == 1) {
        queuesDone +=1;
        wirrqDone[qToDq]=1;
    }
    wirrTemp[qToDq]-=1;
    if(queuesDone >= NumQueues) {
        queuesDone = 0;
        for(i=0;i<NumQueues;i++) {
            wirrTemp[i] = queueWeight[i];
            wirrqDone[i]=0;
        }
    }
    if (i==NumQueues)
        return(-1);
    else {
        queueLen[qToDq]--;
        return(qToDq);
    }
}

void dsWIRR::printWRRcount()
{
    for (int i = 0; i < NumQueues; i++){
        printf("%d: %d %d %d.\n", i, sliceCount[i],queueLen[i],queueWeight[i]);
    }
}

void dsWIRR::EnqueEvent(Packet* pkt, int Queue)
{
    queueLen[Queue]++;
}

```

```

//*****PQ*****/
dsPQ::dsPQ(int NQ, double WinLen):dsScheduler(NQ, WinLen)
{
    printf("NQ %d\n", NQ);
    printf("WinLen %f\n", WinLen);

    NumQueues=NQ;
    winLen=WinLen;
    for (int i=0; i<MAX_QUEUES; i++) {
        queueMaxRate[i]=0;
        queueLen[i]=0;
    }
    Reset();
}

void dsPQ::Reset()
{
    for(int i=0;i<MAX_QUEUES;i++)
        queueArrTime[i] = 0.0;
}

int dsPQ::DequeEvent()
{
    int qToDq=0, i=0;

    while ((i < NumQueues) && ((queueLen[qToDq] == 0) ||
        ((queueAvgRate[qToDq] > queueMaxRate[qToDq]) && queueMaxRate[qToDq]))) {
        //printf("Entrei\n");
        i++;
        qToDq = i;
    }
    if (i == NumQueues) {
        i = qToDq = 0;
        while ((i < NumQueues) && (queueLen[qToDq] == 0)) {
            qToDq = ((qToDq + 1) % NumQueues);
            i++;
        }
    }
    if (i<NumQueues) {
        queueLen[qToDq]--;
        return qToDq;
    }
    else
    {
        return(-1);
        printf("PRIORITY SCHEDULER: no packet to be dequeued\n");
    } // no packet to be dequeued
}

void dsPQ::EnqueueEvent(Packet* pkt, int Queue)
{
    queueLen[Queue]++;
}

//*****/

dsSM::dsSM(int NQ, double WinLen, void *edge):dsScheduler(NQ, WinLen)

```

```

{
    debitoEF = 0.0;
    lasteventEF = 0.0;
    debitoAF = 0.0;
    lasteventAF = 0.0;
    debitoBE = 0.0;
    lasteventBE = 0.0;
    ContadorEF = 0;
    ContadorAF = 0;
    ContadorBE = 0;
    saiu_AF = 0;

    /*daniel ini*/
    auxiliar = edge;

    NumQueues=NQ;
    winLen=WinLen;
    for (int i=0; i<MAX_QUEUES; i++) {
        queueMaxRate[i]=0;
        queueLen[i]=0;
    }
    Reset();
}

void dsSM::Reset()
{
    for(int i=0;i<MAX_QUEUES;i++)
        queueArrTime[i] = 0.0;
}

int dsSM::DequeEvent()
{
    int qToDq=0, i=0;
    double BEQvirtual= 0.0;
    double BEREal= 0.0;
    double AFQvirtual= 0.0;
    double AFReal= 0.0;
    edgeQueue *tmp = (edgeQueue *)auxiliar;
    double now = Scheduler::instance().clock();
    double packetSize;
    if(packet != NULL)
    {
        hdr_cmn* hdr = hdr_cmn::access(packet);
        packetSize = hdr->size();
    }
    //packetSize = 64.0;
    double a = 0.05;
    int retvalue = 0;

    ofstream fileFD;
    fileFD.open("debitos.tr", ios::app);

    ofstream fileFD_Contador;
    fileFD_Contador.open("Contador.tr", ios::app);

    if(tmp){

```

```

if(tmp->policy.getPolicyTableEntry(0)!=NULL)
{
    BEReal = queueLen[2]*packetSize;
    BEQvirtual = (tmp->policy.getPolicyTableEntry(0))->QVirtual_;
}
if(tmp->policy.getPolicyTableEntry(10)!=NULL)
{
    AFReal = queueLen[1]*packetSize;
    AFQvirtual = (tmp->policy.getPolicyTableEntry(10))->QVirtual_;
}
if(tmp->policy.getPolicyTableEntry(11)!=NULL)
{
    //policyTableEntry * p = tmp->policy.getPolicyTableEntry(10);
    AFReal = queueLen[1]*packetSize;
    if((tmp->policy.getPolicyTableEntry(11))->QVirtual_ != 0.0)
        printf("Iolidaada %f\n\n\n", (tmp->policy.getPolicyTableEntry(11))->QVirtual_);
    //printf("AFQvirtual %f\n", AFQvirtual);
    //printf("AFReal %f\n", AFReal);
}

//printf("BEQvirtual %f\n", BEQvirtual);
/*printf("BEReal %f\n", BEREAL);
printf ("now %f\n", now);
printf("AFReal %f\n", AFReal);
printf("AFQvirtual %f\n", AFQvirtual);*/

//Versão 1
//Se não for pacote EF
//if((BEQvirtual < BEREAL) && (AFQvirtual - AFReal) > 200 *64) ||
//if((BEQvirtual < BEREAL) && (BEReal > 0 && AFReal == 0))
/*if((AFQvirtual - AFReal) < 200*64 && AFReal > 0 )
{
    retvalue=1;    //Escalona pacote BE
}
else
{
    retvalue=2; //Escalona pacote AF
}
*/

//Versão 2
//BE a consumir recursos acima da quota?
/*if (BEQvirtual>BEReal)
{
    if(AFReal>0)
        retvalue = 1; //Escalona pacote de AF
    else
        retvalue = 2; //Escalona pacote de BE
}

// BE a consumir recursos abaixo da quota?
if (BEQvirtual<BEReal)
{
    if(AFReal<AFQvirtual && AFQvirtual - AFReal > 500*64)
    //if(AFQvirtual - AFReal > 100*64)
    {
        retvalue=2;    //Escalona pacote de BE
    }
}

```

```

else
{
    if(AFReal>0)
        retvalue = 1; //Escalona pacote de AF
    else
        retvalue = 2; //Escalona pacote de BE
}
}*/

//Versão 3
//BE ta a consumir recursos acima da cota
/*if (BEQvirtual>BEReal)
{
    //printf("BE ta a consumir recursos acima da cota\n\n");
    //dar prioridade a AF
    if(AFReal-0>AFQvirtual)
    {
        retvalue=1;
    }
}

//Escalonar BE
if (BEQvirtual<BEReal)
{
    if(AFReal-640>AFQvirtual)
    {
        //printf("dei a AF\n");
        retvalue=1;
    }
    else
        retvalue = 2;
}*/

//Versão 4
if((BEQvirtual < BEREal && (AFQvirtual-AFReal) > 250 * 64 ) || (BEReal > 0 && AFReal == 0))
{
    retvalue=2; //Escalona pacote BE
}
else
{
    retvalue=1; //Escalona pacote AF
}

//Versão 5
/*if( ( (BEQvirtual < BEREal || (BEReal > (2500 - 500)*64 ))
    && (AFQvirtual - AFReal) > 150*64) || (BEReal > 0 && AFReal == 0))
{
    retvalue=2; //Escalona pacote BE
}
else
{
    retvalue=1; //Escalona pacote AF
}*/

//Versão 6
/*if((AFQvirtual-AFReal) > 500 * 64 || (BEReal > 0 && AFReal == 0) )
{

```

```

        retvalue=2; //Escalona pacote BE
    }
    else
    {
        retvalue=1; //Escalona pacote AF
    }
*/

}
while ((i < NumQueues) && ((queueLen[qToDq] == 0) ||
    ((queueAvgRate[qToDq] > queueMaxRate[qToDq]) && queueMaxRate[qToDq]))) {
    i++;
    qToDq = i;
}
if (i == NumQueues) {
    i = qToDq = 0;

    while ((i < NumQueues) && (queueLen[qToDq] == 0)) {
        qToDq = ((qToDq + 1) % NumQueues);
        i++;
    }
}
/*daniel ini*/
if (i < NumQueues) {
    if (qToDq > 0 && retvalue != 0)
    {
        qToDq = retvalue;
        //printf("retvalue %d\n", retvalue);
    }
    if (qToDq == 0)
    {
        ContadorEF = ContadorEF + 64;
        if (lasteventEF != now && packetSize > 0)
        {
            double debinst = packetSize / (now - lasteventEF);
            debitoEF = ContadorEF / now;
        }
        lasteventEF = now;
    }
    if (qToDq == 1)
    {
        ContadorAF = ContadorAF + 64;
        if (lasteventAF != now && packetSize > 0)
        {
            double debinst = packetSize / (now - lasteventAF);
            debitoAF = ContadorAF / now;
        }
        lasteventAF = now;
    }
    if (qToDq == 2)
    {
        ContadorBE = ContadorBE + packetSize;
        if (lasteventBE != now && packetSize > 0)
        {
            double debinst = packetSize / (now - lasteventBE);
            debitoBE = ContadorBE / now;

```

```

        }
        lasteventBE = now;
    }
    fileFD << now << " " << debitoEF << " " << debitoAF << " " << debitoBE << endl;
    fileFD_Contador << now << " " << ContadorEF << " " << ContadorAF << " " << ContadorBE <<
endl;
    queueLen[qToDq]--;
    return qToDq;
}
/*daniel fini*/

else
{
    return(-1);
    printf("PRIORITY SCHEDULER: no packet to be dequeued\n");
} // no packet to be dequeued
}

void dsSM::EnqueueEvent(Packet* pkt, int Queue)
{
    packet = pkt;
    queueLen[Queue]++;
}

/*****/

dsWFQ::dsWFQ(int NQ, double LBw):dsScheduler(NQ, LBw)
{
    NumQueues=NQ;
    LinkBandwidth=LBw;
    for (int i=0; i<MAX_QUEUES; i++) {
        fs_[i].weight_=1;
        fs_[i].B=0;
    }
    v_time = 0;
    idle = 1;
    wfq_event = 0;
    sum = 0;
    last_vt_update = 0; sessionDelay=0;
    GPSDeparture=-1;
    Reset();
}

void dsWFQ::Reset()
{
    for (int i=0; i<MAX_QUEUES; i++)
        fs_[i].finish_t=0;
    idle=1;
}

void dsWFQ::EnqueueEvent(Packet* pkt, int queue)
{
    double now=Scheduler::instance().clock();
    // virtual time update. Formula 10
    if(idle) {
        v_time=0;
        last_vt_update=now;
    }
}

```

```

        idle=0;
    }
    else
    {
        v_time=v_time+(now-last_vt_update)/sum;
        last_vt_update=now;
    }

    // finish time computation. Formula 11
    fs_[queue].finish_t = (max(fs_[queue].finish_t,v_time))+
        ((double)hdr_cmn::access(pkt)->size())*8/(fs_[queue].weight_*LinkBandwith);

    // update sum and B
    if( (fs_[queue].B++)==0 ) sum+=fs_[queue].weight_;
    // insertion in both lists
    fs_[queue].GPS.push(fs_[queue].finish_t);
    fs_[queue].PGPS.push(sessionDelay+fs_[queue].finish_t);

    // schedule next departure in the GPS reference system
    if (wfq_event!=0) {
        Scheduler::instance().cancel(wfq_event);
        delete wfq_event;
    }
    scheduleWFQ();
}

void dsWFQ::handle(Event *e)
{
    double now = Scheduler::instance().clock();
    //double fTime=fs_[GPSDeparture].GPS.front();

    //update virtual time
    v_time=v_time+(now-last_vt_update)/sum;
    last_vt_update=now;

    //extract packet in GPS system
    if (GPSDeparture!=-1) {
        fs_[GPSDeparture].GPS.pop();
        if (--fs_[GPSDeparture].B == 0) sum-=fs_[GPSDeparture].weight_;
        if ( fabs(sum) < 0.00001 ) sum=0;
        if (sum==0)
        {
            Reset();
            sessionDelay=now;
        }
    }
    else
        printf("DEBUG: ERROR, no active sessions \n");

    // if GPS is not idle, schedule next GPS departure
    delete e;
    if(!idle)
        scheduleWFQ();
    else
        wfq_event=NULL;
}

```

```

void dsWFQ::scheduleWFQ()
{
    wfq_event=new Event();
    double tmp;
    GPSDeparture=-1;

    double minFinishTime=MAXDOUBLE;
    for (int i=0; i<NumQueues; i++)
        if (!fs_[i].GPS.empty()) {
            if (fs_[i].GPS.front()<minFinishTime) {
                GPSDeparture=i;
                minFinishTime=fs_[GPSDeparture].GPS.front();
            }
        }
        if (GPSDeparture!=-1) {
            tmp=(minFinishTime-v_time)*sum;
            // following line is there to recover errors due to finite precision
            if (tmp<0) tmp=0;
            Scheduler::instance().schedule((Handler *)this,wfq_event,tmp);
        }
        else
            printf("DEDUG: ERROR no event to schedule \n");
    }

int dsWFQ::DequeEvent()
{
    int qToDq=-1;
    double minFinishTime=MAXDOUBLE;
    for (int i=0; i<NumQueues; i++)
        if (!fs_[i].PGPS.empty())
            if (fs_[i].PGPS.front()<minFinishTime) {qToDq=i; minFinishTime=fs_[qToDq].PGPS.front();}

    if (qToDq!=-1) fs_[qToDq].PGPS.pop();
    return qToDq;
}

dsSCFQ::dsSCFQ(int NQ, double LBw):dsScheduler(NQ, LBw)
{
    NumQueues=NQ;
    LinkBandwidth=LBw;
    for (int i=0; i<MAX_QUEUES; i++) {
        session[i].weight=1;
    }
    Reset();
}

void dsSCFQ::Reset()
{
    for (int i=0; i<MAX_QUEUES; i++) {
        session[i].label=0;
    }
    tlabel=0;
}

void dsSCFQ::EnqueEvent(Packet* pkt, int queue)
{
    session[queue].label = (max(session[queue].label, tlabel))+

```

```

        ((double)hdr_cmn::access(pkt)->size())/session[queue].weight/(LinkBandwidth/8.0);
        session[queue].SessionQueue.push(session[queue].label);
    }

    int dsSCFQ::DequeEvent()
    {
        int qToDq=-1;
        double minFinishTime=MAXDOUBLE;
        for (int i=0; i<NumQueues; i++)
            if ((!session[i].SessionQueue.empty())&&(session[i].SessionQueue.front()<minFinishTime)) {
                qToDq=i;
                minFinishTime=session[qToDq].SessionQueue.front();
            }
        if (qToDq!=-1)
        {
            tlabel=minFinishTime;
            session[qToDq].SessionQueue.pop();
        }
        return qToDq;
    }

    dsSFQ::dsSFQ(int NQ, double LBw):dsScheduler(NQ, LBw)
    {
        NumQueues=NQ;
        LinkBandwidth=LBw;
        for (int i=0; i<MAX_QUEUES; i++) {
            flow[i].weight=1;
        }
        idle=1;
        MaxFinishTag=0.0;
        Reset();
    }

    void dsSFQ::Reset()
    {
        for (int i=0; i<MAX_QUEUES; i++) {
            flow[i].LastFinishTag=0;
        }
        V=0;
    }

    void dsSFQ::EnqueueEvent(Packet* pkt, int queue)
    {
        PacketTags.StartTag=max(V,flow[queue].LastFinishTag);
        PacketTags.FinishTag=PacketTags.StartTag+((double)hdr_cmn::access(pkt)-
>size())/flow[queue].weight/(LinkBandwidth/8.0);
        flow[queue].LastFinishTag=PacketTags.FinishTag;
        flow[queue].FlowQueue.push(PacketTags);
        idle=0;
    }

    int dsSFQ::DequeEvent()
    {
        int qToDq=-1;
        double MinStartTag=MAXDOUBLE;
        for (int i=0; i<NumQueues; i++) {

```

```

        PacketTags=flow[i].FlowQueue.front();
        if ((!flow[i].FlowQueue.empty())&&(PacketTags.StartTag<MinStartTag)) {
            qToDq=i;
            MinStartTag=PacketTags.StartTag;
        }
    }

    if (qToDq!=-1) {
        V=MinStartTag;
        MaxFinishTag=max(MaxFinishTag,PacketTags.FinishTag);
        flow[qToDq].FlowQueue.pop();
    }
    else if (idle==0) {
        MaxFinishTag=0;
        Reset();
        idle=1;
    }
    return qToDq;
}

dsWF2Qp::dsWF2Qp(int NQ, double LBw):dsScheduler(NQ, LBw)
{
    NumQueues=NQ;
    LinkBandwidth=LBw;
    /* initialize flow's structure */
    for (int i = 0; i < MAX_QUEUES; ++i) {
        flow[i].qcrSize    = 0;
        flow[i].weight     = 1;
        flow[i].S          = 0;
        flow[i].F          = 0;
    }
    V=0; lastTimeV=0;
    //Reset();
}

void dsWF2Qp::Reset()
{
}

void dsWF2Qp::EnqueueEvent(Packet* pkt, int flowId)
{
    int pktSize=hdr_cmh::access(pkt)->size();
    if (!flow[flowId].qcrSize) {
        /* If flow queue is empty, calculate start and finish times
        * paragraph 5.2 formula (23b) and (24)
        */
        flow[flowId].S = max(V, flow[flowId].F);
        flow[flowId].F = flow[flowId].S+pktSize/(flow[flowId].weight*LinkBandwidth/8);

        /* update system virtual clock
        * paragraph 5.2 formula (22)
        */
        double minS = flow[flowId].S;
        for (int i = 0; i < NumQueues; ++i)
            if ((flow[i].qcrSize)&&(flow[i].S<minS))
                minS=flow[i].S;
        V=max(minS,V);
    }
}

```

```

    }
    flow[flowId].flowQueue.push(pktSize);
    flow[flowId].qcrtSize+=pktSize;
}

int dsWF2Qp::DequeEvent()
{
    int i;
    int pktSize;
    double minF = MAXDOUBLE;
    int flowId = -1;
    double W = 0;

    /* look for the candidate flow with the earliest finish time */
    for (i = 0; i<NumQueues; i++){
        if (flow[i].qcrtSize) {
            W+=flow[i].weight;
            if ((flow[i].S<=V)&&(flow[i].F<minF)) {
                flowId = i;
                minF = flow[i].F;
            }
        }
    }
    if (flowId!=-1) {
        pktSize=flow[flowId].flowQueue.front();
        flow[flowId].qcrtSize-=pktSize;
        flow[flowId].flowQueue.pop();

        /* Set start and finish times of the remaining packets in the queue */
        if (!flow[flowId].flowQueue.empty()) {
            flow[flowId].S = flow[flowId].F;
            flow[flowId].F = flow[flowId].S +
flow[flowId].flowQueue.front()/(flow[flowId].weight*LinkBandwidth/8);
        }

        /* update the virtual clock */
        /* looking for min service time of eligibles (=active) flows */
        double now=Scheduler::instance().clock();
        double minS=MAXDOUBLE;
        for (i = 0; i < NumQueues; ++i)
            if ((flow[i].qcrtSize)&&(flow[i].S<minS)) minS=flow[i].S;
            if (minS==MAXDOUBLE) minS=0;

        /* provided service in the last period, this packet sent */

        W = (now-lastTimeV)/W;
        V = max(minS,(V+W));
        lastTimeV=now;
    }
    return(flowId);
}

dsLLQ::dsLLQ(int NQ, double LBw, const char* PFQSchedType):dsScheduler(NQ, LBw,
PFQSchedType)
{
    NumQueues=NQ;
    PFQAssignedBandwidth=LBw;

```

```

    if (strcmp(PFQSchedType, "WFQ")==0)
        xPFQ = new dsWFQ(NQ-1,PFQAssignedBandwith);
    else if (strcmp(PFQSchedType, "WF2Qp")==0)
        xPFQ = new dsWF2Qp(NQ-1,PFQAssignedBandwith);
    else if (strcmp(PFQSchedType, "SCFQ")==0)
        xPFQ = new dsSCFQ(NQ-1,PFQAssignedBandwith);
    else if (strcmp(PFQSchedType, "SFQ")==0)
        xPFQ = new dsSFQ(NQ-1,PFQAssignedBandwith);
    xPQ = new dsPQ(1,1);
// Reset();
}

void dsLLQ::Reset()
{
}

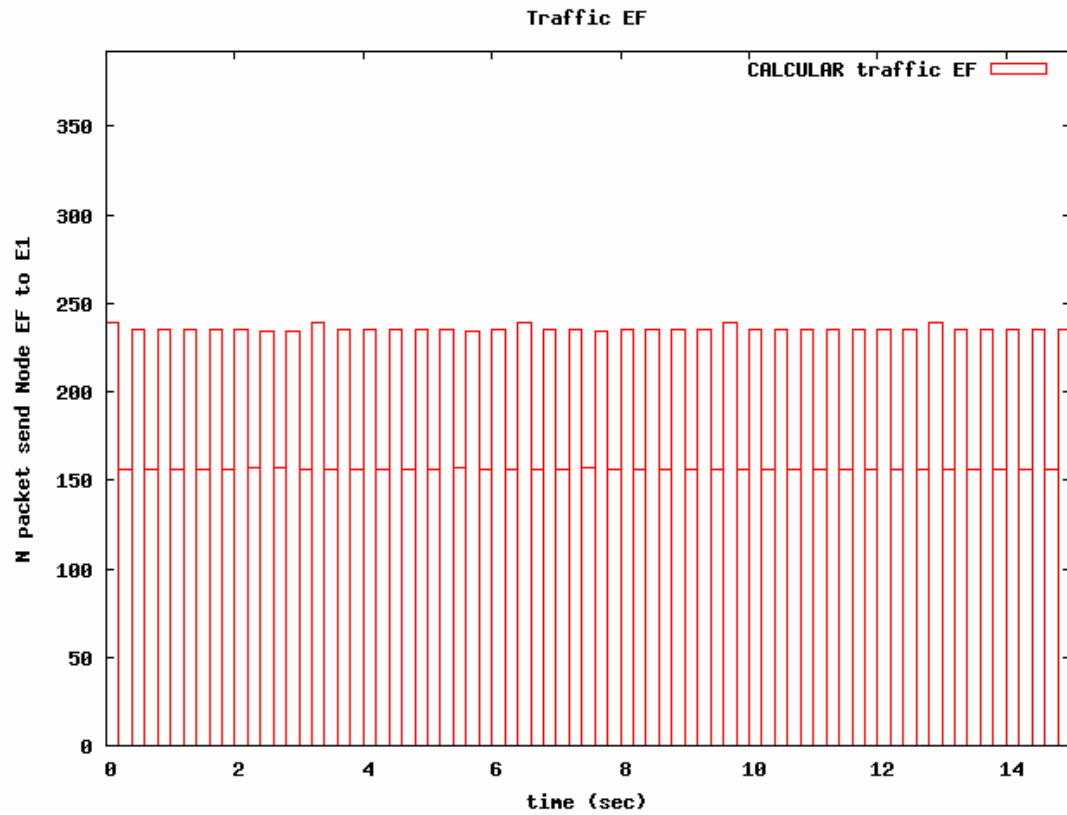
void dsLLQ::EnqueueEvent(Packet* pkt, int queue)
{
    if (queue==0) xPQ->EnqueueEvent(pkt, queue);
    else xPFQ->EnqueueEvent(pkt, queue-1);
}

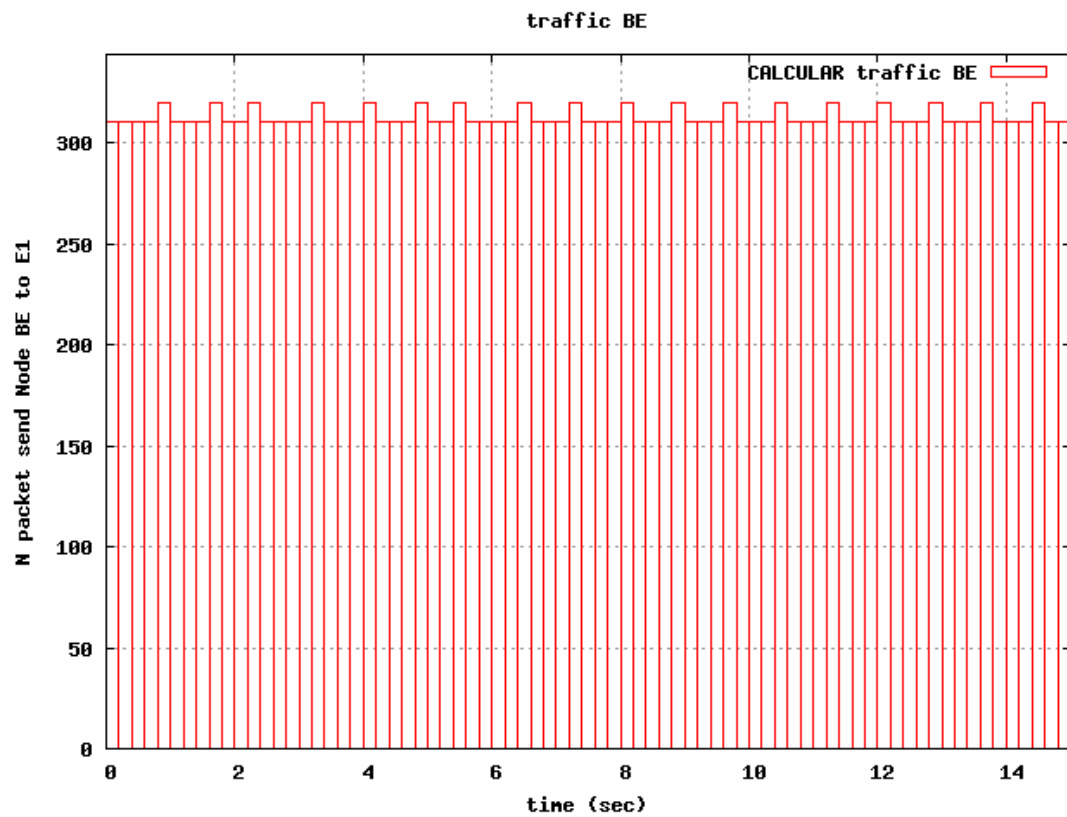
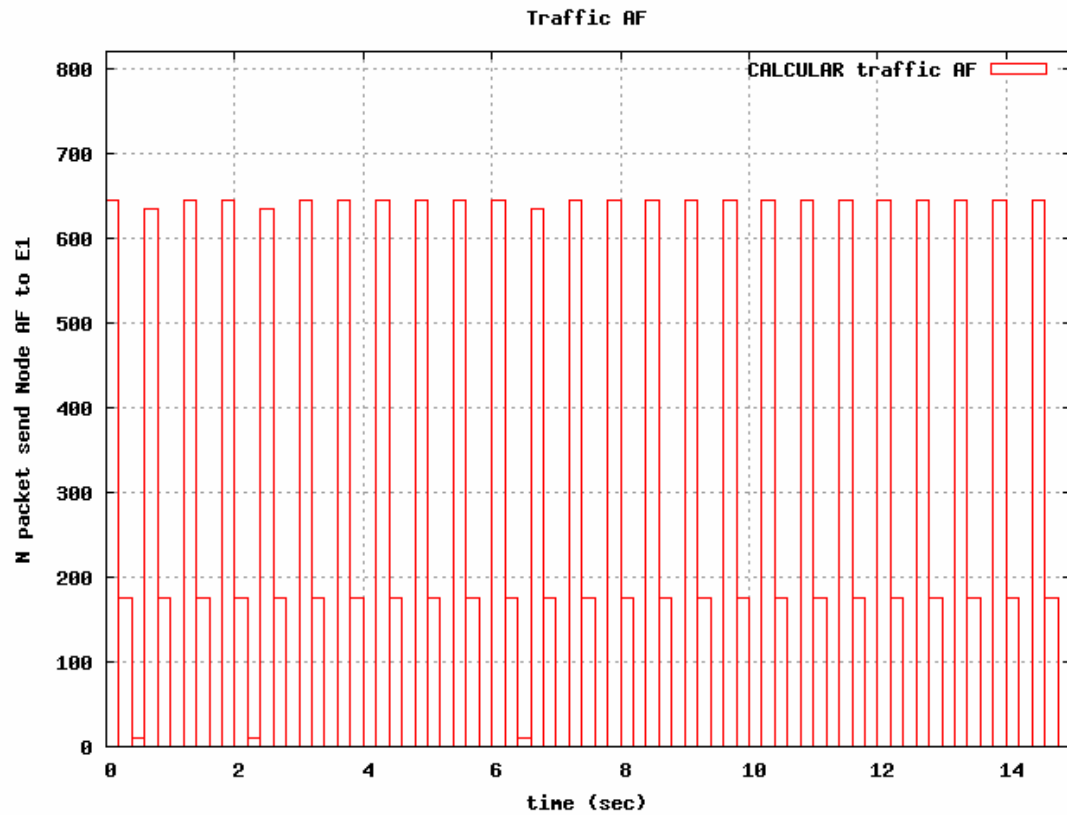
int dsLLQ::DequeEvent()
{
    int qToDq=xPQ->DequeEvent();
    if (qToDq==-1) {
        qToDq=xPFQ->DequeEvent();
        if (qToDq>-1)
            qToDq++;
    }
    return qToDq;
}

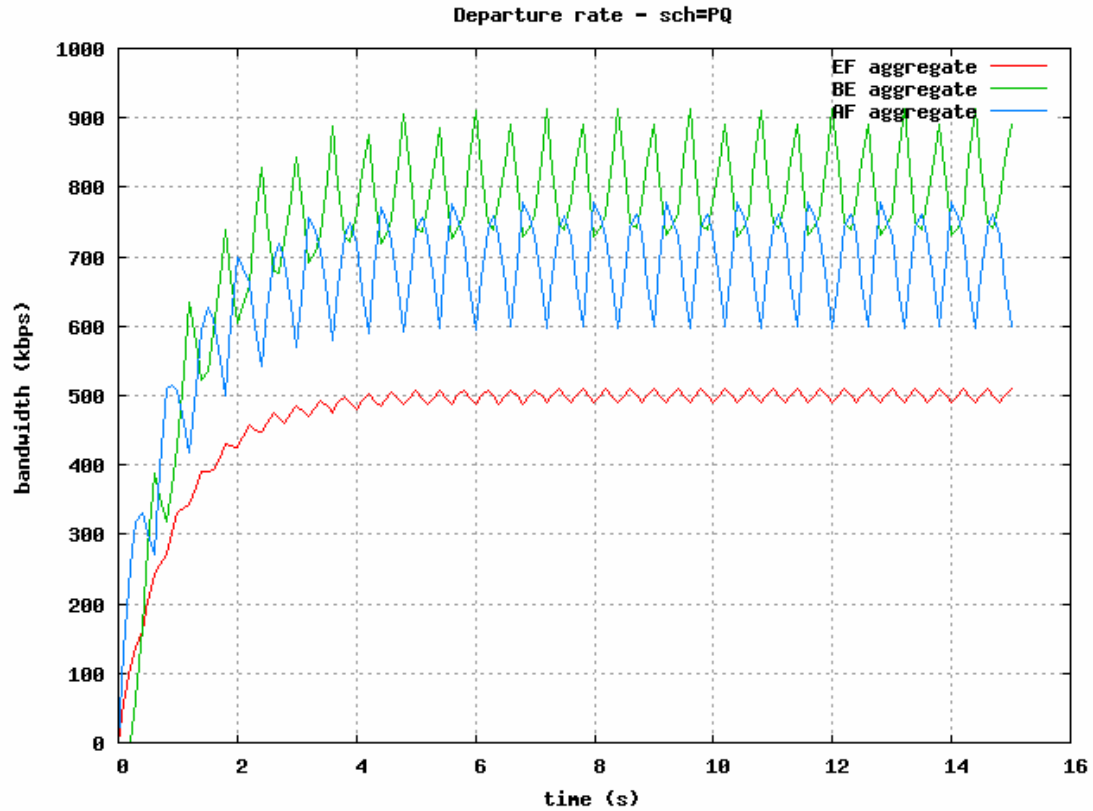
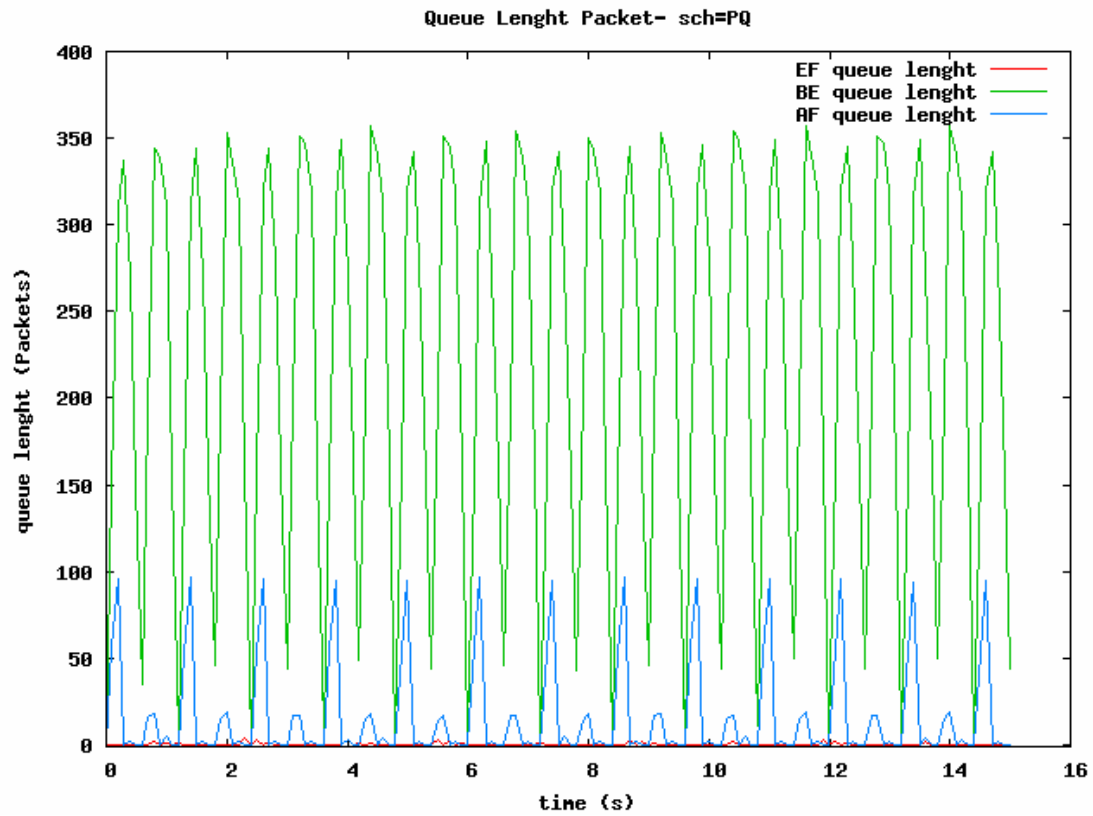
```

ANEXO 4

PQ - Próximo da Saturação







Input:

O número de pacotes gerados BE é de 23430 pkts -> foram enviados 799.744 kbps.

O número de pacotes gerados EF é de 14722 pkts -> foram enviados 502.511 kbps.

O número de pacotes gerados AF é de 20500 pkts -> foram enviados 699.733 kbps.

O número de pacotes perdidos é de 27 pkts.

Output:

Estatística de Pacotes

CP	TotPkts	TxPkts	ldrops	edrops
--	-----	-----	-----	-----
0	23270	100.00%	0.00%	0.00%
10	20500	100.00%	0.00%	0.00%
46	14575	100.00%	0.00%	0.00%
48	27	0.00%	0.00%	100.00%

All	58372	99.95%	0.00%	0.05%

O valor máximo do atraso IPDV de EF é 1.00 ms.

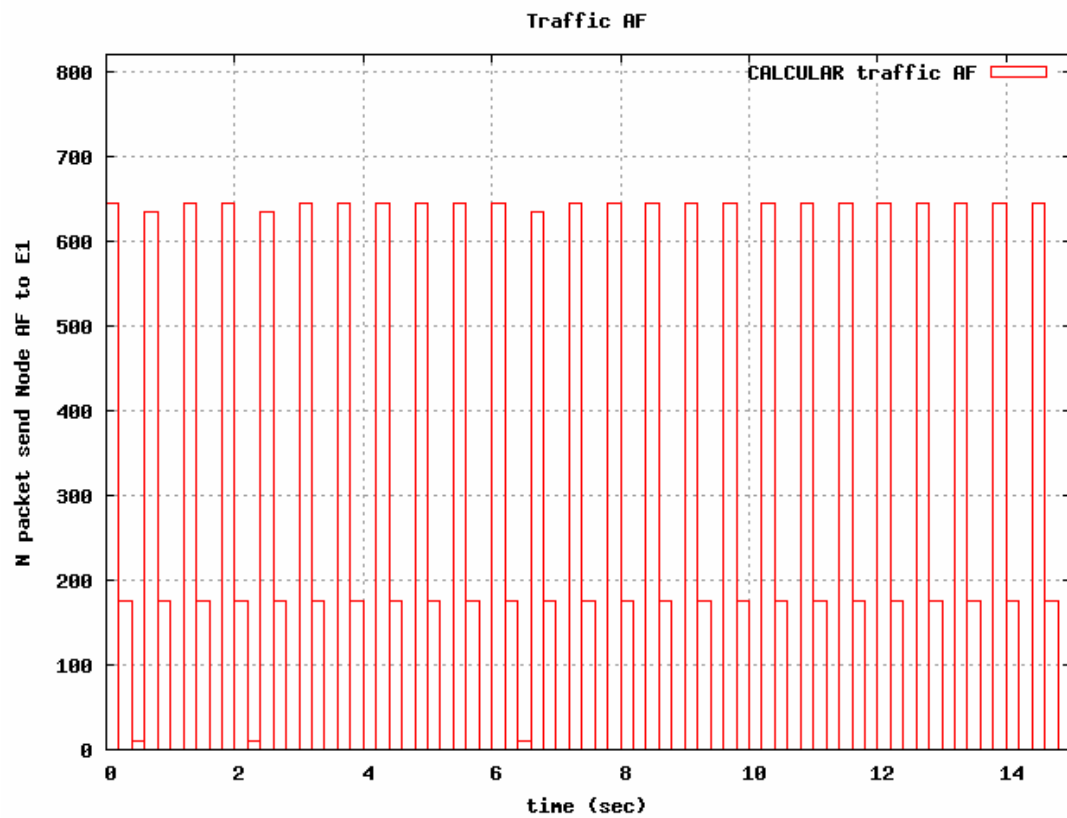
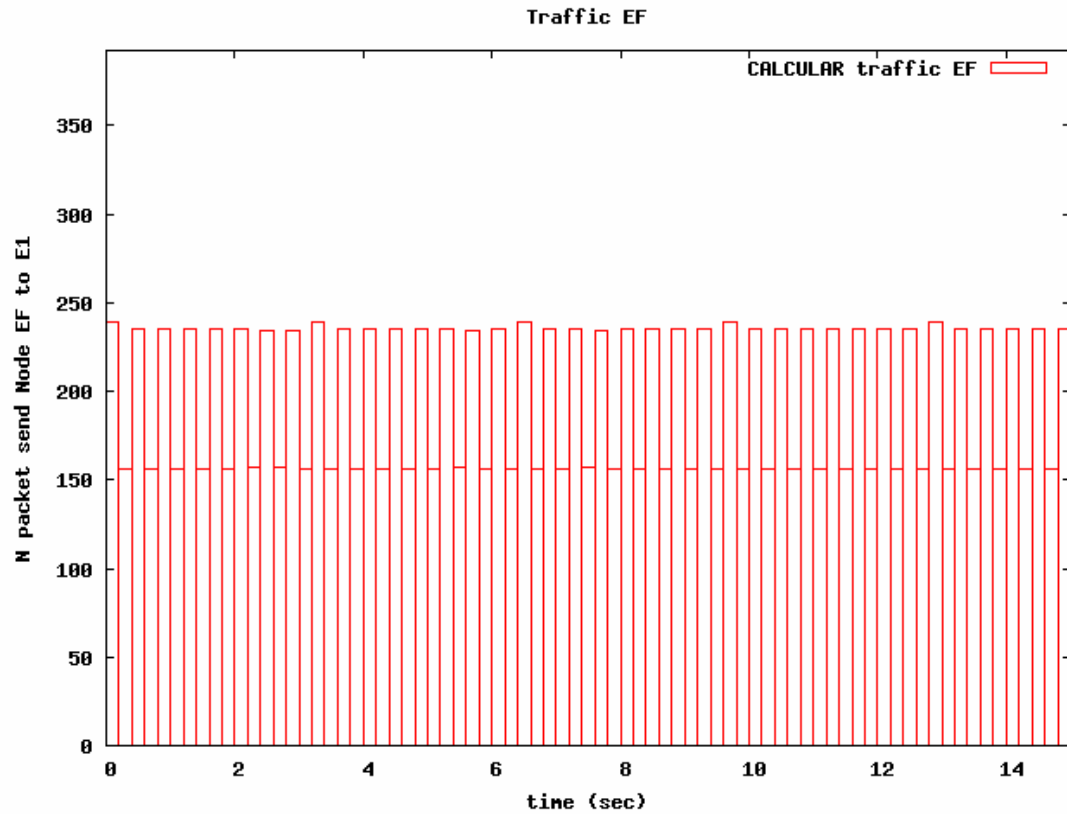
O valor máximo do atraso OWD de EF é 12.00 ms.

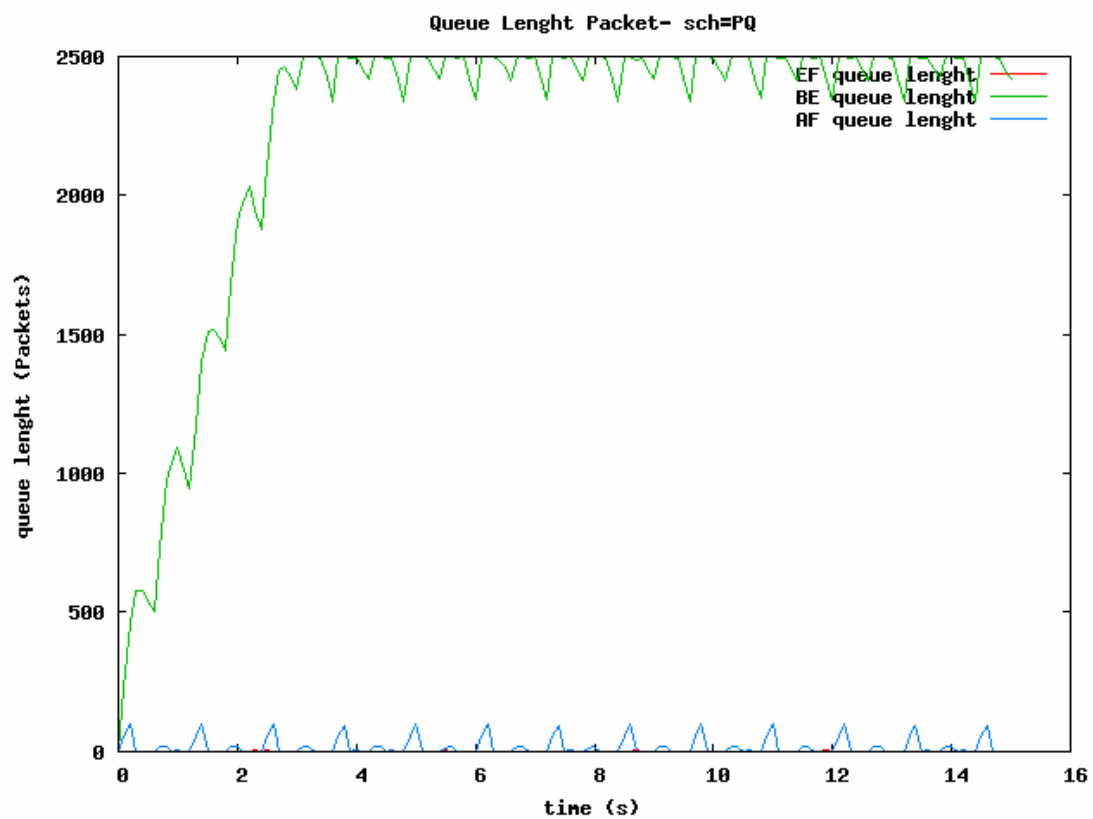
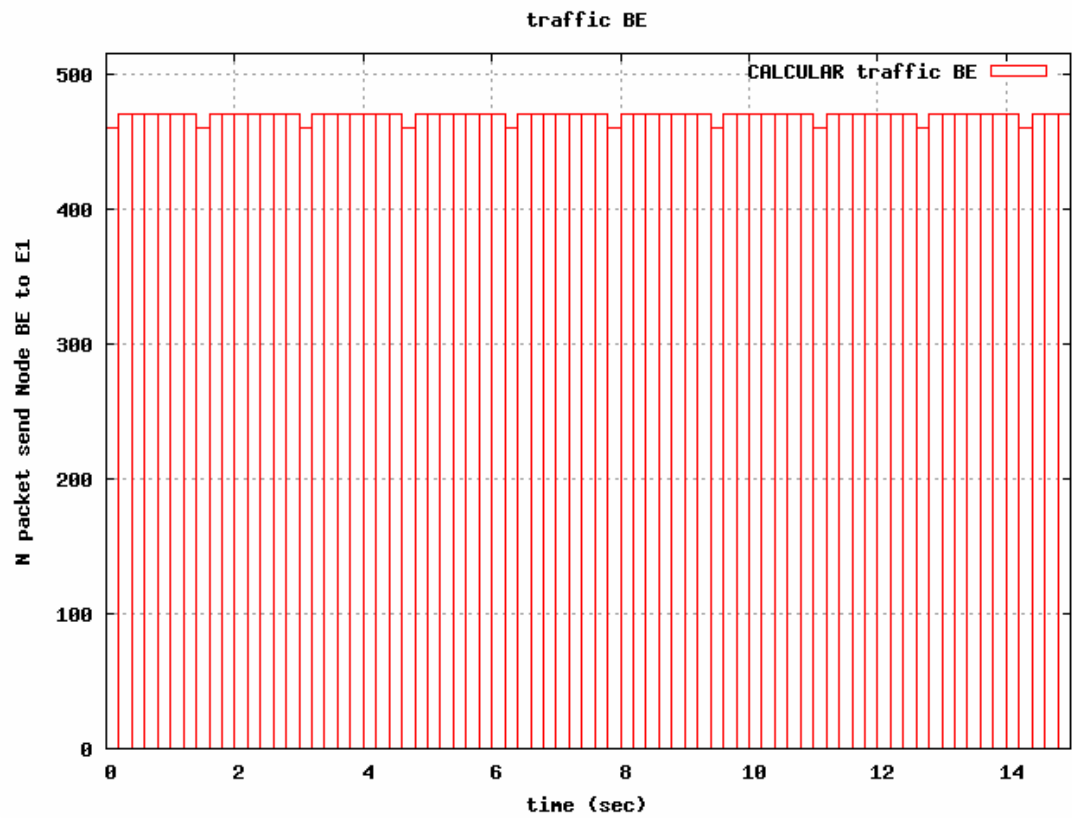
O valor máximo do atraso IPDV de AF é 5.00 ms.

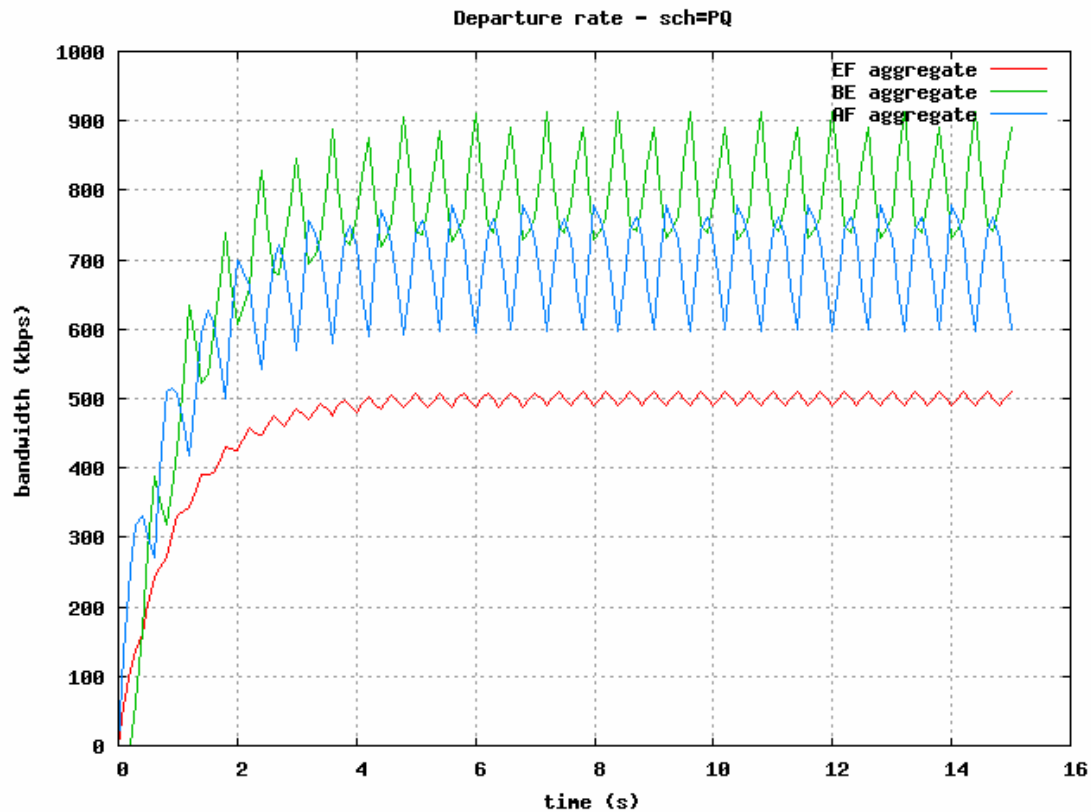
O valor máximo do atraso OWD de AF é 44.00 ms.

O valor máximo do atraso IPDV de BE é 243.00 ms.

O valor máximo do atraso OWD de BE é 260.00 ms.







Input:

O número de pacotes gerados BE é de 35150 pkts -> foram enviados 1199.78666667 kbps.

O número de pacotes gerados EF é de 14722 pkts -> foram enviados 502.510933333 kbps.

O número de pacotes gerados AF é de 20500 pkts -> foram enviados 699.733333333 kbps.

O número de pacotes perdidos é de 9361 pkts.

Output:

Estatística de Pacotes

CP	TotPkts	TxPkts	ldrops	edrops
--	-----	-----	-----	-----
0	34910	73.26%	26.74%	0.00%
10	20500	100.00%	0.00%	0.00%
46	14575	100.00%	0.00%	0.00%
48	27	0.00%	0.00%	100.00%

All	70012	86.63%	13.33%	0.04%

O valor máximo do atraso IPDV de EF é 2.00 ms.

O valor máximo do atraso OWD de EF é 12.00 ms.

O valor máximo do atraso IPDV de AF é 5.00 ms.

O valor máximo do atraso OWD de AF é 44.00 ms.

O valor máximo do atraso IPDV de BE é 245.00 ms.

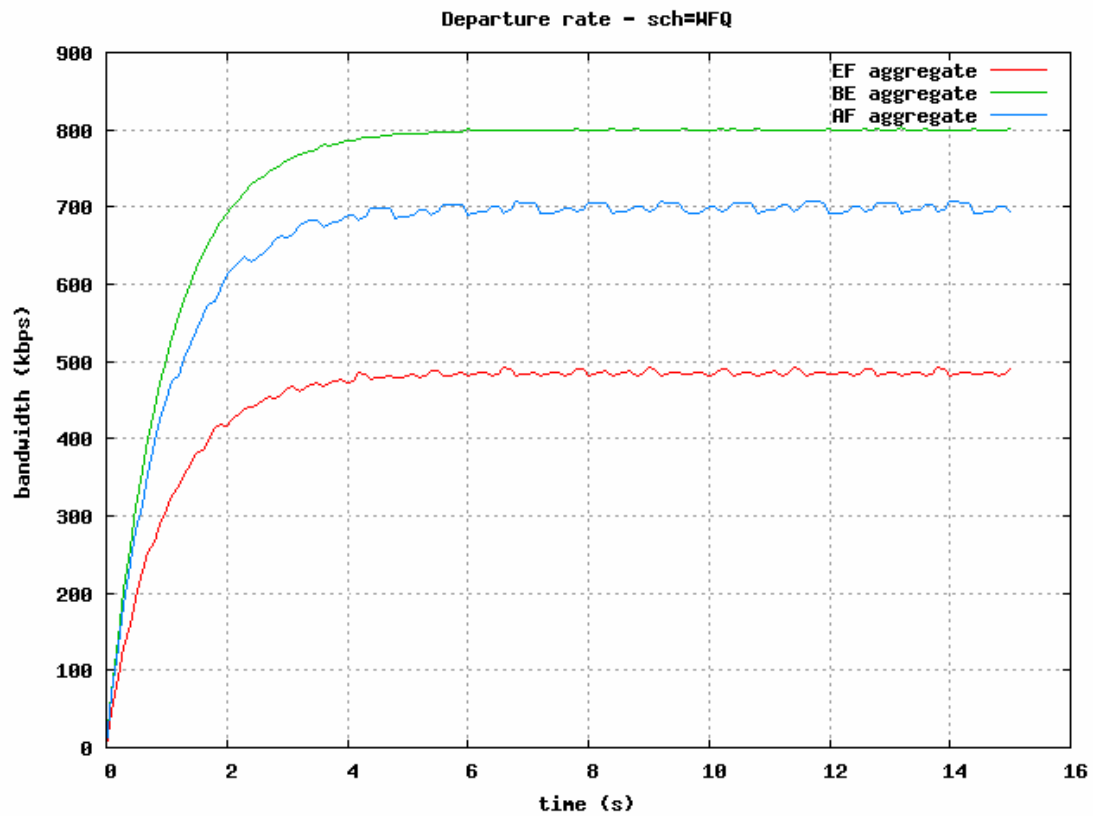
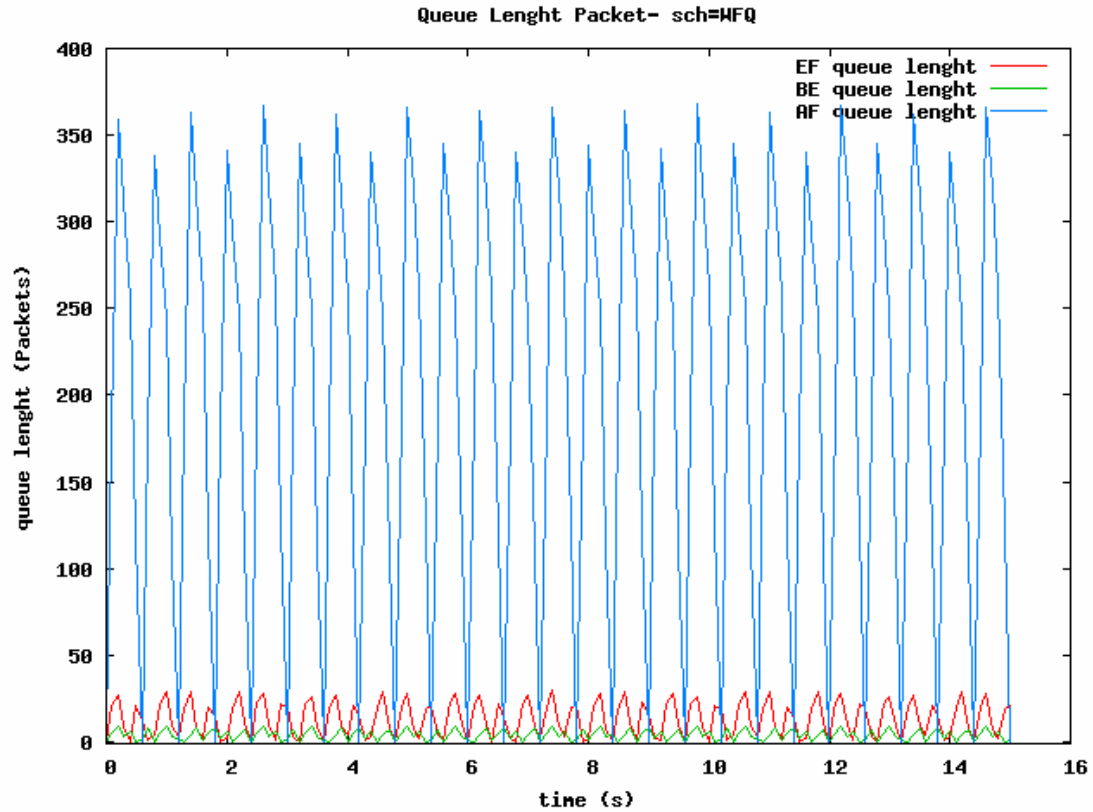
O valor máximo do atraso OWD de BE é 1694.00 ms.



WFQ - Próximo da Saturação

A quantidade de tráfego injectada na rede é igual ao apresentado para o algoritmo PQ, sendo assim apartir deste momento apenas serão apresentados os resultados de saída, tanto para o caso de “Próximo da Saturação” como no caso de “Saturação”.

Pesos: EF→5, AF→7, BE→8



Input:

O número de pacotes gerados BE é de 23430 pkts -> foram enviados 799.744 kbps.

O número de pacotes gerados EF é de 14722 pkts -> foram enviados 502.510933333 kbps.

O número de pacotes gerados AF é de 20500 pkts -> foram enviados 699.733333333 kbps.

O número de pacotes perdidos é de 465 pkts.

Output:

Estatística de Pacotes

CP	TotPkts	TxPkts	ldrops	edrops
--	-----	-----	-----	-----
0	23270	100.00%	0.00%	0.00%
10	20500	100.00%	0.00%	0.00%
46	14575	97.06%	2.94%	0.00%
48	27	0.00%	0.00%	100.00%

All	58372	99.22%	0.73%	0.05%

O valor máximo do atraso IPDV de EF é 31.00 ms.

O valor máximo do atraso OWD de EF é 42.00 ms.

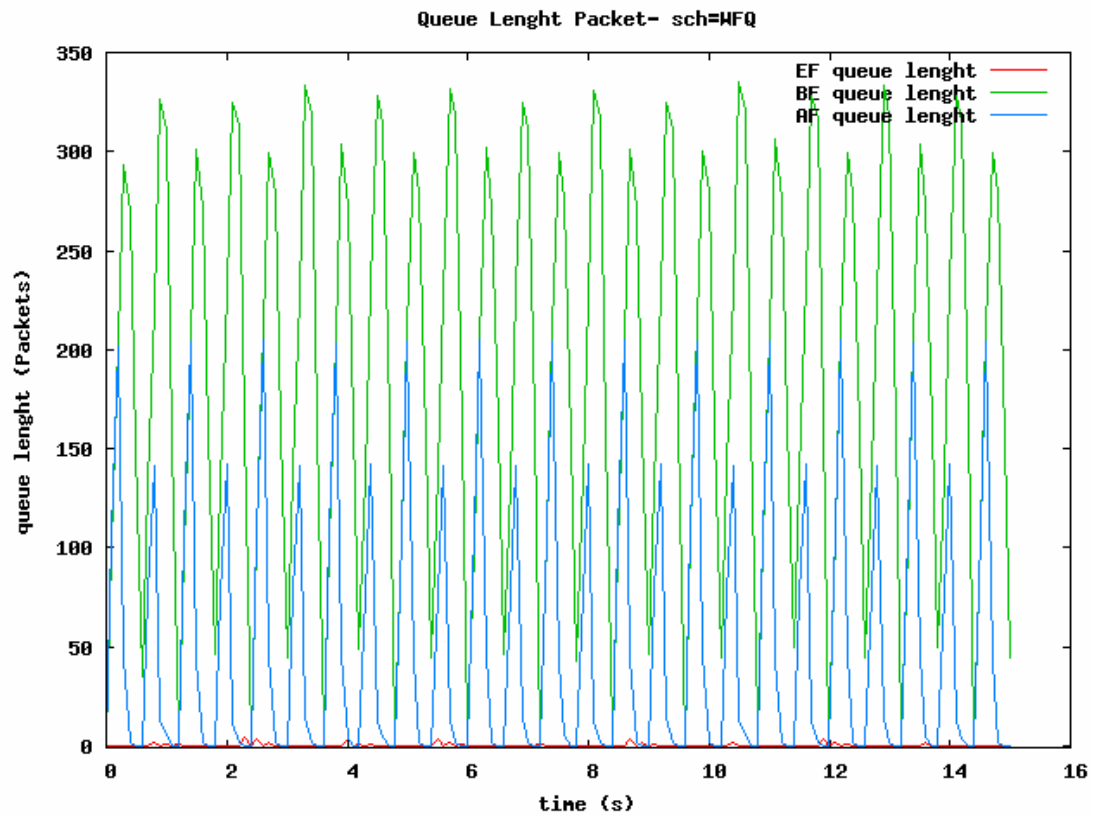
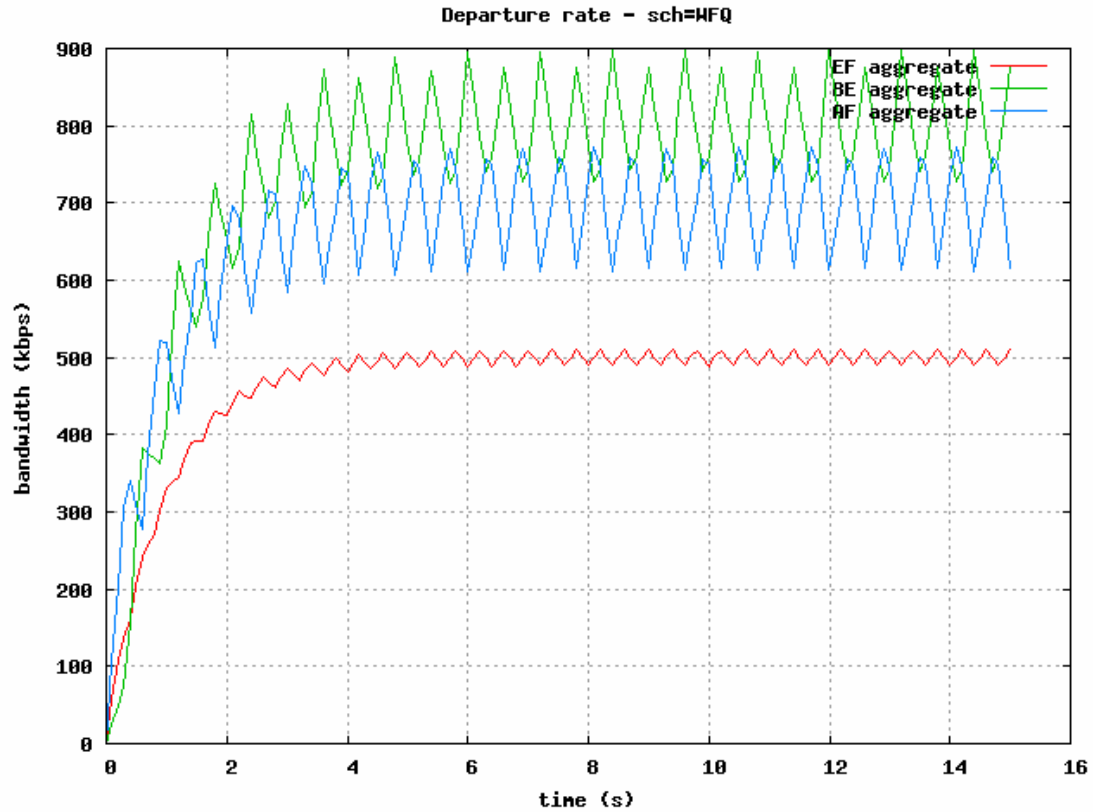
O valor máximo do atraso IPDV de AF é 188.00 ms.

O valor máximo do atraso OWD de AF é 272.00 ms.

O valor máximo do atraso IPDV de BE é 4.00 ms.

O valor máximo do atraso OWD de BE é 17.00 ms.

Pesos: EF→100, AF→4, BE→1



Input:

O número de pacotes gerados BE é de 23430 pkts -> foram enviados 799.744 kbps.

O número de pacotes gerados EF é de 14722 pkts -> foram enviados 502.510933333 kbps.

O número de pacotes gerados AF é de 20500 pkts -> foram enviados 699.733333333 kbps.

O número de pacotes perdidos é de 27 pkts.

Output:

Estatística de Pacotes

CP	TotPkts	TxPkts	ldrops	edrops
--	-----	-----	-----	-----
0	23270	100.00%	0.00%	0.00%
10	20500	100.00%	0.00%	0.00%
46	14575	100.00%	0.00%	0.00%
48	27	0.00%	0.00%	100.00%

All	58372	99.95%	0.00%	0.05%

O valor máximo do atraso IPDV de EF é 2.00 ms.

O valor máximo do atraso OWD de EF é 12.00 ms.

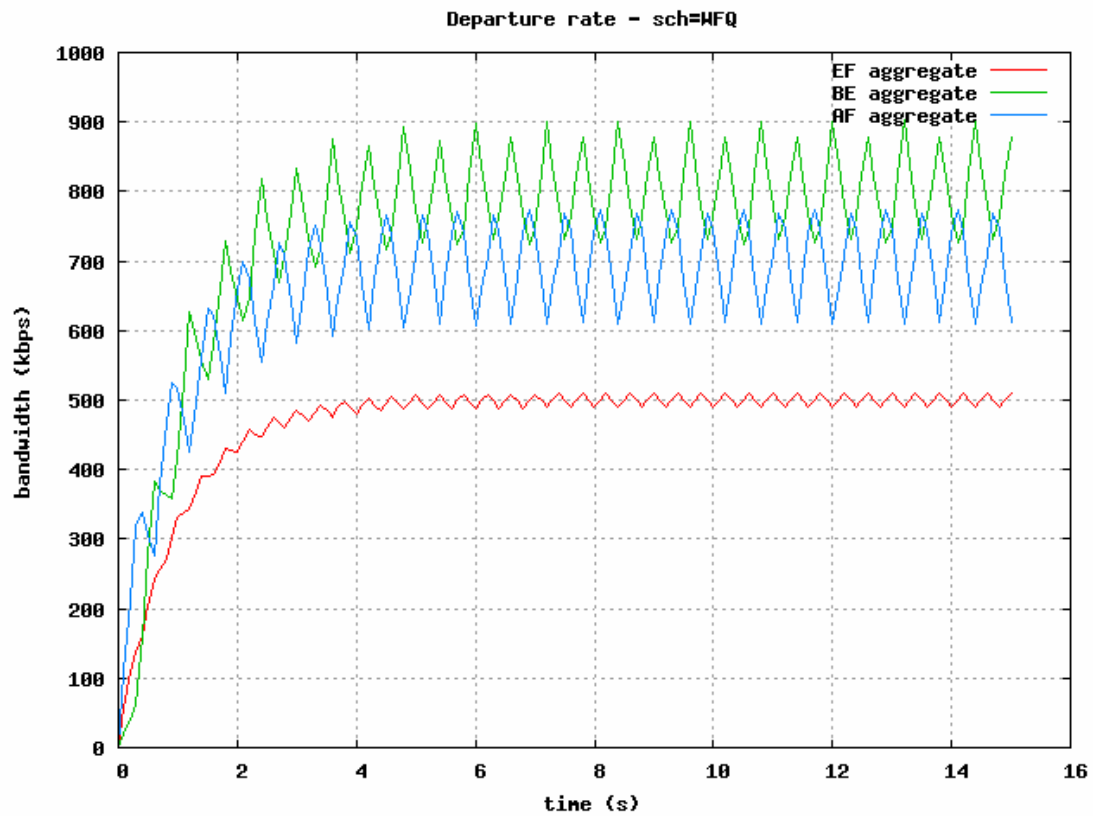
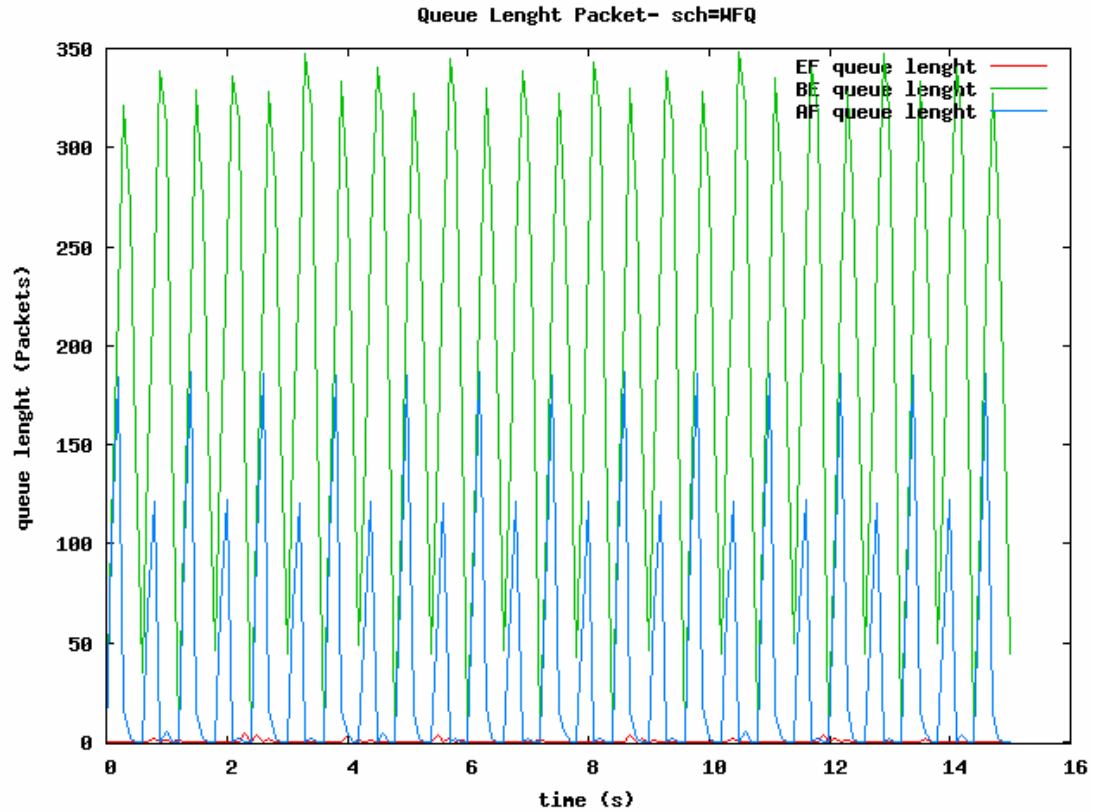
O valor máximo do atraso IPDV de AF é 5.00 ms.

O valor máximo do atraso OWD de AF é 95.00 ms.

O valor máximo do atraso IPDV de BE é 14.00 ms.

O valor máximo do atraso OWD de BE é 234.00 ms.

Pesos: EF→100, AF→5, BE→1



Input:

O número de pacotes gerados BE é de 23430 pkts -> foram enviados 799.744 kbps.

O número de pacotes gerados EF é de 14722 pkts -> foram enviados 502.510933333 kbps.

O número de pacotes gerados AF é de 20500 pkts -> foram enviados 699.733333333 kbps.

O número de pacotes perdidos é de 27 pkts.

Output:

Estatística de Pacotes

CP	TotPkts	TxPkts	ldrops	edrops
--	-----	-----	-----	-----
0	23270	100.00%	0.00%	0.00%
10	20500	100.00%	0.00%	0.00%
46	14575	100.00%	0.00%	0.00%
48	27	0.00%	0.00%	100.00%

All	58372	99.95%	0.00%	0.05%

O valor máximo do atraso IPDV de EF é 1.00 ms.

O valor máximo do atraso OWD de EF é 12.00 ms.

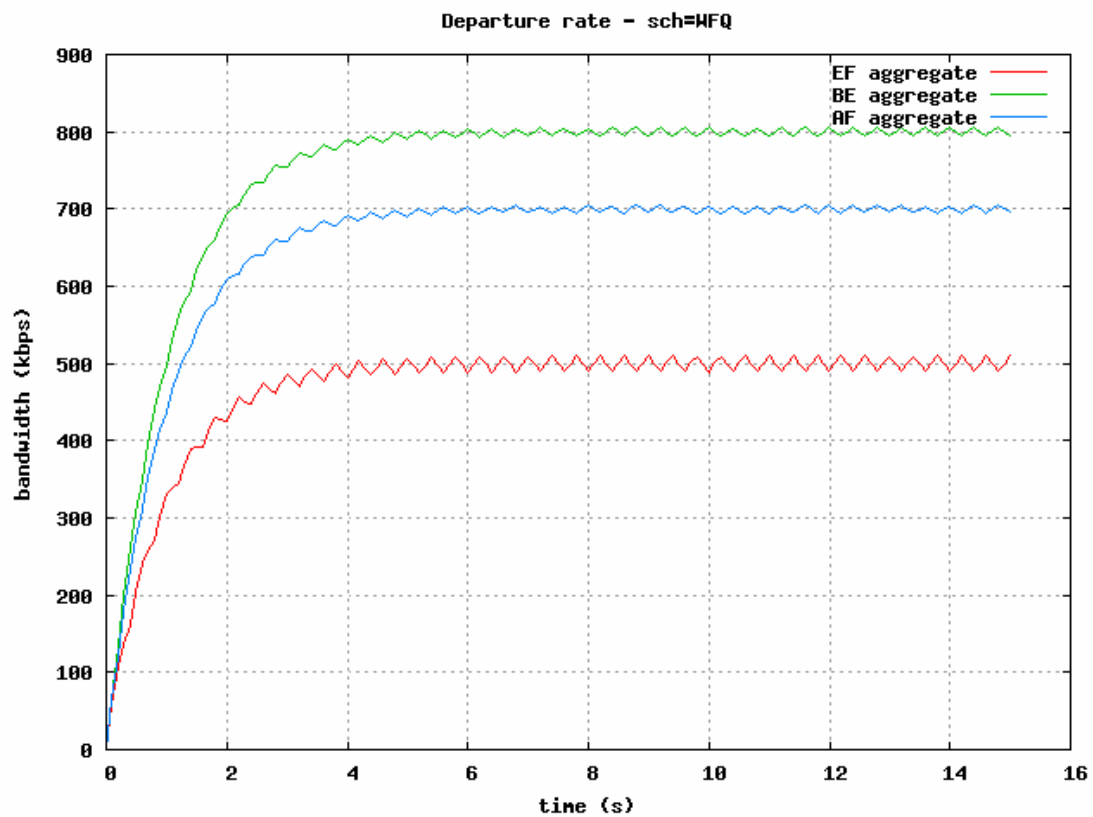
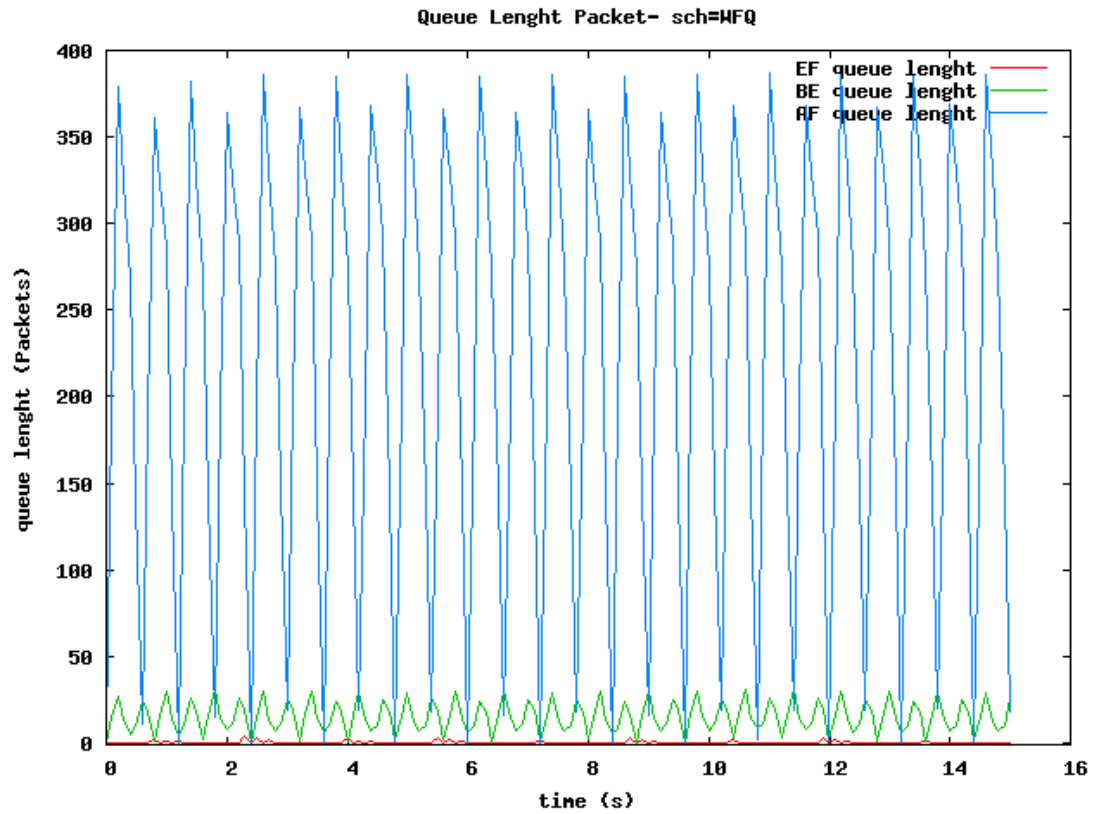
O valor máximo do atraso IPDV de AF é 5.00 ms.

O valor máximo do atraso OWD de AF é 85.00 ms.

O valor máximo do atraso IPDV de BE é 17.00 ms.

O valor máximo do atraso OWD de BE é 236.00 ms.

Pesos: EF→100, AF→7, BE→8



Input:

O número de pacotes gerados BE é de 23430 pkts -> foram enviados 799.744 kbps.

O número de pacotes gerados EF é de 14722 pkts -> foram enviados 502.510933333 kbps.

O número de pacotes gerados AF é de 20500 pkts -> foram enviados 699.733333333 kbps.

O número de pacotes perdidos é de 27 pkts.

Output:

Estatística de Pacotes

CP	TotPkts	TxPkts	ldrops	edrops
--	-----	-----	-----	-----
0	23270	100.00%	0.00%	0.00%
10	20500	100.00%	0.00%	0.00%
46	14575	100.00%	0.00%	0.00%
48	27	0.00%	0.00%	100.00%

All	58372	99.95%	0.00%	0.05%

O valor máximo do atraso IPDV de EF é 2.00 ms.

O valor máximo do atraso OWD de EF é 12.00 ms.

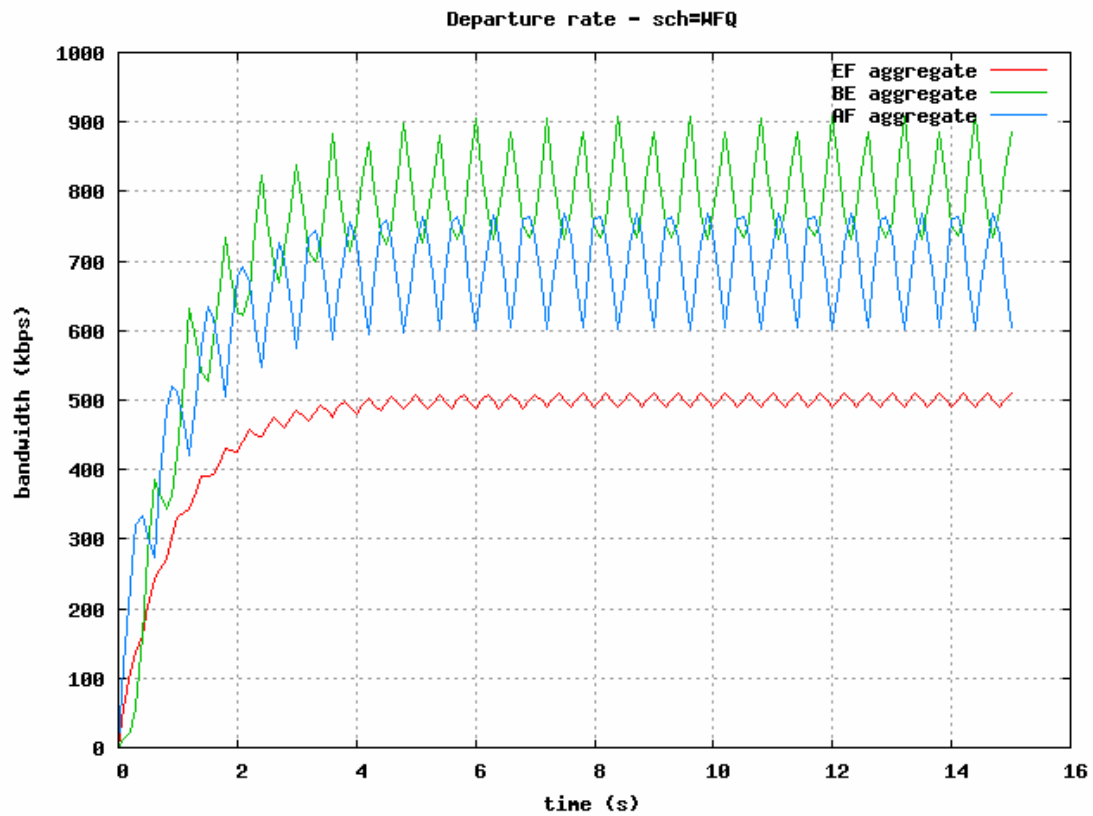
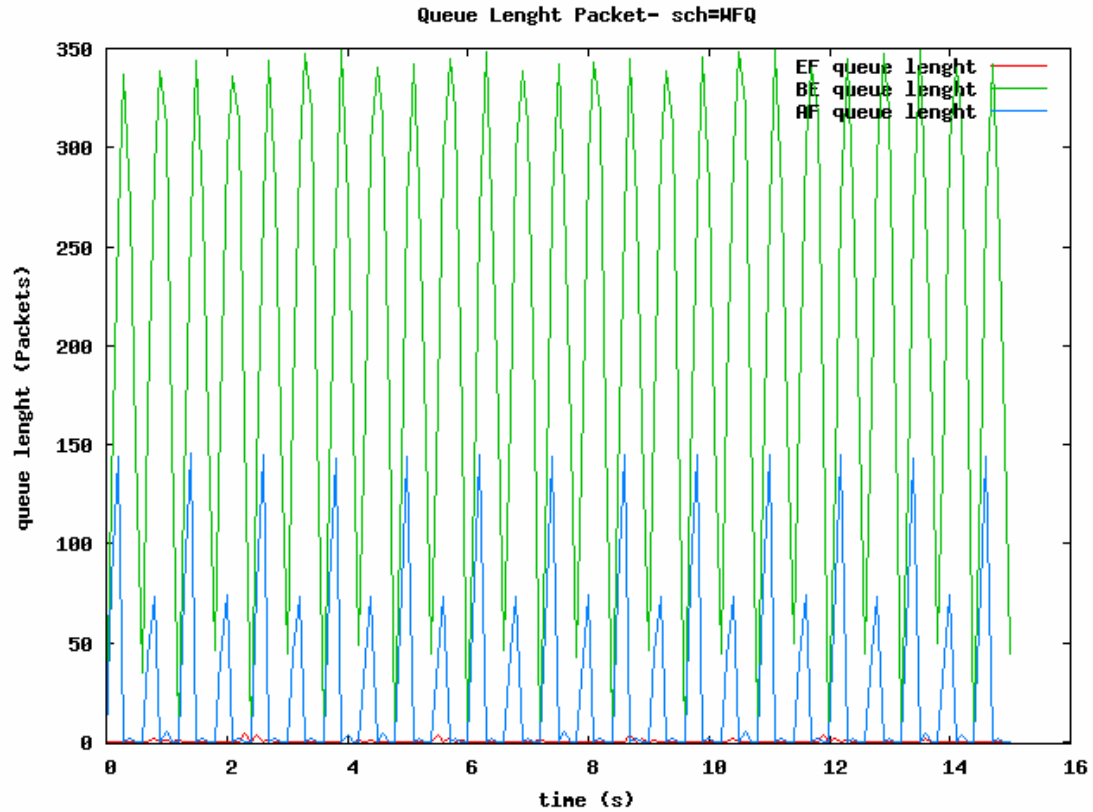
O valor máximo do atraso IPDV de AF é 199.00 ms.

O valor máximo do atraso OWD de AF é 291.00 ms.

O valor máximo do atraso IPDV de BE é 4.00 ms.

O valor máximo do atraso OWD de BE é 31.00 ms.

Pesos: EF→100, AF→10, BE→1



Input:

O número de pacotes gerados BE é de 23430 pkts -> foram enviados 799.744 kbps.

O número de pacotes gerados EF é de 14722 pkts -> foram enviados 502.510933333 kbps.

O número de pacotes gerados AF é de 20500 pkts -> foram enviados 699.733333333 kbps.

O número de pacotes perdidos é de 27 pkts.

Output:

Estatística de Pacotes

CP	TotPkts	TxPkts	ldrops	edrops
--	-----	-----	-----	-----
0	23270	100.00%	0.00%	0.00%
10	20500	100.00%	0.00%	0.00%
46	14575	100.00%	0.00%	0.00%
48	27	0.00%	0.00%	100.00%

All	58372	99.95%	0.00%	0.05%

O valor máximo do atraso IPDV de EF é 2.00 ms.

O valor máximo do atraso OWD de EF é 12.00 ms.

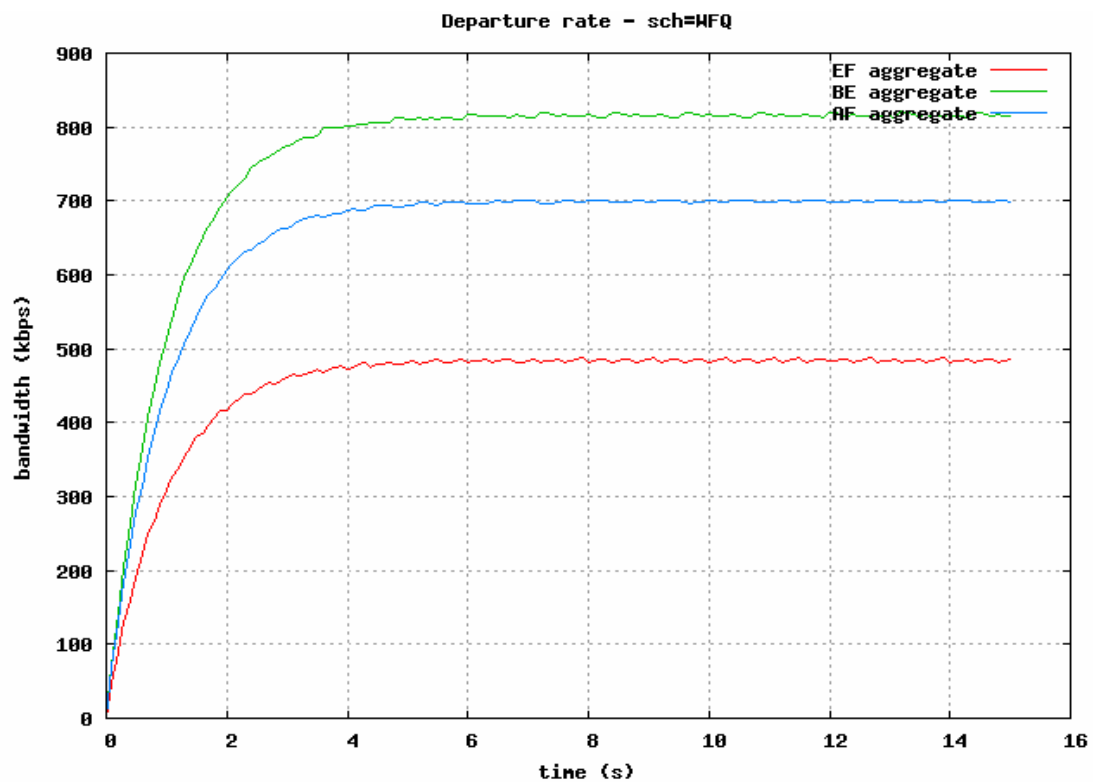
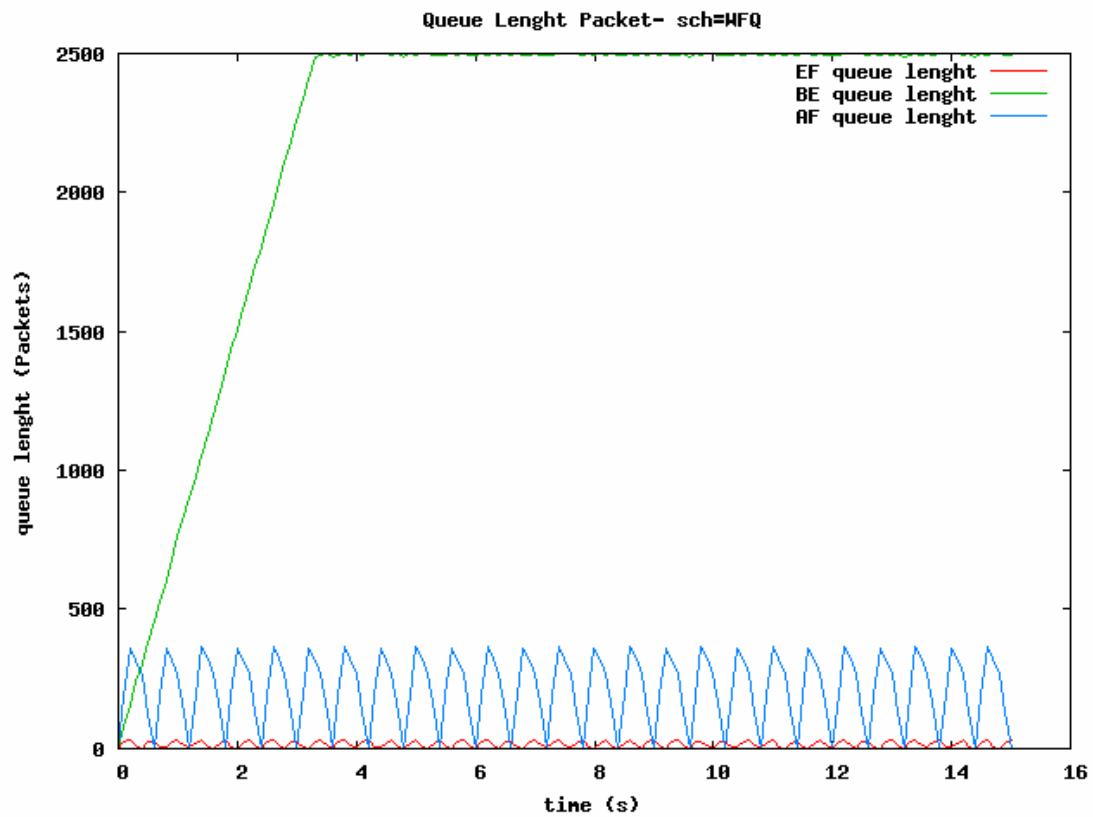
O valor máximo do atraso IPDV de AF é 5.00 ms.

O valor máximo do atraso OWD de AF é 64.00 ms.

O valor máximo do atraso IPDV de BE é 36.00 ms.

O valor máximo do atraso OWD de BE é 246.00 ms.

Pesos: EF→5, AF→7, BE→8



Input:

O número de pacotes gerados BE é de 35150 pkts -> foram enviados 1199.78666667 kbps.

O número de pacotes gerados EF é de 14722 pkts -> foram enviados 502.510933333 kbps.

O número de pacotes gerados AF é de 20500 pkts -> foram enviados 699.733333333 kbps.

O número de pacotes perdidos é de 9257 pkts.

Ouput:

Estatística de Pacotes

```
=====
```

CP	TotPkts	TxPkts	ldrops	edrops
--	-----	-----	-----	-----
0	34910	75.10%	24.90%	0.00%
10	20500	100.00%	0.00%	0.00%
46	14575	96.92%	3.08%	0.00%
48	27	0.00%	0.00%	100.00%

All	70012	86.91%	13.06%	0.04%

O valor máximo do atraso IPDV de EF é 29.000000 ms.

O valor máximo do atraso OWD de EF é 41.000000 ms.

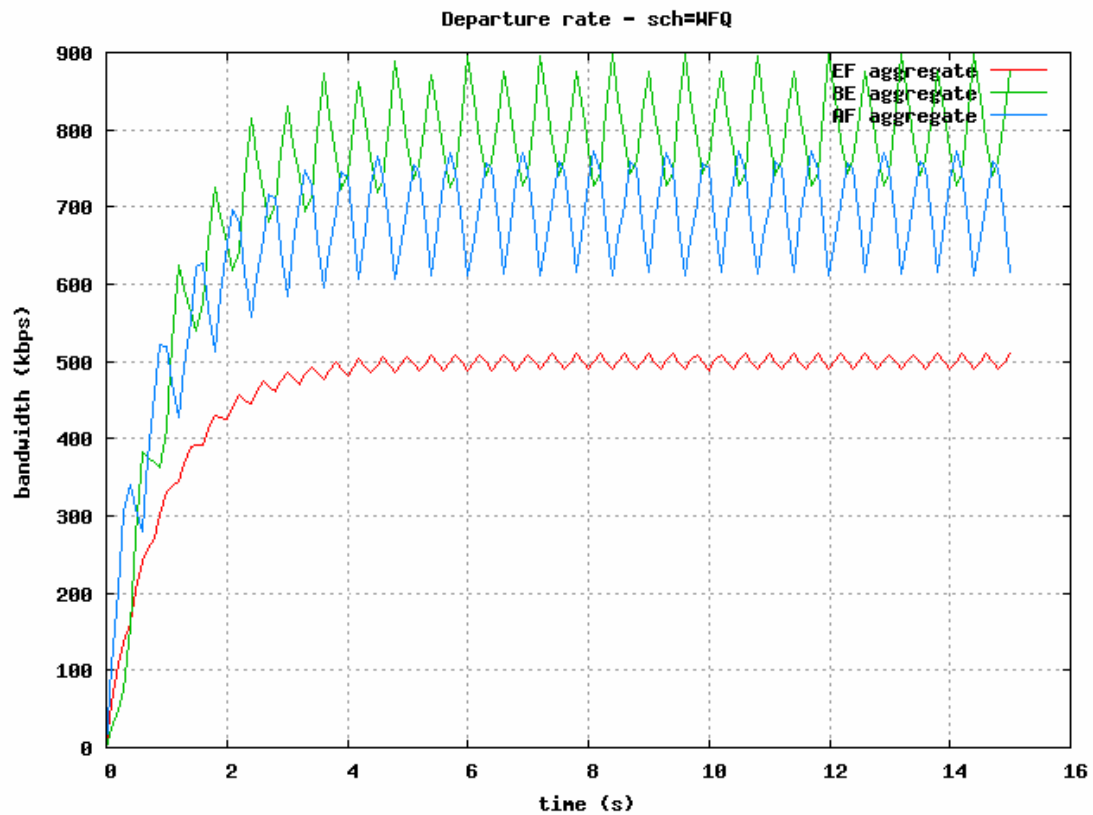
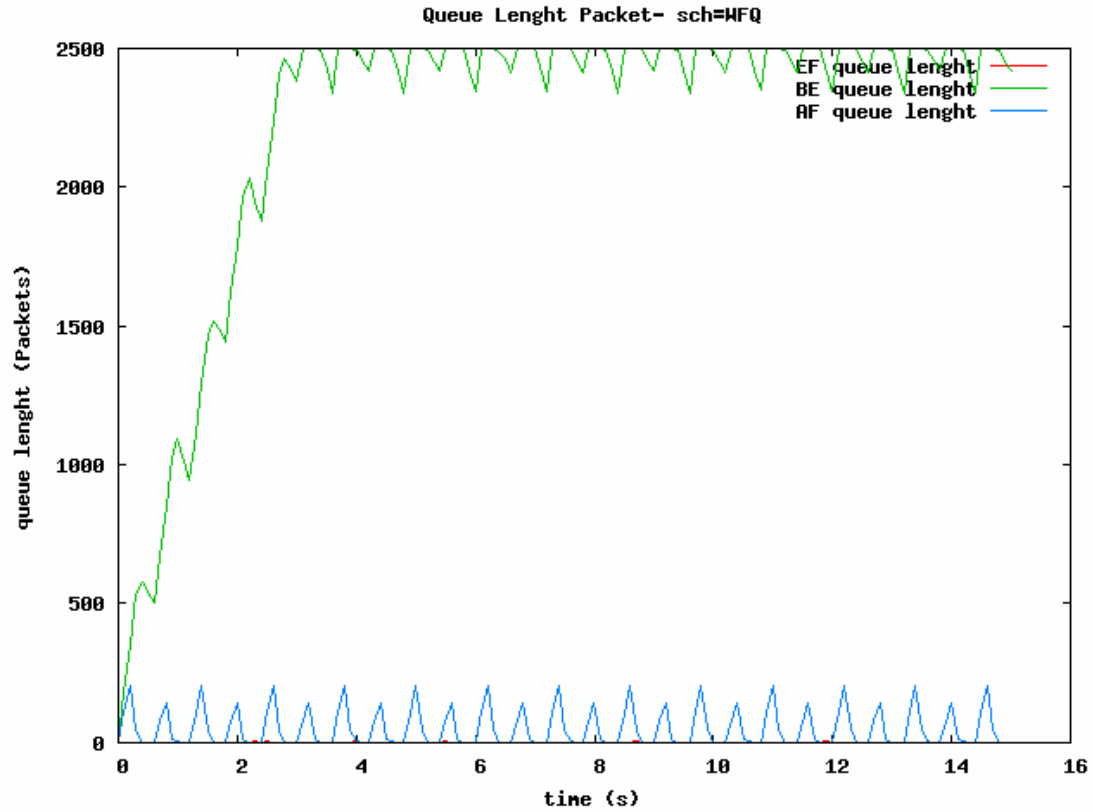
O valor máximo do atraso IPDV de AF é 192.000000 ms.

O valor máximo do atraso OWD de AF é 279.000000 ms.

O valor máximo do atraso IPDV de BE é 6.000000 ms.

O valor máximo do atraso OWD de BE é 1585.000000 ms.

Pesos: EF→100, AF→4, BE→1



Input:

O número de pacotes gerados BE é de 35150 pkts -> foram enviados 1199.78666667 kbps.

O número de pacotes gerados EF é de 14722 pkts -> foram enviados 502.510933333 kbps.

O número de pacotes gerados AF é de 20500 pkts -> foram enviados 699.733333333 kbps.

O número de pacotes perdidos é de 9361 pkts.

Output:

Estatísticas de pacotes

=====				
CP	TotPkts	TxPkts	ldrops	edrops
--	-----	-----	-----	-----
0	34910	73.26%	26.74%	0.00%
10	20500	100.00%	0.00%	0.00%
46	14575	100.00%	0.00%	0.00%
48	27	0.00%	0.00%	100.00%

All	70012	86.63%	13.33%	0.04%

O valor máximo do atraso IPDV de EF é 2.000000 ms.

O valor máximo do atraso OWD de EF é 12.000000 ms.

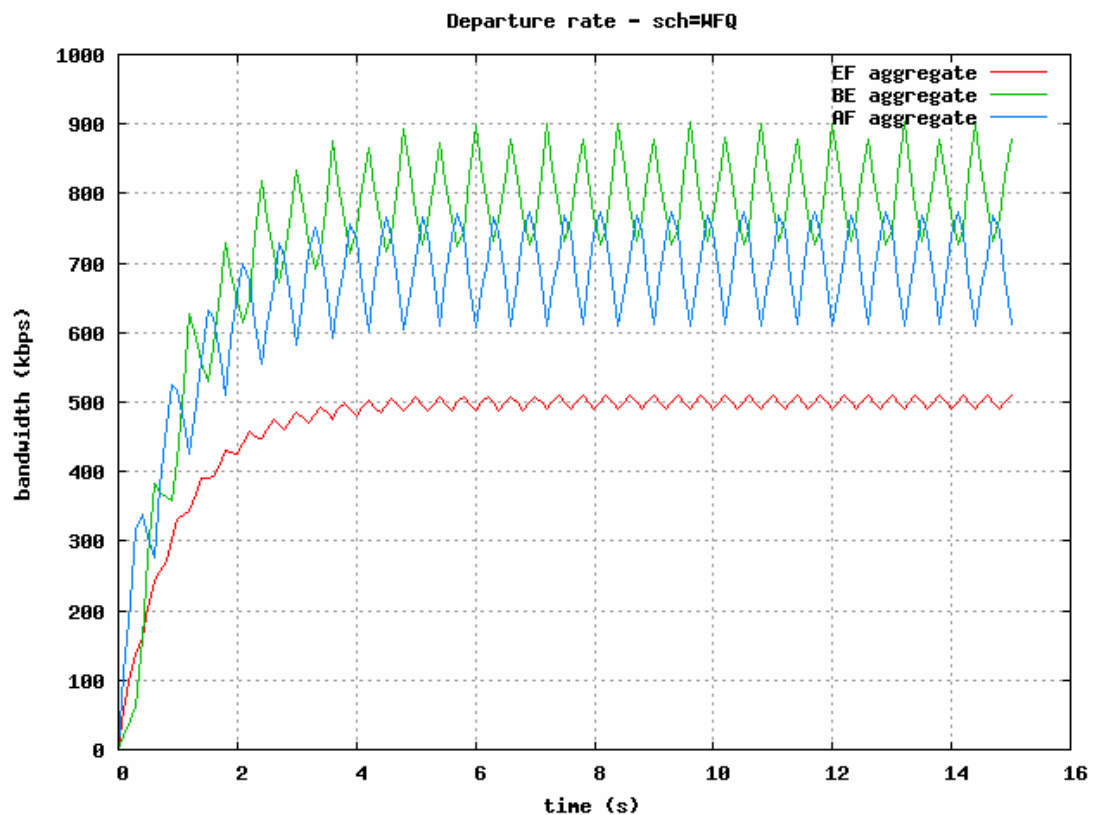
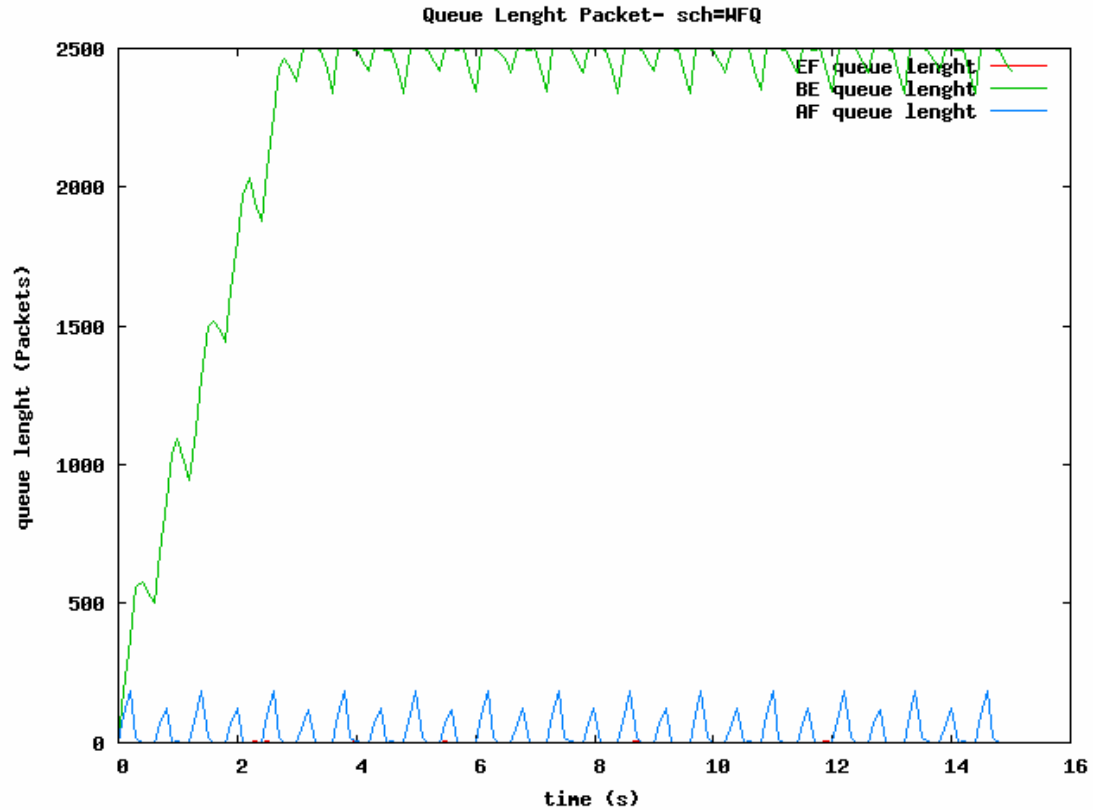
O valor máximo do atraso IPDV de AF é 5.000000 ms.

O valor máximo do atraso OWD de AF é 95.000000 ms.

O valor máximo do atraso IPDV de BE é 21.000000 ms.

O valor máximo do atraso OWD de BE é 1691.000000 ms.

Pesos: EF→100, AF→5, BE→1



Input:

O número de pacotes gerados BE é de 35150 pkts -> foram enviados 1199.78666667 kbps.

O número de pacotes gerados EF é de 14722 pkts -> foram enviados 502.510933333 kbps.

O número de pacotes gerados AF é de 20500 pkts -> foram enviados 699.733333333 kbps.

O número de pacotes perdidos é de 9361 pkts.

Output:

Estatísticas de pacotes

=====				
CP	TotPkts	TxPkts	ldrops	edrops
--	-----	-----	-----	-----
0	34910	73.26%	26.74%	0.00%
10	20500	100.00%	0.00%	0.00%
46	14575	100.00%	0.00%	0.00%
48	27	0.00%	0.00%	100.00%

All	70012	86.63%	13.33%	0.04%

O valor máximo do atraso IPDV de EF é 2.000000 ms.

O valor máximo do atraso OWD de EF é 12.000000 ms.

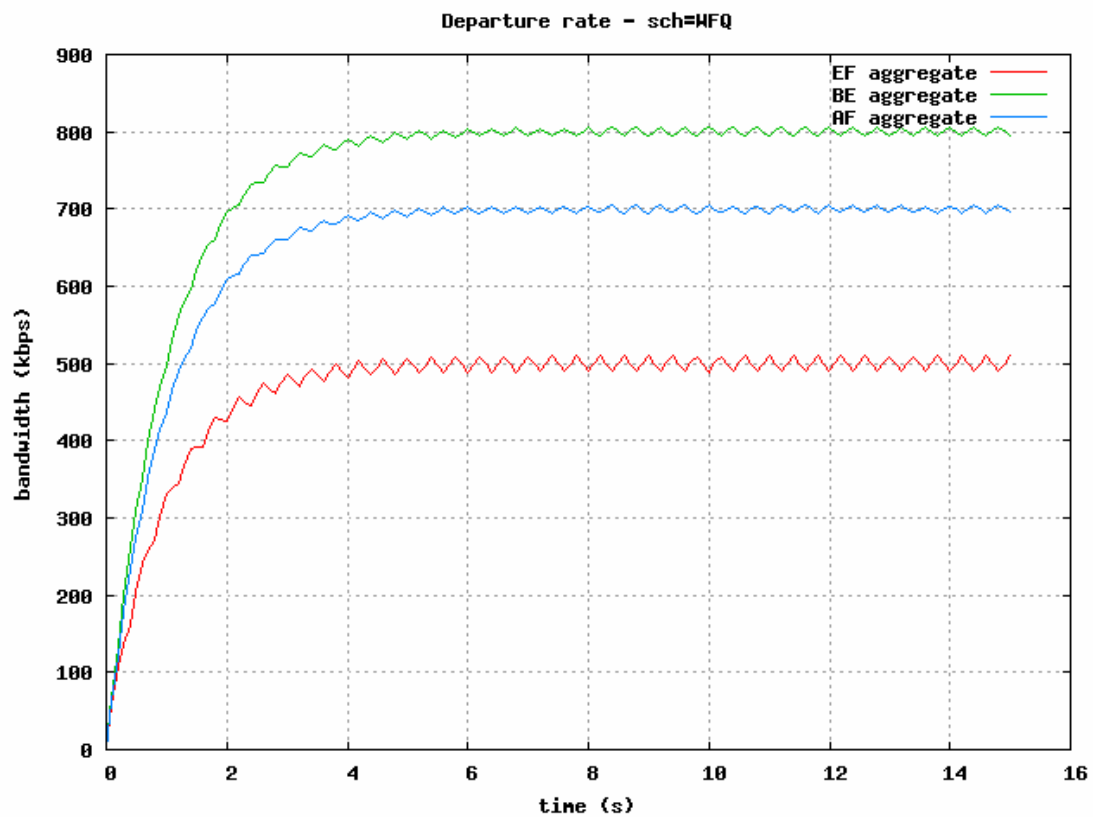
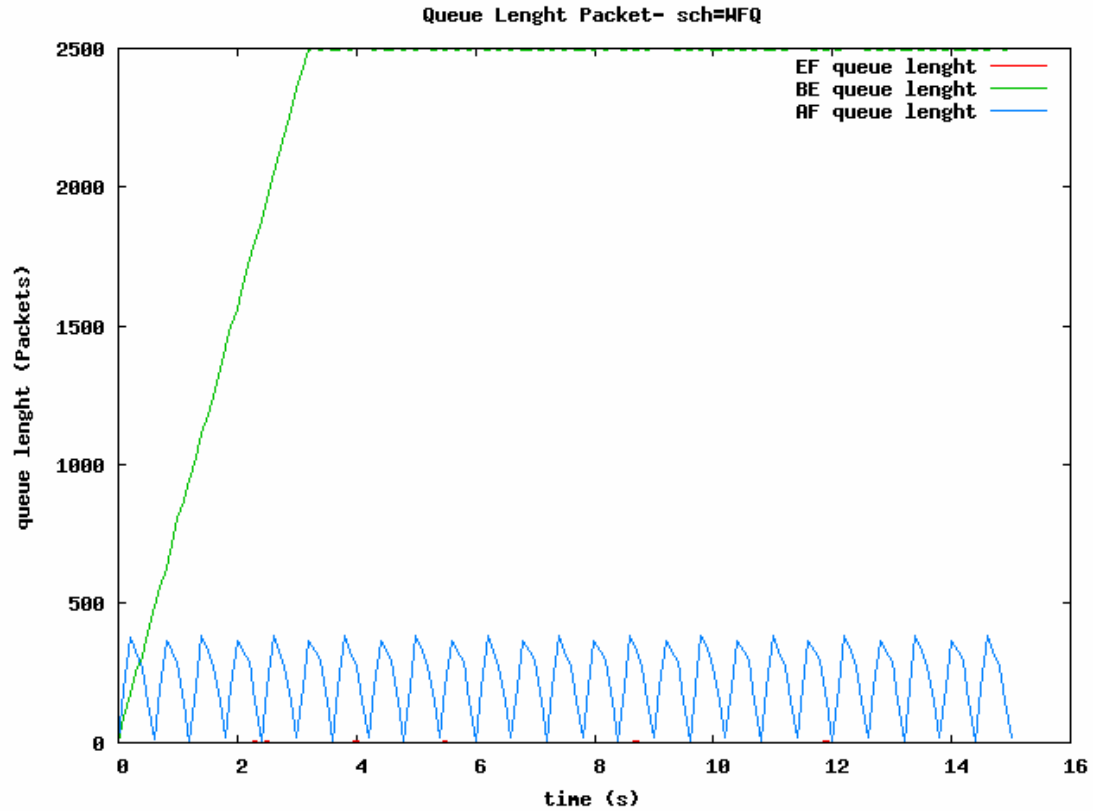
O valor máximo do atraso IPDV de AF é 5.000000 ms.

O valor máximo do atraso OWD de AF é 84.000000 ms.

O valor máximo do atraso IPDV de BE é 228.000000 ms.

O valor máximo do atraso OWD de BE é 1694.000000 ms.

Pesos: EF→100, AF→7, BE→8



Input:

O número de pacotes gerados BE é de 35150 pkts -> foram enviados 1199.78666667 kbps.

O número de pacotes gerados EF é de 14722 pkts -> foram enviados 502.510933333 kbps.

O número de pacotes gerados AF é de 20500 pkts -> foram enviados 699.733333333 kbps.

O número de pacotes perdidos é de 9264 pkts.

Output:

Estatísticas de pacotes

=====

CP	TotPkts	TxPkts	ldrops	edrops
--	-----	-----	-----	-----
0	34910	73.80%	26.20%	0.00%
10	20500	100.00%	0.00%	0.00%
46	14575	100.00%	0.00%	0.00%
48	27	0.00%	0.00%	100.00%

All	70012	86.90%	13.06%	0.04%
-----	-------	--------	--------	-------

O valor máximo de atraso IPDV de EF é 2.000000 ms.

O valor máximo de atraso OWD de EF é 12.000000 ms.

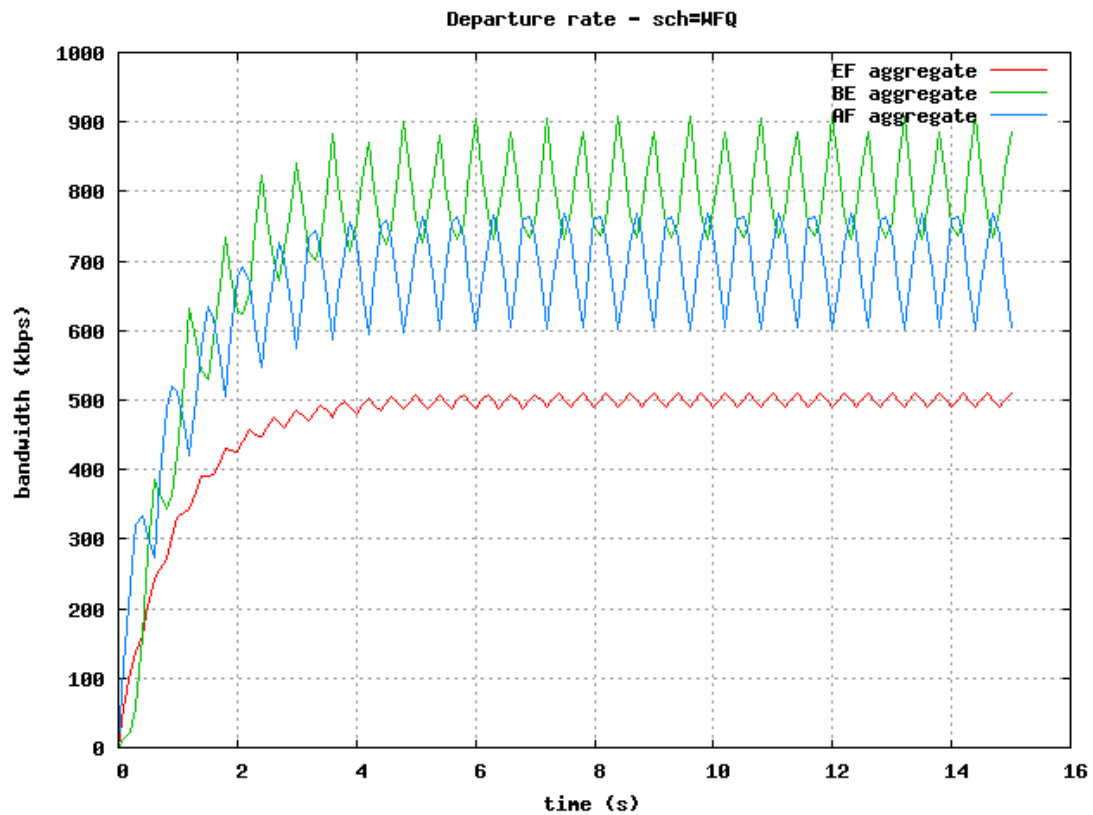
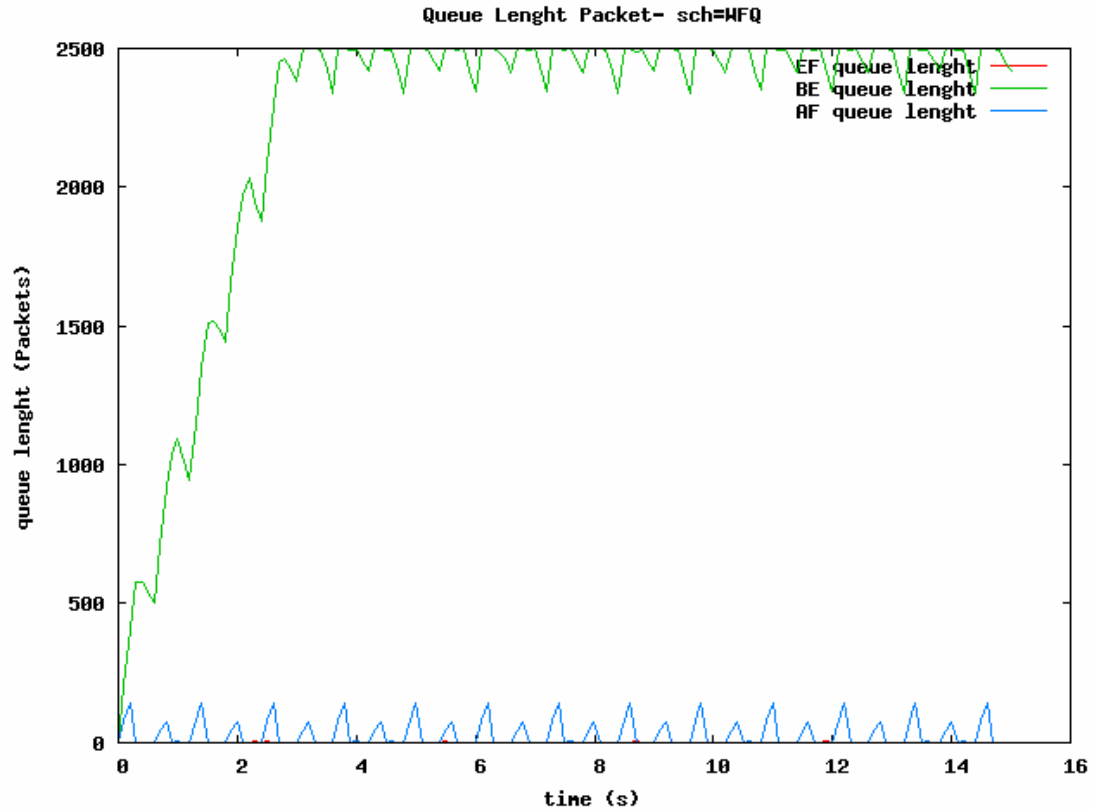
O valor máximo de atraso IPDV de AF é 198.000000 ms.

O valor máximo de atraso OWD de AF é 292.000000 ms.

O valor máximo de atraso IPDV de BE é 4.000000 ms.

O valor máximo de atraso OWD de BE é 1611.000000 ms.

Pesos: EF→100, AF→10, BE→1



Input:

O número de pacotes gerados BE é de 35150 pkts -> foram enviados 1199.78666667 kbps.

O número de pacotes gerados EF é de 14722 pkts -> foram enviados 502.510933333 kbps.

O número de pacotes gerados AF é de 20500 pkts -> foram enviados 699.733333333 kbps.

O número de pacotes perdidos é de 9361 pkts.

Output:

Estatísticas de pacotes

=====

CP	TotPkts	TxPkts	ldrops	edrops
----	---------	--------	--------	--------

--	-----	-----	-----	-----
----	-------	-------	-------	-------

0	34910	73.26%	26.74%	0.00%
---	-------	--------	--------	-------

10	20500	100.00%	0.00%	0.00%
----	-------	---------	-------	-------

46	14575	100.00%	0.00%	0.00%
----	-------	---------	-------	-------

48	27	0.00%	0.00%	100.00%
----	----	-------	-------	---------

All	70012	86.63%	13.33%	0.04%
-----	-------	--------	--------	-------

O valor máximo do atraso IPDV de EF é 2.000000 ms.

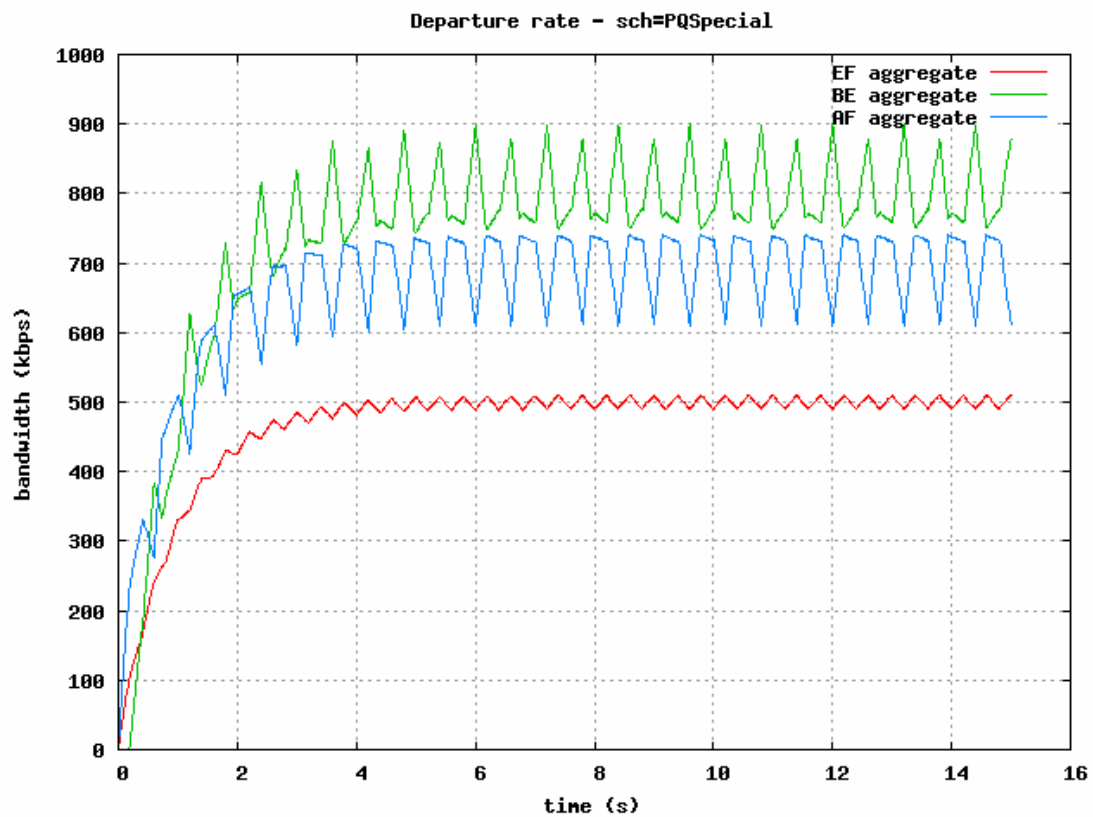
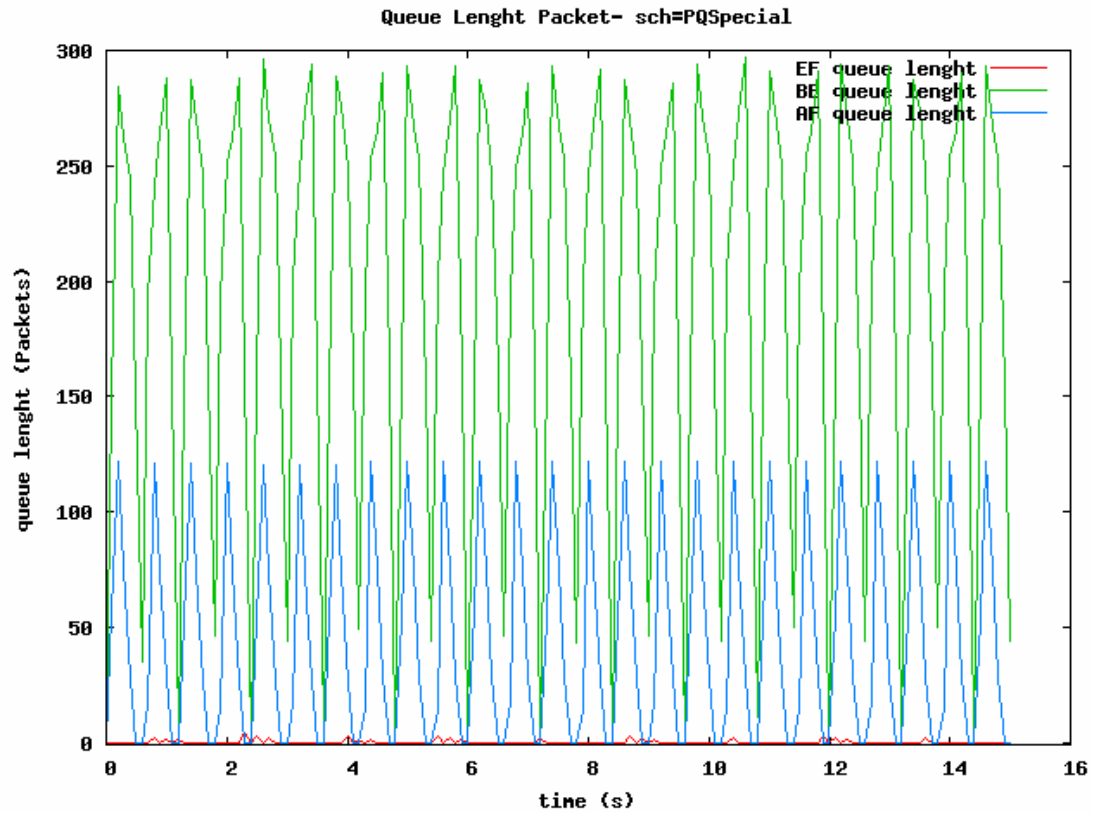
O valor máximo do atraso OWD de EF é 12.000000 ms.

O valor máximo do atraso IPDV de AF é 5.000000 ms.

O valor máximo do atraso OWD de AF é 64.000000 ms.

O valor máximo do atraso IPDV de BE é 207.000000 ms.

O valor máximo do atraso OWD de BE é 1694.000000 ms.



Input:

O número de pacotes gerados BE é de 23430 pkts -> foram enviados 799.744 kbps.

O número de pacotes gerados EF é de 14722 pkts -> foram enviados 502.510933333 kbps.

O número de pacotes gerados AF é de 20500 pkts -> foram enviados 699.733333333 kbps.

O número de pacotes perdidos é de 27 pkts.

Output:

Estatísticas de pacotes

=====				
CP	TotPkts	TxPkts	ldrops	edrops
--	-----	-----	-----	-----
0	23270	100.00%	0.00%	0.00%
10	20500	100.00%	0.00%	0.00%
46	14575	100.00%	0.00%	0.00%
48	27	0.00%	0.00%	100.00%

All	58372	99.95%	0.00%	0.05%

O valor médio do OWD de EF é 10.618432643638409.

O valor médio do OWD de AF é 48.833893960799884.

O valor médio do OWD de BE é 139.89554411126142.

O valor médio do IPDV de EF é 0.31462562729977767.

O valor médio do IPDV de AF é 2.6024261555821453.

O valor médio do IPDV de BE é 3.778823126336647.

O valor máximo do atraso IPDV de EF é 2ms.

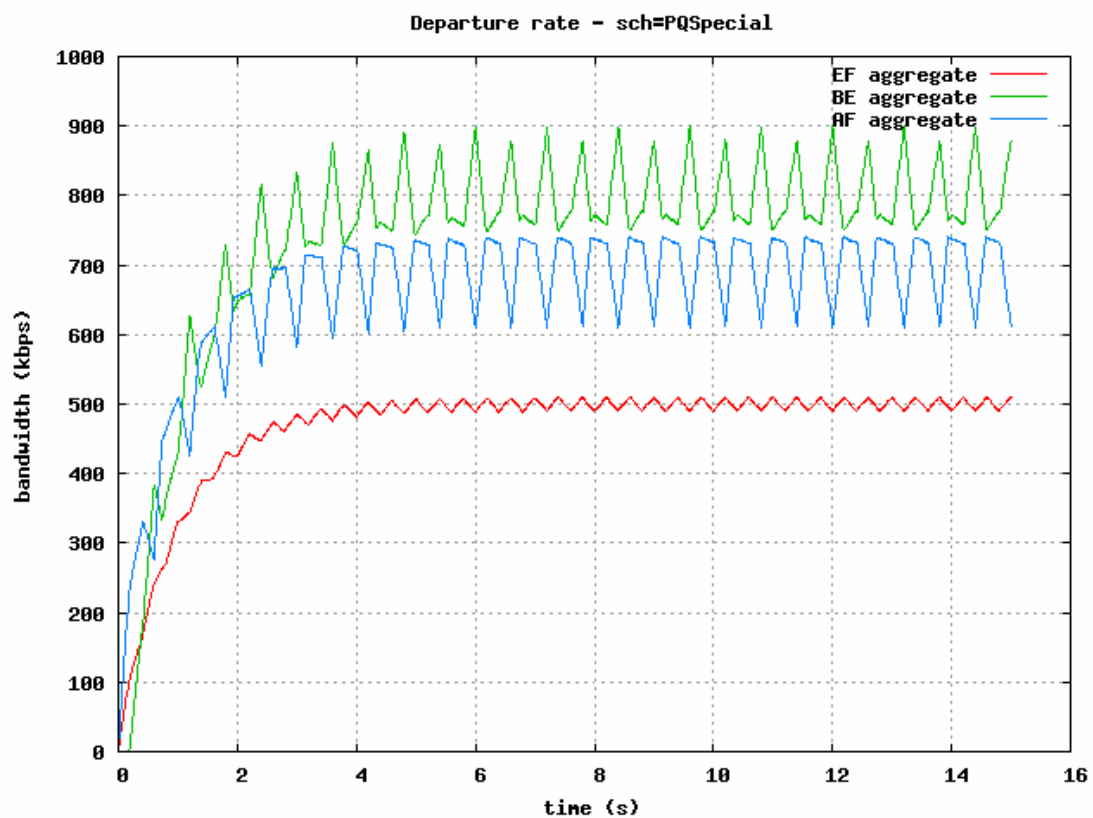
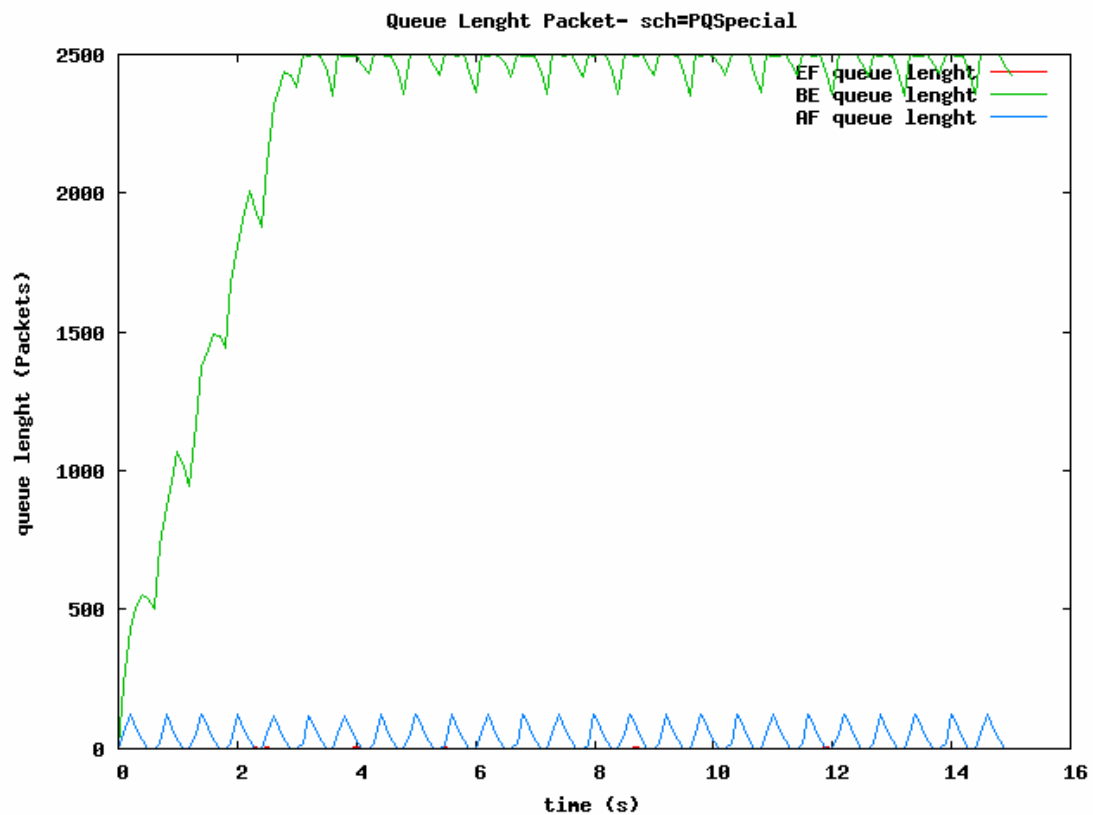
O valor máximo do atraso OWD de EF é 12ms.

O valor máximo do atraso IPDV de AF é 18ms.

O valor máximo do atraso OWD de AF é 102ms.

O valor máximo do atraso IPDV de BE é 183ms.

O valor máximo do atraso OWD de BE é 200ms.



Input:

O número de pacotes gerados BE é de 35150 pkts -> foram enviados 1199.78666667 kbps.

O número de pacotes gerados EF é de 14722 pkts -> foram enviados 502.510933333 kbps.

O número de pacotes gerados AF é de 20500 pkts -> foram enviados 699.733333333 kbps.

O número de pacotes perdidos é de 9355 pkts.

Output:

Estatísticas de pacotes

=====				
CP	TotPkts	TxPkts	ldrops	edrops
--	-----	-----	-----	-----
0	34910	73.28%	26.72%	0.00%
10	20500	100.00%	0.00%	0.00%
46	14575	100.00%	0.00%	0.00%
48	27	0.00%	0.00%	100.00%

All	70012	86.64%	13.32%	0.04%

O valor médio do OWD de EF é 10.620915588312856.

O valor médio do OWD de AF é 48.213580722857984.

O valor médio do OWD de BE é 1354.6786697091411.

O valor médio do IPDV de EF é 0.31933623285530405.

O valor médio do IPDV de AF é 2.6049595390193638.

O valor médio do IPDV de BE é 5.1127008812870187.

O valor máximo do atraso IPDV de EF é 2ms.

O valor máximo do atraso OWD de EF é 12ms.

O valor máximo do atraso IPDV de AF é 18ms.

O valor máximo do atraso OWD de AF é 102ms.

O valor máximo do atraso IPDV de BE é 184ms.

O valor máximo do atraso OWD de BE é 1690ms.