Quality of Service Management in IP Networks

By

Titus Awotula

A doctoral thesis submitted in partial fulfilment of the requirements for the award of Doctor of Philosophy of Loughborough University

September 2006

Dedicated To The Glory of Almighty God
Through The Name of Our Lord Jesus Christ

And

In Remembrance of My Late Parents
Mr Beniah Awotula and
Mrs Luce Awotula

Abstract

Quality of Service (QoS) in Internet Protocol (IP) Networks has been the subject of active research over the past two decades. Integrated Services (IntServ) and Differentiated Services (DiffServ) QoS architectures have emerged as proposed standards for resource allocation in IP Networks. These two QoS architectures support the need for multiple traffic queuing systems to allow for resource partitioning for heterogeneous applications making use of the networks. There have been a number of specifications or proposals for the number of traffic queuing classes (*Class of Service* (CoS)) that will support integrated services in IP Networks, but none has provided verification in the form of analytical or empirical investigation to prove that its specification or proposal will be optimum.
Despite the existence of the two standard QoS architectures and the large volume of research work that has been carried out on IP QoS, its deployment still remains elusive in the Internet. This is not unconnected with the complexities associated with some aspects of the standard QoS architectures.

This thesis presents as the first of two parts, work on an empirical investigation to determine the Optimum Number of Traffic Queuing Classes (ONTQC) that will best support multiservice in IP Networks. The results of the investigation show that a three-queuing-classes is optimum. The result will be of great interest to the research community and the telecommunication industries.

The second piece of work addressed by this thesis concerns the design and evaluation of a novel IP QoS architecture referred to as *Predeterministic Distributed Event Response Resource Management* (PDERRM). Simplicity, elegance and robustness were key concepts employed in the design of PDERRM. The results from the performance evaluation of PDERRM have been very impressive. The results of simulation experiments carried out to compare PDERRM with *best-effort* service showed that PDERRM performed far better than best-effort service on *latency* (delay). Remarkably, results of experiments showed PDERRM outperformed Resource reSerVation Protocol (RSVP) on key performance metrics such as latency and throughput.

Acknowledgements

I am grateful and indebted to my supervisor Professor David Parish, for his support guidance and encouragement throughout the period of my PhD study. His support has been instrumental to the successful completion of this research work.

I wish to express my gratitude to the committee setup in the year 2004, which serve as support to enable me successfully complete my PhD work. The committee was setup to ameliorate the difficulties I mighty have had in coping with my studies during period of convalescence after a period of hospitalisation. The committee members include — Professor Peter Smith (former Head of Dept., and now late), Professor Goodall, Professor David Parish, Dr Brigette Vale and Mr Dan Doran. The existence of the committee gave me a lot of encouragement and boost my morale.

I will always like to say thank you to the former Vice Chancellor of Loughborough University, Sir Professor David Wallace who did not ignore my initial appeal to the University for support in pursuing my PhD program.

I will forever be very grateful and indebted to Dr Brigette Vale who has shown sympathy right from the beginning for my desire to pursue academic goal. She continue her support undaunted during the long period involve in my struggle to achieve my goal. I cannot find the right words to express my gratitude to her.

Last but not the least, I wish to express my gratitude to Mrs Olanrewaju Awotula for her financial support during the long period involved in this study, and the problems that I had. Despite the serious problems between us, she continued to provide most of living expenses throughout the long period of study.

Abbreviations & Acronyms

AAL	ATM Adaptation Layer
ACPL	ATM Centric Protocol Layer
AF	Assured Forwarding
API	Application Programming Interface
ATM	Asynchronous Transfer Mode
BA	Behaviour Aggregate
BECN	Backward Explicit Congestion Notification
BRR	Bit-by-Bit Round Robin
BSTD	Block and State Transition Diagram
CBQ	Class Based Queue
cbqdwrr	Class Based Queue (with) Deficit Weighted Round Robin
CBQ-RR	Class Based Queue Round Robin
CBR	Constant Bit Rate
CLP	Cell Loss Priority
COPS	Common Open Policy Services
CoS	Class of Service
CPCS	Common Part Convergence Sublayer
CPU	Centre Processing Unit
CRC	Cyclic Redundancy Checks
CRCQBA	Classification, Resource Capacity & Quota Brokerage Agent
CR-LDP	Constraint Routed Label Distribution Protocol

CS	Convergence Sublayer
CSMA/CD	Carrier Sense Multiple Access with Collision Detention
DARPA	Defence Advanced Research Projects Agency
DiffServ	Differentiated Services
DS	Differentiated Services
DSBM	Designated Subnet Bandwidth Manager
DSCP	Differentiated Services Codepoint
DSField	Differentiated Services Field
DWRR	Deficit Weighted Round Robin
EF	Expedited Forwarding
FCFS	First-Come First-Serve
FCS	Frame Check Sequence
FD	Forwarding Devices
FDDI	Fiber Distributed Data Interface
FEC	Forwarding Equivalence Class
FECN	Forward Explicit Congestion Notification
FIFO	First In First Out
FLM	Fussy Logic Management
FQ	Fair Queuing
FRBF	Fission of the Rightmost Block First
FSM	Finite State Machine
FSST	Flow Session State Table
FTP	File Transfer Protocol
GFC	Generic Flow Control

GPS	Generalised Processor Sharing
HSN	High Speed Network
HTML	Hyper Text Mark-up Language
HTTP	Hyper- Text Transfer Protocol
ICMP	Internet Control Message Protocol
ICMP	Internet Control Message Protocol
IEEE	Institute of Electrical and Electronic Engineers
IETF	Internet Engineering Task Force
IGMP	Internet Group Management Protocol
IHL	Internet Header Length
IntServ	Integrated Services
IP	Internet Protocol
IPv4	IP version 4
IPv6	IP version 6
ISSLL	Integrated Service over Specific Link Layers
ITU	International Telecommunication Union
LAN	Local Area Network
LDP	Label Distribution Protocol
LER	Label Edge Router
LIB	Label Information Base
LSP	Label Switched Path
LSR	Label Switched Router
LST	Label-Switching Table

MAC	Media Access Control
MAN	Metropolitan Area Network
MBS	Maximum Burst Size
MP3	MPEG Audio Layer-3
MPEG	Moving Picture Expert Group
MPLS	Multi-Protocol Label Switch
MTU	Maximum Transmission Unit
NE	Network Elements
NNA	Neural Network Algorithm
NNI	Network-to-Node Interface
NS-2	Network Simulator Version 2
NSFnet	National Science Foundation Network
ONTQC	Optimum Number of Traffic Queuing Classes
OQD	Optimum Queuing Discipline
OSI	Open Systems Interconnection
PCR	Peak Cell Rate
PDERRM	Pre-deterministic Distributed Event Response Resource Management
Pdf	probability density function
PDF	Probability Distribution Function
PDU	Protocol Data Units
PHB	Per-Hop Behaviour
PSB	Path State Blocks
PSTN	Public Switched Telephone Network
PVC	Permanent Virtual Circuit

QoS	Quality of Service
QPD	QoS Parameter Database
QSD	Queuing and Scheduling Discipline
RR	Round Robin
RSVP	Resource reSerVation Protocol
RSVP-TE	RSVP with Traffic Engineering
RTT	Round Trip Time
SAD	Simulation Actions Domain
SAR	Segmentation And Reassembly
SBM	Subnetwork Bandwidth Manager
SCR	Sustained Cell Rate
SGJW	Stochastic-Gap Jumping Window
SLA	Service Level Agreement
SLS	Service Level Specification
SMTP	Simple Mail Transfer Protocol
SNMP	Simple Network Management Protocol
SP	Simple Priority
SQSTD	Server's Queue State Transition Diagram
SSCS	Service Specific Convergence Sublayer
SVC	Switched Virtual Circuit
TBP	Token Bucket Parameter
TCA	Traffic Conditioning Agreements
TCP	Transmission Control Protocol
TDM	Time Division Multiplex

TE	Traffic Engineering
ToS	Type of Service
UBR	Unspecified Bit Rate
UDP	User Datagram Protocol
UNI	User-to-Network Interface
UPRT	User Priority Regeneration Table
VBR	Variable Bit Rate
VCI	Virtual Circuit Identifier
VoIP	Voice over Internet Protocol
VPI	Virtual Path Identifier
VPN	Virtual Private Networks
VQS	Virtual Queuing System
WAN	Wide Area Network
WFQ	Weighted Fair Queuing
WG	Working Group
WRR	Weighted Round Robin

Contents

CHAPTER 1

Introduction

The notion "***Quality of Service***" (QoS) viewed globally is a key subject for *products* and *services*. It is of great interest to both *service providers* and *end-users* alike. A QoS global view can conceptually be stated as "the degree or measure of satisfaction derived from services."

The concern of this thesis is to examine and discuss a complete set of technologies that have been developed to enable multiservice QoS in the Internet such that it will become a robust global multiservice network. In this introductory chapter, we will discuss the meaning of QoS as it relates to the Internet. We will continue the discussion by justifying why we need a QoS level better and higher in scope than that presently obtainable from the Internet. A shallow dissection and brief look into Internet technologies will follow this. The discussion on Internet technologies will include legacy QoS at each Internet protocol layer whose combinatorial package sum-up to produce a QoS effect known as the ***best-effort*** paradigm. We will also discuss Asynchronous Transfer Mode (ATM), a technology designed to provide multiservice QoS in a network and highlight the argument for and against it as global QoS technology for the Internet. In the same light, we will examine the QoS architectures that have so far been developed for the Internet. We will round up this chapter with a brief presentation of some research problems and the research objective for the work we undertake and then introduce other chapters of this thesis.

1.1 The Internet

The Internet has progressively continued to have a profound impact on the way we live our everyday life, and has become an indispensable part of the way we work and conduct business **[Peu99]** **[Wan01]**. It has been ubiquitous and a powerful invaluable resource in academia, industry and the economic life of most nations. Its popularity and tremendous growth over the past few years has enlisted keen research interest in both academia and the industry. The flexibility and the ability of the Internet Protocol (IP) to run across virtually any network transmission media, and communicate between virtually any system platform, has encouraged the emergence of new applications such as real-time multimedia applications to use the service of the IP Networks **[Mae03] [Sei03]**.

Researchers in academia and industry are trying to exploit the flexibility that led to IP's phenomenal success in bringing convergence of other networks—telephone, radio, and television—to the Internet. The new caption now will be ***"Everything over IP"*** as against the earlier one that says ***"IP over everything"***. The exponentially increasing traffic and the inability of the Internet infrastructure to cope with demands has progressively shown the weaknesses in the IP Networks **[Chen et al.03]**. The situation has been that, service is not denied but gracefully degraded. This is aggravated by the fact that the new breed of applications need a different service model that differs from the service model of traditional Internet traffic. The traditional Internet traffic can tolerate latency (delay) and jitter (delay variations) to some degree. On the other hand, the new breed of applications such as IP telephony or video streaming cannot tolerate latency and jitter at those levels acceptable to traditional Internet applications **[Car et al.02]**. The intolerance of these time-sensitive applications to latency and jitter results in degradation of the received signal, when run over the IP Networks. Degradation or corruption of signals through a network is a situation that nobody likes. Network designers and network users alike are concerned about the quality of service output from the network. The Quality of Service (QoS) issue for the Internet has been a subject of great importance to global telecommunication research communities **[Sol et al.04]**. Judging from the impact that the Internet has had in our everyday life, the subject of QoS Control is of great importance to everybody.

1.2 Concept of Quality of Service (QoS) in the Internet

The Internet provides network service to diverse applications with diverse network performance requirements. It has made possible new applications and ways of communications we never thought possible. Email, e-commerce, digital video streaming, video conferencing, distant learning and Internet telephony are only the beginning of a profound revolution **[Wan01, p.xi]**. Whole ranges of new applications have been developed. Their traffic contests for network resources in rivalry with traditional Internet application traffic. QoS is the network engine that could drive the ***convergence*** of these applications onto Internet Protocol (IP) Networks. As mentioned in Section 1.1 above, the new breed of applications such as voice over IP (VoIP) and video conferencing have network characteristics that are at variance to the network characteristics of legacy applications such as email, file transfer and telnet. The Internet which operates as a datagram network where each packet is transported and delivered individually and independently through the network makes no guarantee for either timely delivery of packets or that the packet will ever be delivered at all. Random congestion could build up in the network that could cause high latency, jitter and packet drop. "The widespread use of World Wide Web (WWW) has caused Internet access to become ubiquitous. However, the exponentially increasing traffic and the inability of the Internet infrastructure to cope with the demands have led to a phenomenon known as the "World Wide Wait" **[DurYar99, p.3]**. IP Networks make all possible effort to deliver their application traffic to their destination, but no assurances are offered. This is known as a ***best-effort*** service. The best-effort service paradigm has thrived well with legacy applications, which could tolerate some inconsistency in network services, but the new breed of applications that are sensitive to timeliness are intolerant of the inconsistency in network services. The new breed of applications, which are mostly real-time applications require more predictable and guaranteed service than offered presently by the Internet. Since the diverse applications have different network characteristics, for the Internet to become a truly multiservice network, it must offer differentiated services to meet the particular need of each of the heterogeneous applications. It must offer predictable, consistent and guaranteed service for a whole range of applications needing such service. These issues which border on poor performance of real-time applications in the network and the general degradation of network performance during peak periods, concern the notion known as ***Quality of Service***, which needs to be addressed.

1.2.1 The Meaning of Quality of Service

The definition given to Quality of Service (QoS) in a multiservice network environment can be subjective in view of its multifaceted nature **[SchWin04]**. In line with this, Geoff Huston refers to QoS as an elusive elephant, in which he narrated the three blind men story: Each of the three blind men in a journey happen upon an elephant and touched different parts of the elephant's body and concluded that they have come across different objects. Huston concluded that different people interpret QoS variably because numerous and in some cases ambiguous QoS problems exist **[Hus00, p.6]**. Thus different people or groups use different shades of syntax to define the term "Quality of Service" in a network, but the connotations, though from different perspectives are the same. ITU-T recommendation E.800 defines QoS as ***"the collective effect of service performance which determines the degree of satisfaction of a user of the service"*** **[HuiIgo03]**. QoS refers to both the performance of a network relative to the applications needs and the set of technologies that enable the network to make performance assurances. In the simplest form, QoS means providing a predictable, consistent and guaranteed data delivery service that satisfies the needs of the end-user. Concisely, David Durham **[DurYav99, p.3]** defines QoS as the performance seen by the recipient (end-user) of an Internet application placed across the network. The performance measure can be taken in a variety of ways, depending on the type of application. For example, the time taken to download a web page, the fidelity of an audio signal placed across the Internet, or the video quality of a real-time video presentation, to mention a few. Measurement techniques for QoS include *subjective quality assessment*—the mean opinion score (MOS) and *objective quality assessment*—use of physical quality parameters **[Tak et al.04]**. QoS is the general ability of the network to differentiate between communication traffic in order to provide different levels of service. QoS functions deal with management and control of network resources in a manner that satisfies the various needs of the diverse applications. The Internet presently treats all application traffic equally, there is no service differentiation.

In considering a QoS real-life analogy, it will be seen that it is a rather an obvious concept. An example of a real life situation on QoS service differentiation is the postal delivery service. Customers usually have a choice of the kind of QoS they want. Letters mailed via first class mail receive a different QoS than the ones mailed via second class mail. In order to get the best out of any type of resource, prudent management is sine qua non. Take a look at the road networks in towns and cities, if traffic lights were not put in place to control traffic on these networks of roads, there would be terrible traffic jams, in other words, great congestion, and the road would become a nuisance to its users.

1.2.2 Quality of Service (QoS) Performance Metrics

QoS performance metrics refers to a set of quantifiable variables or parameters that could be controlled in a network in order to meet the QoS requirements of applications placed across the network. They are a set of quantities that network elements or objects would control in order for networks to make QoS guarantees to applications in terms of providing a certain contracted level of service throughout the application session **[Sha et al.02]**. The following QoS parameters constitute the main component objects of a Service Level Agreement (SLA) / Service Level Specification (SLS):

- Latency
- Jitter
- Packet Loss Rate
- Packet Error Rate
- Packet Rate
- Throughput

Latency: The delay in seconds encountered by a packet as it is being transmitted from source to destination (end-to-end). This quantity is the cumulative addition of ***processing delay or queuing delay, transmission delay and propagation delay***.

Jitter: This refers to variation in latency, which is as a result of randomness in traffic intensity profile in the network. In other words **variances** in end-to-end packet delay across the network.

Packet Loss Rate: The percentage of packets dropped or lost during end-to-end transmission of application's sessions across the network.

Packet Error Rate: The percentage of packets received in error.

Packet Rate: Derived from the desired bit rate (bps) or bandwidth the application would use in order that the network could meet the application QoS requirement.
Throughput: The total amount of packets that the network can move successfully end-to-end in a given period of time. This performance measure quantifies the effectiveness or efficiency of the network.

Added to the above, Heidelberg **[Hei97]** listed other parameters affecting QoS as follows:

- User controlled parameter settings of the application software.
- Characteristic of the application which includes coding, compression, etc.
- Hardware of the end system.
- Geographical separation of the end systems.

1.2.3 Basic Network Resource Elements that determine QoS Parameters

Fundamental network resources elements that control and determine the value of QoS parameters in a network as listed by Robert Malaney **[MalRog]** are:

- Link capacity –the bandwidth.
- Processing speed and power of network nodes---routers, bridges and switches.
- Set of QoS protocols and the architectures that are put in place to handle traffic management and control.

Link capacity is a dominant network resource component that could greatly enhance QoS offerings of a network. Without adequate bandwidth QoS technologies would not be effective. The processing speed and power of network nodes have parallel functions to bandwidth. They together constitute capacity provisioning for the network. The network node's processing speed and power must be made to synchronise with adequate link capacity for a seamless pipeline offering.

Adequate capacity provisioning alone can't adequately address the all round QoS need of the network **[Car et al.02] [Gio03] [Rod et al.03]**. Sporadic and random high intensity congestion (stochastic functions) could occur and jeopardise the QoS needs of some applications even though the network is adequately provisioned. This is where QoS technologies come in. QoS technologies are needed to control and manage the network resources in an equitable manner for all traffic contending for network resources such that, their QoS needs would be met even if random high congestion occurs.

1.2.4 Desirability of Convergence—the IP QoS beauty

The Internet's phenomenal success and its flexibility make it a good candidate for a robust global multiservice network. Convergence of other telecommunication services onto Internet Protocol (IP) Networks is an issue of using one big stone to kill many birds in the Internet. This means turning the Internet into an integrated service network that could effectively support a variety of application's data transmission services, carrying real-time voice, video, and interactive data along with non real-time bulk data transfer. Simply it means, a single cable or medium could integrate or provide a wide range of services over a common underlying network technology. This is the beauty of convergence. The beauty or desirability of convergence is shown in Figure 1.1.

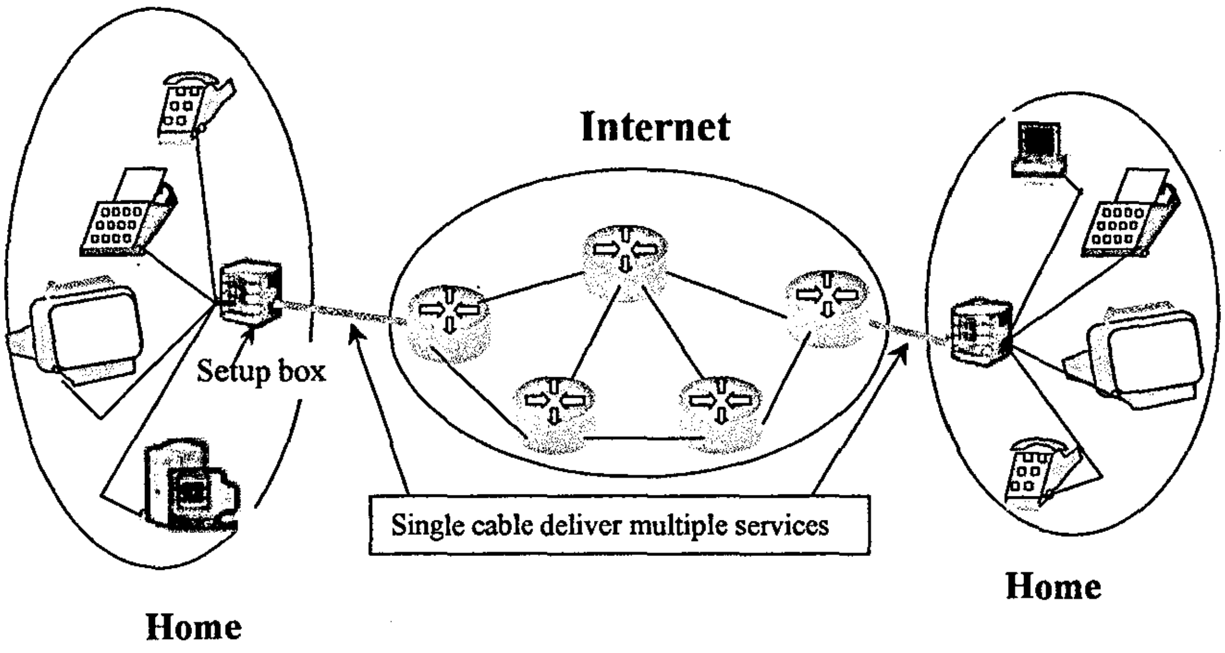


Figure 1.1 Integrated Services Network Showing Beauty of Convergence

Proliferation in development of new diverse applications for use in the Internet has been growing at exponential rates within the last decade. Time sensitive applications such as Voice over IP (VoIP), Internet video conferencing and distance learning are examples of emergent applications converging on the IP Networks. Service providers, corporate organisations and enterprises used to build and support separate networks for voice, video, mission critical and non-mission critical applications traffic. There is a growing trend in recent time in which organisations are now embracing the convergence of all these networks into a single packet based IP Network **[Kli et al.02] [Mae03] [Veg03, p.5] [LinDeV99] [Yan et al.04]**. Convergence has a great appeal to both service providers and end users in view of its numerous advantages.

The obvious advantages for a service provider arise from building, running and maintaining a single network rather than building, running, and maintaining three or more networks. There will be great reduction in both human and material cost in all work phases of the network. This could naturally lead to enhancement in network management and control. Also convergence will guarantee better network efficiency. Since some application traffic is time-critical while others are not, there is always a potential application for any unused bandwidth resources. For example the silent period of time-sensitive applications such as voice which has been a wasted slot in synchronous Time Division Multiplex (TDM) transmission technology could be put to use by inserting a non time-sensitive bulk data packet into the silent slot whenever it becomes available. This will help in maximising the use of bandwidth. Network resources can be fully utilised leaving nothing to waste. Multiservice QoS also will enable interesting applications that can only run over an integrated network. Imagine having a video teleconferencing with several co-workers in which participants can concurrently edit an electronic document. This gives an insight into the power of an integrated network, which could allow for powerful collaboration capabilities.

The advantages of convergence for end users include lower cost and the beauty of having a single set of integrated devices that could deliver all telecommunication services. A single interface to a network providing a variety of services offers an attraction and convenience. For example having the Internet and an Internet phone could allow a person to search online yellow pages for businesses or persons and then immediately call the individual. With enough bandwidth, home users could order and download movies directly to their TVs. The possibilities are only limited by the performance capabilities of the network infrastructure, **[DurYav99, p.19]**. End users will be able to bundle or select the services they need. Summarising the advantages of convergence as highlighted above, Srinivas Vegesna **[Veg03, p.6]** listed IP QoS benefits as follows:

- It enables networks to support the services for existing applications and emerging multimedia applications.
- It gives the network operators capability to control network resources and their usage.
- It provides service guarantees and service/traffic differentiation across the networks. It provides capability to converge voice, video and data traffic to be carried on a single IP Network.
- It enables service providers to offer premium services along with the present *best-effort Class of Service* (CoS*)*. A service provider could rate its premium service to customers as Platinum, Gold and Silver for example and configure the network to differentiate traffic from the various classes accordingly.
- It enables application-aware networking in which network service its packet based on the application information contained within the packet header.
- It plays an essential role in new network offerings such as Virtual Private Networks (VPNs)

1.3 Overview of Internet Technologies

Historically the Internet was conceived as a military and academic project, and was born to serve the U.S. military as a data network to link its computers across multiple bases and institutions around the world **[Wan01]**. Since its inception it has been essentially designed to transport digital data in a popular message unit called ***packets***. The packet switched network was designed to be highly fault-tolerant and can transport data to a varied number of destinations perhaps even under the Armageddon-like conditions of a third world war—doom day situations where most networked nodes and communication links would have been blasted and destroyed **[DurhYav99, p.6]**. The military data network was under the *Defence Advanced Research Projects Agency* (DARPA*)*. Its power and flexibility led to the creation of the *National Science Foundation Network* (NSFnet) which gave birth to today's Internet.

Standard Internet Protocol (IP) Networks provide best-effort data delivery by default. The term "best-effort" as earlier explained in Section 1.2 is used to describe the functionality of IP Networks in doing a good job of transporting data from its source to its destination, but with no commitment or guarantee. Best-effort service in IP Networks allows complexity to stay in the end-hosts, while the core network (network of backbone routers), is made relatively simple. This results in a simple network that scales well, as is evident from the Internet's phenomenal growth. The Internet has undergone great evolution since its inception. As a result of the potential power it wields in the global telecommunication industry, its processes of metamorphosis have been laden with intense research activities in the research community of the telecommunication sector all round the globe. This cumulated in its flexibility being exploited in trying to maximise the advantages that could be derived from its ductile adaptability to various telecommunication needs. As earlier mentioned, a wide range of applications has been developed to use the network services of the Internet. The new emergent applications could not thrive well because the Internet was not designed originally to support their specific network characteristic needs. Thus the Internet service model necessarily must be extended **[IntServ94] [RFC1633]**.

1.3.1 Technological Nature of the Internet

The Internet is an acronym for inter-network. It is a **packet switched** network in which data are transmitted in discrete, self-addressed datagrams of information known as ***packets***. The packet generally contains the ***header*** and the ***payload*** fields. The payload is the bulk user data, and the header consists of sets of information such as, source and destination addresses and other management and control information that would help in safely transporting the packet from its source to its destination. The concept of a packet switching network is similar to the postal system, where a letter or a parcel is put in an envelope that bears the source and destination address before being posted. At the Post Office, sorting, routing and transportation take place, before the letter finally gets to its destination. In this case writing the letter can be seen as one layer of the process, addressing the envelope, posting, sorting, routing and transporting as each separate layer process of the postal system. In an almost similar analogy the Internet, which operates on the

Transmission Control Protocol/Internet Protocol (TCP/IP) suite, operates as layered protocol systems **[Com01] [Ste00] [Tan96]**. The layer model consists of (from the top) *application layer, the transport layer, the network layer and the link layer* as shown in Figure1.2.

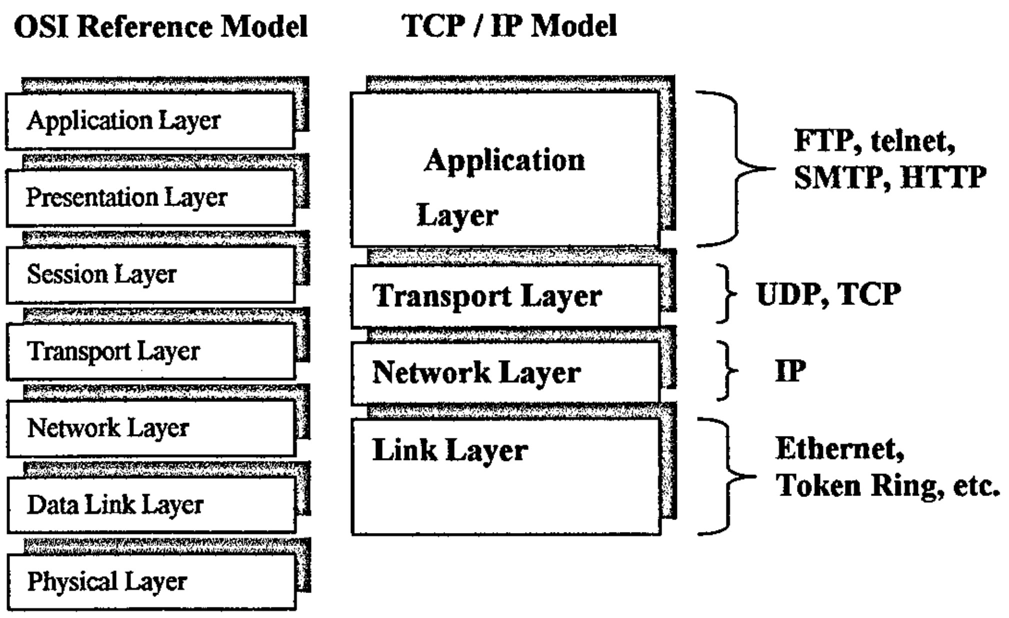


Figure 1.2 Protocol layer structure illustrated with OSI Reference Model and TCP/IP Standard Model.

The **link layer** in the Internet model could be any of the ***Local Area Network*** (LAN) technologies such as Ethernet or Token Ring. The Figure 1.2 shows the layer model of two important computer networks, the OSI (Open System Interconnection) reference layer model and the TCP/IP standard layer model. Most networks are structured as layered models in order to reduce design complexity and allow for modularization or segmentation of network functionality. The standard protocol layer models are a set of universally accepted communication protocols that allows otherwise incompatible network technologies to communicate. This is simply possible because unrelated technologies could provide a mapping to the standard. The standard acts as a common protocol language allowing diverse network technologies to communicate with each other. Thus the Internet which is made-up of a conglomeration of diverse network technologies enables computers in different remote geographical locations to communicate effectively together even though they may be operating on different networked technologies. Starting from the application layer, each layer depends on the one directly below it for its network services. The message is encapsulated (i.e. put into envelopes) from top-to-bottom and then transmitted end-to-end (sender-host-to-destination-host), en-route via the routers. At the destination-host, the data climbs up the protocol ladder from bottom-to-top shelving its envelope as it climbs up before it finally become useful data at the top application layer.

Technical nomenclature in telecommunication differentiates between the word ***internet*** (note the lowercase *i*) and **Internet** as will be seen shortly. The ***internet*** is a conglomeration of interconnected autonomous network systems **[Ste00]**. The autonomous network systems may be a single or sets of ***Local Area Network*** (LAN) interconnected to form a ***Metropolitan Area Network*** (MAN) and this in-turn interconnected to form a ***Wide Area Network*** (WAN). Regional WAN can be interconnected to form global WAN that may be referred to here as the ***internet*** **[Ste00, p.16]**. The LANs and the MANs in an ***internet*** are made up of diverse network technologies. The ***Routers, Bridges and Gateways*** that are used for the interconnections generally do the necessary translation in both hardware and software between the heterogeneous network technologies. The **Internet** is a specific worldwide ***internet*** operating on the TCP/IP protocol suite. This is generally referred to as an IP Network.

1.3.2 Internet Protocol and Communication Mechanism

The goal of a standard protocol suite is to efficiently achieve inter-networking by providing bridging functions between a diverse set of computer network technologies. The core protocols of the TCP/IP suite are the Internet Protocol versions 4 and 6 (IPv4 and IPv6), the User Datagram Protocol (UDP) and the Transmission Control Protocol (TCP). The Internet communication mechanism is built on the four protocol-layers, the *application layer*, the *transport layer*, the *network layer*, and the *link layer* **[Tan96]**. We will now briefly summarise each of these layers in relation to protocols built into them and the part they play in the mechanism of data communication in the Internet.

1.3.2.1 The Application Layer

This layer houses the environment where applications are used. The applications may be email (SMTP), file transfers (FTP), web browsers (HTTP), video or audio applications or whatever the application programmer has invented or will invent. Within this layer, each of these applications has it's own specific built-in protocol or mechanism for effective network communication. Included in the mechanism is the *Application Programming Interface* (API) which provides an interface through which the application can communicate with the network **[RFC 1122]**.

1.3.2.2 The Transport Layer

In the TCP/IP suite, the transport layer consists of two types of transport protocol, which are the User Datagram Protocol (UDP) and the Transmission Control Protocol (TCP). They provide services for the application layer. Each of these transport protocols can encapsulate or de-encapsulate application traffic and provides specific services such as multiplex and de-multiplex functions for applications. Factors that would influence the choice of either UDP or TCP will depend on the application traffic characteristics and Quality of Service (QoS) need in the network.

1.3.2.2.1 User Datagram Protocol (UDP)

UDP is a simple transport layer protocol **[RFC 768]**. It is useful for transporting data over the Internet when timely delivery out-weighs other considerations. The UDP ***datagram*** consists of a simple header, followed by the bulk data (i.e. the information) to be transported. The header consists of ***source*** and ***destination port*** fields for identifying applications at the end stations, a ***length*** field to determine the total length of a datagram, and a ***checksum*** field to detect corruption of the datagram. UDP is a connectionless transport protocol. Unlike its counterpart the TCP, it does not provide mechanisms for the reliable delivery of the datagram it transports. There are no flow control or retransmission facilities.

1.3.2.2.2 Transmission Control Protocol (TCP)

TCP is a more complex transport protocol than UDP. It provides a number of sophisticated connection-oriented data communication services, which include end-to-end virtual connectivity, reliable transport, and flow control over the Internet **[RFC 793]**. The TCP packet has an advanced header format with a lot of built in traffic management and control functions. The header consists of ***source*** and ***destination port addresses*, a *sequence number*, *acknowledgement number*, *data offset value*, *six control bit value*, *window field*, *checksum field*, *urgent pointer field* and an *option field*.**

1.3.2.3 The Network Layer

The Internet Protocol (IP) is the communication protocol that operates on the network layer of the TCP/IP suite **[RFC 791]**. IP would provide services (encapsulation and forwarding to link layer) for either UDP or TCP packets coming from transport layer and as well provide services (de-encapsulation and forwarding to transport layer) for traffic from the link layer. IP is the glue that joins all the diverse and disparate network technologies together **[Fod et al.03]**. It is the network amalgamator whose function includes mediation, interpretation and translation in hardware and software for the diverse network technologies employed for communication within the Internet. The protocol's header consists of information necessary to effectively transport the data portion end-to-end across the Internet. This includes the Internet source and destination addresses as well as control and management fields. In total, the protocol header is a minimum of 20 bytes long in IPv4. The header fields and functions are as follows:

- ***Protocol version number,*** field is used to identify the version of the IP header format. It would be either IPv4 or IPv6.
- ***Internet header length*** (IHL) field represents the length of the header as a number of 32 bit chunks. This value is used to determine the offset to the data portion of the IP packet.
- The ***Type of service*** field is used to determine precedence of the packet over others.
- The ***Total length*** field is the overall size of the packet measured in bytes.

- The next three fields, which include ***fragment identification***, ***flags*** and ***fragment offset***, are used for packet fragmentation management.
- The ***Time to live*** number field is used to control the lifetime of the packet in the network.
- The ***Protocol*** field identifies the transport layer protocol encapsulated in the data portion of the packet.
- The ***Header checksum*** field is used to determine if the packet header has been corrupted during transportation.
- The ***source* and *destination address*** fields contain the IP address of the message originator and its intended receiver. IP address must be unique throughout the network so that network computer can be uniquely identified.

Routing a packet towards its destination is one of the major functions of the IP in the network layer. A number of routing protocols have been developed depending on the network topology to give IP capability to efficiently route packets to its destination.
An IP packet may directly transport ***Internet Control Message Protocol*** (ICMP) in its data field. ICMP provides error-reporting capability with regard to handling IP packet by network nodes. Node that discards packet for any of the possible number of reasons will generate an appropriate ICMP message and send it to the packet source.

IP with all the elaborate communication mechanism as derived from its header fields functionality together with other communication protocols, provides best-effort data delivery by default. Best-effort in the sense that IP will try its best to transport the data to its destination, but no guarantee that the data will ever be delivered or delivered on time. This is so because the IP Network is made from a conglomeration of diverse network technologies with diverse QoS. Along the data path in the disparate network, congestion could build-up, packets could be dropped and nodes could breakdown to mention a few of the situations that could delay or prevent data delivery. Thus IP Networks (best-effort service) do not provide service guarantee **[HsiSiv05]**.

1.3.2.4 The Link Layer

The link layer in the Internet architecture could employ any link layer network protocols and technologies. This could be Ethernet, Token Ring, FDDI, ATM etc, as long as the *data link protocols* and the *physical layer protocols* can encapsulate the IP packet. The link layer protocols provide the means and mechanism for physical connectivity between the networked nodes in the Internet. In the physical layer (a sublayer of link layer), a collection of raw bits in unit called *frame* are transmitted from one node to another until the data gets to its destination.

Summarily, the Internet layered communication mechanism, in structure, can be seen to operate as a kind of ***unique relay system*** as explained here and shown in Figure 1.4. The message or application traffic is generated at the top application layer and passed to the transport layer. The transport layer after adding its own functionality to the application traffic passes the data to the network layer. The IP process at the network layer performs operation on either TCP or UDP datagram and passes the packet to the link layer. At the link layer IP packet is called ***frame***. The link layer protocol process will relay the frame through its own internal sublayer until the frame gets to the physical layer where it is transmitted hop-to-hop in the network until it gets to its destination.

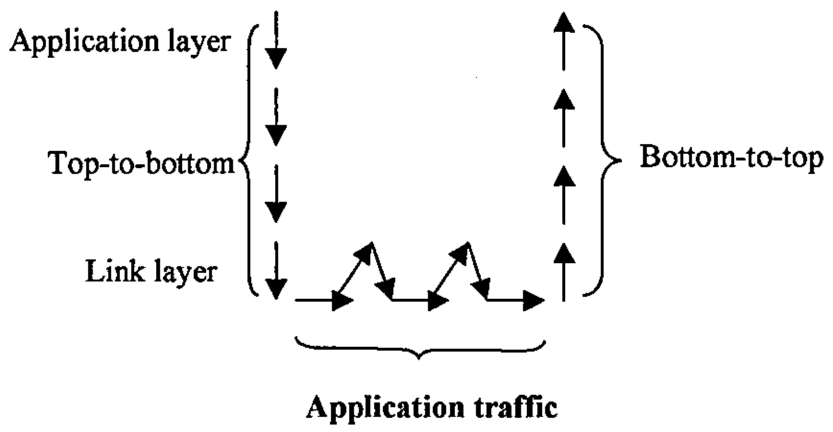


Figure 1.4 Modulated U-shape relay system as illustration of application traffic movement from top-to-bottom, end-to-end and bottom-to-top in the layer model of TCP/IP Networks.

The frame as it travels towards its destination, will have to climb up and down the sublayers of the link layer protocol of each node or hop along its path to its destination. When the frame-networked message gets to its intended destination, the encapsulated message climbs the protocol ladder from bottom-to-top, de-encapsulating or shelving its envelope (i.e. its header) as it climbs up until it gets to the application layer where it becomes useful data or a message. Thus the Internet communication mechanism could be illustrated with a ***modulated U-shaped relay system***. As shown in the Figure1.4, the modulated U-shaped relay system would have its left vertical side represented by application traffic moving from top-to-bottom of the protocol ladder. While the zigzag horizontal side (modulated side) could be represented by transmission of application traffic end-to-end through system of hops, and the right vertical side could be represented by the traffic moving from bottom-to-top of the protocol ladder.

1.4 QoS at each Protocol Layer in the Internet

In view of the layered structure of the Internet architecture, QoS could be achieved in any of the protocol layers. Various QoS protocols and mechanisms have evolved in the past largely to satisfy the need of legacy traffic and to some extent new applications such as multimedia traffic. The sum total of QoS offerings in each of the protocol layers yields the value best-effort service model, which is purely egalitarian. There is no preferential treatment, all traffic receive equal treatment. We will briefly mention below the QoS efforts in each of the protocol layer.

1.4.1 QoS in the Application Layer

Various QoS mechanisms were developed to enhance the network performance of the applications in the application layer environment. These include various compression techniques and a host of coding techniques to adapt application traffic to network conditions and to economise the use of bandwidth. Others include development of various application protocols and standards to meet specific needs of application users. In the realm of compression and coding techniques a few examples are:

HTML (HyperText Markup Language) for publishing information in the Internet.

The "**gzip**" (command used in Unix operating systems) uses Lempel-Ziv coding to compress text based data for economic use of buffer space and bandwidth.

- The **Moving Picture Expert Group** (MPEG) is used for video and audio coding and compression
- **MPEG Audio Layer-3** (MP3) is used for audio coding.
- The **International Telecommunication Union** (ITU)- **H320** is used for audio/ video coding.

In the area of application protocols and standards, the Internet has witnessed the increasing availability of a number of application protocols and architectures that essentially offer the same service but with different performance offerings. Examples include:

- Internet Explorer, Netscape Navigator, etc are used for web browsing.
- Microsoft Outlook Express, Eudora, Elm and Pine are used for e-mail.

These applications are content-based applications with built-in utilities to satisfy the various needs of various users. Application developers have also endeavoured to cater for the QoS need of real-time multimedia applications in the application layer environment by designing and developing ***playback buffer*** to cushion the effect of network vagaries such as jitter.

1.4.2 QoS in the Transport Layer

The two transport layer protocols–Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) use a ***checksum*** facility in the header field of the datagram they transport to check the integrity of the datagram. TCP is a connection oriented transport protocol with built-in traffic management and control mechanisms. It has both a packet rate control and a congestion control mechanism that will reduce packet loss and build-up of packet delay. Each correctly received TCP packet is acknowledged thereby providing guaranteed service to the sender. TCP QoS rendering is suitable for legacy applications, however it cannot provide a consistent level of service that will satisfy the need of real time applications.

1.4.3 QoS in the Network Layer

Although the Internet Protocol (IP)–the main protocol at the network layer does not provide guaranteed service, it has implicit QoS provision. The Type of Service (ToS) byte field in IPv4 packet header, which has so far not been used, was designed for QoS control. A checksum facility is also available in the packet header to check if the packet header has been corrupted. IP can use a protocol, such as Internet Control Message Protocol (ICMP) to communicate error messages such as packet loss. It also could be used to transmit or relay various control messages and configuration information from receiver host to sender host. These functions, add together to enhance QoS for packet delivery in the network layer.

1.4.4 QoS in the Link Layer

The Internet link layer is made-up of diverse network technologies with disparate QoS offerings. These include the various types of link layer technologies as found in the Institute of Electrical and Electronic Engineers (IEEE) Project 802 standards for Local Area Network (LAN) technologies. The IEEE 802-style of LAN technologies includes Token Ring, Ethernet and Fiber Distributed Data Interface (FDDI) among others. Although they generally provide the same best-effort service to datagrams from higher-layer protocols such as Internet Protocol (IP), each of them traditionally has its own style of QoS built into it. Token Ring or FDDI has priority built into their ***frame*** (unit of information at the link-layer) transmission mechanism, to offer limited differentiated service. This allows preference to be given to some time sensitive traffic at the expense of less time sensitive traffic. Shared media link-layer technology such as Ethernet or Token Ring has a contention resolution algorithm built into its medium access mechanism to limit packet loss as a result of traffic collision. The Ethernet Carrier Sense Multiple Access with Collision Detention (CSMA/CD) is a highly efficient and popular media access mechanism. Sublayers of the link-layer QoS mechanism will ensure that a bit '1' sent is received as bit '1' not as bit '0'. This is achieved by an error control algorithm such as checksum and cyclic redundancy checks (CRC) which the hardware or the software must calculate to ensure the frames are received correctly. Other QoS mechanisms include detection of lost frames by ensuring that correctly received frames are acknowledged. When an acknowledgement frame for a received frame is lost, the Data Link layer software ensures that the frame is not retransmitted to prevent duplication of the frame. Frame Relay is a popular wide area network (WAN) technology with limited QoS offering. Operating at layer 2, it has a congestion control mechanism as signified by the Forward Explicit Congestion Notification (FECN) bit and Backward Explicit Congestion

Notification (BECN) bit in the header of the frame it transports. It also has built-in basic prioritisation by using the Discard Eligibility (DE) indicator bit in the frame header. Other popular link-layer technology such as Asynchronous Transfer Mode (ATM) needs to be closely looked into in view of its QoS offering, as we will see in the next section.

The focus on QoS offerings in each protocol layer of the Internet as highlighted above has been essentially based on the traditional Internet architecture which additively sum up to best-effort service. Obviously the existing QoS protocols and architectures in the Internet are inadequate to meet the QoS requirements of integrated services. The power and advantages of integrated service network have been identified long ago and the first concrete effort towards producing such network technology was the birth of ATM, which we will examine in the next section.

1.5 Initial Effort on Integrated Service Network (ATM technology)

Early effort on multiservice networks was concentrated on design and development of Asynchronous Transfer Mode (ATM) network technology that was seen as solution for broadband integrated service delivery. The ATM fundamental concept has been to build a network technology, which incorporates support for integrated services. Its architecture has been designed to support various needs of application QoS in multiservice network environment **[Peu99]** **[Tan96]**. An ATM network is composed of ATM end nodes or hosts and ATM switches. Its concept is to carry all traffic types within the same network infrastructure with all data being transmitted in equal size packets known as ***cells***. It is a connection oriented and fast cell switching protocol. In order to simplify switching and multiplexing operations, each cell has two address fields, the Virtual Path Identifier (VPI) and Virtual Circuit Identifier (VCI). A Virtual Path (VP) can contain a number of Virtual Circuits (VCs). A VC must be setup across the ATM network before any user data can be transferred between two or more ATM attached devices. Fundamentally ATM technology has two Classes of Service (CoS), the Constant Bit Rate (CBR) service and the Variable Bit Rate (VBR) service. The VBR service is further grouped into a range of service classes to cater for diverse application QoS needs. The ATM layer structural model, which could be seen as the high-level view of its architecture, is shown in Figure1.5.

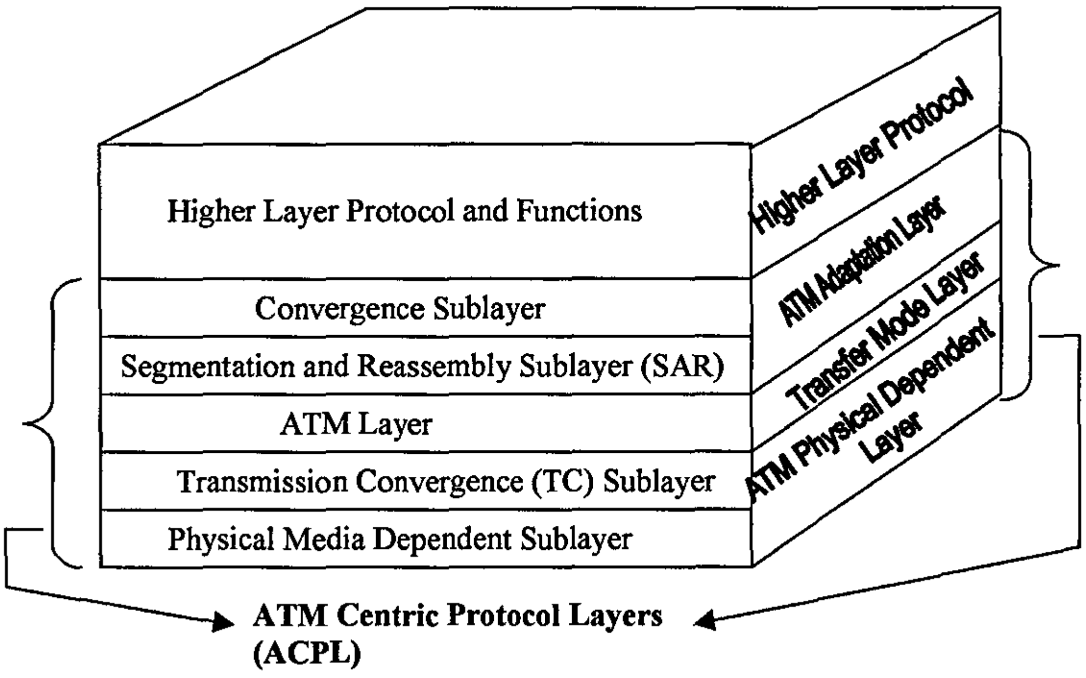

Higher Layer Protocol and Functions
Convergence Sublayer
Segmentation and Reassembly Sublayer (SAR)
ATM Layer
Transmission Convergence (TC) Sublayer
Physical Media Dependent Sublayer
Higher Layer Protocol
ATM Adaptation Layer
Transfer Mode Layer
ATM Physical Dependent Layer
ATM Centric Protocol Layers (ACPL)

Figure 1.5: Showing ATM layered architecture reference model

The Higher Layer Protocol objects, which are not unique to the ATM architecture, are passive in relation to QoS control in the network. By this we mean the higher layer protocol objects do not directly manipulate entities that work together to enable message transfer with QoS features protected. The remaining layers consist of protocol layers that are specific to ATM technology and could be referred to as ATM Centric Protocol Layers (ACPL). The ACPL layers, as shown in Figure1.5 consists of ATM Adaptation layer, ATM (transfer mode) layer and ATM Physical Dependent layer. We will briefly discuss each of the ACPL layers in other to graphically capture the essentials of ATM message transfer mechanism with regard to QoS capability.

1.5.1 ATM Adaptation Layer (AAL)

The AAL is the layer that provides service to the higher layer protocol. It isolates higher layers from the specific characteristics of the ATM layers. It maps higher layer Protocol Data Units (PDU) into information field of cells, and it reassembles them into original PDUs. AAL provides service-specific functionality to support specific service for application traffic. It consists of two sublayers: the Convergence Sublayer (CS) and Segmentation and Reassembly (SAR) Sublayer.

1.5.1.1 The Convergence Sublayer (CS)

The CS also consists of two sublayers, the Service Specific Convergence Sublayer (SSCS) and the Common Part Convergence Sublayer (CPCS).

- **The SSCS** functions depend on actual service requirements of specific application traffic. SSCS provides service for all traffic, both real-time and non real-time. Different real time services require different SSCS, hence multiple AAL services are defined through this function. The different classes of service that have been defined are highlighted in Section 1.5.5.
- **The CPCS** provides common AAL operations such as multiplexing or demultiplexing and cell-loss detection.

1.5.1.2 The Segmentation and Reassembly (SAR) Sublayer

At the sending node, SAR takes CS-PDUs from the layer above and breaks them up such that they are 48 bytes in size after addition of SAR header and tail to become AAL frame before sending it to ATM layer. At the receiving node it joins cells from ATM layer together to form higher layer PDUs. Thus its functions include fragmentation of PDUs to form payload fields of cells and re-assemble of ATM cells to form PDUs.

1.5.2 ATM (Transfer Mode) Layer

The ATM layer is the heart of ATM technology that provides the mechanism for connection-oriented message transfer since it operates at the cell-level. It attaches a header of 5 bytes to an AAL-SAR frame (cell payload) which should be 48 bytes long to make a cell that is 53 bytes long. The use of equal size cell in ATM technology will help to reduce the effect of jitter (delay variation) that arises from queuing, scheduling and transmission of various packet sizes in other packet switched network technologies. It embraces high speed switching capability to reduce traffic delay. Essentially ATM has two types of connections or VCs: Permanent Virtual Circuits (PVCs) and Switched Virtual Circuits (SVCs). PVCs are usually static and require a manual or external configuration to set them up.

SVCs are dynamic and are created based on demand. Their setup requires a signalling protocol between the communicating ATM nodes **[Veg03, p.169]**.
ATM layer's cell-level of operation is independent of the physical layer. Thus any of the physical media technology could be used to carry ATM cells as long as it will support ATM QoS features. The main functions in this layer include:

- Cell routing, multiplexing and demultiplexing.
- The use of virtual path identification (VPI) and virtual circuit identification (VCI) for flexible connection-orientated transfer mode.
- Cell-level monitoring of access rate for connections.
- Cell buffer management and congestion control
- Passes ATM-Service Data Units (ATM-SDU) between peer AAL entities.

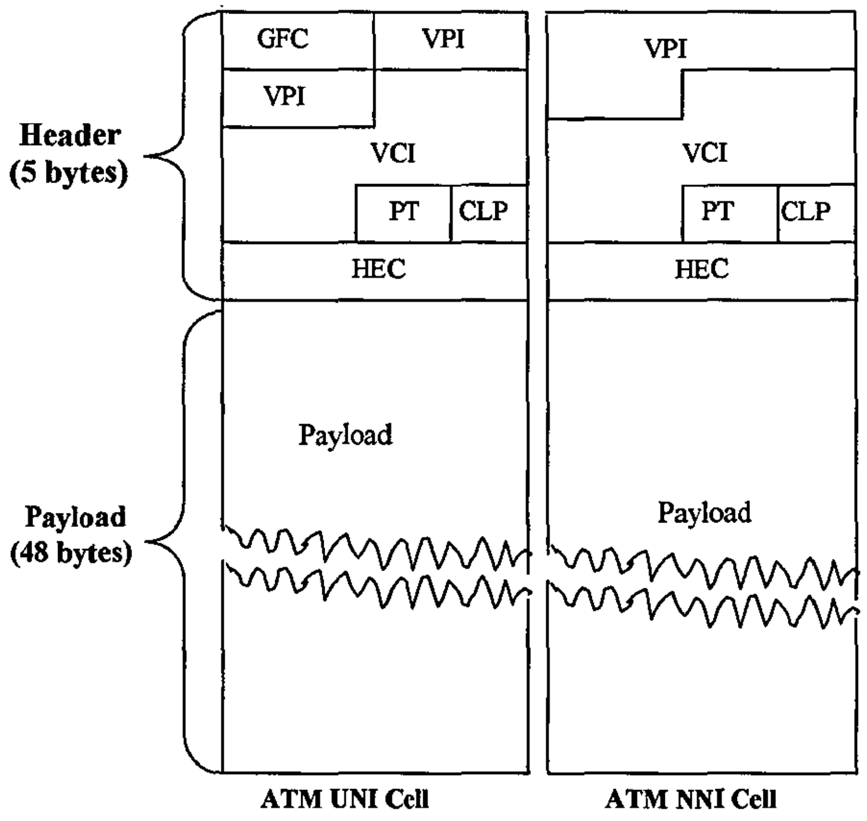
The key concept in ATM cell transfer mode is determinism, which arises from VCs being setup end-to-end before data can be transfer and which consequently minimises delay. A deterministic system, which has a variance that tends to zero will certainly gives the best delay profile.

1.5.3 ATM Cell Structural Types

As mentioned above ATM cell is 53 byte long–5 bytes of header information and 48 bytes of user information or payload. The cell header contains information the ATM switches used to switch cells. There are two types of ATM cell format: the User-to-Network Interface (UNI) defines the format for cells between a user and ATM switch; and the Network-to-Node Interface (NNI) defines the format for cells between switching nodes. The cell structural types are shown below in Figure 1.6.

GFC is a 4-bits field used to control traffic flow from end stations to the edge of the network. It is not used within the core network, NNI cell format thus has no GFC. The GFC has local flow-control importance to the user for traffic flow control. ATM end nodes shape their transmission in accordance with the value present in the GFC field. Communication mechanisms within ATM networks have two modes of operation: traffic that enter the network without GFC-based flow control (NNI traffic), are referred to as ***uncontrolled access;*** and traffic with GFC-based flow control (UNI traffic) are referred to as ***controlled access.*** According to

Srinivas Vegesna **[Veg03, p.170]** most UNI implementations do not make use of the UNI field.



GFC: Generic Flow Control
VPI: Virtual Path Identifier
VCI: Virtual Channel Identifier
PT: Payload Type
CLP: Cell Loss Priority
HEC: Header Error Control
UNI: User-to-Network Interface
NNI: Network-to-Node Interface

Figure 1.6: ATM cells showing UNI and NNI cell header formats

VPI is made up of an 8-bits to 12-bits field depending on its implementation. A Virtual Path (VP) consists of a bundle of Virtual Circuits (VCs) and is assigned to a Virtual Path Identifier (VPI). ATM switches could implement group routing decisions by switching VPIs along with all the VCs within them.

VCI is a16-bits field, used to identify a logical path a cell takes from one node to another. Each VC within a VP is assigned a virtual channel identifier (VCI). VPI and VCI fields are used by ATM switches in making switching decisions.

PT is assigned to a payload type identifier (PTI) which has 3-bits field. PTI is used to identify the type of payload carried by the cell. A payload may either be user data or operation, administration and maintenance (OAM) information.

CLP is a 1-bit field used to indicate cell priority. When congestion occur in the network and buffers overflow, a cell with CLP set (i.e. CLP = 1) will be discarded before cell whose CLP is not set (CLP = 0). Thus Cell with CLP = 0 has higher priority.

HEC is an 8-bits field used for detecting and correcting the errors in the cell header.

1.5.4 ATM Physical layer

The Physical Layer of ATM should be physical media independent. The main requirement is that the physical media should support high-speed networking and broad bandwidth to support multiservice. The layer is viewed as consisting of two sublayers, the Transmission Convergence (TC) sublayer and Physical Media (PM) Dependant sublayer.

- The **TC sublayer** provides cell level interface to higher layers and performs the Header Error Check (HEC) for each cell as well as cell delineation. Its function also includes any channel coding needed by the Physical Media Dependant sublayer.
- The **PM Dependant** sublayer provides functionality for interfacing with the actual communication channel. The functionality will include both the electrical and mechanical components required for the interfacing and also the production and interpretation of the actual waveform representing the bits of the cells. Issues concerning transfer of timing information between the two ATM entities are also handled.

1.5.5 QoS Mechanism and Service Classes in ATM

QoS guarantees are achieved through a deterministic approach adopted by ATM end systems in sending traffic to the network. ATM end nodes explicitly specify a traffic agreement describing their intended traffic flow characteristics to the network. The flow descriptor carries QoS parameters such as Peak Cell Rate (PCR) Sustained Cell Rate (SCR) and Maximum Burst Size (MBS). ATM edge nodes police each user's traffic characteristics to ensure it is in accordance with the traffic agreement offered in the network. In this mode of operation, establishment of traffic agreement and protection of the traffic agreement offers guaranteed service.

The ATM Adaptation Layer (AAL) provides the mechanism and flexibility to support different traffic services carried within the same format. Five AAL service types are defined **[Bou et al.03]**, and these are:

- **AAL 1:** It is a Constant Bit Rate (CBR) connection oriented traffic transfer service with synchronised transmit and receive timing. Suitable for applications with tight time constraint e.g. digitised voice at 64 Kbps and video (H.320 standards).
- **AAL 2:** It is a real-time Variable Bit Rate (rtVBR) connection oriented service with synchronised transmit and receive timing, in which the traffic source may be bursty. Can find application in packetised video and some interactive multimedia traffic.
- **AAL 3:** It is a non real-time Variable Bit Rate (nrtVBR) connection oriented with no strict timing relationship between transmit and receive times. E.g. connection-orientated interactive data transactions.
- **AAL 4:** This service is referred to as Available Bit Rate (ABR). It is in fact a VBR service with no synchronised transmit and receive timing. This service is suitable for applications that thrive well under the Transmission Control Protocol (TCP) service in Internet Protocol suite.
- **AAL 5:** This service is referred to as Unspecified Bit Rate (UBR). It is a VBR service with no congestion control. There is no bandwidth reservation nor delay and jitter bound for this service. This is an equivalent of IP best-effort service. Suitable for massive file transfer such as system backups.

The highlight of ATM technology as discussed above shows that ATM network technology has built-in mechanisms to support multiservice QoS. The issue is that, ATM is just one network technology, it is not the Internet, which is made-up of a conglomeration of diverse network technologies. While some schools of thought in the research community believed that the Internet should be re-invented on the basis of ATM networks, others believed that, the flexibility of the Internet Protocol (IP) should be exploited to extend the service model of IP Networks to support multiservice QoS. The pendulum is swinging in favour of the latter school of thought. Also it has been reported that ATM has its own touch of scalability problem **[DeM et al.00]**.

1.6 QoS Provisioning beyond Best-Effort Service Paradigm in IP Networks

In view of the importance of multiservice QoS provisioning in IP Networks, various standard organisations, forums, working groups and corporate bodies are presently working to ensure the dream of convergence of telecommunication services on to the IP Networks become a reality **[Goz et al.03]**. They are concerned with the development and implementation of QoS protocols and architectures to turn the Internet into robust global multiservice networks. The draft and review of standards, and the formulation of policy guide lines for QoS provisioning in the Internet is gathering momentum every day, and speeding towards its targeted goal. We will briefly highlight the contributions that have been made by the various bodies below.

1.6.1 Contributions from Standard Organisations

Standard organisations that have made concrete effort to the design and development of QoS protocols and architectures for IP Networks include, Internet Engineering Task Force (IETF) and Institution of Electrical & Electronic Engineers (IEEE) to mention those that readily come to mind. These organisations are working top-down in terms of the layered protocols, and from end-to-end in terms of network mechanism to produce QoS models that will be robust and scalable when deployed in the Internet. Some of these standard organisations are working on specific aspect of QoS provisioning to achieve the goal of Integrated Services in the Internet. Prominent QoS protocols and architectures that have been developed are:

- **Integrated Services (IntServ)** architecture, developed by IETF Working Group on integrated services in the Internet. The IntServ architecture consists of a number of building blocks for resource allocation in the Internet. IntServ together with its signalling protocol–the Resource reSerVation Protocol (RSVP) provide means for per-flow resource allocation in the Internet.
- **Differentiated Service (DiffServ)** architecture is developed by another IETF Working Group, working on scalable QoS provisioning in the Internet.

DiffServ architecture is designed to extend the functionality of the Type of Service (ToS) byte field in IP version 4 (IPv4) packet header to offer scalable service differentiation in the Internet.

- **Multi-Protocol Label Switching (MPLS)** is also designed and developed by another IETF Working Group. MPLS represents the convergence of connection-oriented forwarding techniques as found in the use of virtual circuit forwarding technique of ATM and IP routing protocols forwarding paradigm. It could solve problems of QoS routing in the Internet.
- **Subnetwork Bandwidth Manager (SBM)**— yet again, a design and development of IETF Working Group. SBM is an extension to RSVP signalling, and support RSVP-based admission control over the shared medium LAN family of IEEE 802-style networks.
- **IP over ATM**—A Forum of IETF has been working to take advantage of QoS capabilities of the ATM networks in mapping IP precedence values in the Type of Service (ToS) byte field of IPv4 packet header to ATM AAL service classes.
- **IEEE standard on Media Access Control (MAC) Bridges**–IEEE 802.1D has been extended with IEEE 802.1p to support the notion of expedited traffic transmission capabilities. Time-critical traffic could be tagged with a high priority tag such that they enjoy preferential treatment in their transmission from source to destination in the network.

A plethora of IP QoS provisioning titles exists in telecommunication literature, which is impossible to list here in view of time and space **[Myk et al.03]**. We have only filtered out the most prominent standard ones, and highlighted them above. Each of the above mentioned QoS protocols and architectures will be discussed in more detail in Chapter 2.

1.6.2 Contribution from Corporate Organisations and Equipment Vendors

Many companies have produced network devices to implement some of the above listed standards on QoS protocols and architectures. One example of such a company is 3Com, which announced that it has produced high-performance switches that support some of the QoS architecture listed above **[3ComPro&Srv]**.

Cisco Systems also announced the development of its convergence architecture dubbed AVVID (Architecture for Voice Video and Integrated Data), **[eWeek99]**. 3Com, Cisco, and many other network device vendors have taken giant strides towards the realisation of the convergence of Telecommunication Services unto IP Networks.

1.7 Research Problems and Objective of the Research Work

The last two decades have witnessed high levels of research activities on provisioning of Quality of Service (QoS) in IP Networks **[Gio et al.03]**. As mentioned earlier in the previous sections of this chapter, there has been a plethora of publications on the topic and a number of IP QoS architectures and standards have been developed. Despite the elaborate work that has been done and concluded on the topic, IP QoS deployment end-to-end still remains elusive in the Internet **[Gio et al.03] [HsiSiv05] [Krap00] [MooSil03]**. It has not been fully deployed in commercial real life networks. What have been witnessed are ad hoc test networks where IP QoS has been deployed to test its commercial viability. It is therefore correct to state that, there is yet to be equilibrium between the demand and supply on the subject. There still exist some research questions that need to be answered, and some research issues that need to be addressed and resolved on this very popular and very difficult topic.

1.7.1 Research Problems

It has been mentioned in Section 1.6.1 of this chapter that, IntServ and DiffServ architectures are among the prominent QoS technologies that have so far been developed to extend the service model of the Internet to embrace integrated services. The IEEE 802.1p standard can provide limited QoS in layer 2. Although each of the architectures will be discussed to some details in Chapter 2 of this thesis, we need to highlight their salient points in order to add fullness to the discussion points in this section.

1.7.1.1 Integrated Services (IntServ) Architecture

The IntServ architecture is one of the standards QoS architecture developed to transform the Internet to a robust multiservice network, as described in **[RFC 1633]**. It serves to support:

- Real-time multimedia service in the existing Internet.
- Controlled resource sharing.

In achieving its objective the architecture proposes to extend the existing best-effort service model in the Internet to embrace multiservice QoS. One important assumption made in the development of the IntServ model is that the network resources can be explicitly controlled. This means that ***Resource Reservation*** and ***Admission Control*** are the main basic blocks of the IntServ model. Although other basic blocks such as packet schedulers and packet classifier exists in the implementation framework for IntServ model, resource reservation and admission control functions come first in network resource allocation mechanism of IntServ model. **Resource reSerVation Protocol (RSVP)** is the resource reservation signalling mechanism developed for implementation of the IntServ model. RSVP has been found to be too complex and it's per flow paradigm is not scalable in the Internet **[LeoMas03] [Gio et al.03] [Mol et al.05] [Bak98]**. This is an example of a research problem that needs to be addressed.

1.7.1.2 Differentiated Services (DiffServ) Architecture

DiffServ architecture is designed to provide scalable service differentiation in the Internet without the need for maintaining Per-flow State or do per hop signalling **[RFC 2475]**. The architecture achieves scalability by aggregating traffic flow into groups of traffic classes for network resource allocations and services. The Differentiated Service Code Point (DSCP) field **[RFC 2474]** in the IP packet header is used to identify each aggregated service class. The DSCP field is the same as Type of Service (ToS) byte field in IP version 4 (IPv4) header and Traffic Class byte field in IP version 6 (IPv6) header. The value of DSCP field in a packet will determine the Per-Hop Behaviour (PHB) treatment its class should receive in the network. DiffServ can provide a wide range of services through a combination of:

- Setting bits in the DSCP field at network edges and administrative boundaries.
- Using those bits to determine how routers inside the network treat packets.
- Conditioning and marking of packets at network boundaries in accordance with the requirements of each service.

In the DiffServ specification, traffic conditioning and marking could turn out to be a complex process at network edges—a research problem. Also in view of DiffServ traffic aggregation or bundling for network resource allocation, some of the fine details of some application traffic QoS need may be omitted or compelled to hibernate **[Gio et al.03]** **[Mol et al.05]** **[ChrLie03]** **[Bia et al.02]** **[Wel et al.03]**—another research problem.

1.7.1.3 IEEE 802.1D and 802.1p Standards

IEEE 802.1D is the standard for Media Access Control (MAC) Bridges which has been developed to transparently interconnect IEEE 802-style of LANs. IEEE 802.1D was extended with IEEE 802.1p to support the notion of expedited traffic transmission capabilities through the use of user's priority traffic tagging. Through this support, limited differentiated service could be achieved. Each bridge port has a User Priority Regeneration Table (UPRT). The user's priority of a received frame is regenerated using priority information contained in the frame and UPRT. The Bridge forwarding process will use the user's priority information encoded into the frame in deciding the type of QoS the frame should get while forwarding it. The IEEE 802.1D & p standards in 1993 **[802.1D&p Yr.93]** specified 2 Priority Classes for bridged LANs to support multiservice QoS, while the 1998 standard **[802.1D&p Yr.98-05]** specified 8 Priority Classes. Their decision was based on sound heuristic consideration. A number of publications have suggested three or four classes. What then is the optimum number of priority classes for integrated service Internet? A good research question.

1.7.2 Objective of the Research Work

This thesis endeavours to proffer answers to the research questions posted in Section 1.7.1.3 as first part of the research work carried out. In line with this objective, the thesis provides a detailed account of the actions taken, results obtained and analysis of the results on empirical investigations into the research question, *—what is the optimum number of priority queuing classes that would best meet the need of integrated services in the Internet?* The experiment generated very useful results, and this thesis will endeavour to present the technical details in a form that will be beneficial to the Internet research community. The second part of the research work concerns finding solutions to the complexity and scalability problems of the IntServ-RSVP per-flow paradigm and DiffServ complexity as highlighted in Section 1.7.1.1 and Section 1.7.1.2 respectively above. The approach to design a solution specifically takes into consideration the simplicity of Internet core networks that accounts for its phenomenal success in tackling the research problem. This approach resulted into the design of an experimental simple, scalable, elegant and robust IP QoS architecture called *Pre-deterministic Distributed Event Response Resource Management* (PDERRM). This thesis will present the design, experimental test, and experimental result of PDERRM. The performance of PDERRM has been investigated through simulation and the results of the simulation show that it is a very elegant and robust architecture. The technical details as presented in this thesis are engineered towards formalisation of the experimental work carried out as mentioned above. The formality process includes presentation of the procedure adopted, the results we obtained, comments on the results and our conclusions. All the details will be found in Chapters 8 and 9 of this thesis.

1.8 Organisation of Remaining Part of the Thesis

Chapter 2: Overview of QoS technologies developed to extend the service model of the Internet to become a robust global multiservice network will be presented in this chapter. Standard QoS architecture and protocols such as IntServ, RSVP and DiffServ are discussed and analysed with intent to grasp or highlight the salient points of their mechanisms. Other QoS related architectures and protocols are also examined. Included in the group are; Subnet Bandwidth Management (SBM) which is used in shared medium LAN segments for resource allocation, and Multi-Protocol Label Switch (MPLS) which could be employed for QoS routing.

Chapter 3: In Chapter 3 of this thesis, the focus is on generic components of QoS architectures. The various component parts of a typical QoS architecture are presented with emphasis on queuing and scheduling disciplines, which are the bedrock of QoS delivery.

Chapter 4: The introductory work on simulation experiments to determine the *Optimum Number of Traffic Queuing Classes* (ONTQC) that will best meet the needs of integrated services in the Internet is presented in this chapter. The presentation and discussion therein includes motivation for the work, the objectives of the work, the choice of queuing discipline for the experiment and the choice of simulation tools. Also simulation parameters and the number of traffic classes used in the experiment are discussed.

Chapter 5: Simulation methodology and procedures on ONTQC are the subjects of this chapter. Grouping of simulation scenarios into *Simulation Actions Domains* (SADs), and the procedural steps involved in each simulation group are presented. The chain-like pattern of the series of SADs is described with Markovian chain. Algorithm for queue decomposition and novel meta-heuristic model developed to predict simulation outcomes are presented.

Chapter 6: The results of simulation experiments on ONTQC are presented in this chapter. Methods for processing results are outlined. End-to-end delay charts and throughput charts are presented and discussed. Numeric representations of the results in terms of matrices and row vectors are also presented.

Chapter 7: Analyses of results of simulation experiments carried out on ONTQC to support Integrated Services are presented. The presentation includes the use of matrices and row vectors to analyse the results of ONTQS experiments on the bases of *non-work conserving* operation of multiple queues and the consequent loss in their efficiency as the number of multiple queues pass certain threshold.

Chapter 8: The discussion in this chapter concerns the design of a new simple, elegant and robust QoS architecture for IP Networks. The new experimental QoS architecture called PDERRM is described in line with its principle of operation and building blocks. The framework and the main features of the architecture are presented.

Chapter 9: Presents the experimental procedure and processes for PDERRM performance evaluation. The simulation experiments, experimental results and comments are presented.

Chapter 10: Conclusions and future work are the subjects of this chapter.

Summary

This introductory chapter has been a brief all-embracing discussion that covers Quality of Service (QoS), its concept as related to the Internet, its desirability, the

Internet technological architecture, ATM technology, introduction to already developed QoS architectures for the Internet, identification of research problems and research objectives.

QoS was globally defined as the degree of satisfaction derived from a service. Internet architectural flexibility has encouraged the use of the network for new breeds of applications such as real-time voice over IP and Internet video conferencing. The emergent real-time applications could not thrive well in the Internet because the service model of the Internet was not originally designed to support real-time applications. The Internet best-effort service model needs to be extended. QoS was defined as performance seen by the recipient end-user of an application placed across the network. The main advantage of QoS is seen to be the notion of empowering a single network to provide multiple services.

Internet technology is built on the TCP/IP protocol suite, which consists of four layers: *application layer, transport layer, network layer and link layer.* The Internet communication mechanism was described as a modulated U-shaped relay system in which application traffic moves from the top application layer to the bottom link layer and then travels from one hop to another until it gets to its destination. At the destination it climbs the protocol ladder from the bottom link layer to the top application layer where it becomes useful data.

ATM technology is provisioned with capability to support multiservice QoS. The ATM AAL services can provide services for both real time and non-real time applications.

Prominent QoS architectures that have been developed for resource allocation in the Internet are IntServ and DiffServ. MPLS is traffic engineering architecture developed to support QoS routing. SBM handles RSVP based resource allocation in shared medium LAN segments.

The research work is in two parts: (1) An experimental investigation to find the optimum number of service classes that would meet the needs of integrated services in the Internet. (2) The design of a simple scalable and elegant QoS architecture for IP Networks.

CHAPTER 2

IP QoS Architectures Beyond Best-Effort Service

We have seen in Chapter one that real-time application performance in the Internet is below standard. In order to overcome this problem, the Internet Engineering Task Force (IETF) has developed new technologies and standards to provide resource guarantees and service differentiation in the Internet under the broad heading of IP Quality of Service (QoS). In this chapter we will examine the IETF's three main technologies that have emerged as core standards for supporting QoS in the Internet. Integrated Services (IntServ) architecture and Differentiated Services (DiffServ) architecture are two technologies that address the issue of resource reservation and allocation to various types of applications in the Internet. Multiprotocol Label Switching (MPLS) has interesting applications for QoS routing and Traffic Engineering (TE). The Subnetwork Bandwidth Manager (SBM) enables an IntServ-RSVP service model within shared LAN segments.

We will also briefly examine the effort of the Institute of Electrical and Electronic Engineers (IEEE) on the add-on QoS mechanism built into IEEE 802—style LANs. IEEE MAC Bridges standards were designed with capabilities for expedited transmission of time-critical applications.

2.1 Integrated Services Model in the Internet

The concept of Integrated Services (IntServ) originated from the notion that lots of benefits and efficiency could be derived from having one global network infrastructure that would support various types of telecommunication services. A global network that would support a variety of data transmission services, carrying real-time voice, video, and interactive data along with bulk data transfer. The flexibility of the Internet technology coupled with its phenomenal success has encouraged its nomination as a candidate for the convergence of all telecommunication services. This culminated into the Internet Engineering Task Force (IETF) creating the "Integrated Services Working Group" **[IntServ94]**, to develop a unified model for Integrated Services in the Internet. In order to turn the Internet into a robust integrated service communication infrastructure, its service model must be extended beyond the best-effort paradigm. This extension is necessary to meet the growing need for real-time services for a variety of new applications. The result of the multicast backbone (MBONE) experiment that was setup in 1993 for testing real-time applications motivated researchers in the IETF Working Group (WG) in part to work on the requirements and mechanisms for integrated service model for the Internet. The outcome of the experiment as documented revealed that, it would be possible to run multimedia applications over the Internet, but their performances with the best-effort service model were found to be below standard **[RFC1633]**. The result of the MBONE experiment geared IETF into action, and they have since engineered and developed QoS architectures and protocols that would transition the dream of Integrated Services into practice in the Internet environment.

Real-time applications QoS are not the only issue for next generation of traffic management in the Internet **[FloJac95]**. Network operators require the ability to control the sharing of bandwidth on a particular link among different traffic classes. They want to be able to divide traffic into a few administrative classes and assign to each a minimum percentage of the link bandwidth under conditions of overload, while allowing "unused" bandwidth to be available at other times. These classes may represent different user groups or different protocol families. Such management facilities are commonly called *controlled link sharing*. The term Integrated Services (IS) is used for an Internet service model that includes *best-effort services, real-time services, and controlled link sharing.* Thus the Integrated Services Model essentially consists of:

- An extended service model for the Internet, which is called the IS Model.
- A reference implementation framework, which gives a set of semantics and a generic program of actions to realise the IS Model.

2.1.1 Categories of Applications

In designing an enduring service model, the IntServ working group (WG) was concerned with defining classes of applications. The categorisation of various applications into generic groups was informed by taking analysis of the mechanics of their flows in an integrated services environment. An application traffic will generally fall into a category depending on how tolerant or intolerant the application is to inconsistency in network behaviour. In broad terms, applications can be categorised as being either *elastic* or *inelastic*. Elastic applications (non real-time applications), are those that can adapt to network inconsistency while inelastic application (real-time applications) are those that cannot adjust to network behavioural inconsistency and consequently degrade in performance. The IntServ service model divides applications into three main categories: real-time intolerant applications, real-time tolerant applications, and non real-time applications.

2.1.1.1 Real-time Intolerant Applications

They placed high demand on network traffic control in order for the network to meet their QoS requirements. If their traffic is not transmitted consistently and precisely at all times, the application will suffer degradation and will consequently be unacceptable. Examples of such applications are, Voice over IP (VoIP), interactive video and control systems. Two-way conversation cannot tolerate excessive delay or jitter.

2.1.1.2 Real-time Tolerant Applications

These are less sensitive forms of real-time applications. They do expect their data to arrive in a timely fashion but occasional missed or delayed packets will not adversely affect their performance in the network. They tolerate some inconsistency in network services as long as it does not typically exceed a certain well-defined threshold.

Applications can be real-time and still be tolerant of network irregularities through additions to their design **[DurYav99]**. A video application for example may buffer a few frames ahead of time to playback. If timing of frames out of the network is not perfectly consistent, the buffer will hide the resulting distractions from the user by always showing the buffered frame in consistent intervals. Video streaming and Internet games are examples of real-time tolerant applications.

2.1.1.3 Non Real-time (Elastic) Applications

These are the most tolerant to network variation conditions. Applications in this group do not particularly suffer if their traffic is subjected to inconsistency in network behaviour. The group includes traditional best-effort traffic, such as file transfer and e-mail.

2.1.2 Integrated Services Architecture

Integrated Services (IntServ) architecture is the proposed standard for per-flow resource allocation in the Internet. In the IntServ architecture, new service models were developed to meet the requirements of real-time applications. The IntServ architecture consists of a set of mechanisms and protocols used for making explicit resource reservation and resource allocation in the Internet. The basic approach is per-flow resource reservation and allocation in the Internet **[Sol et al.04]** **[RFC1633]**. The architecture assumes that the main QoS about which the network makes commitments is *per packet delay*. Integrated Services will include additional flow-states in routers and an explicit set-up mechanism that will provide the different services required by the various applications.

The main components of the architecture are:

- Specification and definition of general QoS parameters
- Applications reference service model
- Resource reSerVation Protocol (RSVP) **[RFC 2210]**, and
- Reference implementation framework

We will briefly examine each of these components.

2.1.2.1 General QoS Parameters Specification

General QoS parameters are sets of characterisation parameters that describe traffic flow profiles, their QoS needs, and network element ability to support application QoS requirements. These standardised parameters have a common and consistent interpretation and understanding that allows unified specification and practice of QoS end-to-end over the heterogeneous network. Characterisation parameter specifications and definitions provide the semantics which an application can employ to inform the network of its QoS requirements and derive information about available resources along its path of flow in the network.

A two level namespace is designed for each parameter to reflect the type of service with which the parameter is associated— <service_number> <parameter_number>. As indicated, it is designed such that, each parameter ID is composed of two numerical fields. One identifying the service associated with the parameter (<service_number>) and the other identifying the parameter value (<parameter_number>). Each of the numerical fields ranges in numerical value between 1 and 254. The definition of each parameter used to characterise a path through the network describes two types of values, local and composed. A local value gives information about a single network element. Composed values reflect the running composition of local values, specified by some composition rule. Each parameter definition specifies the composition rule for that parameter **[RFC 2215]**. Since characterisation parameters are used to compute the properties of a specific path through the inter-network, all characterisation parameters are conceptually 'per-next-hop' as opposed to 'per interface' or 'per network element'.

The general parameters include:

- NON_IS_HOP —IntServ unaware network elements (NEs).
- NUMBER_OF_IS_HOPS—IntServ aware NEs.
- AVAILABLE_PATH_BANDWIDTH—Bandwidth available along the path.
- MINIMUM_PATH_LATENCY—Cumulative minimum latency incurred by all NEs along path of flow.
- PATH_MTU—Maximum transmission unit (MTU) for packets following the path.
- TOKEN_BUCKET_TSPEC—very important, used to describe traffic characteristics and QoS requirements.

Information on detailed specification and definition of IntServ general parameters could be found in RFC 2215 **[RFC 2215]**.

2.1.2.2 Reference Service Model

In the IntServ architecture, two service models were proposed as standards for integrated services (IS) Internet and these are: *Control Load Service* and *Guarantee Service*. Although three generic classes of applications have been specified or categorised, it has to be noted that, real time tolerant applications and elastic applications share similar characteristics in term of jitter tolerance, thus they both could make use of control load service, while the real time intolerant application could make use of guaranteed service. The default service model is best-effort.

2.1.2.2.1 Control Load Service

The control load service is directed towards the need of real-time tolerant applications and elastic applications **[RFC 2211].** Bandwidth is a basic resource and a fundamental quantity for a QoS control mechanism. Applications that use control load service are designed to operate over a best-effort network with sufficient available bandwidth. In order for controlled load service to support real-time tolerant application, it simulates best-effort quality of service not affected by excessive load. To achieve this, the controlled load service will set aside a specific amount of bandwidth for applications that require the service. This subset of the network device's link capacity will always be available to requesting applications irrespective of other traffic. The end-to-end behaviour provided to an application by a concatenation of network elements providing controlled-load service closely approximates the behaviour visible to applications receiving best-effort service "under unloaded conditions" in its path of flow end-to-end.

Assuming the network is functioning correctly, these applications may assume that:

- Very high percentage of its transmitted packets will successfully be delivered by the network to the receiving end-node.
- The transit delay experienced by a very high percentage of the delivered packets will not greatly exceed the minimum transmit delay experienced by any successfully delivered packet.

To ensure that the controlled load conditions are met, clients requesting the service will provide the intermediate network elements with an estimation of the data traffic they will generate i.e. the TSpec. In return, the service ensures that adequate network element resources are provided to process traffic falling within the descriptive envelope that is provided by the client.

2.1.2.2.2 Guaranteed service

This is suitable for real-time intolerant applications. Guaranteed Service (GS) provides firm (mathematically provable) bounds on end-to-end packet queuing delays, **[RFC 2212]**. This service makes it possible to provide a service that guarantees both delay and bandwidth. This means packets in a flow that is transmitted with a guaranteed QoS specification must arrive at its destination within a defined delay bound. GS provides assurance that, packets will arrive within the guaranteed delivery time and will not be discarded due to queue overflows, provided the flow's traffic stays within its specified traffic characteristics. Applications with stringent time of delivery requirements that make use of "play-back" schemes as a means of mitigating against network vagaries, are intolerant of any packets arriving after their "play-back point". Such applications with hard real-time requirements should take advantage of delay bound assurance inherent in GS. The GS does not attempt to minimise the jitter, it merely controls the maximal queuing delay.

Applications requiring GS must describe their traffic characteristics in the form of *Sender TSpec* (traffic specification) to the network, and also make resource reservation request by specifying the *Receiver RSpect* (reservation specification) to the network. The reservation request must be accepted by concatenation of network element in the path of the application traffic flow end-to-end before traffic can start to flow. The Guaranteed QoS Synthesis (GQS) technique proposed by Hovell et al. provide a method for providing QoS guarantee **[Hov et al.05]**.

2.1.2.3 Resource reSerVation Protocol (RSVP)

Resource reSerVation Protocol (RSVP) is the signalling protocol that provides reservation set-up and control to enable the *Integrated Services* (IS) in the Internet **[RFC 2205]**. It is seen as part of the implementation framework for IntServ architecture. As a result of its complex signalling, it is said to represent the biggest departure from standard "best-effort" IP services and provides the highest level of QoS in terms of service guarantees, granularity of resource allocation and details of feedback to QoS-enabled applications and users.

RSVP signals per-flow requirements of data traffic to IS capable data-forwarding network elements. It is a way to communicate the various applications divergent requirements to network elements along the path of flow and to convey QoS management information. It helps in creating and maintaining *flow specific state* in the endpoint hosts and routers in the path of flow. In order to state its resource requirements, an application must specify the desired QoS, using a list of parameters that is called a "*flowspec*" (flow specification). The flowspec is carried by RSVP messages, passed to admission control to test for acceptability, and ultimately used to parameterise the packet scheduling mechanism.

2.1.2.3.1 Basic Features

Hop-by-hop message sequence: RSVP serves as a hop-by-hop signalling protocol such that network elements along the path of a data flow be aware of a flow's QoS requirements, in order to provide any special treatment for the data flow. A path is effectively pinned down such that each hop knows its neighbouring adjacent RSVP hops for a particular flow end-to-end. RSVP other basic features include:

- **Simplex reservation:** RSVP makes a reservation in only one direction (simplex flow).
- **Receiver oriented reservation:** Receiver is responsible for making resource reservations.
- **Soft-state design:** Reservation state is temporary. Reservation states are refreshed at regular time interval during its lifetime.
- **Support multicast communication:** RSVP is designed to support both unicast and multicast communication.
- **Routing Independent:** It is not a routing protocol nor it's part of the routing architecture of a network device.
- **Act like Internet Control Protocol:** RSVP does not transport application data but is rather like an Internet control protocol such as ICMP or IGMP.
- **Doesn't need transport protocol:** RSVP messages are transmitted directly over the IP protocol as opposed to being transmitted over TCP or UDP. RSVP protocol ID is specified in IP header (RSVP = 46).

2.1.2.3.2 RSVP Messages

RSVP messages consist of seven types: the PATH and RESV Messages, the PATH Error and RESV Error messages, the PATH Tear and RESV Tear messages, and the Confirmation messages. PATH and RESV messages are compulsory while the remaining five are optional in signalling for resource reservation **[RFC 2209]**. Each of the messages is made up of an RSVP header followed by a set of message objects. The objects contain the information necessary for describing resource reservation requests. We will briefly examine the two non-optional messages.

A **PATH Message** originates from the *Sender* and is sent to the *Receiver*. It is used to discover and pin down the path of a data flow, identifies the data flow and its source, describes its traffic characteristics, advertises resource capabilities in path of flow, and installs states relevant to the data flow in network elements along the data path. PATH message objects are; the *Session Class object*, *RSVP Hop Class object*, *Time Values Class object*, *Policy Data Class object*, *Sender Template Class object*, Sender *TSpec Class object* and *AdSpec Class* object.

A **RESV Message** is generated by the *Receiver* and sent to the *Sender*. It retraces the reverse direction of the path created by the path message hop-by-hop upstream from the destination back to the source. The RESV Messages are addressed from the sending downstream hop to the previous upstream hop (PHOP). The PHOP information is obtained from the corresponding path-state installed on each network element in the path setup by the PATH message. When PHOP receives the RESV Message, it will be interpreted by the device's RSVP process. This process will check to see if at least one valid path-state has been installed for the session. If valid path-state exists, the reservation will proceed, and if admission control succeeds, a reservation state will be installed, in other words, resources will be applied to service the corresponding flow. RESV Message objects are similar to PATH Message objects with the exception of; *RESV Confirmation object*, *Style object*, *Flow Spec object* and *Filter Spec object*.

2.1.2.3.3 RSVP Basic Operation

An RSVP session consists of a simplex sender's PATH Message sent *downstream* through a concatenation of forwarding devices to the receiver. A corresponding simplex receiver's RESV (Reservation) Message will be sent *upstream* tracing the path created by the PATH Message to the sender. A unidirectional flow from a sender through series of RSVP-aware devices to a receiver is regarded as *downstream* flow. On the other hand, a flow in the reverse direction from receiver to the sender is referred to as *upstream* flow.

Since the source typically knows the characteristics of the traffic it's capable of sending, RSVP allows the data source to describe the characteristics of the traffic it intends to generate. This information is encoded in the parameter within the sender's PATH Message and transmitted from the source to the destination through the data path. Thus, the destinations as well as every RSVP-aware device along the data path are aware of the sender's traffic generating capabilities. The PATH Message from the sender effectively installs a path state on all RSVP-aware devices along the path of data flow.

Once the path-state has been established on all RSVP-aware devices from source to destination, the RSVP paradigm requires the receiver to issue a reservation request for the particular data flow. The reservation request must then follow hop-by-hop, the path laid down by the corresponding PATH Message. No network element will commit it's resources to service a data flow until a reservation request has been issued for the flow and accepted by the devices along the data path. The reservation request is carried by the receiver's RESV Message, and contains the receiver's QoS specification for the data flow. That is the destination actually determines what QoS the flow will actually receive. Through the hop-by-hop reservation set-up mechanism, all RSVP devices along the original path will become aware of the QoS reservation request made by the receiver. Each network element along the path may then individually decide to accept the reservation or modify appropriate parameter in the RESV Message before sending it upstream or refuse the request altogether due to capacity or administrative constraints.

The sender's Traffic Specification (TSpec) object in the path message describes the characteristics of the traffic the source is able to generate, while the Advertisement Specification (AdSpec) object also in the path message describes the kind of services all devices along the data path can offer. Each forwarding device on the data path updates the AdSpec object. This procedure provides enough information to the receiver to make a choice of its QoS reservation. On the other hand the receiver's Reservation Specification (RSpec) in the RESV message describes the type of QoS the receiver desires to receive. The receiver also uses the Filter Specification (Filter Spec) object to identify the source of the data flow.

An RSVP session is illustrated in Figure 2.1 below. PATH Messages are sent downstream from *Sender* to *Receiver*. The corresponding RESV Message sent upstream from receiver to sender.

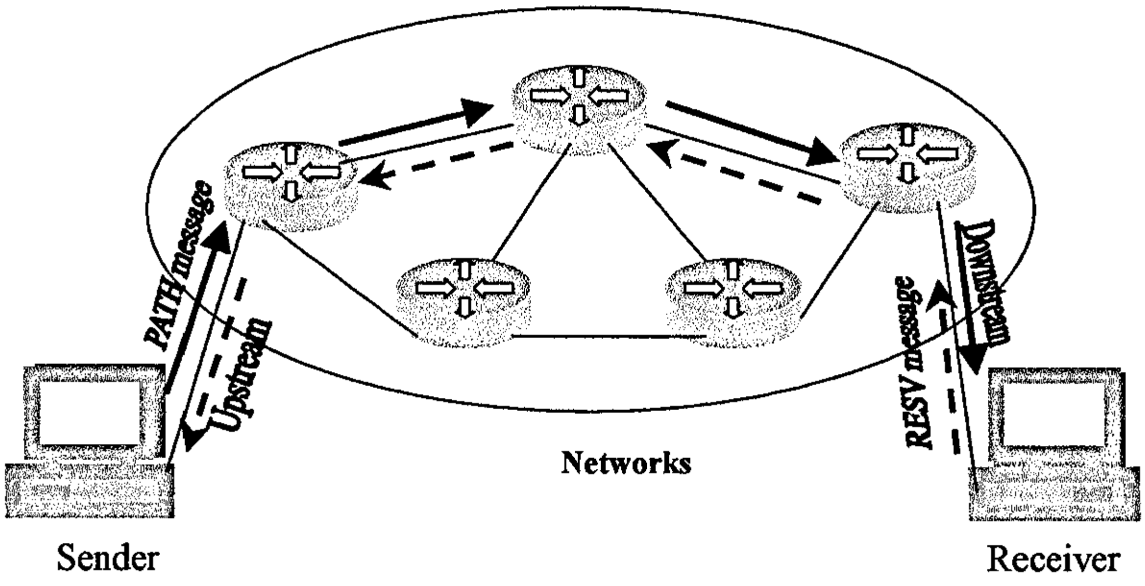


Figure 2.1 RSVP PATH and RESV messages flow

A Reservation must pass both "Admission Control" and "Policy Control" before data can flow. Admission Control determines whether the node has sufficient available resources to supply the requested QoS. Policy Control determines whether the user has administrative permission to make the reservation.

2.1.2.3.4 Reservation Styles

Basically RSVP supports two resource reservation styles—distinct and shared, to cater for both unicast and multicast flows. The shared reservation style is also further categorised into two, consequently three reservation styles are currently defined. The three styles of reservation are briefly explained below and illustrated in Table 2.1.

- Fixed filter (FF) style. The FF style is used when distinct reservation is required for each sender in an explicit sender selection.
- Shared explicit (SE) style. In SE style, reservations are shared for explicit selection of sender list. SE style ensures that a single reservation is created for each group to share.
- Wildcard filter (WF) style. With WF style reservation, all senders share a single reservation. It implies *shared* reservation and wildcard sender selection. The simple algorithm is that, all senders share the largest of the resource request from receivers.

Table 2.1 Reservation Styles

Sender Selection Scope	**Reservation Styles**	
	Distinct reservation	**Shared reservation**
Explicit sender selection	Fixed filter (FF)	Shared explicit (SE)
Wildcard sender selection	(None defined)	Wildcard filter (WF)

2.1.2.3.5 RSVP Scalability Problem

RSVP per-flow packet processing can lead to scalability problems in backbone routers where hundreds of thousands of flows are processed within a short interval of time **[Gio et al.03] [Mol et al.05] [Sol et al.04]**. Per-flow packet classifications, admission controls, policing, shaping and micro-differentiation in scheduling will create serious limitations on scalability in multi-gigabit link environment. Additionally, state information on per-flow reservations needs to be maintained by the routers in order to satisfy the need of each flow. With very large number of flows, routers will find it difficult to maintain such per-flow state information. In view of the above RSVP is considered a nonscalable solution for Internet backbone.

2.1.2.4 IntServ Reference Implementation Framework

The IntServ architecture reference implementation framework includes generic QoS components such as the admission control, the classifier, the packet scheduler, and the reservation set-up protocol. The component implementation details are not mandated to be uniform but the perceived outward behaviour must be uniform for all implementations. We have already discussed the reservation setup protocol—RSVP, the other components of the implementation framework share similar functions as discussed in Chapter 3 under *generic components of QoS architectures* (Section 3.1 to Section 3.5).

Admission Control: This implements the decision algorithm that a router or host uses to determine whether a new flow can be granted the requested QoS without impacting against earlier guarantees. Admission control is invoked at each node to make a local accept/reject decision, at the time a host requests a real-time service along some path through the Internet.

Classifier: In the Integrated Services (IS) model, each in coming packet must be identified in order to be able to map it to its reservation state information. The state information is used to parameterise the treatment the packet will receive from the scheduler.

Packet Scheduler: For more detailed information on packet scheduler, see Section 3.7. Packet scheduler manages the forwarding of different packet streams using a set of queues and perhaps other mechanisms like timers.

2.2 Integrated Services over Specific Link Layers

The Integrated Services (IntServ) standard is essentially a layer 3 QoS architecture. Its signalling protocol—RSVP is designed on the basis of point-to-point link technology. Since IP runs over different link layer technologies—Ethernet, ATM, etc, there is the need to support the IntServ-RSVP resource allocation paradigm over these link layer technologies. This will ensure continuity in end-to-end IntServ QoS support across the Internet. The IETF realised this and created a working group (WG) called Integrated Service over Specific Link Layers (ISSLL) **[ISSLL-802]** WG to address the issues.

To support RSVP resource allocation over shared LANs, the concept of the *Subnetwork Bandwidth Manager* (SBM) has been developed. The SBM Protocol **[RFC 2814]** is an extension to RSVP signalling to support RSVP-based admission control in IEEE 802.3 style of LAN, (Ethernet).

Many Internet backbones are now built on ATM networking technology, it is natural to make use of ATM built-in QoS capabilities by developing a standard way of mapping IntServ QoS support onto ATM QoS support. This involves developing a standard method of inter-operating the QoS functionality of the two technologies. We will briefly discuss the SBM protocol and RSVP over ATM in this section.

2.2.1 RSVP over ATM

ATM (Asynchronous Transfer Mode) has been established as a networking technology with a built-in mechanism for QoS support. It is therefore a natural process to define a standard mapping for translating IntServ-RSVP QoS service classes onto ATM QoS service classes **[And et al.00]**.

The proposed mapping for IntServ service classes and parameters onto ATM service categories and descriptors are shown diagrammatically in Figure 2.2.

RSVP Flow Specification	**ATM Service Specification**
Average Bit Rate (R)	Average Bit Rate (SCR)
Peak Rate (P)	Peak Rate (PCR)
Burst/Bucket Size (B)	Emission Burst (MBS)

IntServ Service Classes	ATM Service Classes
Best-Effort	UBR/ABR
Control Load	nrtVBR or ABR
Guaranteed	CBR or rtVBR

SCR: Sustainable Cell Rate **PCR:** Peak Cell Rate **MBS:** Maximum Burst Size
UBR: Unspecified Bit Rate **ABR:** Available Bit Rate **VBR:** Variable Bit Rate
nrtVBR: non-real-time VBR **CBR:** Constant Bit Rate **rtVBR:** real-time VBR

Figure 2. 2: Mapping of IntServ Services onto ATM Services

2.2.2 Subnetwork Bandwidth Manager (SBM)

Subnetwork Bandwidth Manager operates as a centralised resource broker in a shared LAN environment. The Subnetwork Bandwidth Management protocol allows each RSVP signalling session to transverse the shared link or networks without overcommitting the shared link resources to service a particular host's flows to the detriment of other flows in other hosts in the network. It provides a method of managing and mapping IntServ-RSVP QoS services in the shared resources of a shared LAN segment. For each LAN a single SBM becomes the Designated SBM (DSBM) either through election or static configuration **[Fin02]**. When a new host first becomes active on the LAN, it attempts to discover whether a DSBM exists through a fault-tolerant DSBM discovery-and-election protocol. Once a host finds the DSBM, it becomes the DSBM client to the DSBM. The DSBM client simply passes all its RSVP messages through the DSBM. The DSBM is responsible for RSVP reservation and admission control for all its DSBM clients. It co-ordinate management of resources in the LAN segment.

Managed Segment RSVP Message Processing:

A DSBM client sends its RSVP messages through the DSBM. All incoming and outgoing RSVP messages compulsorily must pass through the DSBM for control. Thus DSBM is inserted necessarily as an extra hop to the normal RSVP operations. As part of processing, the DSBM build-up Path State Blocks (PSB) for each session and update the RSVP_HOP object (PHOP) to include its own DSBM interface address. SBM protocol introduces new RSVP objects called LAN_NHOP object and RSV_HOP_L2 object. The RSV_HOP_L2 object is similar in function to RSVP_HOP object in that it identifies the MAC address of the next hop or previous hop nodes participating in the SBM signalling processes. This helps to simplify processing of RSVP messages by layer 2 devices that operate strictly with MAC addresses. For example when a layer 3 device forwards the PATH message over a shared segment, it includes its IP address in the RSVP_HOP object and the corresponding MAC address in the RSVP_HOP_L2 object, consequently it simplifies processing for layer 2 device that cannot interpret the RSVP_HOP object. The LAN_NHOP object has similar application for RSVP messages that are transmitted from one layer 3 device to another, but must transverse a purely layer 2 network. Another new RSVP object that is introduced is the TCLASS object. Downstream DSBMs will insert the new traffic object (TCLASS object) in the PATH message that specifies the appropriate service class for the flow according to the service mapping determined at the intervening switches. The TCLASS object is treated like the ADSPEC object in normal RSVP PATH messages. All incoming RSVP PATH messages pass through the DSBM. It would parse both the normal and additional RSVP objects in the message and setup PSB before sending the message to its destination (i.e. the DSBM client).

The DSBM makes decisions on both incoming and out going RESV messages based on the resources available in the managed segment. If a request cannot be granted, RESVerror is sent to the requester. On the other hand if the reservation request can be granted, the RESV massage is forwarded to the previous hop address of the incoming PATH message based on the session's PSB.

2.3 Differentiated Services Architecture

The development of the Differentiated Services (DiffServ) architecture was the response of IETF to the need for a relatively simple and coarse resource allocation architecture compared with the complexity of per-flow end-to-end IntServ architecture **[SikTei]** **[Aky et al.03]**. It is a well-known fact that the per-flow processing, involving admission control and flow state maintenance of IntServ-RSVP paradigm could create scalability problems in backbone networks with multi-gigabit links. The limitations of IntServ-RSVP model engineered the revitalisation of the old mechanism of providing simple priority-based QoS in the Internet **[Bia99]**. The IETF DiffServ Working Group (WG) **[RFC 2475]** was charged with the responsibility of redesigning the priority-based Type of Service (ToS) Field in IPv4 header to support broad based resource allocation. The ToS Field was renamed as Differentiated Services (DS) Field **[Vei00]** **[Ste98]**. The definition of services encoded in the DS Field along with the specification of the associated resource allocation policies in the networks is referred to as the Differentiated Services (DiffServ) architecture **[DSFWK]**.

2.3.1 DiffServ Architecture Overview

The DiffServ architecture is based on a simple model where traffic entering a network is classified and possibly conditioned at the boundaries of the network, and assigned to Behaviour Aggregates (BA). A BA is a collection of packets that would receive similar forwarding treatment in the networks. Each BA is identified by a single value of DS Codepoint **[RFC 2474]**, (the 6-bit of ToS byte field of IPv4 header). Within the core of the network, packets are forwarded according to the Per-Hop Behaviour (PHB) associated with the DS Codepoint. The per-hop forwarding behaviour of packets within the interior of the network is guided by service provisional policy adopted by the network administration. Packet forwarding at the boundary of the network (both ingress and egress traffic) must comply with the peer Service Level Agreement (SLA). Functionally, ingress and egress traffic to the network must comply with the Traffic Conditioning Agreement (TCA) which is a sub-set of SLA.

2.3.2 DiffServ Architecture

DiffServ is an architecture for implementing scalable service differentiation in the Internet. The architecture achieves scalability by aggregating traffic into a small number of groups known as *forwarding classes* that is conveyed to the network by means of IP-layer packet marking using the DS Field **[RFC 2474]**. The DS Field is the same as the Type of Service (ToS) byte in the IPv4 Header. Each forwarding class, as encoded in the DS Field represents a predefined forwarding treatment in terms of resources allocation that the class would enjoy in the network. There is no need for resource allocation setup. Packets are classified and marked to receive a particular per-hop forwarding treatment on nodes along their path. Per-hop behaviours are defined to permit a reasonably granular means of allocating buffer and bandwidth resources at each node among competing traffic streams. Per-application flow or per-customer forwarding state need not be maintained within the core of the network. Sophisticated classification, marking, policing, and shaping operations are restricted to network edge nodes. Network resources are allocated to traffic streams by service provisioning policies which govern how traffic are marked and conditioned upon entry to a differentiated services-capable network, and how that traffic is forwarded within that network.

The architecture only provides service differentiation in one direction of traffic flow and is therefore asymmetric. Conceptually, the architecture functional blocks include; *Traffic Classification block, Per-Hop Behaviour* (PHB) *block and Traffic Conditioner block.*

2.3.2.1 Traffic Classification

Traffic classification functions concern criteria employed for classifying traffic into groups of forwarding classes to receive specific treatment in the

Differentiated Services (DS) networks. Various criteria could be employed for allocating traffic into groups and for identifying the groups. These could be based on categories of applications (whether elastic or non-elastic), on customer needs, on protocol classes, on organisational requirements etc. Two types of classifiers are defined. One classification is based on the use of *Differentiated Services Codepoint* (DSCP), (6-bit of ToS byte field of IPv4 header) to mark packets as belonging to a forwarding class, and is referred to as BA Classifier. The other classification is based on the value of a combination of one or more packet header fields, and referred to as MF (Multi-Field) Classifier. After traffic groups have been defined and are allocated to forwarding classes, the packets belonging to a forwarding class are marked for identification purposes using a classifier, usually the BA Classifier. A service will be defined to associate with the forwarding class.

The classification function involves two phases of operation: *pre-flow classification* and *in-flow classification.* The pre-flow classification is done at the end nodes and its criterion is dictated by customer-service-provider agreement. Actually the in-flow classification is under Traffic Conditioning block. It is done at the edge nodes and controlled by TCA

2.3.2.2 Per-Hop Behaviour (PHB)

Per-Hop Behaviour (PHB) is a description of the externally observable forwarding treatment a DS node applies to a particular DS BA. All packets with the same codepoint are referred to as *Behaviour Aggregate* (BA), they receive the same forwarding treatment in the network. The term "externally observable forwarding treatment" relates to queuing, queue management and scheduling characteristics a BA receives from DS-node in the network. The DSCP value on a packet specifies the PHB given to the packet within the DS network. The PHB represents the building block from which a variety of services could be built on. After a service is defined, a PHB is specified on all nodes in the network offering the service, and a DSCP is assigned to the PHB.

The DiffServ architecture is illustrated in Figure 2.3 and it captures the main component functions of the architecture.

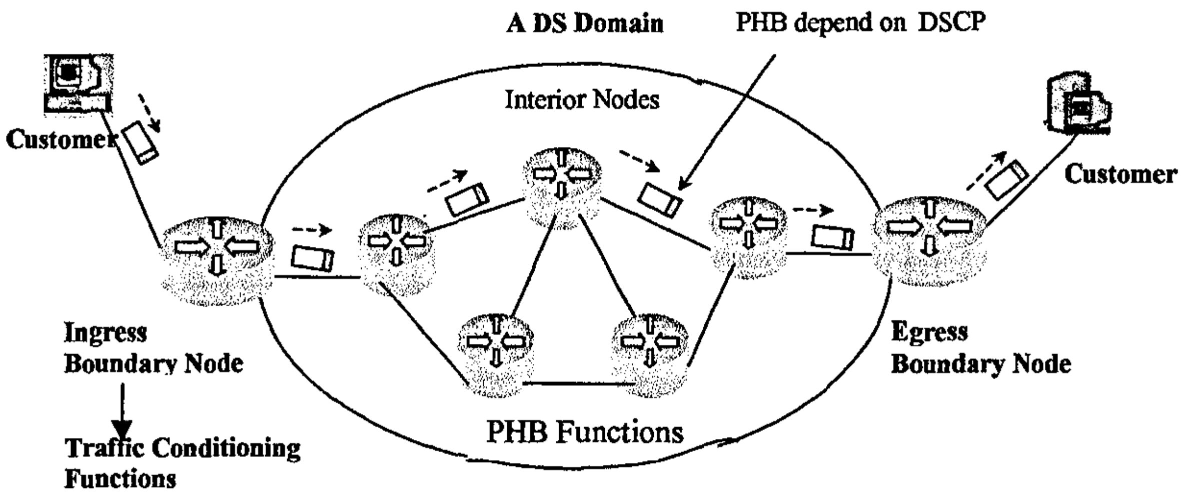


Figure 2.3: Illustration of Main Functions in DiffServ Architecture

2.3.2.3 Traffic Conditioner

Traffic conditioning operation ensures that a traffic flow's profile characteristics stay within its contracted agreement such that excessive traffic is prevented which could offset provisioning-utilisation balance. Traffic flow that stays within its agreed characteristic flow-profile are said to be *in-profile* and traffic flow that exceeds its agreed characteristic flow-profile are said to be *out-of-profile.*

Differentiated Services (DS) may span several DS domain boundaries. To ensure smooth operations, Service Level Agreements (SLA) are established between an upstream DS domain network and a downstream DS domain network. The SLA will specify traffic conditioning rules that may be applied to traffic streams, which are in-profile or out-of-profile. Traffic Conditioning Agreements (TCA) are specified between the DS domains and are derived (explicitly or implicitly) from the SLA.

Traffic conditioning operations include: *classifying, metering, shaping, policing and/or re- marking* to ensure that, traffic entering the DS domain conforms to the rules specified in the TCA, in accordance with the domain's service provisioning policy. The extent of traffic conditioning required is dependent on the specific of the service offering, and may range from simple codepoint re-marking to complex policing and shaping operations.

2.3.3 Terminology

DS Domain: a contiguous set of nodes or a single network capable of employing DiffServ mechanism for network resources allocation and services.

Service Level Agreement (SLA): a service contract between a customer and a service provider that specifies the forwarding service that customer traffic should receive. A customer may be a user organisation (source domain) or another DS domain (upstream domain). A SLA may include traffic conditioning rules, which constitute a TCA in whole or in part **[Cor et al.03]**.

Service Provisioning Policy: a policy which defines how traffic conditioners are configured on DS boundary nodes and how traffic streams are mapped to DS behaviour aggregates to achieve a range of services.

Traffic Conditioning Agreement (TCA): an agreement specifying classifier rules and any corresponding traffic metering, marking, discarding and/or shaping rules which are to apply to the traffic streams selected by the classifier **[Arm00]**.

2.3.4 DS Field

The current IP packet header (IPv4) includes an 8-bit field called *Type of Service* (ToS) *Field.* It consists of 3-bits precedence, 3-bit type of service (ToS) and 2-bits unused that must be set to zero **[RFC 1349] [GunRua99]**. The precedence bits represent the priorities of the traffic, while the ToS bits encode preference for loss, throughput, and delay. This is illustrated in Figure 2.4a.

The IETF DiffServ Working Group (WG) redefines the existing ToS Field and renames it *DS Field.* The existing 3-bits precedence and the 3-bits ToS are now combined to form the 6-bits *Differentiated Services Codepoint* (DSCP), which is also referred to as DS Codepoint as illustrated in Figure 2.4b. The remaining 2 bits are currently unused (CU). The DSCP should be treated as an index, and the mapping of DSCP to PHBs must be configurable, **[RFC 2474]**.

Class Selector Codepoint:

DiffServ WG allocated a set of Codepoint values known as *Class Selector Codepoints* to maintain backward compatibility with known current use of the IP precedence field. The codepoints in the range 000000 to 111000 are reserved as class selector codepoints and have direct relationship with precedence bits 000 to 111. The codepoint value 000000 has been set as default PHB DSCP and has a PHB forwarding behaviour equivalent to best-effort service. Other codepoints could be mapped to the class selector PHBs, but they must meet the class selector requirements. The requirements for class selector PHBs are detailed in RFC 2474 **[RFC 2474]**.

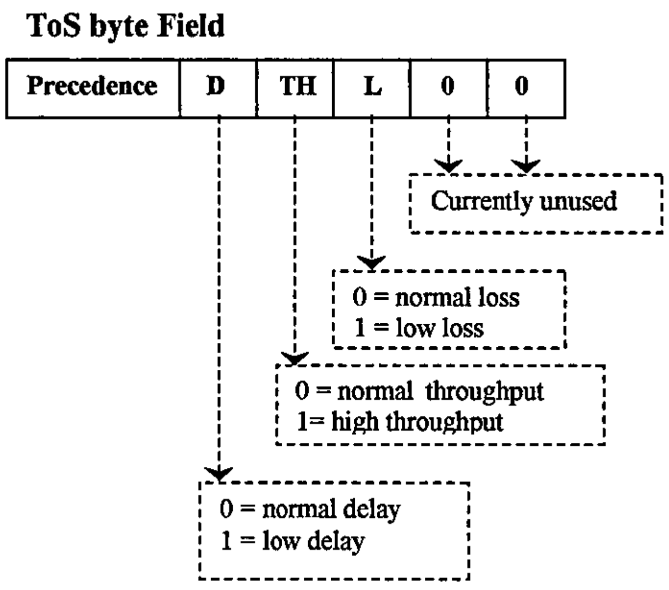


Figure 2.4a: Type of Service (ToS) Field in IPv4 Packet Header

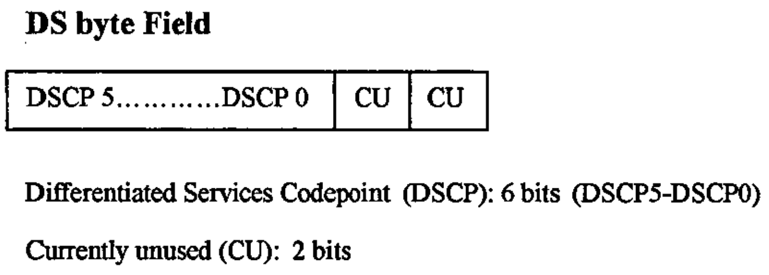

DS byte Field
DSCP 5............DSCP 0
CU
CU
Differentiated Services Codepoint (DSCP): 6 bits (DSCP5-DSCP0)
Currently unused (CU): 2 bits

Figure 2.4b: DS Field in DiffServ Architecture showing DSCP

2.3.5 DiffServ Service Models

Besides the class selector codepoints PHBs and the default PHB codepoint, IETF DiffServ WG has defined two PHBs as DiffServ standard service models. The two are *Expedited Forwarding* (EF) PHB and *Assured Forwarding* (AF) PHBs. The two European funded project, AQUILA and SEQUIN deal with DiffServ-based service models **[Bak et al.03] [Eng et al.03] [Bouras et al.03].**

2.3.5.1 Expedited Forwarding PHB

EF PHB was proposed as a low latency, low jitter and low loss PHB forwarding behaviour aggregate **[RFC 2598] [Arm00]**. EF PHB is defined as the forwarding treatment for a traffic aggregate, in which the departure rates of the packets in the aggregate from any DS node must exceed or equal the arrival rates. The EF traffic must receive this rate independent of the intensity of other traffic requesting transmission service from the node. The specification doesn't mandate a specific implementation approach, but from its low jitter requirement, priority based queuing and scheduling with appropriate control may be more suitable. The codepoint recommended for EF PHB is <101110>.

2.3.5.2 AF PHB Group

The AF standard is a PHB group that is structured into four forwarding classes and within each forwarding class, three drops precedence are defined **[RFC 2597]**. Each forwarding class is associated with a minimum amount of buffer and bandwidth allocation. Three drop priorities within each forwarding class are used to select which packets to drop during congestion. While EF supports services with hard bandwidth and jitter characteristics, the AF group allows more flexible and dynamic sharing of network resources—supporting the "soft" bandwidth and loss guarantees appropriate for bursty traffic.

Table 2.2 AF PHB Codepoints

	Class 1	**Class 2**	**Class 3**	**Class 4**
Low drop precedence	001010	010010	011010	100010
Medium drop precedence	001100	010100	011100	100100
High drop precedence	001110	010110	011110	100110

Two distinct classification contexts are encoded within the DSCP—a packet *service class* and its *drop precedence*. The first 3 bits of the DSCP encode the class and 2 bits encode the drop priorities and this is illustrated in Table 2.2.

2.4 IEEE 802.1D MAC Bridges Support for User's Priority

The IEEE 802.1D is the standard for Media Access Control (MAC) Bridges that specifies a state-of-the-art architecture that can be used to interconnect the various IEEE 802 style of LAN technologies. The IEEE 802 standards for LAN technologies include: IEEE 802.3 style– Ethernet, 802.4–Token Bus, 802.5–Token Ring, Fibre Distributed Data Interface (FDDI), and 802.12–Demand Priority. Some of these LAN technologies, such as FDDI, Token Ring, Token Bus and Demand Priority, support the mechanism of user priority as legacy built-in mechanism to provide minimal service differentiation in the networks.

IEEE 802.1p standard is a supplement to the IEEE 802.1D standard, which defines additional capabilities in Bridged LANs environment, aimed at:

1. The provision of expedited traffic capabilities, to support the transmission of time- critical information in a LAN environment.
2. The provision of filtering services that support the dynamic use of Group MAC addresses in LAN environment.

Since our concern here is on QoS, we will only examine the capabilities specified in 1 above. MAC Bridges are designed to support and maintain Quality of Service (QoS). It is expected that the QoS supported in the Bridged LAN environment should not be significantly inferior to that supported by individual LAN.

IEEE 802.1p specification on expedited traffic capability enables MAC Bridges to support priority based differentiated services in all the Bridged LAN **[Hal98]**. Thus MAC Bridges could signal user priority information over a LAN such as an IEEE-style Ethernet which has no inherent capability to signal such priority information. IEEE 802.1p standard covers the concept of Traffic Classes and their effect on the operation of Forwarding Process of MAC Bridges. MAC Bridges interconnect the separate LANs that comprise a Bridged LAN by relaying and filtering frames between the separate MACs of the Bridged LAN. QoS functions of MAC Bridges in relaying or transmitting frames between LANs include:

- Prevention of frame misordering for frames transmitted with the same user priority and the same combination of destination and source addresses.

- Regeneration of user priority based upon a combination of signalled information and configuration information held in the Bridge.
- Filtering services which increase the overall throughput of the network
- Minimise transit delay
- Use of priority for expedited transmission of time-critical traffic.
- Prevention of frame duplication
- Use of frame check sequence (FCS) to prevent undetected frame error rate.

2.4.1 Bridge Operation on the User's Priority

A Bridge classifies frames into traffic classes in order to expedite transmission of frames generated by critical or time-sensitive applications. The Forwarding Process of the Bridge may provide more than one transmission queue for a given Bridge Port.

Table 2.3 User Priority to Traffic Classes Mappings with Default Mapping

User Priority	Numbers of Available Traffic Classes						
	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0 (default for unspecified)
1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	2
3	1	1	1	2	2	2	3
4	1	1	2	2	2	3	4
5	1	1	2	3	3	4	5
6	1	1	2	3	4	5	6
7	1	2	3	4	5	6	7

Frames are assigned to queues on the basis of received or regenerated user-priority, using the Traffic Class Table that is part of the state information associated with each Port. The table indicates, for each possible value of user-priority, the corresponding value of traffic class that shall be assigned. Queues correspond one-to-one with traffic classes.

Up to eight traffic classes are supported in a Traffic Class Table which will allow for up to eight level of queue classes in a Port. The traffic classes are numbered from 0 through *n*-1, where *n* is the number of traffic classes associated with a given Port. This is illustrated in Table 2.3. Management of traffic class information is optional. In a given Bridge, it is permissible to implement different numbers of traffic classes for each Port.

Where a Bridge Forwarding Process does not support expedited classes of traffic for a given Port, all values of user-priority are mapped to the default traffic class 0. Support for expedited traffic class should include support for default mapping of user-priority to traffic class 0 as shown in Table 2.3.

For a given supported value of traffic class, frames are selected from the corresponding queue for transmission only if any queue corresponding to numerically higher values of traffic class supported by the port are empty at the time of selection. This implies strict priority scheduling which has been criticised for its pre-emptive of lower priority traffic classes **[Arm00]**.

Originally two traffic classes were recommended. The arguments put forward to support this include the claim that, LAN traffic mainly falls into two categories: time-critical and non-time-critical traffic. And that there will be no great advantage to be gained by further categorisation and provisioning for additional expedited classes **[802.1D&p93]**.

2.5 Multi-Protocol Label Switching (MPLS)

MPLS is a networking technology that represents the convergence of two fundamentally different approaches to packet forwarding paradigm: connection oriented and connectionless. It combines the high-performance capabilities of layer 2 switching with the scalability of layer 3 based forwarding **[Arm00-art] [Vei00] [Cisco02] [Ste98]**. MPLS is expected to improve the performance of network layer routing, provide support for QoS routing, and provide greater flexibility in the delivery of new routing services. The initial MPLS effort as reported by IETF MPLS working group, was focused on IPv4, however the core technology was said to be extendible to multiple network layer protocols (e.g., IPv6, IPX, Appletalk, DECnet, CLNP). MPLS provides connection-oriented label based switching, based on IP routing and the control protocols **[MPLSfrwk] [MPLSarch]**.

The core effort includes, using fixed length of labels to label packets and using the labels to route packets to their destination. Label swapping takes place at MPLS-capable routers along the path a packet takes to its destination. The purpose of label-switching is not to replace IP routing but rather to enhance the service provided by IP Networks by offering scope for Traffic Engineering (TE), guaranteed QoS and Virtual Private Networks (VPN).

2.5.1 MPLS Architecture Overview

MPLS is based on a simplified and faster layer 3 forwarding architecture that employs packet labelling as the basis for expedited forwarding. The architecture ensures that, the assignment of a particular packet to a particular *Forwarding Equivalence Class* (FEC) is done once, as the packet enters the network. An FEC defines a group of IP packets that could share a similar forwarding routing treatment and are forwarded over the same *Label Switched Path* (LSP). The FEC to which the packet is assigned is encoded as a short fixed length value known as a "label". The label is attached to the packet header before the packet is forwarded to its next hop. This ensured that, packets are "labelled" before they are forwarded.
Basically the MPLS architecture consists of two principal components: *control* and *data forwarding*. The control component is responsible for providing fixed-length labels entries into the *Label-Switching Table* (LST). It consists of process threads that use a *Label Distribution Protocol* (LDP) to obtain, distribute and maintain label-forwarding information for all destinations in the MPLS network. The data forwarding component is the executive branch, which switches packets by swapping labels using the label information carried in the packet and the label information maintained in the LST.

In conventional IP packet forwarding, each router makes independent forwarding decisions based on the layer-3 destination address in the IP packet header and its longest prefix match in the router-forwarding table. In some cases (especially in multicast or source routing) the layer-3 source address in conjunction with destination address may be used in processing forwarding decision. These processes are repeated at every hop in the networks. With MPLS forwarding paradigm, the header addresses lookup is done just once as the packet enters the network. The packet is then assigned to a particular FEC. Once a packet is assigned to an FEC, no further IP header analysis is done by subsequent routers, all forwarding decision are driven by the labels. Intermediate routers may only perform label swapping on the packets.

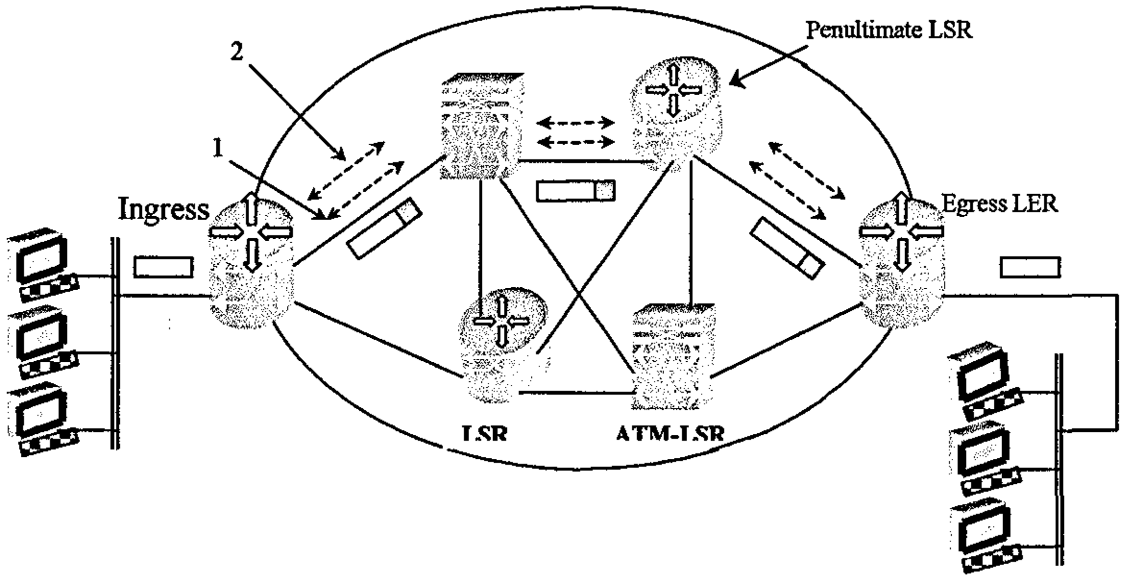
In the MPLS architecture the decision to bind a particular label to a particular FEC is always made by the downstream LSR with respect to the flow of the data traffic. The downstream LSR then informs the upstream LSR of the binding. There are two modes of downstream label distribution: *downstream on demand* and *unsolicited downstream*. In the downstream-on-demand mode, an LSR explicitly requests a neighbour for a label binding for a particular FEC. Whereas the unsolicited downstream mode allows an LSR to distribute label bindings to its neighbours that have not explicitly requested for them.

2.5.2 Data Forwarding Process

Once the control component has converged or has carried out its functions and adequately populated the label-switching table, the data forwarding process becomes simple. The Ingress *Label Edge Router* (LER) to the MPLS network will perform the initial packet classification by looking into the IP header address information and allocating the packet to a FEC and then adding the appropriate label using information from the *Label Information Base* (LIB). The LER will then forward the packet. Subsequent *Label Switched Routers* (LSRs) simply forward the packets by using label-swapping technique. When a packet carrying a label arrives at a LSR, the LSR uses the label on the packet as the index into the entries in its LIB. For an incoming label, the LIB contains a matching entry for the corresponding outgoing label, interface, and link-level encapsulation information to forward the packet.

Using the information in the LIB, the LSR swaps the incoming label with the outgoing label and transmits the packet on the outgoing interface with the appropriate link-layer encapsulation. The data forwarding action performed by an egress LER will depend on whether it forwards the packet to an MPLS-capable node in another MPLS cloud or it forwards the packet to its destination or to a node in non MPLS-capable network. If the egress LER is sending the packet to another

MPLS cloud, it employs the label swapping technique and transmits the packet. In all other cases, it removes the label and uses the IP header destination address information to transmit the packet as in conventional IP router. MPLS architecture as described above is illustrated with Figure 2.5. The figure graphically describes MPLS network operation.



1. Existing routing protocols (e.g. OSPF, BGP, ISIS) establish reachability-route to destination networks.
2. Label distribution protocol (LPD) creates entries to switching-table to provide labels mappings to label switched paths (LSPs) to reach destination networks.
3. On receiving a packet, ingress label edge router (LER) performs normal IP header lookup and classifies the packet to a forward equivalent class (FEC) and adds MPLS label, then forward the packet.
4. Interior label switched router (LSR) simply looks at switching-table to swap incoming label on the packet with outgoing label and then transmits the packet.

Figure 2.5: MPLS Network Functional Components

2.5.3 MPLS with QoS

Explicit *Label Switched Path* (LSP) could be setup based on resource availability in the paths within a network. Traffic flows requesting for such adequate resources that are available in the LSP would be allocated to a FEC, which would carry a label for the relevant LSP. This will ensure that the packets of the traffic flows are routed along the LSP that have the resources the flows requested.

The experimental 3 bits in MPLS label could be setup just like IP precedence values to provide up to eight-levels of per-hop differentiated QoS routing. Thus the class selector field values in DSCP of DiffServ architecture or the IP precedence values in IPv4 could be mapped to experimental bit values of MPLS label for per-hop behaviour specification and treatment.

2.5.4 MPLS with Traffic Engineering

Traffic Engineering (TE) concerns the optimum distribution of traffic loads within networks to achieve efficient utilisation of network resources. With the tools for TE, an explicit path can be specified for certain traffic flows to take based on their resource requirements rather than being based on the shortest hop-by-hop distance-vector or link-state to flow's destination metric employed by IP routers in conventional IP packet forwarding.

MPLS application provides capability for TE **[Aky et al.03] [Bos et al.04]**. The path the conventional IP traffic takes may not be optimal, since it depends on static link metric information without any knowledge of the available network resources along the path end-to-end. MPLS supports the explicit LSP path setup mechanism for network resources reservation for some classes of traffic and for load distribution within the networks. IETF MPLS working group has specified two protocols for such functions: *Constraint Routed Label Distribution Protocol* (CR-LDP) and *RSVP with Traffic Engineering* (RSVP-TE). The features for CR-LDP include: explicit routing, resource reservation for classes, route pinning, path pre-emption, handling failure, and LSP ID. RSVP-TE share similar features as CR-LDP with some minor differences, which include: the concept of nodal abstraction, re-routing of LSPs after failures, and tracking of the actual route of an LSP.

Augmented work on TE in relation to MPLS and DiffServ are treated in, **[AutKir02], [Yen et al.01], [Zha05].**

Summary

The Integrated Services architecture is composed of a framework for a new service model, which includes RSVP as the QoS signalling protocol and a set of implementation mechanisms for per-flow resource reservation and allocation in the Internet. Routers in the path of a flow must maintain Flow States for packet classifiers to identify packets and schedulers to parameterise the treatment flows will receive from the networks.

Differentiated Services architecture adopts a simplified model in which packets are aggregated and classified into groups called Forwarding Classes (FCs). Packets in each FC are marked with a DS Codepoint value to produce a Behaviour Aggregate (BA) that will determine the Per-Hop Behaviour (PHB) the group receives in the network. Per-flow states need not be maintained in the network. Packets are conditioned at the edges of the network and marked to receive PHB relevant to its QoS need within the network.

MPLS represents the convergence of connectionless and connection oriented packet switching technologies. It simplifies layer 3 forwarding and improves its performance. It supports both QoS routing and Traffic Engineering (TE). It is simply based on label swapping in which a packet on entering into the network is assigned to a Forward Equivalent Class (FEC) and given a label. At each hop, the incoming label on the packet will be used as an index to entries in the Label-Switching Table (LST) maintained by the hop (a Label Switched Router (LSR)). The incoming label on the packet will be swapped with the corresponding outgoing label found in the LST and the packet will be forwarded onto a Label Switched Path (LSP) towards its destination.

CHAPTER 3

Generic Components of QoS Architectures

QoS architectures may differ in protocol structure or algorithmic mechanism, but they have the same generic building blocks. Our concern in this chapter is to examine the generic building blocks of QoS architectures and to highlight their importance. The coverage will include: *QoS Specification Template, QoS Signalling Mechanism, Admission Control, Traffic Classification, Traffic Policing and Conditioner,* and *Queuing and Scheduling Disciplines.* We will briefly examine each of these building blocks and place more emphasis on queuing and scheduling disciplines in view of their strategic functions as QoS implementers.

3.1 QoS Specification Template

A QoS specification template concerns the specific format a Quality of Service (QoS) architecture adopts in characterising the resource requirements of an application in the network. A standard way to specify the desired QoS characteristics a flow would need in the network is crucial for the necessary information dissemination on which QoS support depends. The ability to describe the characteristics of a traffic flow is necessary to properly provision resources in the network's infrastructure. This function will include standard semantics to specify and define resource parameters and QoS parameters in the network. Examples of this function could be found in *QoS Specification Parameters* **[RFC 2215]** of Integrated Services (IntServ) architecture, the *interpretation of encoding of DS Codepoint* **[RFC 2474]** of Differentiated Services (DiffServ) architecture, and the *Generic Service Specification* (GSS) framework proposed by Sotiris I. Maniatis et al. **[Man et al.04].**

3.2 QoS Signalling Mechanism

QoS architectures would have a means by which applications can signal their QoS requirements to the network. QoS signalling mechanisms or protocols fulfil this purpose. They are used to describe and carry the characteristics of traffic flows to the network and are also used to reserve resources for the flows from network devices along the paths of data flow end-to-end. This function is very important in the sense that the network must be aware of application traffic requirements to be able to meet its QoS needs. Resource reSerVation Protocol (RSVP) of IntServ is a good example of a QoS signalling protocol **[RFC 2205]**. Also IETF draft on improved signalling protocol is another example **[SignalPro]**.

3.3 Admission Control

Physical resources are finite in the network. QoS can only be achieved if network resources are available. In order to offer guaranteed and predictable services to reserved flows, a network must monitor its resource usage. It should deny network access to new traffic flows when insufficient resources are available. The resources available in the network must be carefully managed in order for QoS to be effectively achieved. This is the function of admission control. For a flow's requirement to be provisioned, there must be adequate resources available that are not already committed to servicing other flows. An admission control agent makes sure that, there are sufficient resources available in the network to meet the needs of both existing flows and new flows before new flows are allowed into the network. It is straightforward reasoning that if there were no admission control to the limited resources available in the network, the resultant contention between flows would yield nothing better than the classical best-effort service. There are basically two functions of admission control: the first is to determine if a new flow can be setup based on the admission control policies and the second is to monitor and measure available resources, **[Myk et al.03] [Wan01,p.25]**.

3.4 Traffic Classification

A multiservice network would employ some kind of differentiated services in order to meet the divergent requirements of the various applications that use the service of the network. In a differentiated services environment, the nodes in the network must have a means of identifying a traffic flow in order to appropriately provisioned discriminatorily for the service of the flow. The process of identifying traffic flows would involve classification of flows into groups and marking each flow to belong to a particular group. This is the role of the flow classification module, thus it identifies and distinguishes packets belonging to one flow from other packets belonging to another flow in order to allow differentiated services for the packets. The classifier uniquely identifies and marks packets based on some predefined rules. One of the rules usually adopted is based on the source and destination addresses, protocol ID, source and destination ports of the packet. These five fields of the packet header are together usually referred to as a *five-tuple.* Packets can also be identified and marked on the basis of Type of Service (ToS) byte field of Internet Protocol (IP) version 4 (IPv4) header, the Traffic Class field of IP version 6 (IPv6) header, and the DS Field **[RFC 2474]** of DiffServ architecture–the three are same object in different specification environments.

3.5 Traffic Policing and Traffic Conditioner

It is important to ensure that a traffic flow stays within its traffic specification and does not send excess traffic into the network. A completely free and unmonitored access of traffic flows into the network can encourage misbehaving sources to flood the network with traffic flows that will grievously consume network resources and consequently pre-empty resources availability for well-behaved traffic flows. Apart from not sending excessive traffic to the network, it is also necessary that a flow maintains its traffic characteristics as it travels through the network. These are the functions of traffic policing and traffic conditioner modules of network elements **[GuiDup02]**.

Policing is the term used to describe the enforcement function, for example, of *Service Level Agreement* (SLA) between the customer and service provider. It is typically implemented at the edges of the network where a traffic flow is first introduced into the network. It carries out its functions to determine if a traffic *flow is in-profile* or *out-of-profile*. The *token bucket scheme,* which will be described in Section 3.5.1, is usually used as tool for policing. The token bucket scheme is a variation of the *leaky bucket scheme* initially proposed by Turner **[Tur86].** The policing function will measure the traffic flow to determine if the flow is in-profile or out-of-profile, if the flow is in-profile, its packet is allowed into the network, but if it is out-of-profile the policer can take one of two actions. It can be intolerant and *drop* the packet or adopt a degree of tolerance and pass the packet to the traffic conditioner who will *mark* the packet for wait-and-see-action such as *dropping* the packet if congestion deteriorate in the network.

The *Traffic Conditioner* performs such actions as *shaping* and *marking* of the traffic flows. Any customer traffic that is out-of-profile of the agreed temporal properties of its traffic stream, is either *rate-limited*, that is *shaped*, or *marked* for subsequent event determinant action in the network. The shaping function can employ either the *token bucket or leaky bucket algorithm* which involves buffering of the packets of the flow and then sending at a rate conformant to agreed traffic characteristics. The relevant leaky bucket scheme is briefly described in Section 3.5.2. The marker can set some fields of the packet header for example to indicate drop probability in the network.

3.5.1 Token Bucket Scheme–Tool for Policing

A token bucket scheme is used as a means of measuring traffic flow to ensure it conforms to its agreed traffic temporal characteristics **[Abe et al.06]**. Basically it consists of a fixed rate *r* token generator, a bucket of finite depth *b*, and a meter-scheduler. As shown in Figure 3.1, tokens are generated and placed in the bucket at a constant rate of *r* tokens per second and the rate *r* should be equivalent in value to the long-term average rate-limit allowed for the flow.

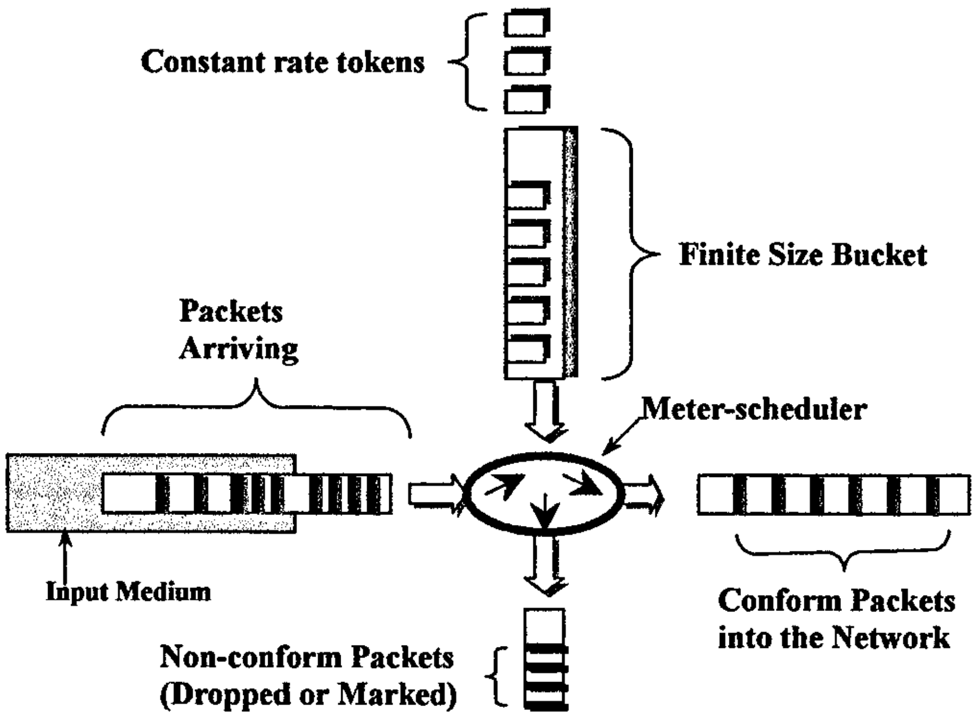


Figure 3.1 Token Bucket Scheme.

The bucket has a fixed capacity of depth *b*. Assume each token corresponds to one byte of data, and tokens are added into the bucket at the rate of *r* bytes/sec, if the bucket fills up, newly arriving tokens are discarded. When a packet arrives, and equivalent tokens are available in the bucket, the equivalent tokens are removed from the bucket, and the packet of the flow is considered in-profile. The meter-scheduler will allow the packet into the network. When a packet arrives and there are not enough tokens in the bucket, the packet is considered out-of-profile. The meter-scheduler will either drop the packet or send it to a traffic conditioner for marking and consequent action.

The token replenishment rate *r*, represents the long-term average rate-limit allowed for the flow, which can allow some bounded burstiness in the flow. If the bucket is full and a burst of packets arrives, the meter-scheduler will immediately transmit packets equal to *b* bytes (the bucket capacity) with no gaps in between and remove the same amount of token from the bucket. In a time interval *t*, the burst of packets that can be allowed is bounded by (r x *t* + *b*) bytes.

3.5.2 Leaky Bucket Scheme–Tool for Shaping

Shaping is an act of rate limiting the traffic flow to the desired packet rate profile. It will prevent very bursty sources from over-stretching network resource capacities. The leaky bucket scheme provides the easiest method of shaping traffic flows, in that, it can convert a variable rate packet input into a uniform rate packet output. The leaky bucket model as illustrated with Figure 3.2, consists of a *fixed size bucket* with capacity β bytes, and a *constant rate packet scheduler* for each flow.

As traffic flow arrives, the packets are placed in the bucket. At any instance of time, the bucket can only take a maximum of β bytes. If a packet of size *k* bytes arrives (*k* < β), it can only be placed in the bucket if the bucket has *k* bytes space left, if not, the packet is discarded. The constant rate scheduler at the bottom of the bucket (head of queue) sends the output traffic into the network at an allowed constant rate α bytes/sec. The depth of the bucket is computed based on the queue delay allowed for the flow, and the uniform speed of the scheduler. Thus a packet of size *k* bytes at the tail of the bucket would be held for a period of time equals to β/α before being transmitted.

The leaky bucket can also be used for traffic policing with little orientation in its operation as described above, **[GanMcK99] [Mer et al.91] [Cha et al.00]**.

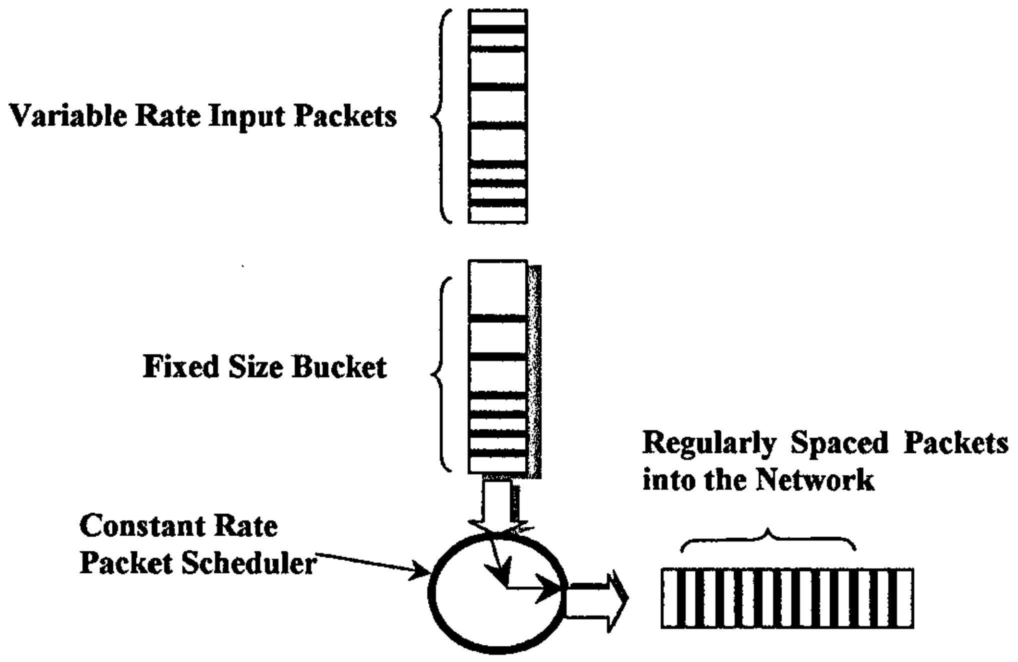

Variable Rate Input Packets
Fixed Size Bucket
Regularly Spaced Packets into the Network
Constant Rate Packet Scheduler

Figure 3.2: Leaky Bucket Scheme

3.6 Queuing and Scheduling Disciplines

Queuing and scheduling mechanisms are the most important component of QoS architectures since they are responsible for implementation of QoS resource allocation policies. All QoS architectures assume deployment of an associated queuing-scheduling discipline that implements the architecture's functionality. The queuing-scheduling mechanism plays a key role in ensuring that traffic flows receive the network resources allocated to them through the allocation mechanisms of the QoS architectures. As traffic flow travels from its source to its destination, it requires instantaneous waiting room offered by the queuing method adopted in the network, in order for the flow to receive resource allocations from network elements along its path end-to-end. The scheduler ensures that, the queued traffic-flow, while being serviced, receives the appropriate network resources allocated to it from each network element in its path from source to destination. Thus queuing and scheduling mechanisms provide the important function of transforming QoS objectives of QoS architectures into reality. In view of their importance we will examine various types of queuing and scheduling disciplines in the next section.

3.7 Types of Queuing and Scheduling Disciplines

Various queuing and scheduling disciplines are available to meet the service requirements of applications traffic in networks. Each is tailored to meet the service objective of the network concern. In practice, queuing systems could either be a mono queuing systems in which all traffic shares a single buffer space in the network elements, or multiple queuing systems for which each flow or class of flows have different buffer space allocated to it in network elements to simplify differentiated services. In addition to the functionality of providing buffer space for transit traffic, each of the queuing system could have appropriate sorting mechanism to provide priority service for flows deserving it. The most prominent among the queuing and scheduling disciplines are: *First In First Out* (FIFO), *Simple Priority* (SP), *Round Robin* (RR), *Weighted Round Robin* (WRR), *Deficit Weighted Round Robin* (DWRR), *Weighted Fair Queuing* (WFQ). Before we examine the various types of queuing-scheduling disciplines, it is helpful to look at the basic requirements needed to achieve their objectives.

3.7.1 Basic Requirements of Queuing-Scheduling Discipline

Queuing-scheduling disciplines are required to meet some basic requirements in order to fulfil their objectives in multiservice network environment. The basic requirements are necessary to meet the need of both integrated and differentiated services. The basic requirements are examined next.

3.7.1.1 Protection among Flows

In a multiservice network environment, queuing-scheduling disciplines are required to resolve contention for network resources among flows. They must try to ensure that misbehaving flows do not affect the performance of well-behaved flows that stay within their contracted traffic characteristics. If queuing-scheduling disciplines do not differentiate between flows, a rogue flow can send traffic at a very high rate into the network and capture a large portion of the link bandwidth. This will prevent other flows from getting their due share.

3.7.1.2 Flexibility in Resource Allocation

Queuing-scheduling actions must be flexible in allocation of resources so that they can separately control per-packet delay through the network for each flow. The buffer space and bandwidth resource allocation for each flow must reflect its required delay bound through the network.

3.7.1.3 Support for Real-time and non-Real-time Traffic

The queuing-scheduling algorithm must be tuned to meet the needs of both real-time and best-effort traffic. It must ensure that meeting the needs of real-time traffic will not adversely starve best-effort traffic.

3.7.1.4 Implementation Simplicity and Efficiency

The queuing-scheduling algorithm must be amenable to easy implementation. It must be devoid of implementation complexities that will hinder scalability. Should support simple implementation schemes that will be elegant for its purpose.

3.7.2 First-In First-Out (FIFO) Scheduling

In FIFO, also known as *First-Come First-Serve* (FCFS) scheduling, packets are queued in the order in which they arrived in the input queue and then serviced or transmitted in the output queue in the same order. As illustrated in Figure 3.3, packets 1, 2, 3 and 4 arrived in that order (i.e. 1 arrives before 2, etc) in the input queue of FIFO queuing and scheduling system, and are transmitted in the same order 1, 2, 3 and 4 (i.e. 1 serviced before 2, etc) at the output queue.

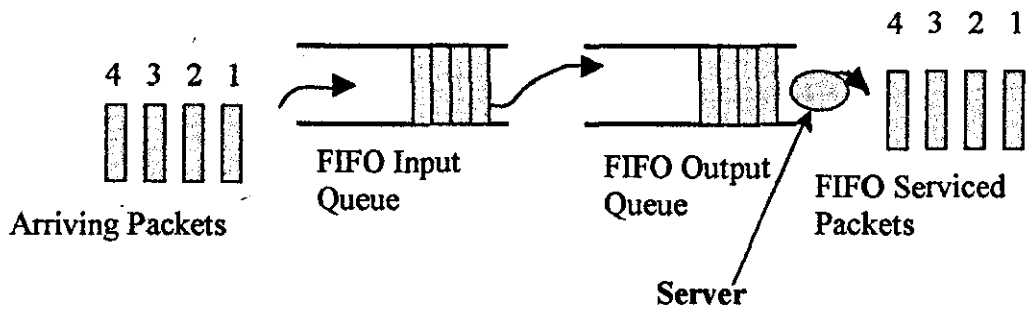


Figure 3.3: FIFO Scheduling Discipline.

FIFO is the traditional packet queuing and scheduling mechanism most commonly employed in Internet networks. It is simple, easy to implement and scalable. However it has no mechanism to differentiate between flows, hence it cannot prioritise among them. Besides not being able to prioritise one flow over others, FIFO also cannot offer either protection or fairness to well-behaving traffic flows because badly behaving or rogue flows can consume most of the network resources thereby denying service to other flows that stay within the conservative rate. Adaptive flow-control mechanism in the Internet, such as the Transmission Control Protocol (TCP) cannot offer protection for well-behaved flows against rogue flows, if such a situation exists with FIFO, since FIFO flows receive service approximately in proportion to the rate at which they send data to the network.

3.7.3 Priority Scheduling

In priority scheduling systems, packets arriving from multiple interfaces or ports are identified and classified into a number of output queues. Theoretically many levels of priority queues can be maintained in the multiple output queuing system, but for manageable practical purposes three or four output queues have been proposed **[Hus00,p.203]**, **[Veg03,p.91]**. The proposed four output priority queues—high, medium, normal and low, accordingly are in decreasing order of priority. Figure 3.4 is used in illustration of priority queuing and its scheduling discipline.

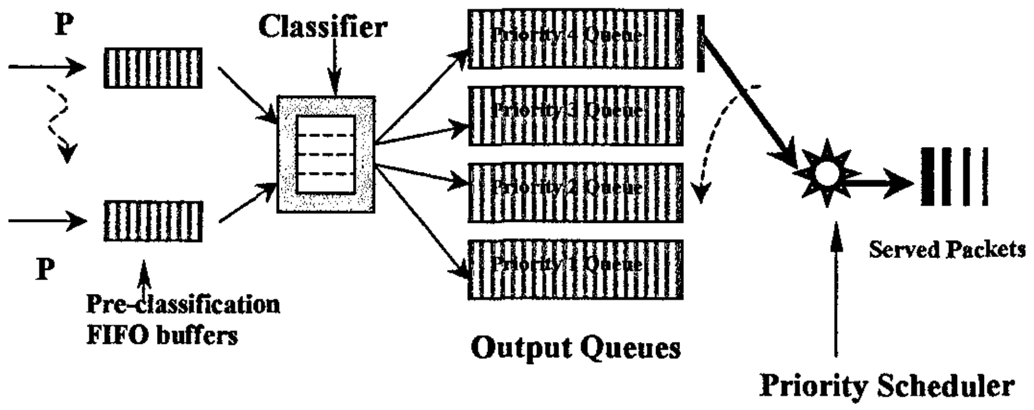


Figure 3.4: Priority Queuing and Scheduling Scheme.

In the arrangement here, the priority-4 queue has the highest priority and priority-1 queue the lowest priority. For example network control traffic could be placed in the highest priority queue while voice packets are placed in the next priority queue and then followed by the video packets. Best-effort packets could be placed in the lowest priority queue. Service within each output priority queue is based on the FIFO scheduling discipline. Non empty high priority queues are serviced before any lower priority queues could be serviced. Any lower-priority packets are held in the queue until no further higher-priority packets are awaiting transmission. Thus a packet is scheduled from the head of the priority-*n* output queue as long as all queues of higher priority are empty. Priority scheduling provides a simple form of differentiated service. It could be used to give preference to some classes of traffic, which would then be placed on high-priority queues and will consequently enjoy lower delay and adequate bandwidth resources. Combined with admission control to limit the input rate of high-priority traffic, low-priority traffic may still get enough network resources to perform marginally well.

However strict priority scheduling has a number of disadvantages, which are intuitively inherent in the nature of the scheme. High priority traffic enjoys network resources to the detriment of lower priority traffic. While high priority traffic is serviced, low priority traffic are backlogged in queues and continue to have increased queue delay. If high priority traffic arbitrarily increases, the priority scheduler will devote almost the whole of the network resources to the service of the high priority traffic and the low priority traffic will be starved of network resources. The *pre-empting* of network resources by high priority traffic to low priority traffic could degenerate to very serious performance degradation of low priority traffic. Thus strict priority scheduling is not suitable in multiservice environment where each flow or class of flows require a predictive share of network resources.

3.7.4 Round-Robin Scheduling

To mitigate against pre-empting of network resources by high priority traffic to low priority traffic, an alternative form of scheduling uses a *Round-Robin* (RR) scheme in which the scheduler visits each nonempty queue in turn, and services one packet on each visit. This enforces some degree of sharing bandwidth resources among contending flow and prevents one class of traffic from starving others of network resources. Figure 3.5 illustrate RR scheduler with four output queues.

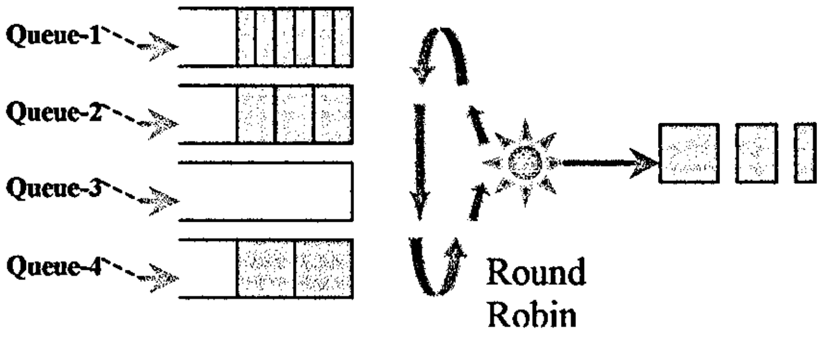


Figure 3.5 Round Robin scheduling.

As shown in Figure 3.5, the scheduler services one packet from queue-1 then moves to queue-2 and services one packet, and because queue-3 has no packet, the scheduler moves to queue-4 and service one packet. The cycle continues again from queue-1, any queue that has a packet will have one packet transmitted in every cycle. RR scheduling prevents stagnation of any backlogged queue of traffic flow. The scheduling decision is simple and efficient, it only needs to look at the next nonempty queue to select the next packet for transmission. Also RR will ensure that each flow get some percentage of the link bandwidth.

Nonetheless RR scheduling has some drawbacks. First, it does not allocate bandwidth proportional to the requirements of individual flow. Second, it does not take into account the variable packet sizes of the Internet Protocol (IP) datagram, which could result in unequal sharing of bandwidth resources. For example, consider Figure 3.5 in which the scheduler transmits one packet at the head of each of the three nonempty queues in one cycle. If the packet sizes in queue-1, queue-2 and queue-4 are 64 bytes, 192 bytes and 384 bytes respectively. This will result in a ratio of 1:3:6 for bandwidth allocation, which certainly is an unequal and unfair allocation. The third setback concerns the *delay* and *jitter* introduced as a result of cycle *interval* and its *variation*, which is a function of variable packet sizes in the queues and the link bandwidth. This problem will be exasperated if many of the packet sizes in the queues are very large and tend to the *maximum size* of IP datagram—65535 bytes, and the link bandwidth speed is low. For illustration, consider an interface with *n* number of queues, link speed of *R* bytes/sec and

Maximum Transmission Unit (MTU) of B bytes. The time it takes to transmit one MTU-size packet is (B/R) seconds. Thus in the worst case scenario, a packet that arrives in an empty queue that has just been skipped will have to wait $(n - 1) \times (B/R)$ seconds until the scheduler comes back in its round to service the packet. The parameters of the variables in this computation are random, and causes the packets involve to incur high queue delay and jitter.

3.7.5 Weighted Round-Robin Scheduling

The *Weighted Round-Robin* (WRR) scheduler is a variation of the RR scheduler in which each queue is assigned a weight corresponding to a proportional share of the bandwidth allocated to it. WRR is also known as **Custom Queuing [Veg03]**. For a hypothetical case, assume each queue-i is assigned a weight W_i in bytes. In WRR scheduling, the scheduler services a number of packets proportional to W_i from queue-i every time it visits the queue **[BhaCro00]**. The weights are assigned to take into account the different packet sizes and the required bandwidth allocation for the flows in each queue. This is based on the assumption that the average packet size of each flow is known in advance, which may not be the case for many flows. The algorithm works in the following way: the weights are allocated in byte-counts. When the scheduler visits a queue and starts to transmit packets, the number of bytes transmitted is deducted from the byte-count. As long as the byte-count is not reduced to zero the scheduler will continue to transmit packets from the queue. If the scheduler has started to transmit a packet and the byte-count reduce to zero before completing transmitting the packet, the scheduler must complete transmission of the packet before moving to another queue. WRR scheduling can be a useful tool for link bandwidth sharing. It will ensure that each flow or class of flow gets a minimum of the bandwidth allocated to it. However, it does not take into account the delay requirements of individual flows. The delay and jitter incurred from cycle intervals, which is inherent in the RR model may not be suitable for time sensitive applications such as voice over IP which must have its small size packets interleave or interspersed with larger packets from all other queues. Also WRR scheduling may not provide accurate link bandwidth sharing. For example, consider Figure 3.5 again, in which queue-1, 2, 3 and 4 have allocated weights of 100, 200, 400 and 300 byte-counts respectively. The sizes of queued packets in queue-1, 2, 3 and 4 are 64, 192, 0 and 384 bytes respectively. When the scheduler completes transmission of the first packet on its visit to queue-1, the queue will still have (100 - 64), 36 byte-counts left, so the scheduler will transmit the second packet. Thus a total of 128 bytes are transmitted in queue-1. In a similar manner queue-2, 3 and 4, will have 384, 0 and 384 bytes transmitted respectively. The ratio of bandwidth allocation for nonempty queues is 1:2:3 but the actual ratio of bandwidth utilisation is 1:3:3. This proves that WRR scheduling may not be accurate in sharing link bandwidth. In order to improve its accuracy in link bandwidth sharing, the weighted byte-counts and the largest average packet size may need tuning for synchronisation, such that, more packets will be transmitted at each queue visit, and the total bytes transmitted will not overly exceed the allocated byte-count. This again will increase the cycle interval time and increase the introduced jitter.

3.7.6 Deficit Round-Robin Scheduling

Another adaptive scheduler similar to *Weighted Round-Robin* (WRR) scheduler is called the *Deficit Round-Robin* (DRR) scheduler **[ShaVar95]**. Unlike WRR, DRR does not require that average packet sizes of flows be known in advance. With proper configuration it can provide both accurate and fair sharing of system resources and bandwidth among contending flows. In a DRR scheduler, each flow or class of flow has a usage regulating parameter which is a variable called *deficit counter* that is initially set to 0. In addition, the scheduler uses another parameter called *quantum size* to decide how many bytes of data to service from each flow during each visit. Let us denote the *deficit counter* for flow-queue i as D_i. In a DRR that is strictly fair, the *quantum size* in bytes is uniform for all flow-queues and is here denoted as Q. The algorithm works as follows:

- Before the start of flow, the D_i is initialised to 0.
- During the first round that the scheduler visits each active flow-queue, the *D*i is set in value to the *Q bytes*, and the scheduler tries to service a number of packets whose total size is not more than *D*i —the *current operative deficit counter value*.
- If the size of the packet at the head of a flow-queue is not larger than *operative D*i bytes, the scheduler services the packet. But if the packet size is larger than

*operative D*i bytes, the scheduler postpones servicing it to the next round and retains the current value of *operative D*i as the next round *D*i. Or if the aggregate size of a number *n* of consecutive packets at the head of a flow-queue are not larger than *operative D*i bytes, the scheduler service all the *n* packets and subtract the total size of the *n* packets serviced from the flow's *operative D*i to produce the next *D*i.

- During the subsequent visit to each of the flow-queues, the scheduler adds *Q* to *D*i to produce *operative D*i, and then service a number of packets whose aggregate size is equal to, or less than (*Q* + *D*i)—the *operative D*i. The scheduler will then compute the difference, (*operative D*i – aggregate size of packets serviced during the visit) as the next Di.
- On any visit, if a flow-queue is empty, the flow's *D*i is re-initialised to 0.

In brief, for each active flow-queue, the scheduler transmits as many packets as the *operative deficit counter value* allows on a flow-queue visit. The *operative deficit counter value* is given as (*Q* + *D*i), that is the sum of the *quantum size Q* and the variable *deficit counter value D*i obtained from previous visit. When a packet is transmitted, the *current operative deficit counter value* is reduced by the packet size. The aggregate size of total packets transmitted on a visit to any active flow-queue should be less than or equal to *current operative deficit counter value*. The aggregate size of transmitted packets is deducted from the *current deficit counter value*. The unused part of *current deficit counter value* is recorded as the next *deficit counter value D*i, and represents the amount of quantum that the scheduler owes to the flow during the visit. At the next round of visiting, the scheduler will add the *Q* to *D*i to produce the new *operative deficit counter* on which the number of packets to be transmitted will depend.

DRR main advantages are its simplicity and ease of implementation which made it attractive for both software and hardware implementation **[DurYav99]**. Its disadvantage is that, by itself, it could not provide a strong latency bound because of the *round intervals*, especially if the *quantum size* is large, it will introduce variable delay to small packet size flow at each node along the path of flow end-to-end.

Weighted Deficit Round Robin (WDRR) is the weighted version of DRR. It allows for preferential bandwidth allocation. Link bandwidth could be shared in any desirable percentage. It operates on three parameters: the weight for each flow or class of flows, the normal quantum size, and the deficit counter for each flow-queue. Unlike DRR, which has one quantum size for all flows, WDRR assigns each flow-queue a quantum size based on the product of the flow's weight and uniform quantum size in operation. WDRR operates like DRR and they both share the same properties of predictable system resource and bandwidth usage by contending flows.

3.7.7 Max-Min Fair Sharing

The problem of how to evenly or fairly divide a resource among competing unequal requests has been the subject of careful studies among network researchers. One form of fair-sharing scheme that has been widely considered in the literature is called *max-min fair sharing*. The max-min fair sharing approach set out to meet the requirements of small requests and fairly distribute the remaining resources among larger requests. The algorithm is as follows:

- Flow-requests are sorted in the order of increasing resource requests.
- Resources are allocated in the order of the increasing resource requests (smallest request first)
- No flow-request is allocated resources more than its demand.
- All Flow-requests with unsatisfied demand get an equal share of the resource

The algorithm can be implemented in four steps **[Wan01,p. 61]:**

1. Calculate the initial fair share = (total resource capacity / total number of requests).
2. For all flow-requests that have a demand equal to or less than the initial fair share, allocate the flow's actual demands.
3. Remove the satisfied flow and subtract the already allocated resource sizes from the total available resource capacity.
4. Repeat step 2 and 3 for remaining flows, with the current fair share = (remaining resource capacity / remaining number of flows) until each of the remaining demands are larger than the current fair share. All the remaining demands with larger request than the current fair share will all have equal share.

It is useful to illustrate this with numbers in order to precisely understand the algorithm. Let us imagine that a resource has capacity of 16 and will service the needs of 4 users—U_1, U_2, U_3 and U_4 with resource request sizes 3, 4, 6 and 8 respectively. Using the max-min fair sharing algorithm, U_1 is entitled to resource allocation (16 / 4) = 4 which is more than its requirement. Its request will be met and the unused excess of 1 returned to the resource capacity pool. U_2 the next-smaller demand is allocated with resource size, ((16-3) / 3) = 4.33, its request will be met and the excess 0.33 will be returned to the resource capacity pool. Repeating the same procedure, U_3 gets ((13-4) / 2) = 4.5 resource allocation size. The resource allocated to U_3 is less than its demand, it is serviced but its demand is unsatisfied. In the same manner, U_4 gets 4.5 resource allocation, it is serviced but its request is unsatisfied. Thus U_3 and U_4 got an equal share of resource allocations but their demands are not satisfied because their requests are higher than the fair sharing of resources. *The scheme is referred to as max-min fair sharing because it maximises the minimum share of a user whose demand is not satisfied*

The max-min fair sharing algorithm as described above does not differentiate between requests, all requests are treated equally. An extension to the scheme that will take differentiated services into consideration will be to associate a weight to each request. The evolved scheme will be referred to as max-min weighted fair sharing in which each user's share is proportional to its assigned weight.

3.7.8 Generalised Processor Sharing

The Generalised Processor Sharing (GPS) is an ideal queuing-scheduling model based on the approach of the max-min fair sharing of resource allocation. GPS implements max-min fair sharing resource allocations using an infinitely small service quota. GPS puts each flow in its own logical queue and services an infinitesimally small amount of data from each nonempty queue in a round-robin fashion **[Par92]** **[Pao04]**. It services only an infinitesimally small amount of data at each turn so that it visits all the nonempty queues at any finite time interval, thus being fair at any moment of time.

If a weight is assigned per-flow, GPS services an amount of data proportional to the assigned weights at each round of service. This GPS extension is equivalent to the *max-min weighted fair sharing* resource allocation.

GPS is an idealised algorithm, which is not possible to implement since it does not take into account the fact that a packet is a discrete unit that must be transmitted as such. The GPS's utility is that, it defines a metric of effectiveness, to measure implemented queuing-scheduling disciplines, in relation to how close the queuing-scheduling discipline is to the ideal GPS.

3.7.9 Fair Queuing and Weighted Fair Queuing

In the preceding sections we have examined scheduling algorithms that are increasingly effective in providing resource partitioning among multiple flows and also provide fair or proportional sharing of bandwidth among contending flows. *Fair Queuing* (FQ) and *Weighted Fair Queuing* (WFQ) offer scheduling algorithms, which exhibit accurate fair-resource sharing or proportional-resource sharing among contending flows with the property that the allocated resources are guaranteed and latency can be bounded. The basic concept of FQ or WFQ is that of bit-wise round robin scheduling. The ideal algorithm, which has the property of fair sharing, was proposed by Demers et al, **[Dem et al.89]** named, *Bit-by-bit Round Robin* (BRR), in which the scheduler sends one bit at a time from each flow in a round robin fashion. The idealised scheme simulates a Time Division Multiplexing (TDM) model over a number of flows. In the ideal model, each flow is assigned to each TDM channel and a bit from a flow will be transmitted in its corresponding time slot. As shown in Figure 3.6(a) for illustration, when Packet 1 arrives in its flow, a single bit from the packet will be transmitted and a bit each from the other two flows will be transmitted before a bit will be transmitted again from Packet 1. Consequently, at any point in time, each flow gets an equal share of the bandwidth. In the Figure 3.6(a), the order in which the packet assembler fully assemble each packet will be determined by the order in which the last bit of each packet is transmitted. The idealised bit-by-bit round robin (BRR) described so far shares the same modus operandi with the Generalised Processor Sharing (GPS) discussed in Section 3.7.8. They both present the same concept in a different constructional language. They both lay the foundation for algorithm on which fairness in resource allocation can be built.

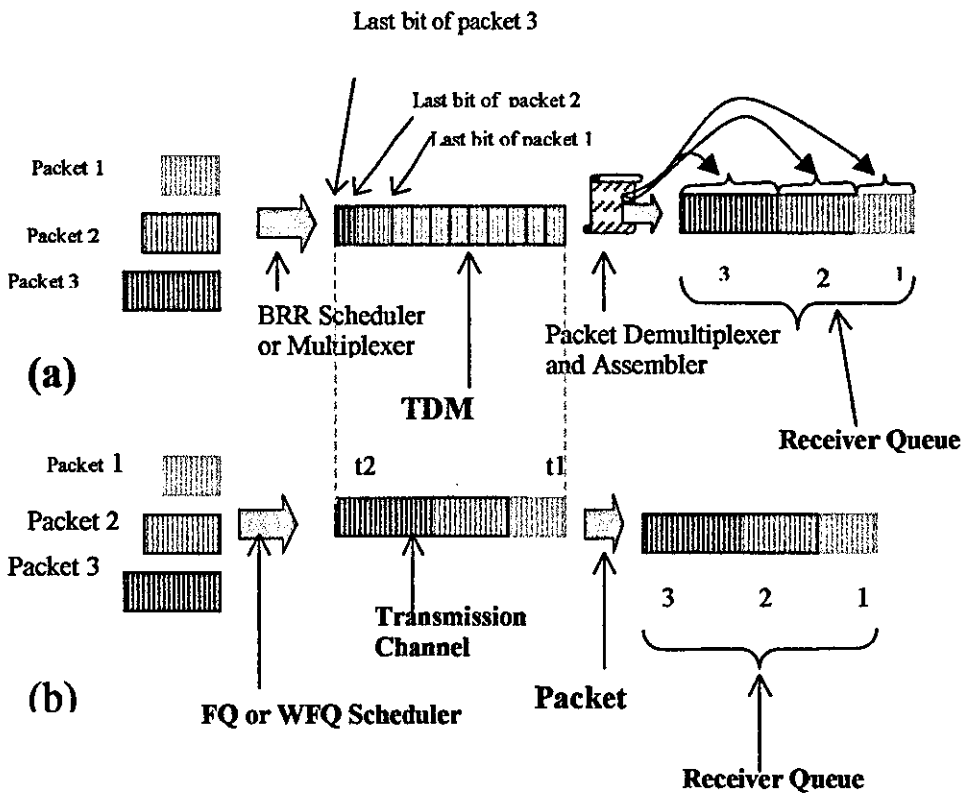


Figure 3.6: Comparison of FQ / WFQ with Idealised BRR Scheduling

(a) BRR Scheduling Model

(b) FQ or WFQ Scheduling Model

Both the BRR and GPS scheme described above are impossible to implement in practice, since packets are discrete units that should be transmitted as a whole unit. An approximation to BRR called *fair queuing* (FQ) was also proposed by Demers et al., and it simulates BRR scheduling model. FQ packet management algorithm ensures that, the time interval it will take to transmit a number of packets from a number of flows will be the same as the time interval the idealised BRR scheduling model will expend in transmitting the same packets. As illustrated in Figure 3.6(b), the time interval between t1—the instant the first bit of packet 1 is transmitted and t2—the instant the last bit of packet 3 is transmitted will be the same in both the idealised BRR scheduling model and the FQ scheduling model.

Under the FQ model, when a packet arrives in a queue, the scheduler computes the time when the packet's last bit would have been transmitted using the BRR scheme. Based on this departure time, the packet is then inserted into a queue sorted by packets departure times, and the packet is transmitted or serviced according to its position in the ordered queue. This guarantees that every active flow gets a fair share of the bandwidth, and it is serviced at a fine granular resolution of *time interval* not larger than the time to transmit the largest packet in the network at the relevant link speed. A variation of FQ is *weighted fair queuing* (WFQ), in which flows are assigned different weights to reflect their bandwidth requirements. The service discipline of WFQ is the same as that of FQ except that, in each round the number of packets transmitted in a flow is proportional to its weight rather than single packet as assumed under FQ model.

WFQ offers a way for a network operator to guarantee the minimum level of resource that will be allocated to each defined service class. This guarantee holds regardless of flow intensities of other service classes. WFQ also provides great flexibility in resource allocation to meet a variety of resource objectives such as link sharing or providing guaranteed delay bounds. If a flow rate is uniform, since each hop in the path of the flow end-to-end can guarantee the flow's share of bandwidth based on the flow's assigned weight, WFQ can be used to provide end-to-end guarantee on both bandwidth and worst-case delay bound. This is of course an intuitive result—if a bounded load is processed by a committed resource, then the throughput and delay incurred should be predictable. Besides the intuitive rationale, Parekh and Gallager **[ParGal94]** have proved an important bound on worst-case end-to-end delay suffered by each packet in a flow traversing a series of WFQ schedulers. Their result is based on the service rate R assigned to a flow at each WFQ scheduler. The higher the service rates, the faster the packets in the flow are serviced at each hop, resulting in lower end-to-end delay.

The main setback of WFQ (or FQ) is the implementation complexity. The maintenance of sorted queues and insertion into sorted queues would be expensive especially when a large number of flows are involved **[Veg03]**. Also WFQ requires the maintenance of per-flow Scheduler State which increases the memory space required in WFQ router implementation. Another setback is that, WFQ does not intrinsically bound jitter below any useful value less than delay bound it offers. If smaller jitter bound is required for some service classes, then some element of priority queuing must be re-introduced into the queuing discipline coupled with admission control.

Summary

The generic components of QoS architectures have been fairly closely examined in this chapter. A QoS Specification Template is the semantics used to describe application traffic characteristics. A QoS Signalling Protocol is used to inform the network of application traffic characteristics and resource requirements. Admission control is used to balance network loading with resource availability. Traffic Conditioner performs traffic shaping functions and Traffic Policer performs enforcement actions.

There are various Queuing-scheduling disciplines. They together implement QoS resources allocation policies.

CHAPTER 4

Empirical Investigation to Determine the Optimum Number of Traffic Queuing Classes for Multiservice IP Networks

There exists a dualism in the nature of queuing—a pleasant attribute on one part of its functions and an unpleasant attribute on the other part. The pleasant aspect of queuing concerns it's *civilized organizational structure* from which fairness in service could be derived. The "*First-Come First-Served*" (FCFS) service discipline could be optimized for fairness in the absence of any other constraint, such as the characteristic nature of queue occupants and their unique service requirements. On the other hand the unpleasant nature of queuing concerns the "*waiting time*" wasted in the queue. This may have far reaching effects such as, negatively impacting on the economy (man-hours wasted) and the quality of service received by the queue occupants. The latter case is more prominent in telecommunication services, especially in communication networks where the *waiting time* suffered by queued packets may seriously affect the quality of the messages received at the destination. Thus queuing study and analysis (queuing theory) is desirable in order to truly understand the nature of queues. Queuing management (application of queuing theory) is equally desirable in order to be able to proffer solutions to queuing problems.

In chapter three, we highlighted the importance of an appropriate queuing discipline as the engine that drives Quality of Service (QoS) architectures in a multiservice network environment. We noted that, the effectiveness of a QoS architecture depends on its queuing discipline and queue management, which jointly serves as implementers of the QoS architecture's resource allocation policies. Integrated Services (IntServ) and Differentiated Services (DiffServ) architectures, which are the proposed standards for QoS architecture in the Internet, support the need for segregation of application traffic flows in order to discriminatorily meet the various QoS requirements of the traffic flows. A natural implementation strategy for each of these architectures resides in ***multiple queuing systems***, which will allow easy resource partitioning for each of the flows. The central parameter to judge the performance of any QoS paradigm is its ability to control ***packet delay and its variation*** to meet the QoS needs of application traffic. The process of packet delay minimization as a function of queue discipline adopted by a QoS architecture, is the central yardstick for measuring the effectiveness of any IP QoS paradigm. While it has been recognized that a *multiple queuing* system is necessary for multiservice IP Networks, the *Optimum Number of Traffic Queuing Classes* (ONTQC) that will meet the need of integrated services in the IP Networks is yet to be determined through analytical or empirical investigation.

In this chapter, we present the introduction of work on empirical investigation to determine the ***optimum number of traffic queuing classes*** (ONTQC) that will meet the QoS requirements of integrated services in IP Networks. In Section 4.1, we present the case for motivation for the work, which includes input for IntServ and DiffServ architectures and the lack of experimental proof and consensus on the subject. In Section 4.2, we highlight the factors that influenced our choice of tools for the investigation. The description of input quantity and output quantity to our experimental queuing systems is the theme of brief discussion in Section 4.3. We proceed to present classification of applications in relation to their priority and QoS need in Section 4.4.

4.1 Motivation for the Work

It has been recognized that *multiple queuing systems* are inevitable for multiservice QoS in IP Networks, the issue is how many traffic queuing classes would be optimum. While a number of Standards Organizations, proprietary bodies, and others in the research community have specified or proposed different numbers of traffic queuing classes to support integrated services in the Internet, none has supported its proposal or specification in the form of analytical or empirical investigation. In view of the varied numbers of traffic queuing classes that have been specified or proposed (i.e. *class of service* (CoS)), it can be said that, there is lack of a consensus on the issue. Some organizations have proposed or specified two, some three, some four, some five, and some eight traffic queueing classes to support QoS in multiservice Internet. As far as we know, none of the proposals or the specifications have been supported with standard verification in the form of analytical or simulation investigation.

The main thrust for this work then includes:

- The need for verification of what should be the optimum number among the different proposals or specifications for the number of traffic queuing classes that would meet the QoS needs of multiservice IP Networks.
- To resolve the issue of lack of consensus on the subject among those concerned.
- To provide a proven point of reference for QoS architectures that adopts multiple queuing systems in their implementation strategy.
- To provide input to the effort of standard organizations on specifications of QoS deployment that involves multiple queuing systems for resource partitioning.

The issue of variation in CoS as highlighted above provides the impetus for action and will be discussed in more detail in this section. The discussion will focus on IEEE 802.D 1993 standard and subsequent standards on number of priority queuing classes, ITEF IntServ and DiffServ QoS architecture specifications on the subject and the proposals of other organizations on the subject.

4.1.1 IEEE 802.D MAC Bridges Specifications on Traffic Queueing Classes

The 1993 IEEE 802.1D MAC Bridges standard, specified two traffic queuing classes to support integrated services in a LAN environment **[802.1D&pYr.93]**. This was based on the heuristic consideration that application traffic flows are broadly dichotomous in grouping, with regard to their QoS requirements in the network. They are either elastic (non real-time) or non-elastic (real-time) applications. Within these broad groupings, there are ranges of sub-groupings, which command respect from the research community. The argument that was put forward to justify a two-class multiple queuing system was that, there would be ***no appreciable gain*** by increasing the number of queuing classes, when weighed against the complexities of the multiple queuing systems which by assumption might increase linearly with the number of queues in the multiple queuing systems. The later version of IEEE 802.1D and p standards specified eight traffic classes (0 to 7) for which mapping could be one to one with regard to traffic classes mappings to queues in the multiple queuing systems **[802.1D&pYr.98-05]**. Here again, the specification is based on heuristic considerations that traffic may benefit from the assumed finer granular allocation of resources when eight traffic queuing classes are supported compared with the coarse granular allocation of resources when only two traffic classes are supported. While the current specification broadens the scope of actions of would be implementers, it does not state that the current specification has a performance advantage over the earlier specification on the subject. Thus an empirical investigation or analytical modelling to determine the optimum, in a range of two to eight traffic queuing classes is worthwhile.

4.1.2 IETF IntServ and DiffServ Service Classes

IntServ Service Classes: In the earlier stages of Integrated Services QoS architecture's design, there were proposals that besides best-effort services, there would be three additional service classes, which were; *Controlled Load, Predictive Service, and Guaranteed QoS* **[RFC 2211] [PREDV-SRVC] [RFC 2212]**. This in effect proposed *four service classes*, which would imply four-priority level, multiple queuing systems. This categorization would find justification in the fact that, within the best-effort service group of applications there are some applications that require some level of timeliness and could be categorized to use the controlled load services. An example is interactive email. Also the real-time group of applications could be sub-classified into real-time tolerant and real-time intolerant applications which could be mapped into predictive service and guaranteed QoS respectively. At some intermediate point in time, the idea of four classes traffic grouping, was short lived by the proposal for *three classes traffic grouping*, and these are best-effort, controlled load and guaranteed QoS services. This was based on the view that the controlled load and the predictive service could be merged together without impacting serious performance problems on the affected applications. The current specification prescribes two service classes, the controlled load and guaranteed QoS services. This means that best-effort and controlled load could be combined into one service class. The justification for reduction in traffic classes grouping from four to two is well-grounded in heuristic considerations. The need for experimental investigation to support the heuristic explanation is well founded on the issue under discussion.

DiffServ Service Classes are, *Expedited Forwarding (EF) PHB [RFC 2598] and Assured Forwarding (AF) PHB Group [RFC 2597]*. AF is a service class with built-in embedded service classes. Beside the two standard service classes, the DiffServ architecture defines the *Class Selector Codepoint*, (see Section 2.3.4) for backward compatibility with IP precedence field. In effect, the service classes in DiffServ architecture ranges from 2 to 8.

4.1.3 Proposals from Organizations on Applications Service Classes

The European Research Agency QoS architecture known as Adaptive Resource Control for QoS Using an IP-Based Layered Architecture (AQUILA) proposed a set of *four traffic classes* beyond the standard best-effort service. The traffic classes are, *premium constant bit rate, premium variable bit rate, premium multimedia and premium mission critical* **[Eng et al.03]**. Thus AQUILA proposed five service classes including best-effort service. This yet again was based on heuristic considerations that was not founded on either analytical or empirical investigation.

Both the Third-Generation Partnership Project (3GPP) for cdma2000 and the Universal Mobile Telecommunications System (UMTS) proposed four QoS classes which are, *Conversational, Streaming, Interactive and Background* service classes **[Che et al.05] [Cho et al.05]**. There is no information that indicates the proposals are based on either theoretical or practical investigation work.

4.1.4 Merging of Motivation Points

The IEEE 802.1D standard specifications through a period of time, increases the number traffic classes from two to eight. That is decomposed from two traffic classes into eight traffic classes. We refer to the decomposition process as *divergent* or *convex process*. This is illustrated in the Figure 4.1(a). On the other hand, IETF IntServ working group specifications through period of time systematically reduces the number of service classes from four to two. The composition process is described as convergent or concave process. This is also illustrated in Figure 4.1(b). The two standard organizations share opposing view on the subject, which could be resolved through either analytical or empirical investigation.

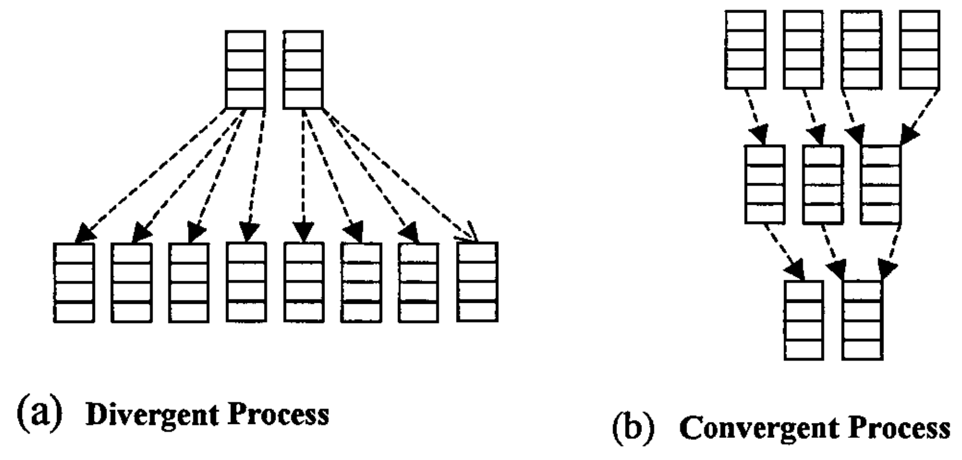


Figure 4.1: Processes of decomposition and composition of traffic queuing classes.
(a) Decomposition Process (b) Composition Process

4.2 Factors that influence the choice of Tools for the Investigation

In every engineering situation, where there are problems to be solved or questions to be answered, the natural approach to the solution is either through analytical modelling, using mathematical tools or computer simulation employing some software packages. Here in this section, we will discuss why we have opted for computer simulation in the investigation work, and also discuss which software package we have utilized. We will also provide insight as to which multiple queuing algorithms we use as queuing disciplines for the investigation.

4.2.1 The Choice of Simulation Modelling instead of Analytical Modelling

The great advancement in processing power of computers coupled with the great advancement in the development of powerful software packages has enabled almost a one-to-one relationship in the representation of real-life situation with computer simulation modelling. Powerful software packages employed in simulation have tremendously expanded the scope of engineering simulation modelling to embrace almost every engineering situation. The accuracy in equivalency between simulation modelling effort and the real-life situation it is meant to represent has been tremendously leveraged. Communication networks and distributed systems typically encompass a wide range of technologies ranging from low-level communications hardware to high-level decision-making software. A successful system modelling effort must represent each of these subsystems and their interactions at a level of detail that is sufficient to obtain valid predictions of performance and behaviour. The complexities involved in such a modelling effort could be appreciated if a communication network is summarily decomposed into its component parts. A communication network will typically consist of the topology, which is a system of nodes interconnected by links. The nodes typically consist of a system of hardware modules, which are controlled, by a system of software modules. Network dynamic objects such as packets would make use of the services provided by these infrastructural resources (i.e. the topology, the links and the nodes—its hardware and software modules). It should be noted that it would be difficult to represent the behaviour and structure of each of the subsystems and their interactions in a single model framework. The flexibility of simulation software packages enables us to partition the modelling effort into a number of modelling domain with defined interface between them. That is breaking down of the big problem into smaller problems, which are now easy to manage. This is the beauty of computer simulation. Another advantage of simulation is reusability. A single network model could be made to answer a number of network questions by creating different scenarios.

On the other hand, analytical modelling using mathematical tools for real-life engineering situations has a lot of limitations especially when modelling large complex systems. For example, it will be difficult to represent the behaviour, structure and interactions of all component objects (both hardware and software) of a communication network with analytical mathematical modelling without loosing great detail. Analytic modelling would mainly be useful in a situation meant to conceptualize the behaviour and performance of a particular generic object and this will entail loss of information on other aspect of the system. For example when using analytical modelling to model communication systems and using queuing systems in other to determine the performance of the telecommunication systems, the details of the topology, the links, other hardware and software in the nodes beside the buffers and servers are not always represented. This is not the case with computer simulation. The impact of the limitation of mathematical analytic modelling can be captured from a statement by Leonard Kleinrock. I quote verbatim, "The mathematical structures we have created in attempting to describe these real situations are merely idealized fictions, and one must not become enamoured with them for their own sake if one is really interested in practical answers" **[Klel,76]**. There are so many assumptions made in mathematical analytic modelling such that, the solutions obtained are mere approximations of the solutions to the real-life system they represent. The fact that a single modelling effort in analytical modelling could not be used to represent other component parts of the system makes it not flexible and not simple to extend it to cover the whole system under modelling effort.

Thus computer simulation modelling has a lot of advantages over its counterpart, the analytic mathematical modelling. Summarily these advantages include better accuracy in representing real-life systems, and better flexibility and re-usability.

4.2.2 Software Packages used for the Simulation

Two software packages were used for the simulation:

- The Network Simulator Version 2 (NS-2), a freeware software package. Released originally by University of California, under the VINT Project.
- The OPNET Modeler a commercial simulation package released by OPNET Technology Inc. USA.

Both were found to be invaluable, flexible, accurate and powerful in modelling communication networks. The commercial simulator (OPNET) was found to have some edge over the freeware (NS-2) simulator. The advantages of OPNET over NS-2 could mainly be found in the hundreds of well-defined, well-organized, powerful communication utilities and functions, which OPNET provides to make communication modelling very easy. OPNET also provides ready-made standard models of many network protocols, network architectures and vendor network devices that one could use as plug and play to model for any specific communication performance and behaviour.

4.2.3 The Queuing Discipline employed for the Investigation

Since we are investigating the optimum number of traffic queuing classes in multiple queuing system, a natural choice of queuing discipline would be either the Class Based Queue (CBQ) with Round Robin (RR) scheduling and its variants or Weighted Fair Queueing (WFQ) and its variants. We have chosen to use CBQ-RR scheduling and its variants for the empirical work. We will now briefly discuss the factors that influence our choice.

4.2.3.1 Reason for chosen CBQ-RR and its Variant for the Investigation

We have seen in Chapter 3 the performance of Class Based Queues with Round Robin (CBQ-RR) scheduling and its variants in providing resource partitioning to classes of application traffic. We need to recall that the variants include; CBQ Weighted Round Robin (CBQ-WRR) scheduling, CBQ Deficit Round Robin (CBQ-DRR) scheduling, and CBQ Weighted Deficit Round Robin (CBQ-WDRR) scheduling. We noted that, the accuracy in providing fairness in terms of allocated resources start with CBQ-RR and increases until CBQ-WDRR, which is the highest in this category of queueing discipline. Since our aim is to determine how many traffic queuing classes (optimum number of queuing classes) that would best satisfy the need of integrated services QoS in the Internet; a natural choice for such an investigation is the CBQ generic queuing discipline. This is because the effect of the number of queues in relation to which class to service next will be more pronounced in view of CBQ *round robin* scheduling discipline. In support of this natural choice, we noted that there has been a simulation result that showed that CBQ might be more advantageous to meet the needs of real-time multimedia applications than WFQ **[Call et al.00].**

4.2.3.2 Reasons Why WFQ Queuing Discipline is not chosen for the Experiment

We have seen in Chapter 3 how Weighted Fair Queueing (WFQ) could perform excellently in terms of fairness in allocation of resources. We noted that, WFQ employed sorted algorithms in queuing and scheduling traffic flows. The very sorted algorithm used by WFQ could mask the effect of ***number of queue*** in the multiple queue system, which is our objective for this experiment. This could be seen from the fact that, whatever number of traffic queuing classes that has been established for service, the sorted algorithm will select packets from any queue class and prioritise its service only on the resource partitioning policy built into the sorting algorithm. This is unlike the round robin scheduling of CBQ where the scheduler as a rule must visit each service class every circle. The effect of the sorting algorithm is such that, the number of queuing classes plays less role in determining which class to service next. The number of established queues will not control the functionality of the sorted algorithm. What would control the function of the sorted algorithm would be the class to which the packet belonged which would be encoded in the packet header Another limitation of the sorted algorithm employed by WFQ concerns its scalability problems in very high-speed network environment. The sorting of the queue will require keeping states of queued packets. Obviously this will cause scalability problems at gigabytes rates in the heart of the Internet. These reasons with the points highlighted in Section 4.2.1 above led us to favour simulation and disfavour the WFQ discipline for the experiment.

4.3 Notation of Parameters of the Experiment

In order to adequately understand experimental actions and its results, the processes must be described quantitatively. In view of this, we will describe in this section the quantities employed in the experimental investigation, and provide a summary of notation for the quantities. The notations that are used to describe the parameters of our systems of multiple queues can be described in terms of the following component processes.

- The *arrival process*: This is a quasi-random process describing how packets arrive into the multiple queuing system.
- The *waiting room*: This refers to the queue capacity, which is the limit to the number of packets that can be accommodated to wait for service, including those currently being served. The queue capacity can be finite or infinite.
- The *service process*: This is a stochastic process describing the length of time the *server* is busy serving a packet.
- The *waiting time*: This is another stochastic process describing the length of time the packet spends in the system. It has two components; the length of time the

packet spends in the queue before it starts to receive service, and the length of time its service takes.

- The *number of queues*: A deterministic process controlled by certain sequence of procedure in the experiment.
- The *number of servers*: Another deterministic process dictated by the conditioning policy of the experiment.
- The *service discipline*: The rule for deciding which packet or packets, within the queuing systems to serve.

Please note that, we deliberately omit the *customer or packet population* in defining quantities employed in our experimental multiple queuing systems, because we see it as being embedded in the arrival process in this case. This will not be the case if we are treating the queuing theory of the systems, since finite or infinite customer population will come into formulation of equations, which could yield analytic solution to the systems.

A useful shorthand notation for specifying five of the above statistical quantities is the *Kendall's notation*, which consists of series of letters and numbers separated into fields by a forward slash. The five field descriptor notation can be represented as *A/B/c/n/p*, where *A* and *B* describe the arrival and the service processes, respectively, *c* gives the number of servers, *n* the waiting room, and *p* the customer population **[Woo93]**. The final two fields are optional and if omitted are assumed by default to be infinite. The letters, which represent the arrival and service processes, can be chosen from a small set of descriptors, which are :

D: This stands for deterministic, which implies constant inter-arrival and service times.

M: Stands for Markovian or memoryless. In continuous time systems, interarrival time or service time will be exponentially distributed. In discrete time systems the interarrival and service time will be geometrically distributed

G: Stands for Generally distributed. Any distribution or combination of distribution is allowed.

4.3.1 Notation for Quantities

Standard quantities employed in the modelling of queue systems have been adopted in the work on empirical investigation under discussion. The definitions for notations of such quantities are found in most standard texts on queue theory. Particularly, reference could be made to the book "Queueing Systems" by Leonard Kleinrock **[Kle11,76]**. The definitions for notation of quantities employed in queue systems are detailed in Chapter 1 and 2 of the textbook. The notations for quantities as related to our empirical investigation are summarily presented in the next section.

4.3.2 Queue Quantities Notation in Summary

The interarrival times for packets to the queues have the following shorthand notation:

t_n = interarrival time between P_n and P_{n-1}

(P_n = nth packet to arrive into the queue facility)

$A_n(t) = P[t_n \leq t]$ (the probability distribution function (PDF) of t_n)

$\alpha_n(t) = \frac{\partial A_n(t)}{\partial (t)}$ (probability density function (pdf) of t_n)

For asymptotic condition,

$t_n \rightarrow \bar{t}$ as $n \rightarrow \infty$, $A_n(t) \rightarrow A(t)$ as $n \rightarrow \infty$, $\alpha_n(t) \rightarrow \alpha(t)$ as $n \rightarrow \infty$.

$\bar{t}_n \rightarrow \bar{t} = \frac{1}{\lambda}$, $\bar{t}_n^2 \rightarrow \bar{t^2}$

λ = average arrival rate

In a similar approach to above, we specify the notation associated with service time x_n, waiting time w_n, total delay s_n, their PDF, and pdf as follows:

x_n = service time for P_n

$B_n(x) = P[x_n \leq x]$ (PDF of x_n); $b_n = \frac{\partial B_n(x)}{\partial (x)}$ (pdf of x_n)

$x_n \rightarrow \bar{X}$ as $n \rightarrow \infty$, $B_n(x) \rightarrow B(x)$ as $n \rightarrow \infty$, $b_n(x) \rightarrow b(x)$ as $n \rightarrow \infty$.

$x_n \rightarrow \bar{X} = \frac{1}{\mu}$, $x_n^2 \rightarrow \bar{X}^2$

μ = average service rate

w_n = waiting time for P_n

$W_n(w) = P[w_n \leq w]$ (PDF of w_n); $\omega_n(w) = \dfrac{\partial W_n(w)}{\partial(w)}$ (pdf of w_n)

$w_n \rightarrow \bar{w}$ as $n \rightarrow \infty$, $W_n(w) \rightarrow W(w)$ as $n \rightarrow \infty$, $\omega_n(w) \rightarrow \omega(w)$ as $n \rightarrow \infty$.

$\bar{w}_n \rightarrow \bar{w} = W, \quad \overline{w_n}^2 \rightarrow \overline{w}^2$

W = average waiting time

s_n = total delay in the system for P_n

$S_n(s) = P[s_n \leq s]$ (PDF of s_n); $\varsigma_n(s) = \dfrac{\partial S_n(s)}{\partial(s)}$ (pdf of s_n)

$s_n \rightarrow \bar{S}$ as $n \rightarrow \infty$, $S_n(s) \rightarrow S(s)$ as $n \rightarrow \infty$, $\varsigma_n(s) \rightarrow \varsigma(s)$ as $n \rightarrow \infty$.

$\bar{s}_n \rightarrow \bar{S} = T, \quad \overline{s_n}^2 \rightarrow \bar{s}^2$

T = average total delay

The quantities, λ (average arrival rate), μ (average service rate), *W* (average waiting time), and *T* (average total delay) were the quantities employed in the simulation processes. λ and μ were the input to the process of the empirical investigation. They were combined and manipulated to produce values for *W* and *T* which constitute output results either in part or whole.

4.4 Classification of Traffic used to drive the Experiment

Considering the taxonomy of applications in a network environment of integrated services, applications can be broadly categorized into dichotomous groups; real-time applications and non real-time applications. Within these two broad categories of applications, further classification will produce a wide range of generic groups with diverse traffic characteristics and QoS needs. A full description of possible network traffic that could be generated together with their characteristics and QoS requirements could be very complex, we will only consider simple traffic classification that have commanded widespread support. The traffic grouping that we will consider and adopt for our experiment is based on the IEEE traffic classification, **[802.1D&pYr.05]**, which will be discussed briefly in the next section.

4.4.1 Traffic Types

The IEEE 802.1D Standard on MAC Bridges specified eight traffic classes, in the range 1 through 8 that could be supported on a Bridge Port. The Standard actually defined seven classes of application traffic leaving one class undefined. We have emulated the IEEE 802.1D standard specification on a number of traffic classes but extended it by adding one traffic class to the specification. The resulting traffic types with their acronyms are defined as follows:

- **Background** (BK); Bulk data transfer and other activities that are permitted in the network which will not pose any limitation to other users.
- **Best-effort** (BE); Traditional Internet traffic.
- **Excellent Effort** (EE); "Premium best-effort", the best-effort type of service that an information services organization would deliver to its most important customers.
- **Controlled Load** (CL); Important business applications subject to some form of "admission control".
- **Audio / Video Streaming** (AVS); Streaming audio or video such as mp3 or video on demand.
- **Video Interactive** (VI); Interactive video applications such as video conferencing whose QoS needs would be characterized by less than 100-millisecond delay.
- **Voice** (VO); Voice traffic applications such as voice over IP (VoIP) whose QoS requirement would be characterized by less than 10-millisecond delay.
- **Network Control** (NC); Traffic critical to network maintenance and control.

The IEEE 802.1D Standard specification on class of traffic is shown in Table 4.1. The standard associated the undefined traffic class to class 2 in the traffic classes range 1 to 8, with Background traffic as class 1 and Network Control as class 8.
We extend the IEEE traffic types by categorizing Audio/Video Streaming as a class of traffic and allocated it to traffic class number 5. Although we consider Voice and Interactive Video as the most *time critical* applications in terms of their temporal reconstruction at the destination, Network Control takes precedence, since it is the network *initial-engine* that drives and coordinate networking functionality. If we assume traffic class number 1 has the least priority and traffic class number 8 has the highest priority, then in our arrangement here, Network Control is given class

number 8, Background traffic is given class number 1 and Audio/Video Streaming is given class number 5. Thus with this extension, Controlled Load, Excellent Effort and Best-effort are shifted 1 class below their usual class number in IEEE 802.1D standard specification. After modification, the resulting traffic types are shown in the Table 4.2.

Table 4.1: IEEE Traffic Types

Traffic Class Number	Traffic Types	Acronym
1	Background	BK
2	Spare	—
3	Best-effort	BE
4	Excellent Effort	EE
5	Controlled Load	CL
6	Video	VI
7	Voice	VO
8	Network Control	NC

Table 4.2: Modified IEEE Traffic Types

Traffic Class Number	Traffic Types	Acronym
1	Background	BK
2	Best-effort	BE
3	Excellent Effort	EE
4	Control Load	CL
5	Audio/Video Streaming	AVS
6	Video Interactive	VI
7	Voice	VO
8	Network Control	NC

4.4.2 Traffic Class Types Mapping into Queues

Table 4.3 is used to illustrate traffic class types mapping into relevant queues. The numbers under the column named 'Number of Queues' indicate the number of queues that may be implemented in an *output interface* of a network node. The number under the column named 'Subqueue Number' gives the convention of naming multiple queues under a buffer. When multiple queues are implemented in an interface, each queue in the set is referred to as *subqueue* and the subqueues, in some cases may be virtual. The subqueues are numbered from 0, and increase as the number of subqueue increases. Thus an output interface with 2 queues will have the two queues named as subqueue 0 and subqueue 1. This is shown in Table 4.3 under the Subqueue Number heading. The third column in Table 4.3 shows the traffic types mapping into relevant queues.

Table 4.3: **Traffic Class Types mapping into relevant Queues**

Number of Queues	Subqueue Number	Traffic Types Mapping
1		[Background, Best-effort, Excellent Effort, Controlled Load, Audio/Video Streaming, Video Interactive, Voice, Network Control]
2	0	[Background, Best-effort, Excellent Effort, Controlled Load]
	1	[Audio/Video Streaming, Video Interactive, Voice, Network Control]
3	0	[Background, Best-effort, Excellent Effort]
	1	[Controlled Load, Audio/ Video Streaming,]
	2	[Video Interactive, Voice, Network Control]
4	0	[Background, Best-effort, Excellent Effort]
	1	[Controlled Load, Audio/Video Streaming]
	2	[Video, Interactive, Voice]
	3	[Network Control]
5	0	[Background, Best-effort]
	1	[Excellent Effort, Controlled Load]
	2	[Audio/Video Streaming]
	3	[Video Interactive, Voice]
	4	[Network control]
6	0	[Background, Best-effort]
	1	[Excellent Effort, Control Load]
	2	[Audio/Video Streaming]
	3	[Video Interactive]
	4	[Voice]
	5	[Network Control]
7	0	[Background, Best-effort]
	1	[Excellent Effort]
	2	[Control Load]
	3	[Audio/Video Streaming]
	4	[Video Interactive]
	5	[Voice]
	6	[Network Control]
8	0	[Background]
	1	[Best-effort]
	2	[Excellent Effort]
	3	[Controlled Load]
	4	[Audio / Video Streaming]
	5	[Video Interactive]
	6	[Voice]
	7	[Network Control]

The IEEE 802.1D standard specified *strict priority* in implementation of the multiple queuing system. Strict priority is widely suspected of its *preventive* *service discipline*. That is the starvation of lower priority traffic by high priority traffic. We believe that, strict priority implementation of multiple queuing system is not suitable in multiservice network environment. Thus for the multiple queue discipline, we propose that, CBQ algorithm, with the variants of round robin (RR) scheduling will be more appropriate. With RR scheduling, the issue of starvation of lower priority traffic by high priority traffic will be absent.

Our traffic types mapping shown in Table 4.3 is almost in-line with the IEEE 802.1D standard specification on mappings of traffic types to priority queues, except for our introduction of Audio/Video Streaming as a class of traffic.

Summary

This chapter has provided the introductory information for the work on experimental investigation to determine the *Optimum Number of Traffic Queuing Classes* (ONTQC) that will best support integrated services in IP Networks. The motivation for the work includes the lack of consensus on the issue of ONTQC.

The notation for input and output quantities to our simulation processes are; λ (average arrival rate), μ (average service rate), W (average waiting time), and T (average total delay). Traffic used in the simulation are classified into eight groups in line with IEEE 802.1D standard.

CHAPTER 5

Simulation Methodology to Determine ONTQC

In this chapter, the procedure for the experimental work, and the actual simulation work carried out on *Optimum Number of Traffic Queuing Classes* (ONTQC) to support integrated services will be presented. It should be recalled that, in Chapter 4, which serves as a precursor to this chapter, the discussion was focused on the simulation input and output quantities. This chapter builds on Chapter 4 in providing a detailed summary of the actual simulation work carried out. A summary of the simulation work and procedures is presented in Section 5.1. This is followed by the detailed simulation methodology in Section 5.2 where we found the concept of Markovian chains very helpful in illustrating the sequence of actions taken. In Section 5.3 we discuss the simulation work, while in Section 5.4 the focus is on the actual simulation work carried out in a typical *Simulation Action Domain* (SAD).

5.1 Procedure in Brief

In setting up a simulation to model the *Optimum Number of Traffic Queuing Classes* (ONTQC) for integrated services, we need to identify and specify all objects and processes that would be employed in the simulation. We also need to specify and define the behaviours of the objects and the way they combine and interact with one another. The logic of the processes that manipulate the objects should be adequately specified and defined.

The objects and their behaviours together with the processes that manipulate them was organised to produce systems of behaviours, which in totality would be equivalent to the characteristic behaviour of the multiple queuing systems we were modelling. The steps involved in these actions will include the following:

- Identifying, specifying and defining static objects associated with the network to be modeled. This will include, specifying both the hardware and software of the communication node, specifying the link or communication channel and their pipeline processes, and defining the topology.

- Identifying, specifying and defining dynamic objects of the communication network. This will include specifying and defining application traffic (packets) and the processes that generate, store, process, send, manipulate, control, and receive the traffic.
- Identifying, specifying, and defining statistics of interest to measure the performance of the system. This will include defining processes to extract statistics and prepare them for display when needed.

The organization of this system follows a highly structural trend, starting from the low level hardware processes to the high level service oriented application software programs. Summarily, the hierarchy in this trend of organization is as follows:

- The processes are combined to form modules, (may be either software stack or hardware module).
- The modules are combined to form nodes
- The nodes are interconnected by links or communication channels to form network topology, (type of topology may be point-to-point, ring, star, tree, etc).

The topology we constructed for the simulation allowed for resource contention among traffic flows that were transmitted through the network. The logic built into the processes of traffic flow mechanism was such that the resolution of resource contention among traffic flows reflect our objective for the simulation modelling. Thus after the appropriate topology had been setup, traffic classes were defined. Each traffic class was parameterised to generate its characteristic flow profile, for which the combined effect for all the traffic classes produced the appropriate traffic intensities suitable for our modelling effort in the network. The simulation actions and processes as briefly highlighted above were combinations of sequences of deterministic and embedded stochastic processes, which will be discussed in more detail in this chapter. The brief discussion here covers the generic topology employed for the investigation, the traffic used and statistics collected.

5.1.1 The Generic Topology

The topology consisted of two point-to-point (P-t-P) star-like LANs (LAN_A and LAN_B) which were connected by a single bottleneck duplex link. As shown in Figure.5.1, each host in each LAN was connected by a P-t-P duplex link to a centre Internet router, which served as gateway to each LAN. Each of the hub-like Internet routers (router-A or router-B, Fig. 5.1) was capable of routing traffic within its own LAN and also capable of directing traffic meant for the other LAN to that LAN's router. For the purpose of this simulation, traffic generated by hosts in one LAN were destined for hosts in the other LAN. The topology is generic for modelling QoS control algorithms and its performances. It is generic in the sense that the bottleneck link together with the two routers to which the ends of the link are terminated could be conditioned in various ways to model various QoS control mechanisms. Such a topology, is found to be widely used for QoS control performance modelling in the literature **[FloJac95]**, **[Cal et al.00]**.

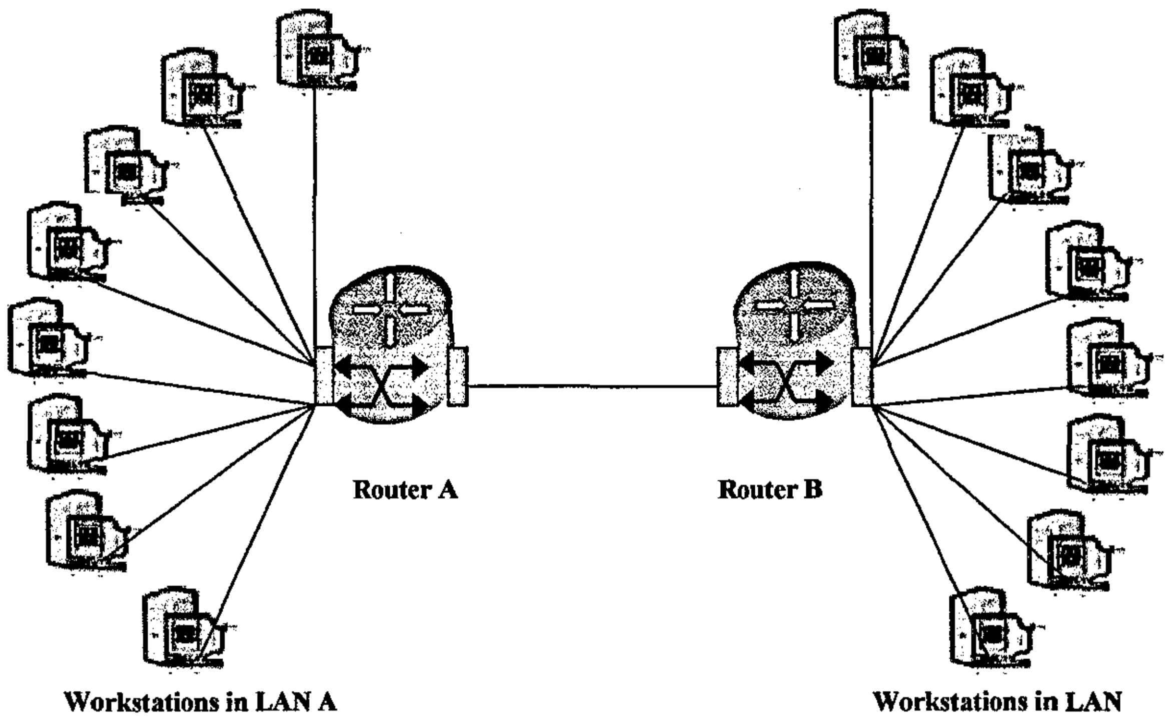


Figure 5.1: Generic Topology Used for Simulation

5.1.2 Application Traffic used for Simulation

We adopted eight traffic classes for the simulation in emulation of IEEE 802.1D standards on number of traffic classes for integrated services. In each simulation run, traffic combination in the network consisted of eight traffic classes, which were configured to generate the appropriate traffic intensity. Traffic intensity in the network varies from one simulation run to another. For each traffic class, we use a wide range of parameters for characterisation of its flow. Traffic characterization parameters were derived from the heterogeneous lists whose elements include packet size, packet interarrival time, packet rate, packet start time, packet stop time, etc. The traffic parameters have various distributions such as, exponential (Poisson distribution), uniform, normal and Pareto distributions. We included in the definition of the processes that generate the traffic, the element of randomness in the values of parameters of the traffic. This instituted stochastic processes in the traffic generation processes of the hosts.

5.1.3 Statistics for the Performance Metric

The main statistics of interest were *end-to-end delay* (sojourn time) and *throughput*. We noted that, other statistics of interest could be derived from the main two statistics of interest (end-to-end delay and throughput). For example, *jitter* could be derived from end-to-end delay, while *packet* losses could be inferred from throughput. Thus we defined or specified processes to extract end-to-end delay and throughput statistics for each of the simulation run.

5.2 Simulation Methodology

The method adopted for the simulation consisted of sequences of simulation actions that had the pattern of a connected action-chain. Each simulation action in the chain had an embedded set of simulation steps and each step also had embedded sub-steps of actions to complete a particular simulation action. In other words the simulation actions consisted of embedded hierarchical simulation actions. Let us denote each main simulation action with a streamline term—*Simulation Action Domain* (SAD). We noted that the whole system of simulation actions assumed a chain-like pattern with similar steps repeated at each SAD, but with each SAD differently parameterised. Itemizing the points highlighted above will give the following format:

1. The action train consisted of sets of simulation actions called *Simulation Action Domains* (SADs)
2. Each SAD was made up of simulation steps numbered from 1 to 8.
3. Each step had a number of sub-steps centred on configuration actions to build the behaviour of simulation model that was intended for our modelling effort.

If we denote SAD_n as the nth action domain (n = 1, 2, 3, -----------) and step_i as the ith step (i = 1, 2, --------, 8), then a particular step in an SAD can be represented by SAD_n; step_i. Let us assume that the topology of the network has been setup and all the underlining processes are working correctly, the sequences of action in SAD_n; step_i can typically be represented as follows:

1. Specify traffic characterization flow parameters for each of the eight traffic classes to be used to load the network. (Values of parameters were fixed for the eight steps in an SAD).
2. Define resource allocation policy for each of the traffic classes.
3. Specify statistic probes and statistics to be collected at the end of the simulation (fixed for all steps in an SAD).
4. Specify the number of queue classes in each of the routers (see Fig. 5.1) and define the appropriate queue discipline to be adopted in serving the queues.
5. Specify simulation attributes for the simulation.
6. Run the simulation.
7. Extract statistics and display the statistics.
8. Analyse statistics and store to combine with other statistics in the SAD.
9. Move to the next step if the current step is less than 8.
10. If step is 8, analyse the combine statistics in the current SAD in order to determine the traffic parameters for the next SAD.

The simulation actions as described above can be said to be quasi-deterministic. The term quasi-deterministic would be relevant when one considers that, some of our simulation actions were deterministic while others depended on outcome of simulation runs, which were unpredictable before the run. The point was that, at the initial stages or current stage, we had a determined set of simulation actions to take, but the next stage of the experiment (in terms of configuration parameters) would be a random action dependent on the outcome of the current experiment. Even if our simulation actions were deterministic all through, the processes they invoked were stochastic. In view of this and for its esoteric beauty, we will like to describe our simulation methodology with classical terms. A number of definitions will enhance the clarity of our descriptions in its classical form.

5.2.1 Probability Systems

The discussion under this section is optional for readers. The inclusion is for background knowledge necessary to understand the description of our simulation methodology, which is presented in relation to Markov chain. Understanding of Markov chain requires knowledge of probability functions. Readers who have basic knowledge of theory of probability functions can omit this section.

The importance of probability theory as a mathematical tool for modelling and analysing communication networks cannot be over-emphansised. We introduce the notion of the triplet of probability systems here, (S, $\mathcal{E}$, P), **[Kle1,76]**. The triple (S, $\mathcal{E}$, P) stands for the probability system, which is defined as follows:

S denote the *sample space*. That is the set of all possible (mutually exclusive exhaustive) outcomes in chanced experiments, such as tossing coins several times. Each possible outcome ρ, ($\rho \in S$) in the set S, is referred to as *sample point.*

$\mathcal{E}$ denote a family of events {A, B, C,......} in which each event is a set of sample points {ρ}

P denote or represent the probability measure of the outcome of chance events defined on the sample space S in terms of real numbers. In other words P is an assignment (mapping) of events defined on S into a set of real value numbers.

5.2.2 Definitions

The definitions presented in this section are optional for readers. The definitions are presented as background knowledge to aid understanding of terminology used in description of our simulation methodology. Reader with basic knowledge of theory of stochastic processes can omit this section.

Definition: A *random variable* X(ρ) is a variable whose value depends upon the outcome of a random experiment. E.g. tossing a coin six times, the number of times a head showed up is a random variable. To clarify the concept, let us represent the outcome of the random experiment with *sample point* ρ ∈ *S*, then to each such outcome ρ we associate a real number X(ρ) which is the value the random variable takes on when the experimental outcome is ρ. Thus the random variable X(ρ) is a function defined on the sample space *S* that assign a real number X to every ρ.

Definition: A *stochastic process* or random process X(t, ρ) is a family of random variables in which each sample point ρ ∈ *S* of each random variable is assigned a time function, which taken together form a family of functions. In other words a stochastic process is a family of random variables X(t, ρ) whose sample points are associated with time functions. For example the atmospheric pressure in a lecture theater as a function of time is a stochastic process.

Stochastic process can be specified and characterized by their *Probability Distribution Function* (PDF) and *probability density function* (pdf). If we denote the PDF of the stochastic process by $F_x(x; t)$, then the pdf is defined by:

$$f_x(x; t) \ {=}D \ \frac{\delta F_x(x; t)}{\delta x} \qquad (5.1)$$

The mean value of the stochastic process is given by:

$$\overline{X}(t) = E[X(t)] = \int x f_x(x; t) \delta x \qquad (5.2)$$

The autocorrelation function of X(t) is given by:

$$\begin{aligned} Rxx(t_1, t_2) &= E[X(t_1)X(t_2)] \\ &= \iint x x f_{xx}(x_1 x_2; t_1 t_2) \delta x_1 \delta x_2 \qquad (5.3) \end{aligned}$$

* =D *To be read as defined by*

Definition: A *stationary stochastic process* is one whose characteristics remain invariant over all time. This implies its PDF Fx (x: t) is invariant to shift in time for all value of its argument. That is given any constant r, the following relation hold:

$$Fx\,(x;\ t + r) \;=\; Fx\,(x,\ t) \qquad (5.4)$$

(*Where the notation* $t + r$ *is defined as a vector* $(t_1 + r,\ t_2 + r,\ t_3 + r, \text{----------} t_n + r)$)

Other characteristics of stationarity include:

1. The mean value must be independent of time: i.e. $\overline{X}(t) = \overline{X}$
2. The autocorrelation function must depend only on time difference $t_1 - t_2$:

   i.e. $Rx\,x\ (t_1\ t_2) = E[X(t_1)\ X(t_2)] = Rx\,x\,(t_1 - t_2)$

If these two conditions hold then the process is said to be *wide-sense stationary*.

Definition: An *ensemble,* denoted by X(t, s) is a collection of time functions that are generated as a result of stochastic processes. In order words it is a family or collection of stochastic processes. Each member of the stochastic processes is called a *sample function.*

Concerning an ensemble, we now define a related term in terms of deterministic process rather than stochastic process.

Definition: A *deterministic ensemble* denoted as X(n; s) is a collection of n (n = 1, 2, ---) sequenced related actions or processes carried out, whose priori probability of selecting the set of actions equals 1 and the outcome of such processes determine a decision.

Definition: *Ergodic process* is a stationary process in which all ensemble averages equal the corresponding time averages. Ergodicity imposes additional condition to stationarity that a single sample function is a representation of the entire processes.

Definition: A set of random variables $\{X_n\}$ form a *Markov chain* if the probability that the next value (*state*) x_{n+1} depends only upon the current value (*state*) x_n and not upon any previous values. That is the way in which entire past history affects the future of the process is completely summarised in the current value (state) of the process. Information on the above definitions can be obtained from **[Car86]** and **[Kre93]**.

The purpose of the definitions we made above will become clear if we adopt classical terms and graphics in the description of our simulation methodology. The definitions will help in suppressing the strangeness that arises from the esoteric effect of using some of the terminology defined above.

Simulation Methodology Continues

To recapture our description, we must recall that our simulation method consisted of sets of simulation actions, each of which was called SAD. Each SAD consisted of eight simulation steps numbered from 1 to 8. The traffic characterisation parameters and traffic intensities for all steps in each SAD were invariant or *ergodic*. While traffic characterisation parameter and traffic intensity differs from one SAD to another, the processes of statistic specification, definition and collection methods for all SADs were ergodic. The main difference in actions taken as we move from one step to another in a SAD was that, the number of queues implemented at each step in the routers A and B (Figure 5.1) was at parity with the step. This means, in step_1, one queue was implemented, in step_2, two queues was implemented, --------------, in step_8, eight queues was implemented.

5.2.3 Use of Markovian Chain to Describe Simulation Methodology

The author leans on earlier work in employing a Markov chain to describe the simulation methodology. Leonard Kleinrock used a Markov chain to describe the action of a hippie who hitchhiked from city to city, **[Klel,76 p26]**. A.J. Bayes used a Markov chain to analyse sampling of the output from a simulation process **[Bay72]**.

A close analysis of our simulation methodology as described above reveals that it could be modelled (descriptively and graphically) with a *Markovian chain*. Each SAD constitutes each *state* of the Markovian chain. The action we took in SAD_n (n = 1, 2, 3, ----) depends only on the immediate previous action in SAD_{n-1} and not on any previous actions in all other previous SAD. This is the characteristic of the Markov chain. Another definition will make the point clearer.

Definition: The sequence of random actions A_1, A_2, ---------A_n, form a discrete-time *Markov chain* if for all n (n = 1, 2, ---------) and all possible values associated with the random actions, we have the *posteriori probability* (for $i_1 < i_2$ ------ $i_{n-1} < i_n$) which state that;

$$P[A_n = j \mid A_1 = i_1,\ A_2 = i_2, \text{-----},\ A_{n-1} = i_{n-1}] = P[A_n = j \mid A_{n-1} = i_{n-1}] \qquad (5.5)$$

In relating Markov chain definition to our simulation methodology, the definition simply states that, the action to be taken at the next SAD, i.e. SAD_{n+1} depends only upon the action taken at the current SAD, i.e. SAD_n and not on all previous actions taken on all previous SADs. In this sense the memory of our random simulation action or Markov chain goes back to the most recent SAD. The conditional probability relation in Eqn. 5.5 is merely emphasizing that, all the actions in SAD_1, -----------, SAD_{n-1} are summarized

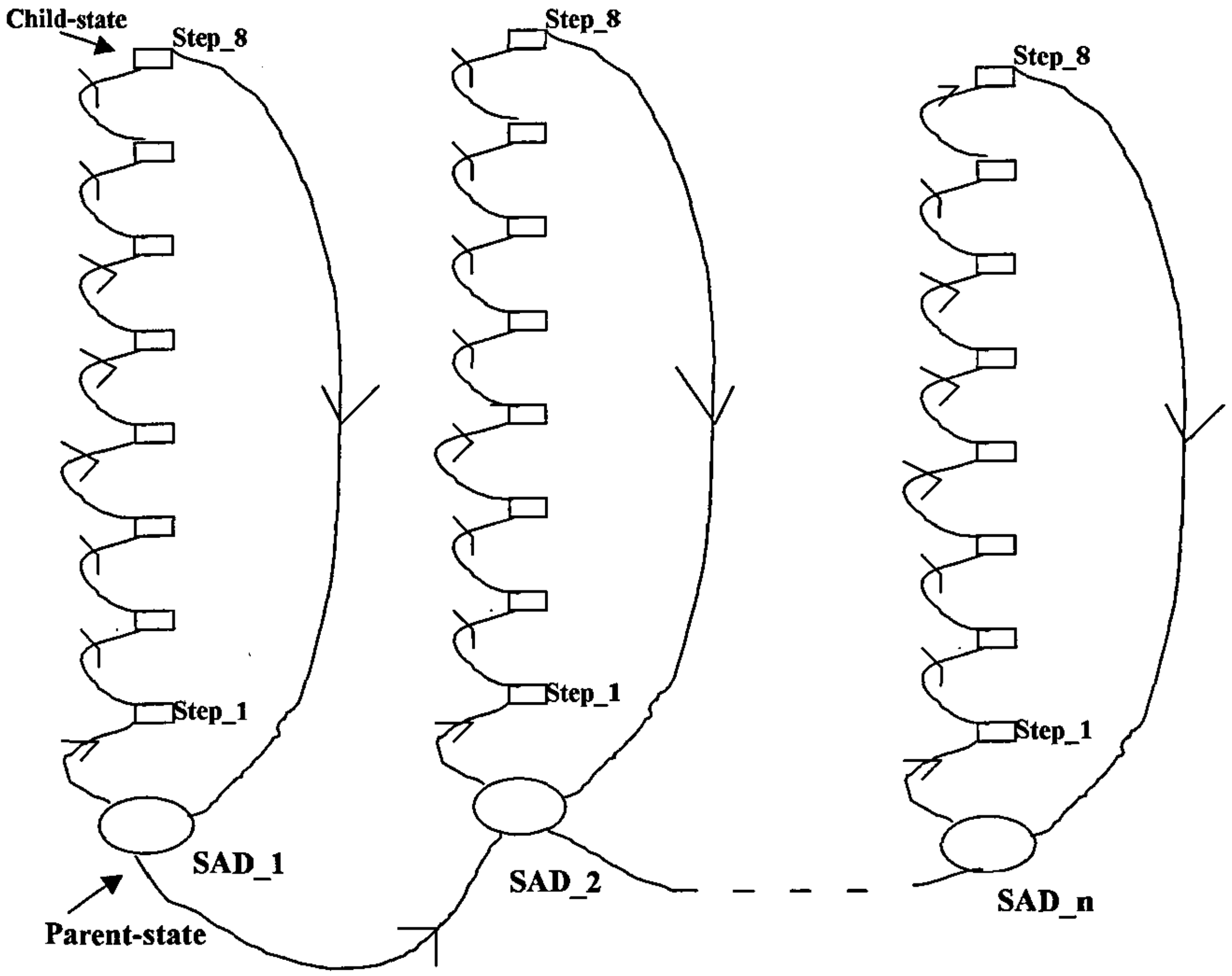


Figure 5.2: Abstraction of Simulation Methodology with Markovian Chain

in SAD_{n-1} and that it is the action in SAD_{n-1} that will affect the action in SAD_n. Another way of conveying the same message is to say that the way the entire past history will affect future action is completely summarized in the current action. This is the key characteristic of the Markov chain. The main difference between pure Markov chain and the Markovian chain that could effectively model our simulation methodology is that our type of Markovian chain has embedded multistates within each state. The unique Markovian chain is shown in Figure. 5.2.

Each main *state* (i.e. SAD) has embedded state transition which loop back to the main state after the last embedded state. Let us denote main state as *parent-state* and embedded state as *child- state*. Each parent-state (SAD) consisted of eight embedded child-states (i.e. steps numbered from 1 to 8) and each child-state (step) consisted of an embedded set of simulation actions. Using the derived terminology for this description, we say each parent-state (SAD) is made-up of a two-level embedded (or stack of) *deterministic ensemble*. The number of child-states that could exist was bounded, fixed and invariant. Bounded in the sense that a child state could only take value i if $1 \leq i \leq 8$, fixed because there would always be 8 child states in each parent-state, and invariant in the sense that child states were the same from one parent-state to another. Transition from one child state to another were constrained. The constraint is in the fact that transition between the child- states were one-way. Transition started from the parent state and moved to child-state 1 (step 1) and from there moved to child-state 2 and continued progressively until child- state 8 was reached. At the end of child-state 8, the transition moved directly back to parent-state. The transition back to the parent state from the highest number child state was necessary because the next parameter for simulation at the next parent-state would be fashioned on the simulation parameter of the current parent-state. Thus we say embedded state transitions starts from the parent-state and form a unique loop unto the parent-state. And that inter child-state transitions were deterministic, monotonically increasing and non reversible. The foregoing explanation simply means that, within any simulation action domain (SAD), simulation actions will move from step 1 to step 2 and from step 2 to step 3, etc until we get to step 8 where at its end we move back to the parent-state. Transition will not be from step 8 to step 7, or step 7 to step 6, etc. By this we mean, a simulation action that has been carried out with certain parameter will not be repeated. Simulation actions in each child-state were *deterministically ergodic* except for queue discipline configuration parameter in each child-state.

The modes of actions in all parent-states were deterministically ergodic. That is the patterns of actions were uniform from one parent-state to another. Traffic characterisation parameters used in parent-states were at variance from one state to another. The values of traffic parameters that would be used in the next parent-state would depend on the values used in current parent-state, but the values would be different for the two parent-states. Transitions from one parent state to another were also one-way, increasing monotonically as the parent state number increased. The parent state transitions were none reversible and non-periodic. This implied that transitions took place from SAD_0 to SAD_1 and from SAD_1 to SAD_2 progressively in that order. Not from SAD_2 to SAD_1 or from SAD_1 to SAD_0. In reality this meant we did not repeat the exact same traffic parameter for two SADs or repeat an SAD that had been concluded.

5.3 Simulation Work

The simulation work will be described in terms of SADs. The sets of all SADs were partitioned into three groups. We name the groups; simulation segment_one, simulation segment_two and simulation segment_three. Brief description of each now follows.

5.3.1 Simulation Segment_one

The simulation work in this segment consisted of a large number of SADs. In each SAD we employed the User Datagram Protocol (UDP) as transport protocol for inelastic (real-time) application traffic, while we used the Transmission Control Protocol (TCP) as transport protocol for elastic (non real-time) application traffic. The intention of using Internet protocols was to increase the equivalency of our simulation modelling effort to the real-life communication mechanisms of the Internet. The use of Internet protocols helped us to measure the effect of TCP traffic control mechanism on the simulation experiment.

Under this segment, we ran several simulations under the headings of SADs, using various values of traffic characterisation parameters. The sets of all simulation actions in each SAD made use of a fixed set of traffic input parameters that were constant for all simulation within the SAD. Other simulation work description relating to this segment will follow in sections under *simulation input, queues decomposition algorithm* and *queuing discipline*.

5.3.2 Simulation Segment _two

The same processes that were followed in segment_one were repeated in this segment. The only difference was that TCP was not used for any of the traffic, all the traffic made use of UDP as the transport protocol.

As in segment_one, we ran several simulations with several traffic parameters maintaining the same procedure as in segment_one. More detail will follow in the related sections.

5.3.3 Simulation Segment_three

The same procedure that was adopted in segment_two was repeated here, but here we did not use any Internet protocol for any of the traffic. We neither used UDP or TCP as transport protocol for any of the traffic. We simply generated traffic and used the traffic packet sizes to identify which class they belong to. The intention here was to find out if applications performance in an environment of multiple queues system were independent of the Internet protocol or any protocol whatsoever. The simulations in this segment help to make the situation clear.

As in other segments, the simulation in this segment consisted of several simulation runs with various traffic parameters. Each simulation was constituted to answer specific questions of our simulation modelling effort. More details follow in subsequent sections.

5.3.4 Simulation Input

The three simulation segments highlighted above consisted of the same number of SADs. Corresponding SADs in each segment used the same traffic parameters and had the same processes. As stated in previous sections, each SAD had 8 simulation steps and each step used 8 traffic classes, and the set of traffic parameters used in each of the steps within a SAD were constant. This implied that, in step_1 we used one queue with 8 traffic classes, in step_2 we used two queues with 8 traffic classes, etc. Traffic characterisation parameter specifications include:

1. Specification of packet sizes.
2. Specification of interarrival time or packet generation rate.
3. Specification of distribution for packet sizes and interarrival times.
4. Start time and stop time of flow sessions.

The parameter of one traffic class was different from the parameter of another traffic class, but the parameters used for one traffic class in a step (e.g. step_1) would be the same as would be used in another step (e.g. step_2), etc. This means, for each traffic class, parameters remain constant as we move from one step to another within a SAD. Each traffic class used different parameters as we move from one SAD to another.

5.3.5 Decomposition of Traffic Queue Classes

In the three simulation segments highlighted above, eight traffic classes were used. The eight traffic classes will form a single queue in step_1 of each SAD, employing a FIFO queuing discipline. We needed to decompose the single queue that consisted of the eight traffic classes into smaller queues in a sequence of steps that would reflect our set objectives. We must develop a particular algorithm for splitting the queues in a manner that would enable us to achieve our goal. We have developed an algorithm, which we found useful in this case. The algorithm is referred to as *fission of rightmost branch or group block first* (FRBF). The algorithm is applicable in a situation where a job to be done or a problem to be solved consists of many component parts. And there is the need to successively split the job or problem into two halves for easier actions or solutions, and priority attached to one-half in the splitting process. FRBF could be applicable in the work presented in **[Coh et al.82]**.

Consider a job or problem to consist of *elements*, which are all packaged into a block (set) known as a *root-block*, then applying the algorithm (*fission of the rightmost branch or block first*) will produce the decomposition format shown in Figure. 5.3.

The algorithm is as follows:

1. Start from the root block, re-arrange the elements into two sets, and move higher priority elements into one set and lower priority elements into the other set.
2. Split the root-block into two-halves to form two child-blocks, one block with higher priority elements and the other block with lower priority elements.
3. Move the higher priority block to the right and the lower priority block to the left as branches under the root-block in the hierarchical tree.
4. Split the higher priority block in the right into two-halves again, in such a way that the priority of the elements placed in one of the two resulting child blocks is higher than the other. The left block (or blocks) will remain constant (i.e. not split) until all the right blocks are each reduced to one element.
5. Move the higher priority block in the last split to the rightmost side in the current hierarchical level of the tree branch and split the block into two halves again, reflecting priority.
6. Arrange the blocks in each hierarchical level of the tree in such a way that priority of the blocks start from the lowest in the left side of the tree and increases towards the right with highest priority at the rightmost side. Repeat the rightmost block splitting and arranging until there is a single element in the lowest block in the rightmost branch of the hierarchical tree.
7. Move inward towards the left to the next block that contain more than one element and continue the rightmost splitting of the block and child-blocks until the rightmost and lowest block in the current branch that is acted on, contains only one element.
8. Repeat the steps 4 to 7 until the last block in each last branch of the hierarchical tree has only one element.

The algorithm was applied in our experiment on queue decomposition into subqueues. This is shown in Figure. 5.3. After the initial splitting of the root queue into real-time traffic (priority) and non real-time traffic (less priority) queues, the subsequent splitting processes concentrate on splitting of priority queues until the highest priority queue contains only one class of traffic. After that, the splitting process moves to the next lower priority queue that has more than one-traffic class and repeats the splitting pattern. Figure. 5.3 represents typical levels of activity in an SAD. The number of blocks in each level represents the number of queues that was worked upon on that level. At each hierarchical level, we run the simulation using eight traffic classes. This was the case in each of the three simulation segments highlighted above.

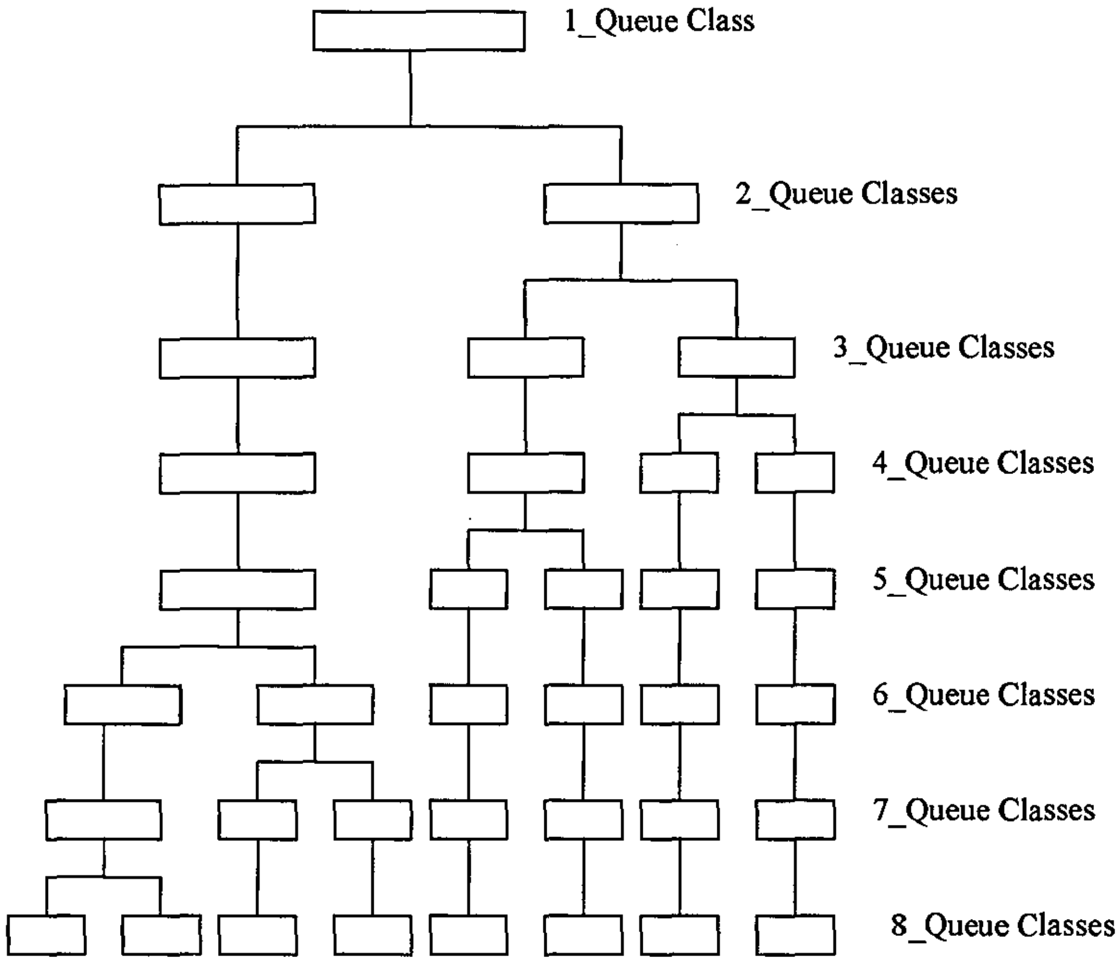


Figure 5.3: Decomposition of queues using algorithm known as *Fission of Rightmost Block First.*

5.3.6 Queue Disciplines Employed for Simulation

In each SAD of each of the three simulation segments highlighted above, we employed a FIFO queue discipline for step_1 in router A and B (see Figure. 5.1). Please recall that step_1 means, simulation with one queue for all the eight traffic classes. When we move to step_2 (employing two queues) and above, we employed a mixture of three queue disciplines, which are variants of the Class Based Queue (CBQ) algorithm. The variant of CBQ discipline used are; the native *round robin* (RR) scheduling, the *weighted round robin* (WRR) scheduling and the *weighted deficit round robin* (WDRR) scheduling. The reason we used the mixture of these CBQ disciplines will be highlighted next.

5.3.6.1 Use of Round Robin (RR) Scheduling

Details of the mechanism of the RR scheduling will be found in Chapter 3, Section 3.7.4. The reason we used RR scheduling was to observe the pure effect of *number of queues* without the modulating effect of preferences such as weight which could shift the performance of application traffic towards the weight of resources allocated to them rather than the pure number of queues. With RR scheduling there is no weight, the queues are visited on a round robin basis. We envisaged that effect of queue numbers would be pronounced in such a situation. The results we obtained will be discussed in Chapter 6, which deals with results.

5.3.6.2 Use of Weighted Round Robin (WRR) Scheduling

The mechanism of WRR can be found in Chapter 3, Section 3.7.5. We made use of WRR scheduling to determine the effect of differential or discriminatory allocation of resources to application traffic on the performance of the application traffic. We seek to find out the extent to which differential allocation of resources affects optimum number of queue classes. Our finding will be discussed in Chapter 6.

5.3.6.3 Use of Weighted Deficit Round Robin (WDRR) Scheduling

The WDRR was discussed in Chapter 3, Section 3.7.6. We observed that the WDRR scheduling was accurate in delivering the share of resources allocated to each application traffic. We used WDRR to determine the extent to which fairness in resource allocation will influence the optimum number of queues. We will discuss our findings in Chapter 6.

5.4 Simulation Work in a Typical Simulation Action Domain (SAD)

We now focus on the simulation work in each step of a typical SAD. We will now discuss each step of a typical SAD in more detail.

5.4.1 Step_1 of a Typical SAD

All the eight traffic classes were concentrated on a single queue in each of router A and B of Figure. 5.1. The layout of the queue mechanism is shown in Figure. 5.4. The queue employs the FIFO queuing discipline. We simulated with both infinite queue capacity and finite queue capacity to determine effect of queue length on *delay* and *throughput*. We experimented on various values of combinations of traffic intensities and capacities of the bottleneck link. Statistics were specified before simulation runs, and collected after each run.

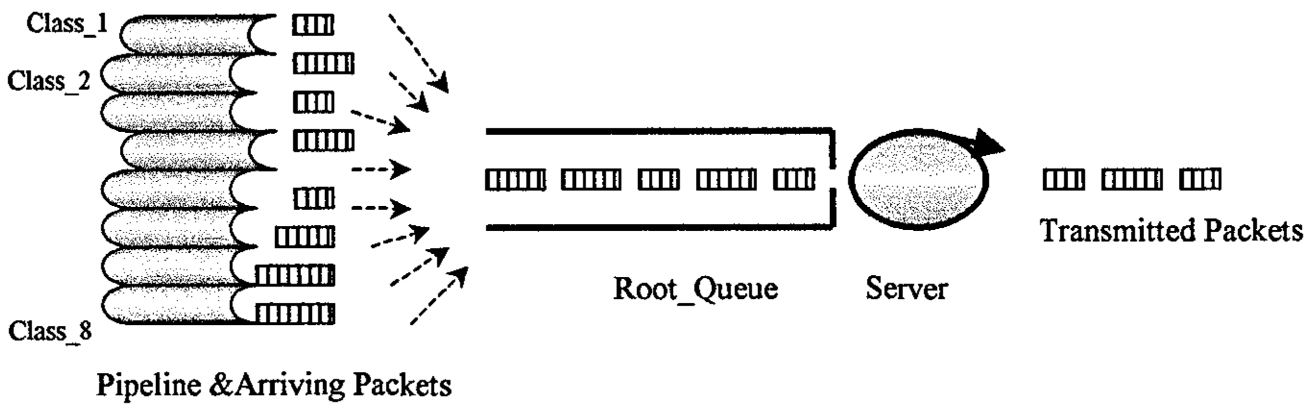


Figure 5.4: The Root_Queue (single queue) with FIFO Queue Discipline

Let the 8 traffic classes be Network Control, Voice, Interactive Video, Audio & Video, Control Load, Excellent Effort, Best-effort, Background or Bulk traffic. The shorthand notation for each of the traffic classes follows:

Network Control	→	NC
Voice	→	VO
Interactive Video	→	VI

Audio & Video	→	AV
Control Load	→	CL
Excellent Effort	→	EE
Best-effort	→	BE
Background Traffic	→	BT

The single queue, which is denoted as the *Root_Queue*, is represented as a *set* with *8 different elements* and given as;

Root_Queue = {NC, VO, VI, AV, CL, EE, BE, BT}

To support the notion we are about to develop, we make the following assumption:

Assumption: We assume that, the elements are arranged in the queue in such a way that no same element takes two positions that are directly next to each other. In other words queue positions are taken alternately by packets of different traffic classes.

The different ways in which the packets of the different traffic classes can be arranged in the queue according to permutation and combination law equals 8! (8 factorial).

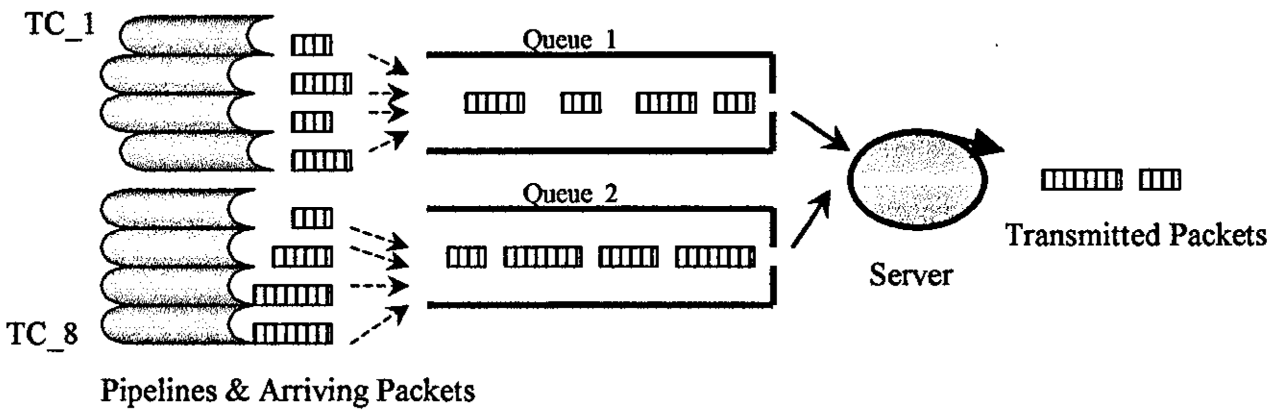
We now consider the state of the queue at the smallest possible time t_0 when all the traffic classes are present in the queue. According to our assumption, the *priori probability* that any one of the traffic classes will be selected for service by the server at time $t_0 = 1/8 = 0.125$. We call this *Expedited Factor*. Low expedited factor becomes a serious limitation for time sensitive application such as Voice in a low capacity link. We now develop a new notion for making similar assessments for timely service. We introduce the notion of *queue quantum molecular length*, which is defined as the number of different traffic classes or packets present in a queue at time t_0. In our case, the Root_Queue has a quantum molecular length of 8. According to the permutation and combination laws, the position any of the 8 traffic classes can take ranges from 1 to 8 at time t_0. In this case, position 1 is nearest to the server while position 8 is the least near to the server. We also introduce the new notion of *proximity number of a packet to server*. This is defined as the number of packets to be served before the *packet* under consideration is served. If the packet at the head of the queue takes position 0 and the *packet* position is *n*, then its proximity number will simply be *n*.

This led us to introduce a related notion of *limiting proximity number* of a packet to server. And it is defined as the worst case scenario of the number of packets to be served before the *packet* when the queue size equals its quantum molecular length at time t_0. This is simply saying the *limiting proximity number* of a packet inside a queue equals its worst case scenario *proximity number,* when the queue size equals the queue's quantum molecular length. If we denote the *limiting proximity number* of a packet inside a queue by *LP* and the *quantum molecular length* of the queue by *ML,* then a packet $LP = ML - 1$. $LP = 0$ gives the best value for occupants of a queue. Higher values of *LP* for queue occupants indicate disadvantages, which increases as *LP* values increases. Thus for the Root_Queue, a time sensitive application such as Voice will have its $LP = 7$ and its theoretical Expedited Factor = 0.125.

5.4.2 Step_2 of a Typical SAD

In the simulation under step_2 of a typical SAD, the eight traffic classes were divided into two groups on the criterion of real-time and none real-time traffic. Each group had four traffic classes. Two queues were implemented in each of the two routers, and each queue served four traffic classes. One of the two queues served real-time traffic while the other served non real-time traffic. The layout of the queue mechanism is shown in Figure 5.5. We used the same traffic parameters as used in step_1 for each of the traffic classes. The values for capacity of the bottleneck link remained the same as used in step_1. Processes of statistics collection and implementation remained the same as in step_1.

We noted that the theoretical Expedited Factor for traffic in each of the queues has improved and the Limiting Proximity Number has also reduced (an advantage over step_1). Thus for step_2 queues, traffic has theoretical Expedited Factor = 0.25 and Limiting Proximity Number = 3.



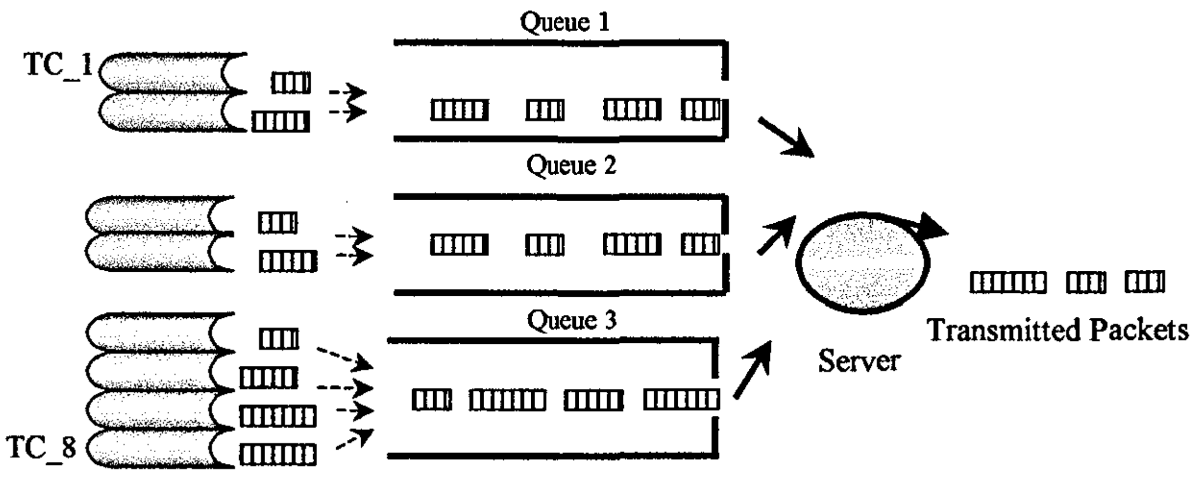
TC = Traffic Class

Figure 5.5: Format of the two Queues Implementation. Queue_1 served real-time traffic (i.e. VO, VI, NC & AV), while Queue_2 served non real-time traffic (i.e. CL, EE, BE & BT).

5.4.3 Step_3 of a Typical SAD

In step_3, we applied the algorithm, *fission of rightmost block first* (see Section 5.3.5). Thus we kept the state of the elastic application queue constant and split the inelastic application queue into two halves. We then have three queues in which Queue_1 had two traffic classes (NC & VO), Queue_2 had two traffic classes (VI &AV) and Queue_3 had four traffic classes (CL, EE, BE & BT). Queue_3 was the same as Queue_2 in step_2. Every other action remained the same as in step_2. All traffic parameters remained the same. The same statistics were specified as did in step_1 and step_2. Figure 5.6 serves as illustration of the format of the three queues.

With the three-queue implementation, Expedited Factor for Queue_1 and Queue_2 occupants increased to 0.5 and their Limiting Proximity Number reduced to 1. Showing theoretical performance improvement. Queue_3 is the same as Queue_2 in step_2, thus Queue_3's Expedited Factor and Limiting Proximity Number remain the same as in step_2. That is Queue_3 occupants Expedited Factor = 0.25 and their Limiting Proximity Number = 3.



Pipelines & Arriving Packets

TC = Traffic Class

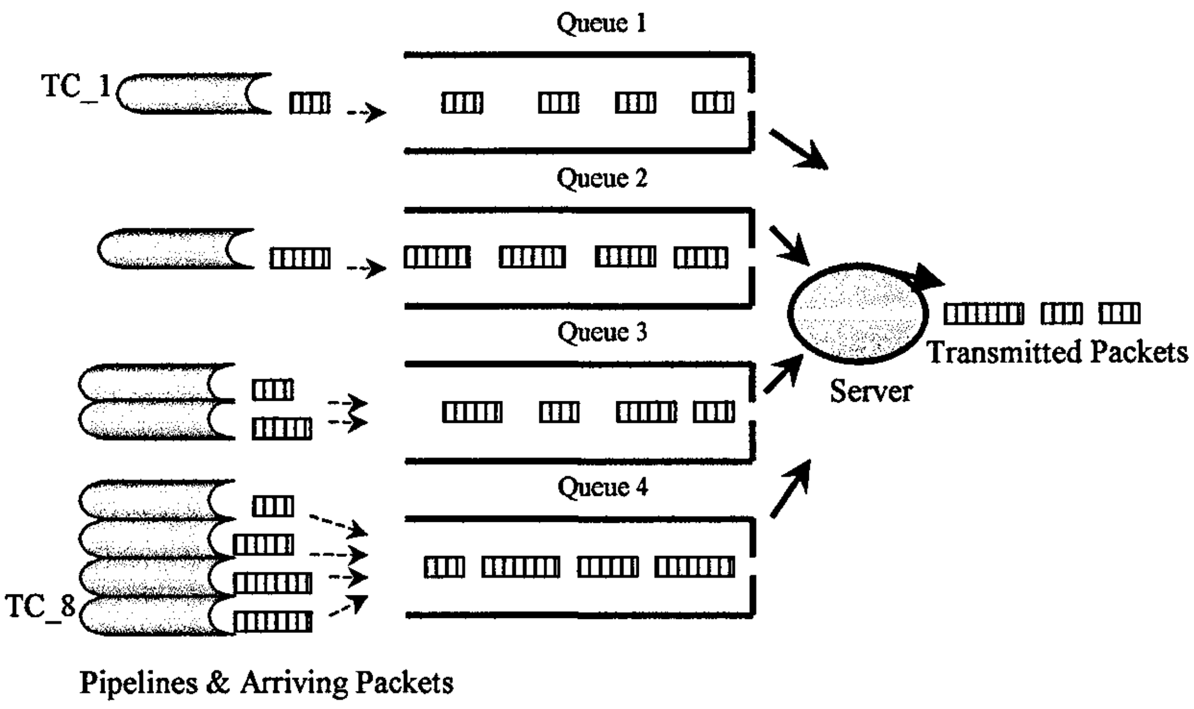
Figure 5.6: Three Queues Implementation. Queue_1 served two classes of traffic (NC & VO), Queue_2 served two classes of traffic (VI & AV), Queue_3 served four classes of traffic (CL, EE, BE & BT).

5.4.4 Step_4 of a Typical SAD

In step_4, we also applied the *fission of the rightmost block first* algorithm, and we got four queues to manage. We split Queue_1 of step_3 into two-halves, and kept the remaining two queues in step_3 the same, resulting in four queues in step_4. Then in step_4, Queue_1 had one class of traffic (NC), Queue_2 also had one class of traffic (VO), Queue_3 had two classes of traffic (VI & AV) and Queue_4 had four classes of traffic (CL, EE, BE & BT). All traffic parameters remained the same as in all previous steps, and so also all statistics. As in step_2 and Step_3, we also experimented with various values of the combination of traffic intensities in relation to capacities of the bottleneck link. Figure 5.7 provides illustrations of the four-queue implementation.

With the four-queue implementation, Queue_1 served only one class of traffic (NC). Queue_2 also served only one class of traffic (VO). The Expedited Factor for occupants of Queue_1 and Queue_2 equals 1 and the Limiting Proximity Number equals 0. These values are the best that could be obtained. Queue_3 was the same as

Queue_2 in step_3, and Queue_4 was the same as Queue_3 in step_3. Thus Queue_3 served two classes of traffic (VI & AV), and Queue_4 served four classes of traffic (CL, EE, BE & BT). The Expedited Factor for occupants of Queue_3 = 0.5 and their Limiting Proximity Number = 1. The Expedited Factor for occupants of Queue_4 = 0.25 and their Limiting Proximity Number = 3. These values show that the traffic in Queue_4 theoretically do not have performance improvement as we change from three-queue implementation to four-queue implementation.

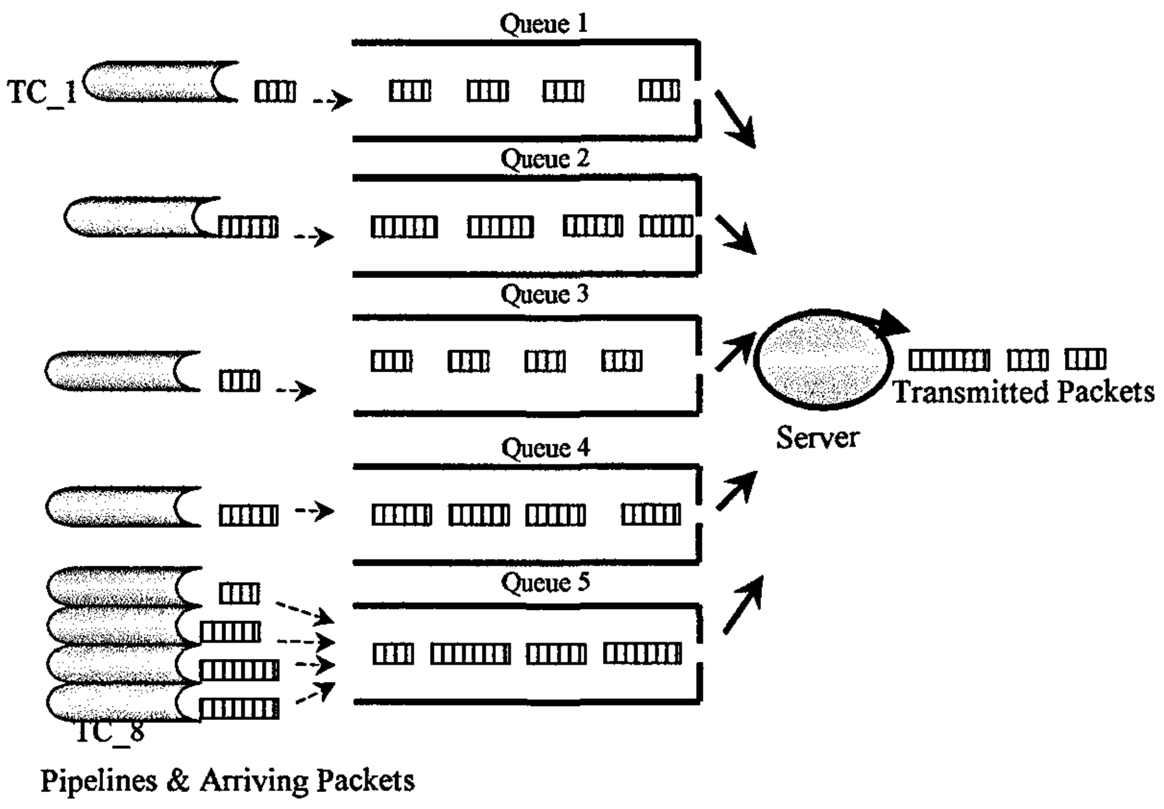


TC = Traffic Class

Figure 5.7: Four Queues Implementation. Queue_1 served one class of traffic (NC), Queue_2 served one class of traffic (VO), Queue_3 served two classes of traffic (VI & AV) and Queue_4 served four classes of traffic (CL, EE,BE & BT).

5.4.5 Step_5 of a Typical SAD

In applying the *fission of the rightmost block first* (FRBF) algorithm in this step_5, we noted that, each of queues 1 and 2 in step_4 had only one traffic class to serve. Thus the application of the FRBF algorithm was not applied to them. The algorithm compelled us to focus on queue 3 of step_4. The queue was split into two, which gave us five queues to manage in step_5. This is illustrated with Figure 5.8.



TC = Traffic Class

Figure 5.8: Five Queues Implementation. Queue_1 served one class of traffic (NC), Queue_2 served one class of traffic (VO), Queue_3 served one class of traffic (VI), Queue_4 served one class of traffic (AV) and Queue_5 served four classes of traffic (CL, EE, BE & BT).

The arrangement then in step_5 was: Queue_1 had one class of traffic (NC) to serve, Queue_2 had one class of traffic (VO) to serve, Queue_3 had one class of traffic (VI) to serve, Queue_4 also had one class of traffic (AV) to serve, and Queue_5 had four classes of traffic (CL, EE, BE & BT) to serve. All traffic parameters remained the same as in previous steps, and also the types of statistics collected were the same as in previous steps. Queue disciplines implemented were the same as in step_4. The Expedited Factor for Queue_1, Queue_2, Queue_3, and Queue_4 equals 1 and their Limiting Proximity Number equals 0. In this case, the theoretical service condition for

Voice and Interactive Video remain the same as in step_4 (four-queue implementation). While the theoretical service conditions of Network Control and Audio/Video have improved compared to step_4. Queue_5 was the same as queue_4 in step_4. Thus the theoretical service conditions of occupants of Queue_5 remain the same as in step_4. Their Expedited Factor remain as 0.25 and their Limiting Proximity Number remain as 3.

5.4.6 Step_6 of a Typical SAD

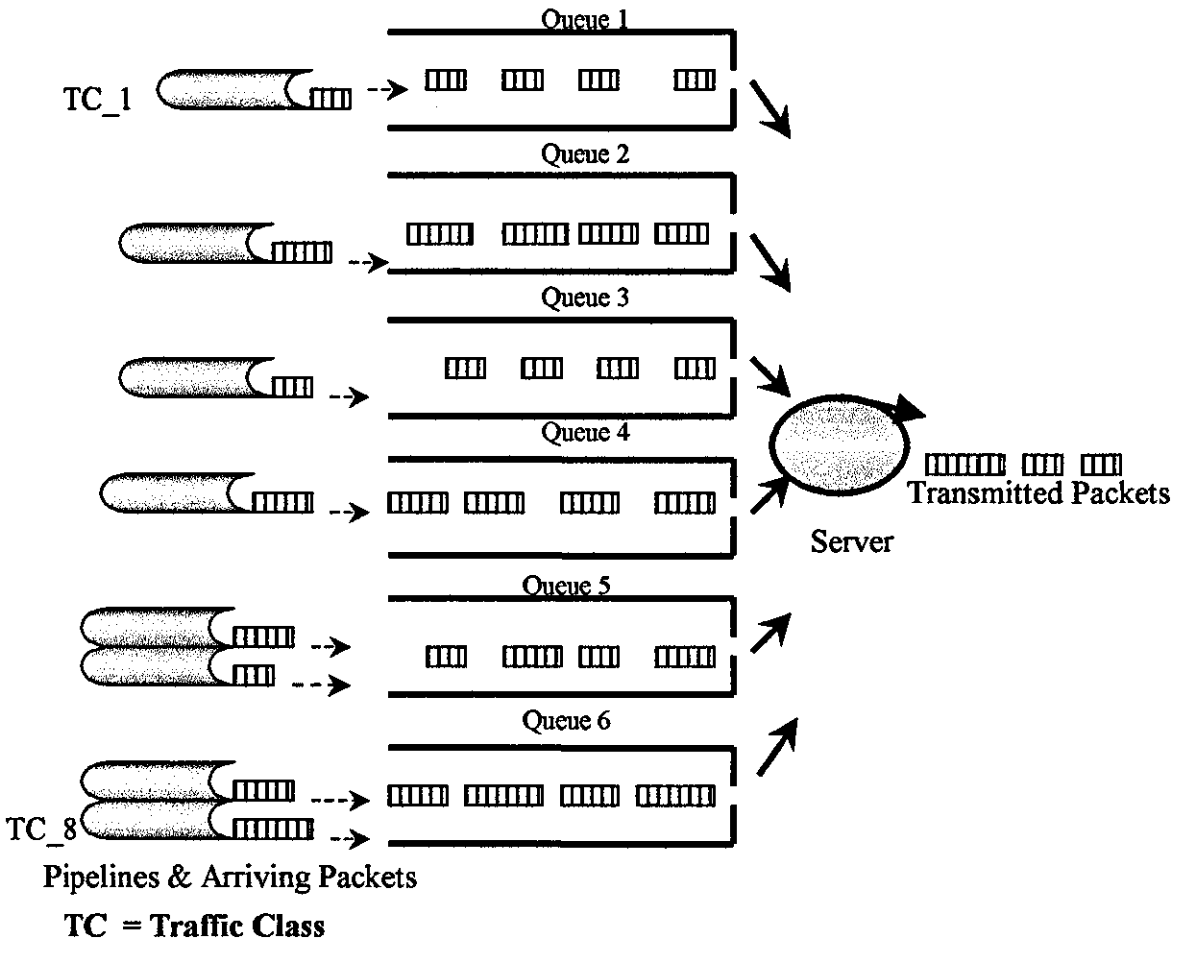


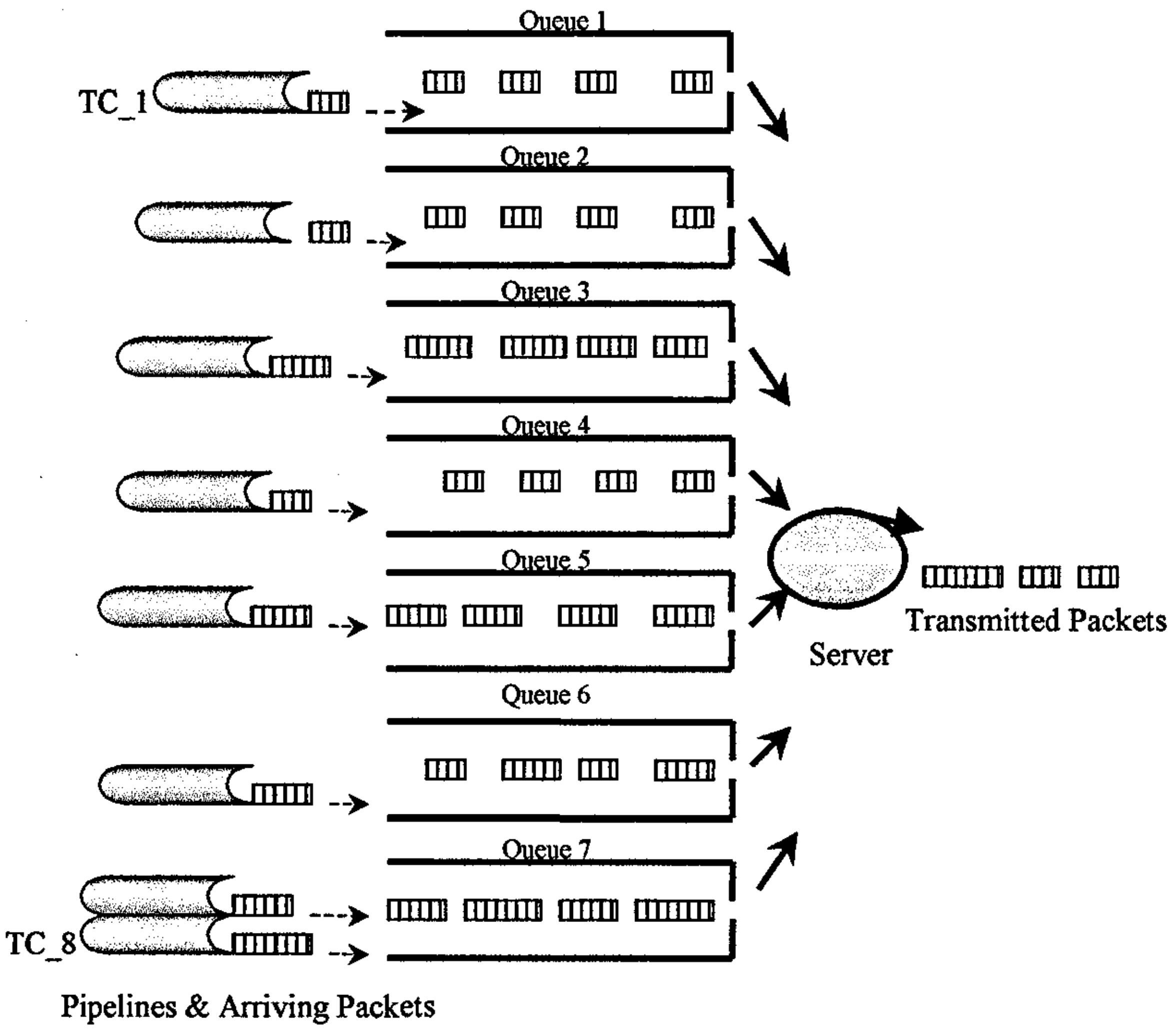
Figure 5.9: Six-Queue Implementation. Queue_1, Queue_2, Queue_3, and Queue_4, each served one class of traffic, which were NC VO, VI, & AV respectively. While Queue_5 served two classes of traffic (CL, & EE), and Queue_6 also served two classes of traffic (BE & BT).

In considering the processes of action in step_6, we noted that, in step_5, queue 1, 2, 3 and 4 respectively had one class of traffic to serve. Thus in applying the FRBF algorithm, these queues were skipped. This brought us to queue_5 of step_5, which had four classes of traffic to serve. We then split it into two-halves. Thus in step_6 we had a six-queue implementation in which Queue_1, Queue_2, Queue_3 and Queue_4 had one class of traffic to serve respectively. Queue_5 and Queue_6 had two classes of traffic to serve respectively. This is illustrated with Figure 5.9. All traffic parameters and types of statistics collected were the same as in previous steps. The Expedited Factor for occupants of Queue_1, Queue_2, Queue_3 and Queue_4 remain the same as in step_5 and is equal to 1 and their Limiting Proximity Number equals 0. The Expedited Factor for occupants of Queue_5 and Queue_6 equals 0.5 and their Limiting Proximity Number equals 1. These values show improvement for the occupants of the two queues compare with their values in step_5.

5.4.7 Step_7 of a Typical SAD

In applying the FRBF algorithm, we noted that in step_6, queue 1, 2, 3 and 4 respectively had only one class of traffic to serve. These queues were therefore skipped as we applied the FRBF algorithm. The next queue to split according to the algorithm was queue 5 of step_6, and the queue was split into two queues, each having one class of traffic to serve. This resulted into seven-queue implementation in this step. The queue arrangement is illustrated with Figure 5.10.

All traffic parameters and types of statistics collected were the same as in previous sections. Queue_1, Queue_2, Queue_3, Queue_4, Queue_5 and Queue_6 respectively had one class of traffic to serve. Their occupant's Expedited Factor equals 1 and the occupant's Limiting Proximity Number equals 0. With these values, Queue_5 and Queue_6 occupants theoretically had improvement in their service conditions compared with their service condition in step_6. Queue_7 had two classes of traffic to serve and the occupants of the queue Expedited Factor equals 0.5 and their Limiting Proximity Number equals 1. These values were the same for the occupants of the queue as they were in step_6.

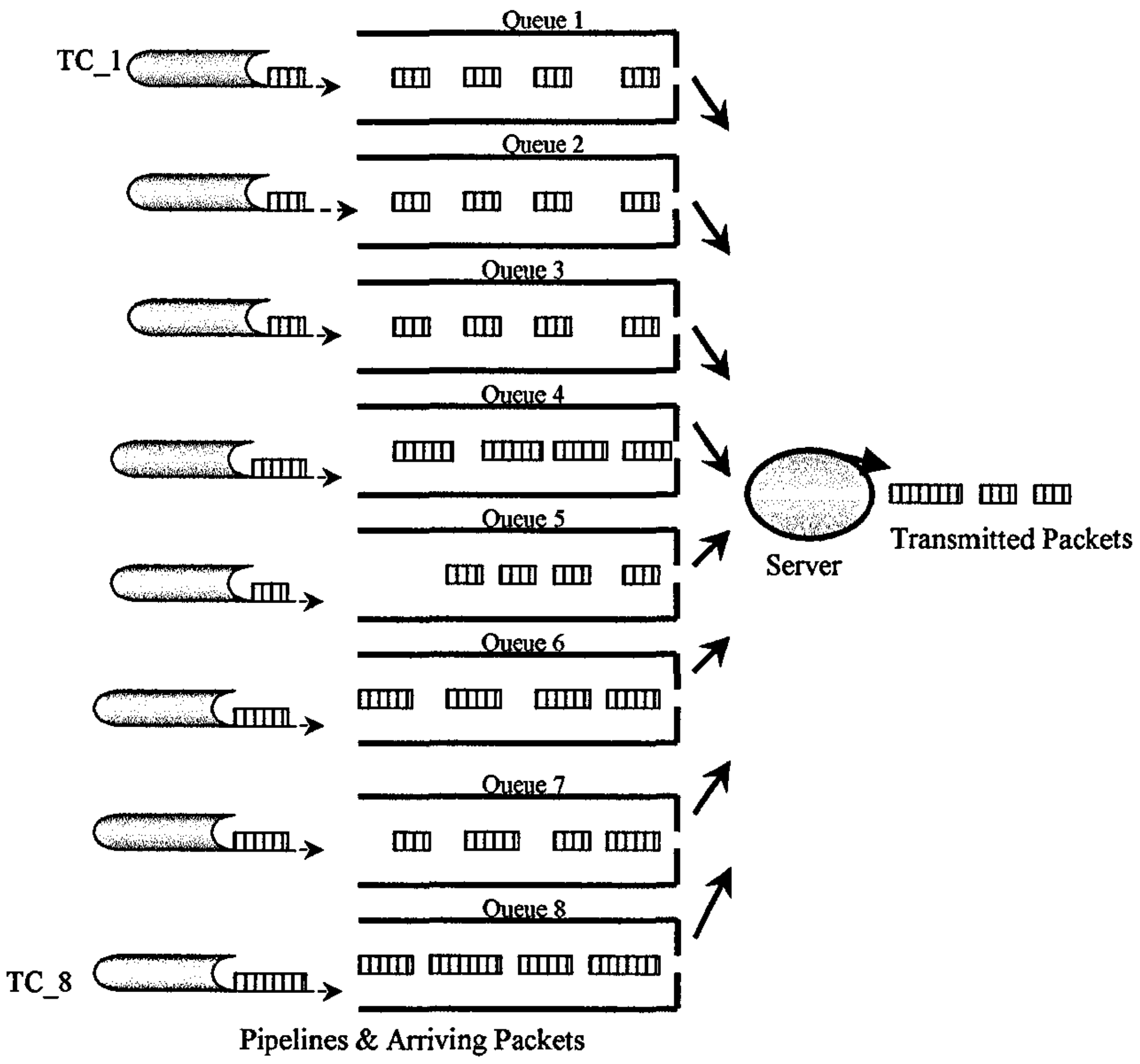


TC = Traffic Class

Figure 5.10: Seven-Queue Implementation. Queue_1, Queue_2, Queue_3, Queue_4, Queue_5 and Queue_6, each served one class of traffic which are NC, VO, VI, AV, CL and EE respectively. While Queue_7 served two classes of traffic (BE & BT).

5.4.8 Step_8 of a Typical SAD

As usual we applied the FRBF algorithm and noted that queue 1, 2, 3, 4, 5 and 6, of step_7, each had one class of traffic to serve and thus they were skipped. The next queue to be acted upon was queue 7 of step_7 and the queue was split into two queues. Thus we have eight queues to manage in this step.



TC = Traffic Class

Figure 5.11: Eight-Queue Implementation. Each queue served one class of traffic

All traffic parameters and types of statistics collected were the same as in previous step. The queue arrangement is illustrated with Figure 5.11.

Each of the eight queues served only one class of traffic. The Expedited Factor of occupants of each of the eight queues equals 1 and their Limiting Proximity Number equals 0. Occupants of Queue_7 and Queue_8 have theoretical improvement in their Expedited Factor and their Limiting Proximity Number if compared with Step_7.

Summary

The simulation methodology for the empirical investigation to determine the *Optimum Number of Traffic Queuing Classes* (ONTQC) that will best support integrated services has been presented in this chapter. The similarity in our simulation methodology with the classical Markovian chain has been expounded. Simulation operations were partitioned into sections known as *Simulation Action Domains* (SAD). The simulation consisted of several SADs and each SAD was made up of eight simulation steps (numbered from 1 to 8). Detailed actions on a typical SAD are described with illustrative diagrams.

CHAPTER 6

Traffic Parameters and Simulation Results for ONTQC

The simulation work on the empirical investigation to determine the optimum number of traffic queuing classes to support integrated services was very extensive. As stated in the previous chapter, it consisted of several simulation scenarios which were grouped or partitioned into *Simulation Action Domains* (SADs) with wide ranges of traffic input parameters. It should be recalled that, there were eight simulation iterative *steps* in each SAD, and each *step* employs eight different traffic classes. Each traffic class in the eight simulation steps of an SAD utilises identical traffic input parameters to generate identical traffic profile. Each SAD generated 64 output results for end-to-end delay and 64 output results for throughput. Due to the large number of SADs we cannot present all the results of every SAD in this thesis. Fortunately, there will not be lots of information lost by not presenting all the results since results from all the SADs follow a similar trend or pattern.

As this chapter title has indicated, the chapter will contain traffic parameters and the simulation results. In Section 1, we will summarise the wide range of traffic parameters employed for the simulation. Although there are four broad groupings of simulation scenarios— (1) simulation with TCP and UDP, (2) simulation with only UDP to remove the effect of TCP traffic management, (3) simulation with equal packet sizes to emulate ATM and (4) simulation without the Internet protocol, we cannot present all the details of the simulation results in this thesis. Since the results have some similarities, we consider it reasonable to present a typical SAD results to represent the simulation results. Thus in Section 2, we will present the results from a SAD augmented with very short summary results from one other SAD to represent the whole simulation results.

6.1 Traffic Parameters in Summary

Here in this section we present an abridgement of traffic input parameters that generate the wide range of traffic profile for the simulation experiment. The input parameters for each of the eight traffic classes is presented in a table (a table each for each class) as illustrated in Table 6.1a to Table 6.1g. Since we cannot represent all the various numerous values of the traffic parameters used in the simulation in a table, an abstraction technique is adopted. The technique consists of presenting the lowest few values of a particular parameter followed by ellipses as an embedded compression notation, which is then followed, by the largest value and then the range in a table row. This idea can be captured graphically from Table 6.1a to Table 6.1g. This represents the wide range of traffic parameter values for a typical broad-group simulation scenario.

Table 6.1

(a) Traffic Class 1

	Low values	High value	Range
Packet sizes in bytes	32, .., 48, ...	, 256	224
Packet rate in kb/s	56, .., 64, ...	, 2048	1992
Burst time in seconds	0.25,.., 0.35, ...	, 2.0	1.75
Idle time in seconds	0.45, .., 0.65, ...	, 3.0	2.55
Inter-packet-arrival distribution	Constant and Exponential		

(b) Traffic Class 2

	Low values	High value	Range
Packet sizes in bytes	210, ...	, 1024	814
Pkt Inter-arrival time	0.0165, ...	, 0.05	0.0335
Packet Inter-arrival distribution	Constant and Exponential		

(c) Traffic Class 3

	Low values	High value	Range
Packet sizes in bytes	105, ...	, 640	535
Packet rate in Kb/s	32, ...	, 448	416
Packet Inter-arrival distribution	Constant		

(d) Traffic Class 4

	Low values	High value	Range
Packet sizes in bytes	200,.., 512..............................	, 1024	824
Packet rate in kb/s	200,......................................	, 300	100
Burst time in seconds	1.0,.., 2.0, ...	, 5.0	4.0
Idle time in seconds	0.45, .., 0.65, ...	, 3.0	2.55
Inter-packet-arrival distribution	Pareto		

(e) Traffic Class 5

	Low values	High value	Range
Packet sizes in bytes	128, ..., 625		814
Pkt Inter-arrival time	0.005, ..., 30.0		29.995
Packet Inter-arrival distribution	Normal		

(f) Traffic Class 6

	Low values	High value	Range
Packet sizes in bytes	400, ..., 800		400
Pkt Inter-arrival time	0.005, ..., 1.0		0.995
Packet Inter-arrival distribution	Normal and Constant		

(g) Traffic Class 7

	Low values	High value	Range
Packet sizes in bytes	700, ..., 800		400
Pkt Inter-arrival time	1.0, ..., 6.0		5.0
Packet Inter-arrival distribution	Exponential		

(h) Traffic Class 8

	Low values	High value	Range
Packet sizes in bytes	1000,....................................., 1500		400
Pkt Inter-arrival time	1.0, ..., 6.0		5.0
Packet Inter-arrival distribution	Exponential		

6.2 Simulation Results

The approach adopted in extracting the main useful results from the extensive simulation results involves a selective extraction approach that will produce the essential results that are needed to make our decision. As described earlier, this consists of choosing a typical SAD and using its results as an approximation of the generality of simulation results and where necessary, we supplement the results with results from other SADs.

The locus of the simulation effort is to compare results of the various simulation scenarios, which employ a ***set*** of different queuing systems, with each member of the set having a different number of traffic-queuing-classes. Thus the results will be combined and presented graphically in composite comparison charts instead of individual result charts. The ***objects*** under consideration in extracting and selecting results are ***eight traffic classes***, ***a set of eight queuing systems***, and ***a set of eight iterative simulation scenarios.*** The objective functions in our decision making process then would consist of a set of relational functions that define a mapping or assignment of these objects onto one another. The statements of such relational functions are as follows.

- Eight traffic classes $\Rightarrow$ map into **one-queue** $\Rightarrow$ produce 2 x (eight instances of output results).
- Eight traffic classes $\Rightarrow$ map into **two-queues** $\Rightarrow$ produce 2 x (eight instances of output results).
- Eight traffic classes $\Rightarrow$ map into **three-queues** $\Rightarrow$ produce 2 x (eight instances of output results).
- Eight traffic classes $\Rightarrow$ map into **four-queues** $\Rightarrow$ produce 2 x (eight instances of output results).
- Eight traffic classes $\Rightarrow$ map into **five-queues** $\Rightarrow$ produce 2 x (eight instances of output results).
- Eight traffic classes $\Rightarrow$ map into **six-queues** $\Rightarrow$ produce 2 x (eight instances of output results).
- Eight traffic classes $\Rightarrow$ map into **seven-queues** $\Rightarrow$ produce 2 x (eight instances of output results).
- Eight traffic classes $\Rightarrow$ map into **eight-queues** $\Rightarrow$ produce 2 x (eight instances of output results).

6.2.1 Results of One-Queue Compared with Two-Queues

The first step in our iterative simulation actions was to determine if the lowest number in the multiple queuing system (**two-queues CBQ**) offered better services than the mono-queuing system (**FIFO**) in Integrated Services environment. The result of the first step in our simulation actions, which consisted of two simulation scenarios — (1), *FIFO queuing system* and (2), *two-queues CBQ system* with both having identical traffic input parameters to generate identical traffic profile is shown in Figure 6.1. *This result shows a tremendous improvement in service quality when the multiple queuing system is employed in multiservice-network compared with the use of mono queuing system.* The result served as motivation to continue with the empirical investigation to determine the optimum number of traffic queuing classes that will best support Integrated Services. It was observed from the simulation results that, the end-to-end delay results for all the eight traffic classes were similar in the single (FIFO) queue system. In the results of the two-queue multiple queue system, it was also noted that, both the four high-priority traffic classes and the four low-priority traffic classes had lower end-to-end delay compared with the single queue system. Thus in order to economise on reading pages, we present here the results of *a traffic class* in both the single (FIFO) queue system and the two-queue (CBQ) queue system, since the results shown in the composite chart capture the trend of the whole results under consideration. The results shown in Figure 6.1 are the end-to-end delay composite chart for traffic class I (Voice) in both the one-queue system and the two-queue system.

In the subsequent result presentation, we will eliminate the mono queuing system (FIFO queuing system) in the comparison of results, since it has been established that multiple queuing system offers better services than the mono queuing system.

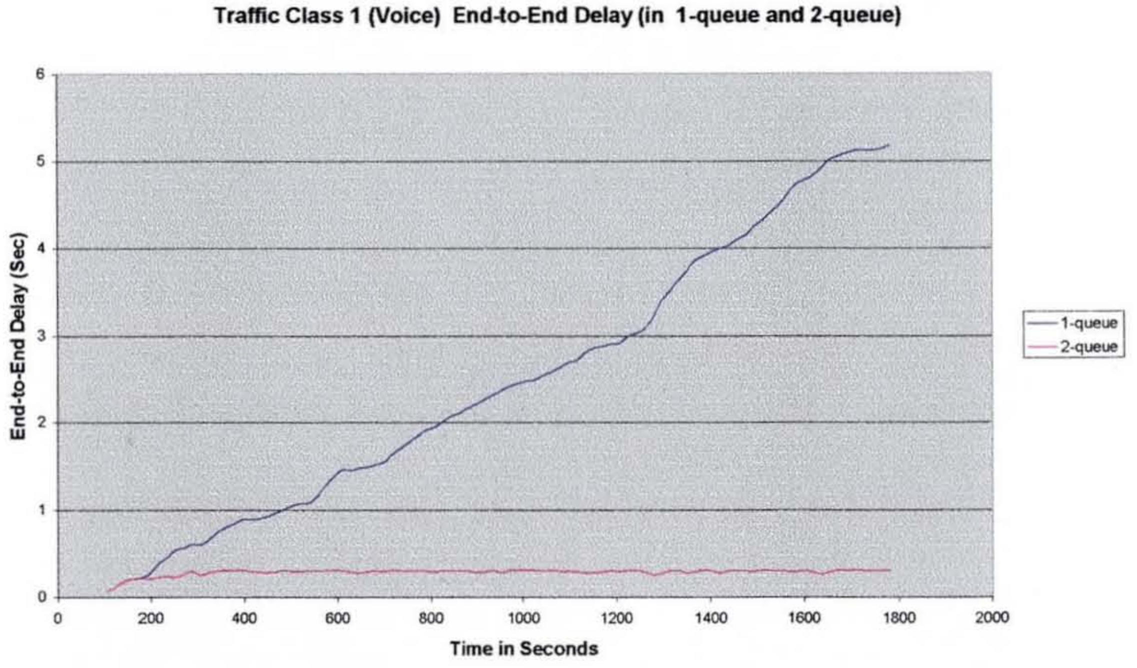


Figure 6.1: Showing end-to-end delay for 1-queue (FIFO) system and 2-queue (CBQ) system.

6.2.2 Typical Results for Traffic Class 1

The results consist of two parts — a graphical part, which consists of an X-Y chart and a statistical numerical part, which consists of a row-vector. The graphical part presents the display of a composite comparison charts for simulation using 2-queues, 3-queues, ..., up to 8-queues, in this case, for traffic class 1. The end-to-end delay results for traffic class 1 is shown in Figure 6.2. In the statistical numerical part, we present end-to-end mean delay row-vector for the traffic class 1.

As shown in the Figure 6.2, the results for simulation with 4-queues and 5-queues have the lowest end-to-end delay for traffic class 1. The performance of the multiple queues in terms of end-to-end delay decreases as we increase the number of queues above 5-queues for traffic class 1.

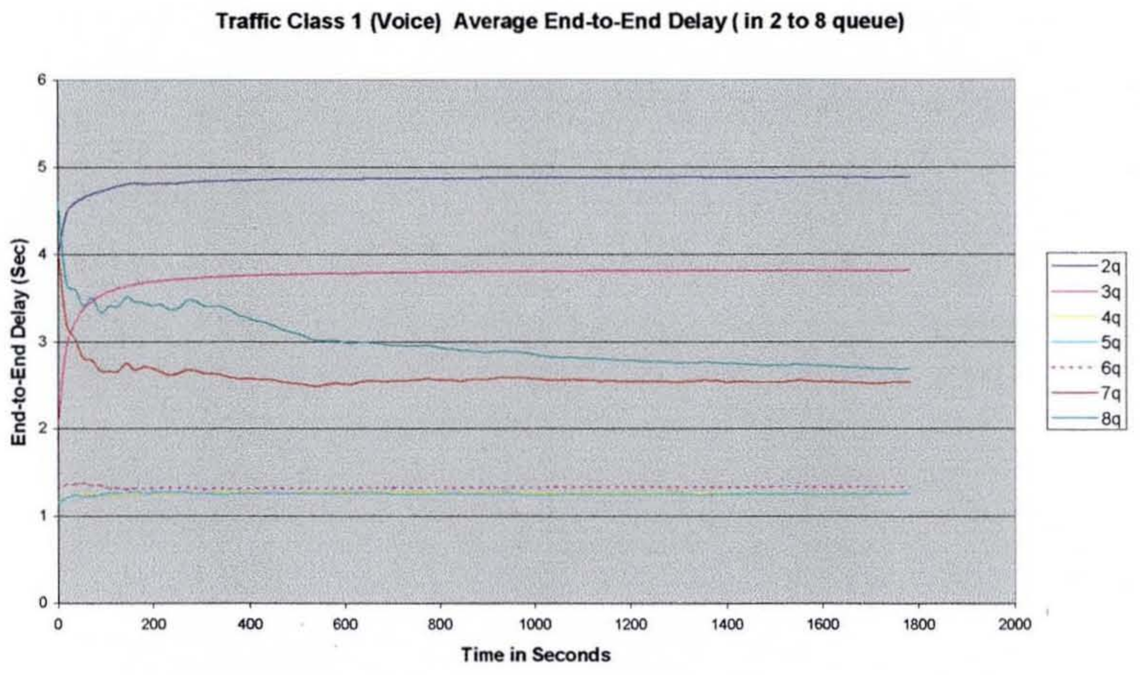


Figure 6.2: Showing end-to-end delay for traffic class 1, in 7 simulation scenarios, with each scenario employing different numbers of queues varying from 2-queues up to 8-queues. Identical input traffic parameters used in all the seven scenarios.

The results for the *mean* end-to-end delay for the set of simulation scenarios employing multiple queues ranging from 2-queues up to 8-queues for traffic class 1 are shown in the row-vector mean delay that follows:

Traffic-class	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Class-1	4.846	3.744	1.259	1.253	1.326	2.597	2.995

6.2.3 Typical Results for Traffic Class 2

The results also consist of two parts —a graphical part and a statistical numerical part, which is in the same format as in the previous section. The graphical result for traffic class 2 is shown in Figure 6.3 for end-to-end delay. The figure is a composite chart, which shows the relationship between the multiple queuing systems for traffic class 2. The chart results show that the lowest end-to-end delay has been recorded for traffic class 2 with simulation scenario employing 3-queues. The results also show that, end-to-end delay performance continues to deteriorate as we increase the number of traffic queuing classes above 3 for traffic class 2.

The *mean* end-to-end delay row-vector for traffic class 2 is as follows:

Traffic-class	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Class-2	4.880	3.723	7.390	7.232	7.628	15.402	12.588

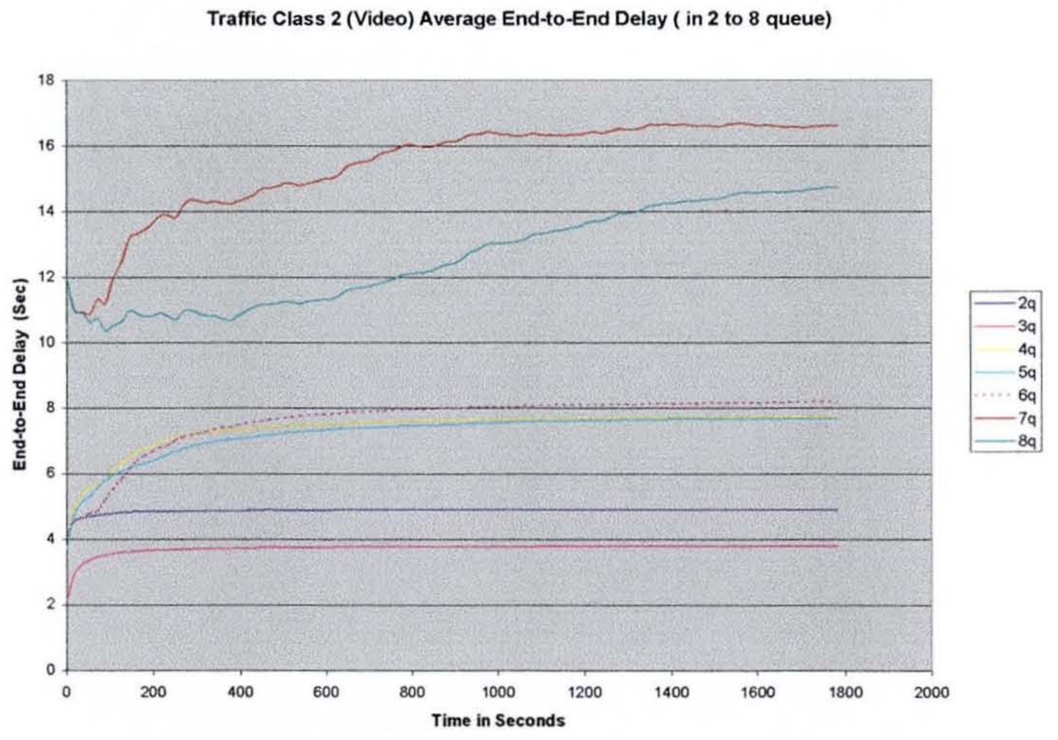


Figure 6.3: Showing end-to-end delay for traffic class 2, in 7 simulation scenarios with each scenario employing a different number of queues varying from 2-queues up to 8-queues. Identical input traffic parameters used in all the seven scenarios.

6.2.4 Typical Results for Traffic Class 3

The results, just as in previous sections consist of two parts —a graphical part and a statistical numerical part. The graphical result for traffic class 3 is shown in Figure 6.4 for end-to-end delay. The composite display as shown in the chart, made it easy to compare the performance of one multiple queuing system with another for traffic class 3. The chart results showed that, the best performance for end-to-end delay has been recorded with the simulation scenario employing 3-queues in the experiment. As revealed in the chart, the end-to-end delay performances continue to deteriorate as the number of queues employed in the simulation scenario increases above 3 for the traffic class 3.

The *mean* end-to-end delay row vector for traffic class 3 is as follows:

Traffic-class	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Class-3	4.886	2.387	3.399	8.092	8.769	17.387	14.071

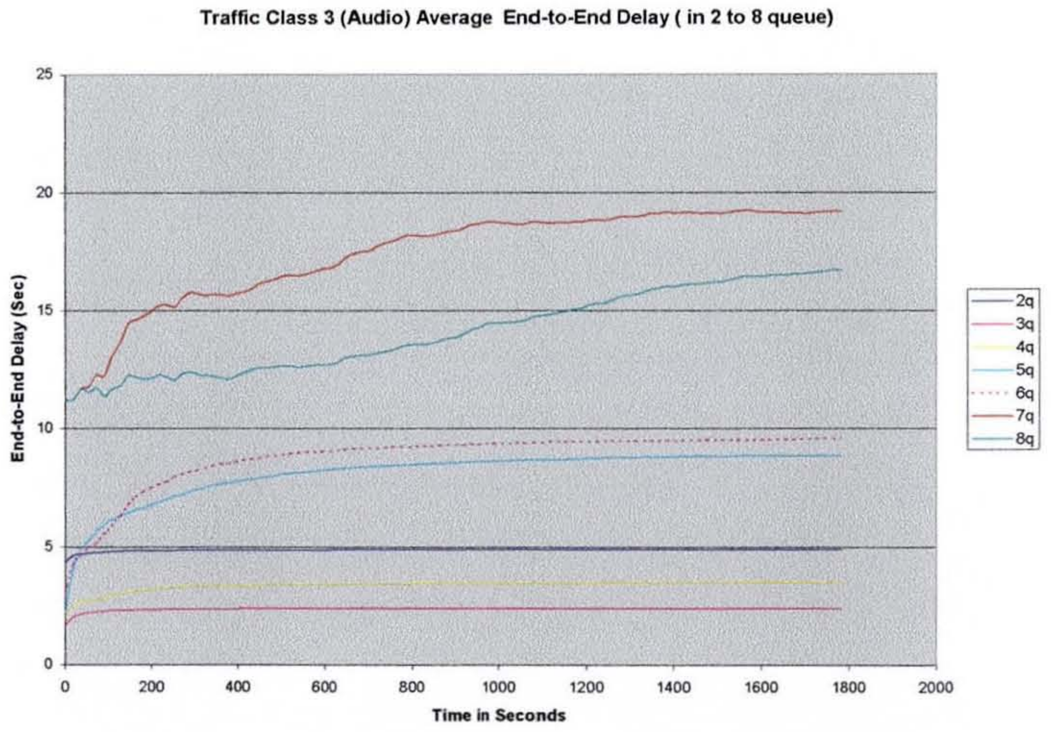


Figure 6.4: Showing end-to-end delay for traffic class 3, in 7 simulation scenarios with each scenario employing different numbers of queues varying from 2-queues up to 8-queues. Identical input traffic parameters used in all the seven scenarios.

6.2.5 Typical Results for Traffic Class 4

The results are in the same format as in previous sections and consist of two parts —a graphical part and a statistical numerical part. The graphical results are shown in Figure 6.5. The composite chart provides an easy means for performance comparison between the elements in the *set* of multiple queuing systems employed in the simulation experiment.

The chart results showed that, the best performance for end-to-end delay has been recorded with simulation scenario employing 5-queues for the traffic class 4. The results revealed poorer end-to-end delay performance when the number of multiple queues employed in the simulation scenarios are higher than 5 for class 4 traffic.

The *mean* end-to-end delay row vector for traffic class 4 is as follows:

Traffic-class	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Class-4	4.882	2.390	3.405	1.966	2.121	5.009	4.878

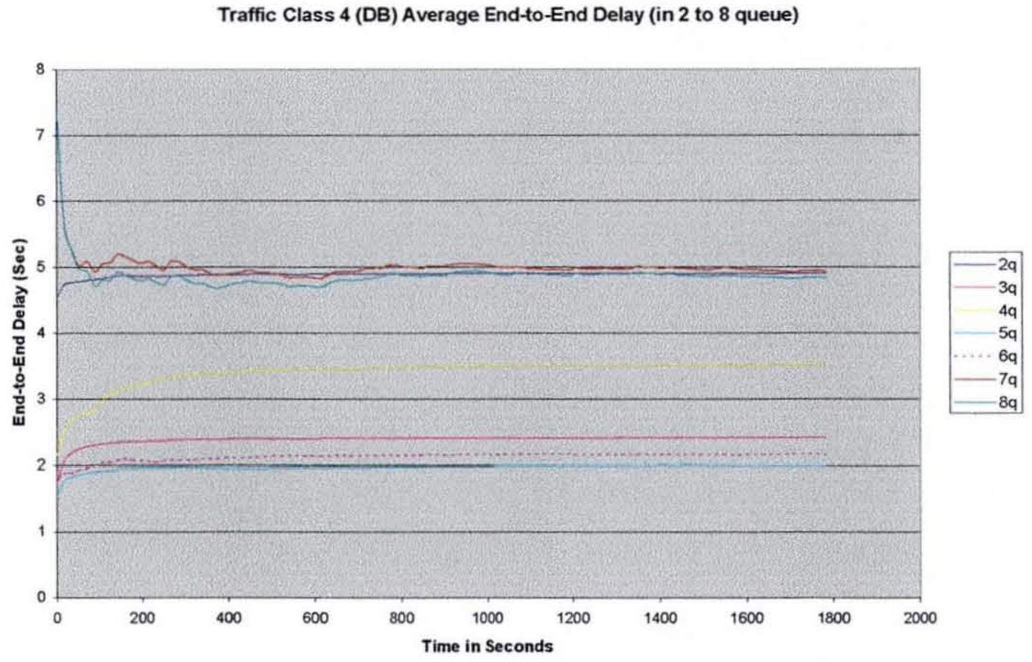


Figure 6.5: Showing end-to-end delay for traffic class 4, in 7 simulation scenarios with each scenario employing different number of queues varying from 2-queues up to 8-queues. Identical input traffic parameters used in all the seven scenarios.

6.2.6 Typical Results for Traffic Class 5

The results consist of two parts as in the previous sections —a graphical part and a statistical numerical part. The graphical result for traffic class 5 is shown in Figure 6.6. The composite chart provides an on-the-sport graphical result performance comparison, between the elements in the *set* of multiple queuing systems employed in the simulation.

The chart results revealed that, the best performance for end-to-end delay has been recorded with the simulation scenario employing 6-queues for traffic class 5. The result showed very poor end-to-end delay performance and a drastic degradation change when the number of queues in the multiple queuing systems is above 6 for the traffic class 5.

The *mean* end-to-end delay row vector for the traffic class 5 is as follows:

Traffic-class	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Class-5	5.875	8.252	14.032	12.943	2.128	145.252	141.014

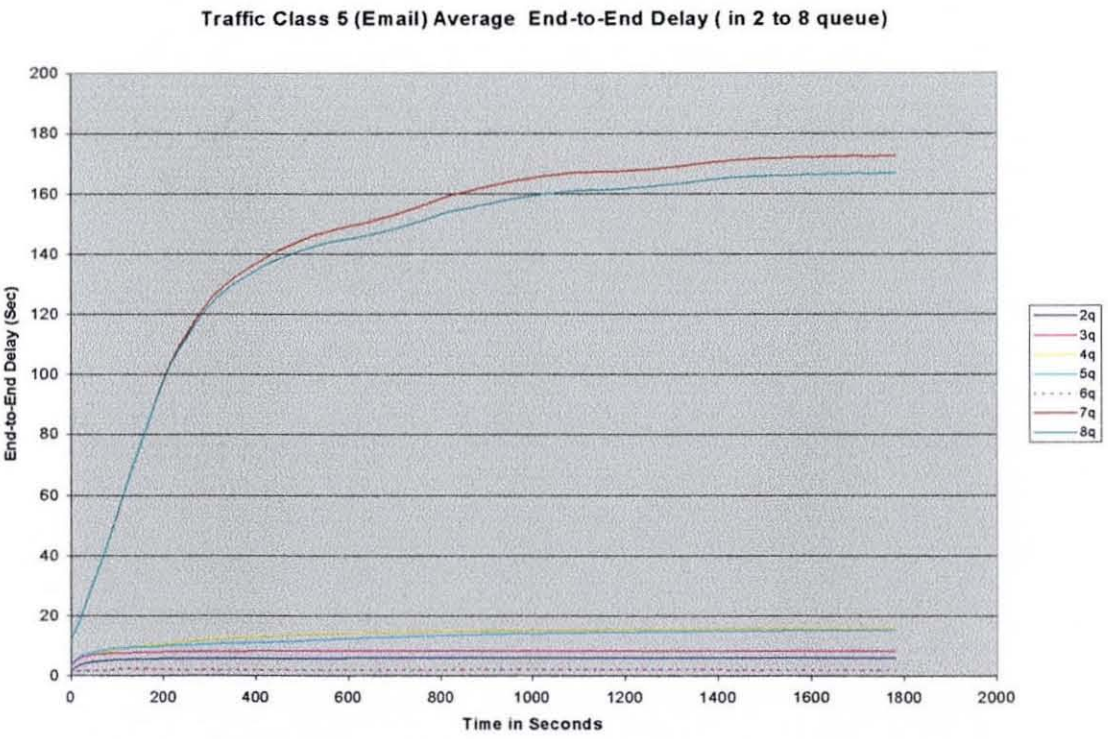


Figure 6.6: Showing end-to-end delay for traffic class 5, in 7 simulation scenarios with each scenario employing different number of queues varying from 2- queues up to 8-queues. Identical input traffic parameters used in all the seven scenarios.

6.2.7 Typical Results for Traffic Class 6

The end-to-end delay results for traffic class 6 as in previous sections, consists of two parts—a graphical part and a statistical numerical part. The graphical results are shown in Figure 6.7.

The chart results revealed that, the best performance for end-to-end delay has been recorded with the simulation scenario employing 7-queues for traffic class 6. The results showed that, changing the number of queues from 7 to 8 in the multiple queuing systems did not improve performance for traffic class 6. The results show slight performance degradation in end-to-end delay as we change from 7-queue to 8-queue system.

The *mean* end-to-end delay row vector for the traffic class 6 is as follows:

Traffic-class	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Class-6	5.845	8.169	13.960	12.869	2.143	1.380	1.390

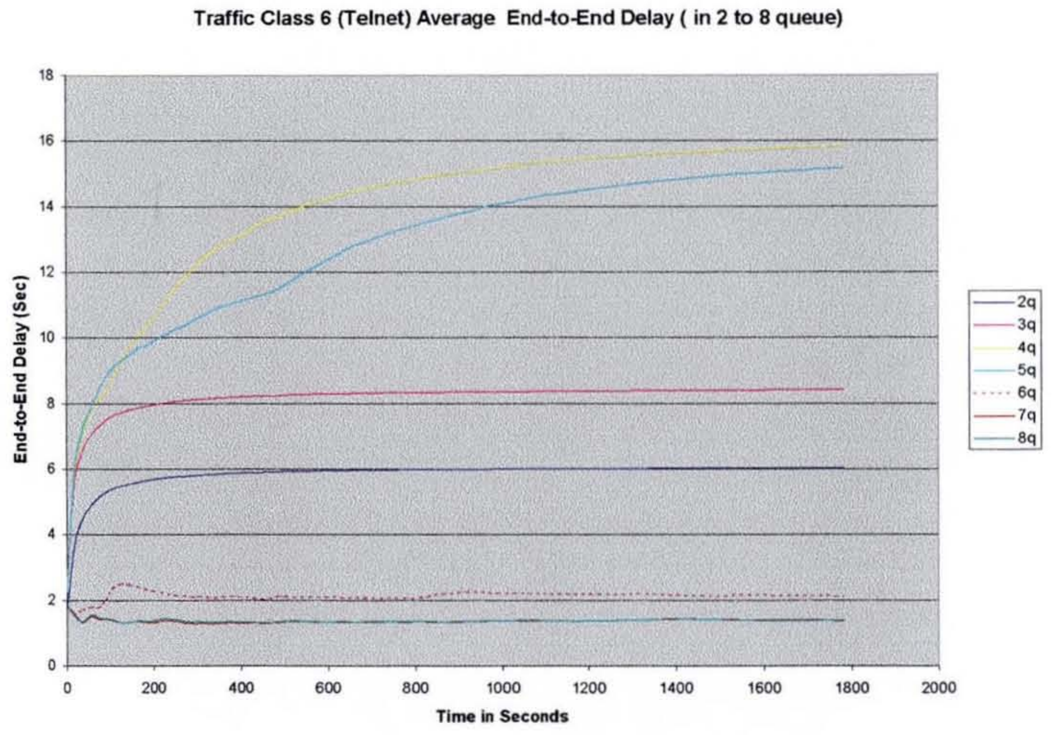


Figure 6.7: Showing end-to-end delay for traffic class 6, in 7 simulation scenarios with each scenario employing different number of queues varying from 2-queues up to 8-queues. Identical input traffic parameters used in all the seven scenarios.

6.2.8 Typical Results for Traffic Class 7

The end-to-end delay results for traffic class 7 as in previous sections, are made-up of two parts—a graphical part and a statistical numerical part. The graphical results are shown in Figure 6.8.

The chart results revealed that, the best performance for end-to-end delay has been recorded with the simulation scenario employing 2-queues for traffic class 7. The results showed that, as the number of queues in the multiple queuing systems is increased above 2, wide range of very poor performances are recorded for end-to-end delay for traffic class 7.

The *mean* end-to-end delay row vector for the traffic class 7 is as follows:

Traffic-class	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Class-7	5.870	8.241	14.045	12.955	11.814	20.997	13.897

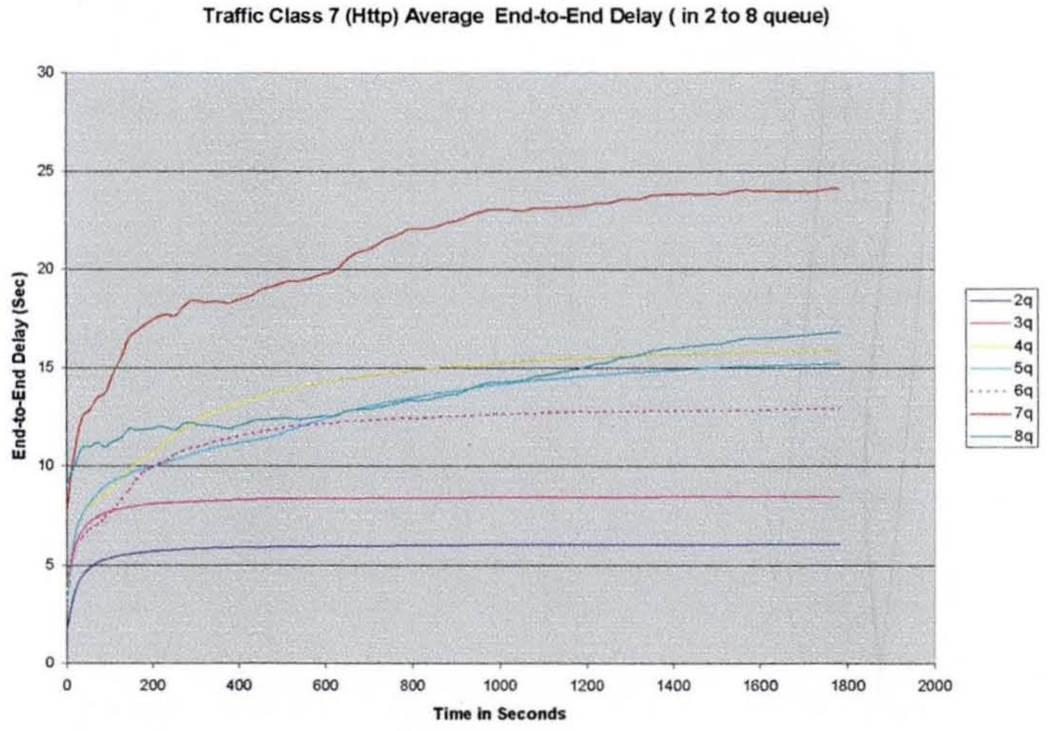


Figure 6.8: Showing end-to-end delay for traffic class 7, in the 7 simulation scenarios with each scenario employing different numbers of queues varying from 2-queues up to 8-queues. Identical input traffic parameters used in all the seven scenarios.

6.2.9 Typical Results for Traffic Class 8

The end-to-end delay results for traffic class 8 as in previous sections, are made-up of two parts—a graphical part and a statistical numerical part. The graphical results are shown in Figure 6.9.

The chart results revealed that, the best performance for end-to-end delay has been recorded with the simulation scenario employing 2-queues for traffic class 8. The results are similar to that of traffic class 7 and showed that, as the number of queues in the multiple queuing systems is increased above 2-queues, wide range of very poor performances are recorded for end-to-end delay for the traffic class 8.

The *mean* end-to-end delay row vector for the traffic class 8 is as follows:

Traffic-class	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Class-8	5.931	8.307	14.106	13.014	11.868	21.104	19.530

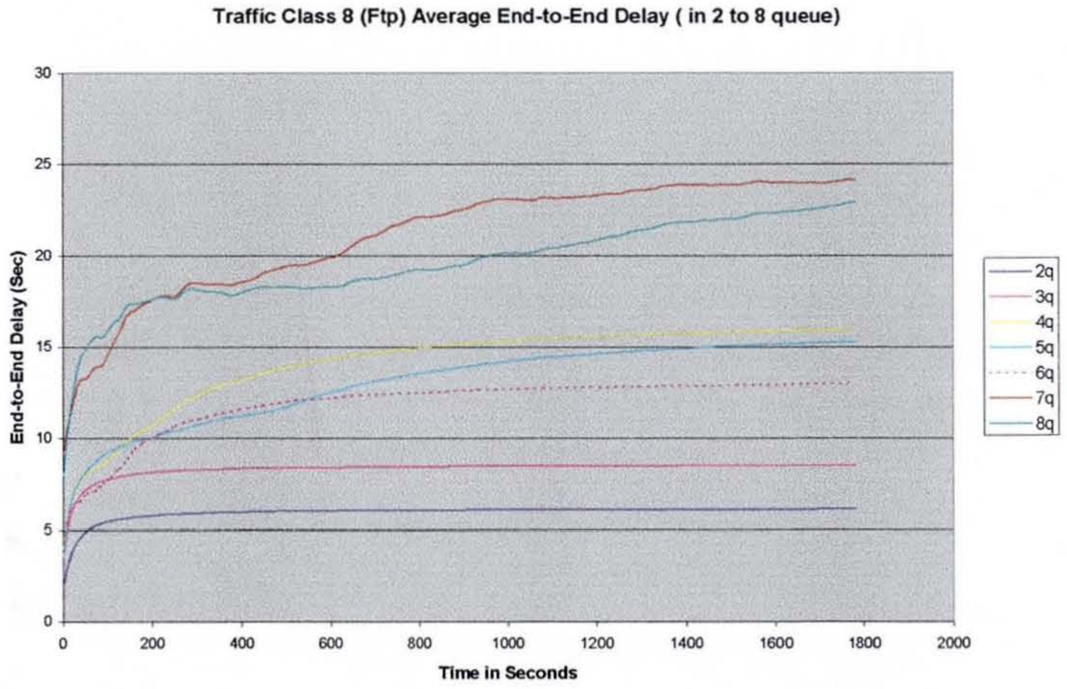


Figure 6.9: Showing end-to-end delay for traffic class 8 in the 7 simulation scenarios with each scenario employing different number of queues varying from 2-queues up to 8-queues. Identical input Traffic Parameters used in all the seven scenarios.

6.2.10 Global End-to-End Delay for the results of each Queue System that have been presented

As a means of abstraction and concise result presentation, the combined average delay for all traffic classes in each of the queuing systems are displayed in a composite global chart. The chart is referred to as All Traffic Class Average Global End-to-End Delay, and is shown in Figure 6.10.

The concise chart provides a means to capture graphically at a glance, the essential details of the whole results of the simulation experiment. As can be seen from the chart, ***the best performance in terms of global end-to-end delay was recorded with the use of 3-queues*** in the *set* of multiple queuing systems employed in the experiment. Analysis of the results will be carried out in Chapter 7 where more details from the chart will be discussed.

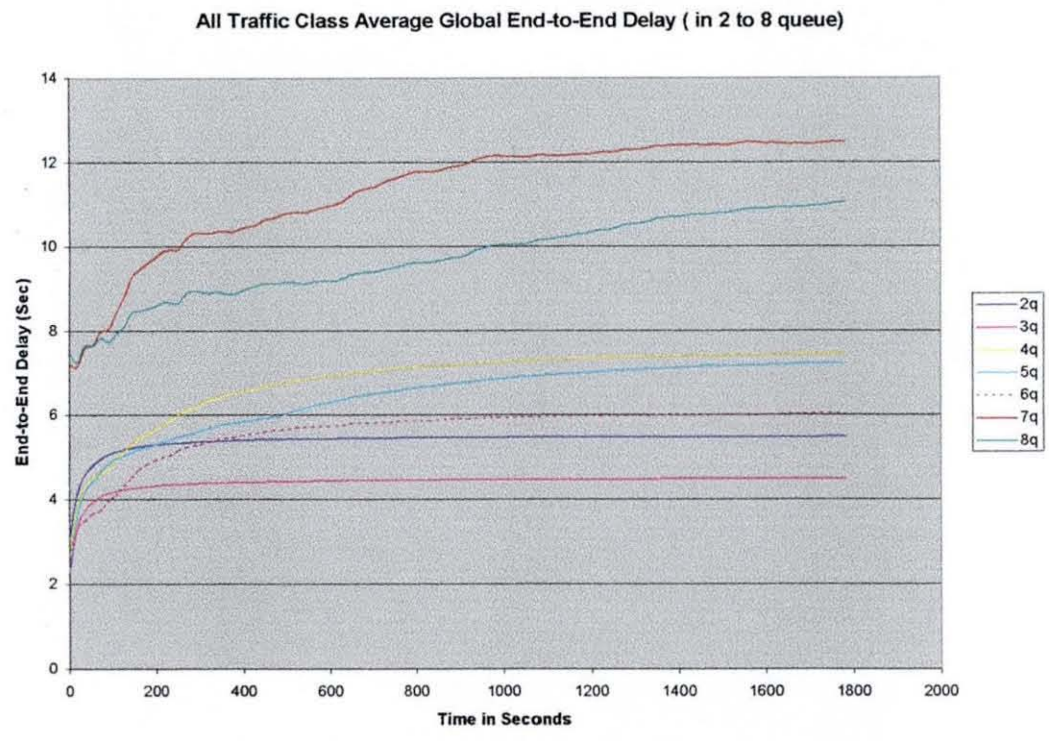


Figure 6.10: Showing Global end-to-end delay in the 7 simulation scenarios with each scenario employing different numbers of queues varying from 2-queues up to 8-queues. Identical input traffic parameters used in all the seven scenarios.

Higher abstractions are obtained through the processes of numerical compression and manipulation, which involves carrying out arithmetic operations on the data of the global average end-to-end delay to produce a *statistic parameter matrix* (Table 6.2.) This provides a high level summary, which enables the trend in delay performances of each multiple queue system to be captured as we change from one multiple queuing system to another in the simulation experiment. The *mean, variance and standard deviation* (S.D) of end-to-end delay computed from the global data for each i-queue, ($2 \leq i \leq 8$) simulations, are shown in the *matrix* of

Table 6.2. This is a combination of the global *mean delay row vector*, the global *variance row vector*, and the global *standard deviation* (S. D) *row vector*.

Table 6.2

	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Mean	5.384	4.399	6.818	6.417	5.616	11.308	9.712
Variance	0.090	0.071	0.823	0.801	0.451	1.804	0.926
S. D	0.299	0.267	0.907	0.895	0.672	1.343	0.962

The results for *end-to-end delay* (sojourn time) presented so far, are from one SAD in which no Internet protocol is directly utilised. The results from SADs in which Internet protocols are utilised are also similar to the ones presented above. The summary of results from an SAD in which Internet protocols was utilised in the simulations will be presented next to capture the trend in the results.

6.2.11 Summary of Results for an SAD with Internet Protocols

In view of limitation of space, the results presented here will be a very concise abstraction that can capture the essentials of the simulation results under the SAD with Internet protocols. The results presented will be the numerical values rather than graphical values that will consume space. Thus the results will consist of a *mean* end-to-end delay matrix and a global *mean-mean* delay row vector. These are shown in Table 6.3 and Table 6.4 respectively.

Table 6.3

Traffic-class	Q2	Q3	Q4	Q5	Q6	Q7	Q8
cls_1	0.0247	0.0242	0.0237	0.0231	0.0231	0.0240	0.0243
cls_2	0.0280	0.0291	0.0265	0.0263	0.0263	0.0262	0.0262
cls_3	0.0258	0.0267	0.0260	0.0262	0.0262	0.0266	0.0264
cls_4	0.0276	0.0284	0.0276	0.0278	0.0280	0.0281	0.0274
cls_5	0.6607	0.6586	0.6600	0.3179	0.3188	0.1589	0.1574
cls_6	0.6644	0.6626	0.6636	0.3186	0.3196	1.0138	0.9795
cls_7	0.6868	0.6849	0.6879	4.4770	4.0524	3.2582	3.1363
cls_8	0.6907	0.6899	0.6908	4.5761	4.2542	3.2452	3.2175

Table 6.4

	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Global_D	0.3511	0.3506	0.3508	1.2241	1.1311	0.9726	0.9494

Note: Global_D to read Global Delay

The results are to some extent, similar to those presented earlier, especially in the trend of the global mean-mean delay. It can be seen from the global mean-mean delay row vector that, the best performance for end-to-end global delay (i.e. the lowest) for all traffic classes is recorded with the use of 3-queues in the investigation of the *optimum number* in the multiple queuing systems.

6.2.12 Typical Global Throughput Results for All Traffic Classes

The results for throughputs presented here are based on the SAD without Internet protocol whose end-to-end delay results have been presented earlier. The results consist of two parts— a graphical part and numerical statistical part. The graphical part consists of a compressed composite chart in which each trace of the chart represents the combined average throughput for all traffic classes in each multiple queuing system simulated. The global throughput composite chart is shown in Figure 6.11. The numerical statistical part consists of *mean throughput* values presented in a matrix form for quick reading, and a global *mean-mean throughput* row vector also for quick reading. These are shown in Table 6.5 and Table 6.6.

The matrix of Table 6.5 shows the values of the *mean* throughput for each of the traffic classes in each multiple queuing implementation, arranged in a row vector. This allows comparison of the results of one multiple queue system against another.

The average throughput matrix is as follows:

Table 6.5

Traffic Class	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Cls-1	1240	2860	2933	2931	2932	2833	2830
Cls-2	39146	69530	48610	48581	45613	22181	22348
Cls-3	25688	42596	39310	40093	37662	18317	18457
Cls-4	22145	43975	40670	39993	37625	18279	18417
Cls-5	25365	19790	18030	18279	32390	3390	3416
Cls-6	755	704	692	677	776	778	778
Cls-7	112973	84910	78381	78686	61686	30638	33950
Cls-8	168823	131903	122231	122192	110213	53428	50815

The global mean average-throughput row-vector is as follows:

Table 6.6

	Q2	Q3	Q4	Q5	Q6	Q7	Q8
GlobMnThrput	49517	49534	43857	43929	41112	18731	18876

It should be noted that the highest throughput is recorded globally by the use of the 3-queue system.

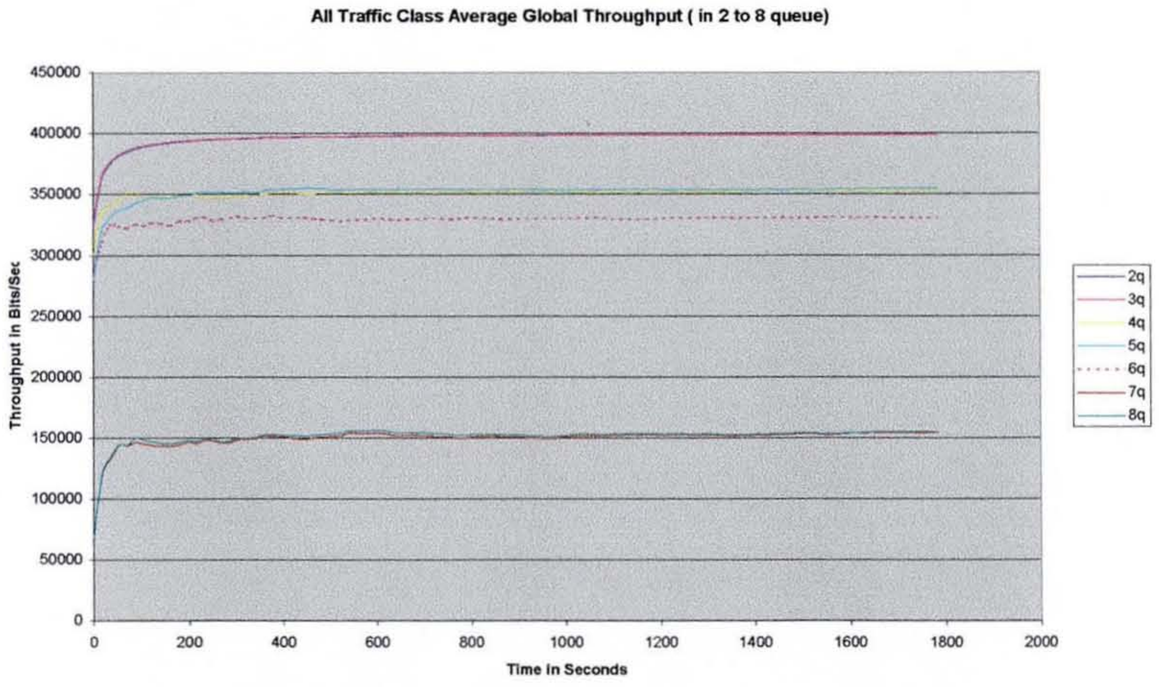


Figure 6.11: Showing Global Throughput for the 7 simulation scenarios employing different numbers of queuing classes ranging from 2-queuing classes to 8-queuing classes. Identical input traffic parameters used in all the seven scenarios.

The chart results showed that, the highest global throughput is recorded when 3-queue is adopted for the multiple queuing systems.

Summary

The parameters used to parameterise the various applications traffic for the simulation and the simulation results have been presented in this chapter.

The parameters are presented in a concise manner, in the format of a set of tables for quick reading.

The results from the large number of *Simulation Action Domains* (SADs) have some similarities and as such, results from one SAD are used to represent the whole simulation results. The results of each simulation step (step 1 to 8 exist) of a typical SAD are presented and illustrated with composite charts. Numerical statistics results such as *mean, variance and standard deviation* are presented in form of row vectors and matrices as summaries, to enhance quick capture of the important points of the results.

CHAPTER 7

Analysis of Results from the Simulation on ONTQC

The results of the simulation were presented in Chapter 6. Analysis of those results will be the focus of this chapter. As noted in the previous chapter, the results for the extensive simulation follow a similar trend, which consequently removed the necessity for presenting and analysing the whole simulation results. Thus the results analysed here in this chapter are for the typical *Simulation Action Domain* (SAD) as presented in Chapter 6. The results analysis will be two-dimensional in perspective. One viewpoint will be based on the true or native results while the other viewpoint will be based on the derived meta-heuristic analysis that was presented in Chapter 5 to predict the results of the simulation experiments.

The thrust of the result analysis is based on comparison of one queuing system with another to deduce the most suitable among all the queuing systems investigated for support of Integrated Services. The typical result analysis is based on eight simulation scenarios, each for each of the eight queuing systems. The ground for comparison is found in the fact that each of the eight queuing systems (1-queue system, 2-queue system, ..., 8-queue system) receive equal load and the server capacities are constant. Although traffic loads in each of the of the eight steps of a SAD are identical, the traffic interarrival time distribution of each of the traffic classes differs from one to another. Thus we could not apply the *central limit theorem** on the sum of the arrival distributions of the eight traffic classes S_N. This is because the asymptotic condition (i.e. $N \rightarrow \infty$) of N, the number of independent random interarrival times, and the need that, all the N random variables should have identical distributions, do not hold in our case. In our case, $N = 8$ and the arrival distributions of the eight traffic classes were not identical.

The central limit theorem state that, if S_N represents the sum of N independent identically distributed random components, and if each component makes only a small contribution to the sum, then the cumulative distribution function (CDF) of S_N approaches a Gaussian CDF as N becomes very large* **[Kre93].

The sum of the traffic input load to our set of queuing systems has arbitrary interarrival time distributions. In which case we assume the mono-queuing system (FIFO) is of the type G/M/1, while the remaining seven queuing systems are of the type G/G/1.

This chapter consists of three main sections, the first section is for analysis of the results for mono-queuing system, while the second section is for analysis of the results for multiple queuing systems and the third section for combining the results to produce a single criterion for decision making. In Section 7.1, the analysis of the results of the simulation of the FIFO mono-queuing system is presented. The results obtained from the queuing discipline are analysed and related to the heuristic analysis covered in Chapter 5. Analyses of results for the multiple queuing systems are covered in Section 7.2, which consists of seven subsections. The results are analysed in relation to the heuristic approach. Comparative analyses of the results of each of the queuing systems with one another are also carried out in the subsections. In Section 7.3, we briefly present a method for combining the results of the simulation to form a single point criterion on which we can base our decision.

7.1 Analysis of Results for Mono Queue System

The end-to-end delay (sojourn time) results for each of the eight traffic classes were similar with very little variation in the results recorded for each of the traffic classes. The results shown in Figure 6.1 chart, are an approximate representation of the results for all the traffic classes. As shown in the chart, the end-to-end delay increases with time almost monotonically (quasi-linear). The results reflect the true performances of applications in an overloaded network with saturated queue capacities. The similarity in delay profile for each of the traffic classes shows that the FIFO queue discipline in the mono-queuing system is truly egalitarian, no preferential treatment is given to any of the traffic classes. The throughput for each of the traffic classes is a function of its packet arrival rate.

A close study of the queue structure and its service discipline shows that the results follow intuitively that which is expected from such a queue structure. Since the FIFO mono-queue system accepts and services packets indiscriminately, the probability of each of the eight traffic classes being served at any instant in time is the same for all. The probability for each traffic class to be served at any point in time then equals 0.125. Since all the traffic classes have equally likelihood of being served and are all given equal treatment by the FIFO queue, the output profile of services will be the same for all. This is exactly what the simulation results show.

Relating the results to the heuristic analysis given in Chapter 5 to predict the simulation results revealed that, the prediction of the results with our heuristic analysis was quite accurate. FIFO mono-queue has the highest *limiting quantum molecular length* in all the sets of queuing systems employed in this investigation. The queue limiting quantum molecular length equals 8 (the number of different traffic classes that the queue accepts). The queue size increases with time in relative multiples of the of the queue's quantum molecular length. The distribution of the queue size is a function of the combined arrival distribution of each of the eight traffic classes. The elongated queue size, which is a consequence of the high quantum molecular length of the queue, accounts for the high end-to-end delay seen in the results.

The queue is a work-conserving system, by this we mean work is neither created nor destroyed at the point of service. All server effort is concentrated on serving the queue. This is illustrated in the Figure 7.1 in which the server has a single *fixed state*, which is in continual transition or looping back to itself as long as the queue is backlogged and service continues. This is not the case with multiple queuing systems, as we will see in the next section, the server has to make transitions from one queue to another. Intuitively, throughput ought to be maximum for this queue in view of its work conserving nature. But the simulation result did not reflect better performance of this queue on throughput than the multiple queuing system. Throughput output was lower for this queue compared with the multiple queuing systems, as we will see in the next section.

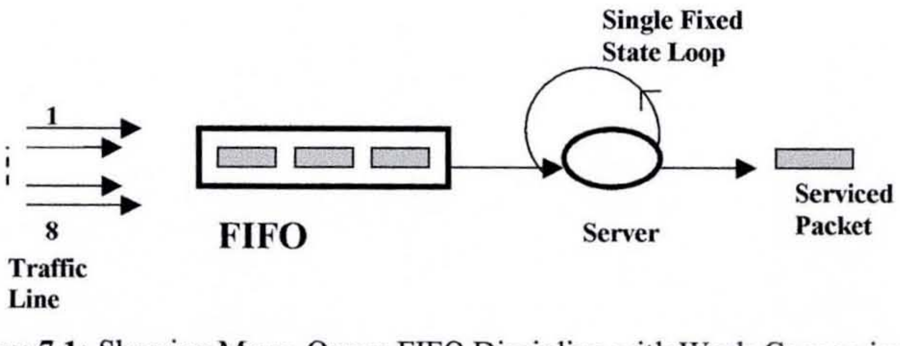


Figure 7.1: Showing Mono-Queue FIFO Discipline with Work-Conserving Server. (No server queue state transition)

7.2 Analyses of Results for Multiple Queue Systems

Beside analysing results for each of the multiple queue systems in this section, we will compare the results of the queuing systems with one another in order to isolate superior performance. This in essence is the thrust of our simulation objective. In order to bring home the analysis and comparison of the results, we will re-present the global delays chart and global throughput chart for closer examination. The charts are shown in Figure 7.2 and Figure 7.3 respectively. These results are augmented with the corresponding statistical parameter row-vectors as presented below.

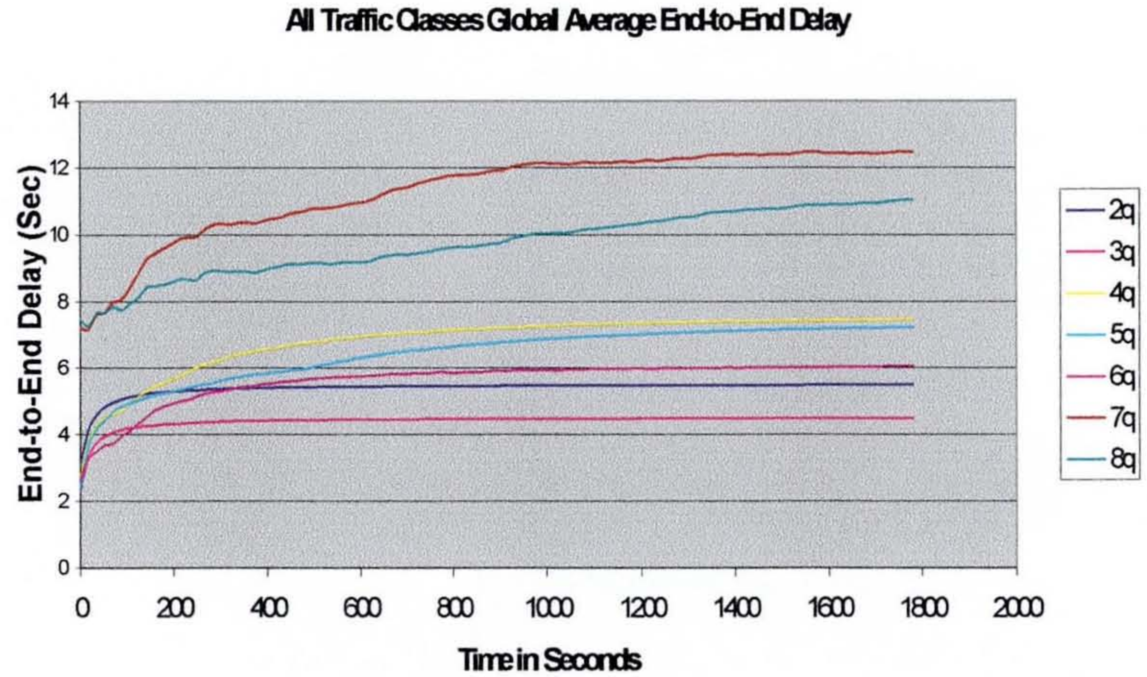


Figure 7.2: Showing Global end-to-end delay in 7 simulation scenarios with each scenario employing a different number of queues varying from 2-queues up to 8-queues. Identical input traffic parameters used in all the seven scenarios. The composite chart shows the simulation with deficit weighted round robin scheduling queuing discipline.

The global *mean* end-to-end delay, the *variance* (jitter) and *standard deviation* (S. D) are as follows:

	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Mean	5.384	4.399	6.818	6.417	5.616	11.308	9.712
Variance	0.090	0.071	0.823	0.801	0.451	1.804	0.926
S. D	0.299	0.267	0.907	0.895	0.672	1.343	0.962

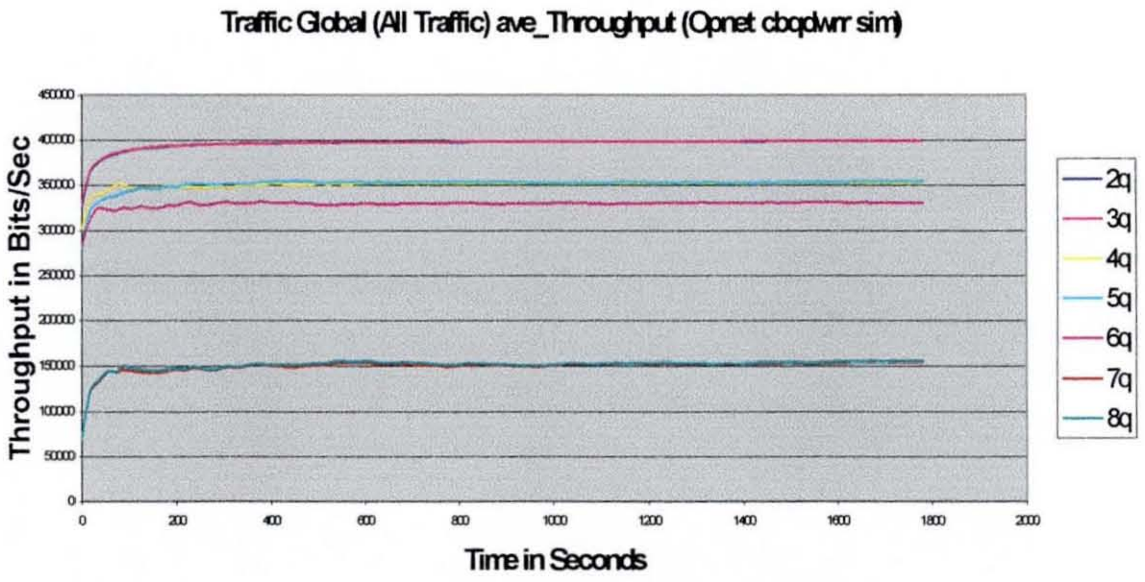


Figure 7.3: Global throughput for 7 simulation scenarios employing different number of queuing classes ranging from 2-queuing classes to 8-queuing classes. Identical input traffic parameters used in all the seven scenarios.

The *average* throughput matrix is as follows:

Table 7.1

Traffic Class	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Cls-1	1240	2860	2933	2931	2932	2833	2830
Cls-2	39146	69530	48610	48581	45613	22181	22348
Cls-3	25688	42596	39310	40093	37662	18317	18457
Cls-4	22145	43975	40670	39993	37625	18279	18417
Cls-5	25365	19790	18030	18279	32390	3390	3416
Cls-6	755	704	692	677	776	778	778
Cls-7	112973	84910	78381	78686	61686	30638	33950
Cls-8	168823	131903	122231	122192	110213	53428	50815

The global *mean-mean* throughput row vector shown below is derived from the average throughput matrix and is a numerical statistical summary of the global throughput.

The global *mean* average-throughput row-vector as shown in Chapter 6 is as follows:

	Q2	Q3	Q4	Q5	Q6	Q7	Q8
GlobMnThrput	49517	49534	43857	43929	41112	18731	18876

Each of the seven subsections under this section covers the analyses and comparison of results for each of the multiple queuing systems.

7.2.1 Analysis of Result for Two-Queue System

In the two-queue system, one queue was dedicated for inelastic applications and the other queue for elastic applications. Thus we have a dichotomous queuing system. The results of the simulation showed a tremendous improvement in the end-to-end delay results for all the traffic classes when compared with the mono-queuing system. While in the mono-queuing system, the end-to-end delay increased almost linearly with time, the results of the two-queuing system showed that, the end-to-end delay converges to a very low limiting value (Figure 6.1). Delay variation (jitter) for the traffic was very high in the mono-queuing system, whereas in the two-queue system, jitter for the traffic was very low due to the almost constant equilibrium value of the end-to-end delay value. In relating the results to our heuristic analysis, we noted that, in changing from the mono-queuing system to the two-queue (multiple-queue) system, the limiting quantum molecular length of the queue was reduced by fifty-percent. This in effect is a change from the series arrangement to the parallel arrangement, which consequently reduces the *limiting proximity number* (see Section 5.4.1) of all the traffic classes to the service point. (Note that high limiting proximity number is a disadvantage). Some of the arriving packets into the two-queue system will now have to wait approximately half the period of time they would have to wait if they were to arrive into the mono-queuing system before reaching the server. From the results, we noted that, this has the effect of transforming the packets quasi-linear increasing waiting time distribution in the mono-queuing arrangement to a uniform waiting time distribution in the two-queue arrangement. The heuristic analysis predicted that all the traffic will have an improvement in delay performance compared with the mono-queuing system, and this is exactly what the results show.

Although we have recorded a great performance improvement as we change from the mono queuing system to the two-queue multiple queuing system, such great performance improvement is not manifested as we change from the two-queue multiple queuing system to higher-queue multiple queuing systems. Looking closely at the results, classes 7 and 8 traffic have their lowest mean delay with the two-queue system and thus perform best on this queuing system. The global *mean-mean* delay row-vector shows that, the two-queue system takes second position behind the three-queue system as the global optimum for multiple queue systems in this experiment. Also, a very close examination of the global throughput chart and global *mean-mean* throughput row-vector shows that, the two-queue system and the three-queue system have the highest throughput and thus perform best. This makes them eligible as candidates for our choice of optimum number of traffic queuing classes that best support Integrated Services.

The multiple-queue systems (*Class Based Queue* (CBQ)) are non-work conserving systems in that the server has to make state transitions in moving from one queue to another in rendering services. We noted that extra work was created in the server's queue state transitions, which were not solely for serving packets. The wasted work can be quantified by normalising each server's transition from one queue to another to have an absolute magnitude value of 1. The total wasted work can be derived from the server's state transition matrix or from the transition diagram. The structure of the two-queuing system and its server's transition matrix are illustrated in Figure 7.4 (a) and (b). The convention adopted for numbering the multiple-queue systems here is that, the queue number starts from 0. Thus in the two-queue system, we have queue-0 and queue-1.

The server's state transition matrix is formed by allocating 1 when there is a direct transition of server from one queue to the other, and where there is no direct transition, 0 is allocated. Thus the server's transition matrix for the two-queuing system is as follows:

	To q0	To q1
From q0	0	1
From q1	1	0

If we denote the server queue state transition matrix by **A**, thus

$$\mathbf{A} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

The *Euclidean or Frobenius Norm* of matrix **A** is given by the following relation:

$$||A|| = \sqrt{\sum_{J=1}^{n} \sum_{k=1}^{n} c_{jk}^{2}} = 1.414$$

The work wasted has a hypothetical nominal absolute value of 1.414. This is highly negligible based on the simulation results, which show impressive performance leverage compared with the mono-queuing system.

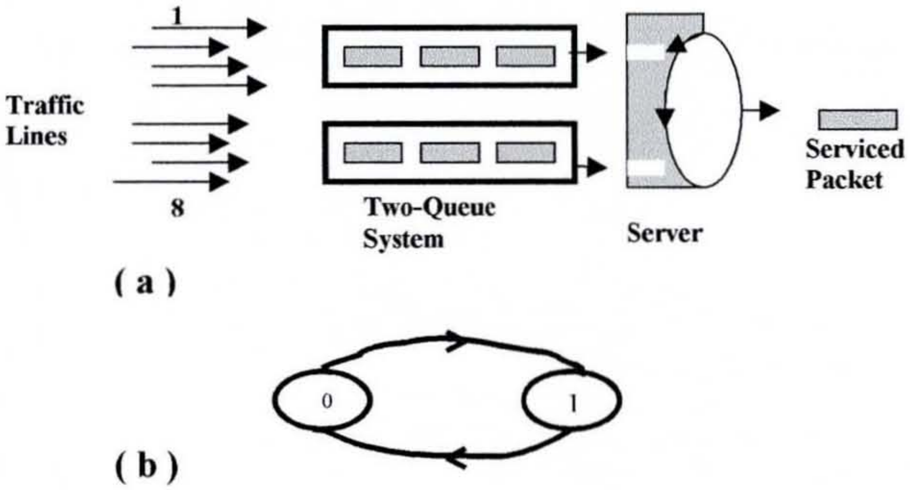


Figure 7.4: (a) Two-Queue system servicing structure
(b) Server Queue State Transition Diagram (SQSTD)

7.2.2 Analysis of Results for the Three-Queue System

The results of the three-queue system show that there is no great difference between its performance and the two-queue system for both end-to-end delay and throughput. The results show slight improvements in the end-to-end delay and throughput for traffic classes 1 to 4, but lower performance in the end-to-end delay and throughput for traffic classes 5 to 8. Global throughput for the three-queue system and the two-queue system are almost the same as recorded by the results of the simulation. The results of the global end-to-end delay and global throughput show that, *the three-queue system has the best overall performance and hence is the candidate for the optimum number of traffic queuing classes to support Integrated Services.*

The heuristic analysis predicted that, there would be improvement in the delay of traffic classes 1 to 4, but for others, the delay would remain constant. The prediction is quite correct up to the limitation of the heuristic analysis as the simulation results revealed above. The limitation of the prediction concerns the inability of the heuristic analysis to define degraded performances.

The non-work conserving nature of the three-queue system is illustrated in Figure 7.5 (a) and (b), which show the queue servicing arrangement and the server's state transition diagram.

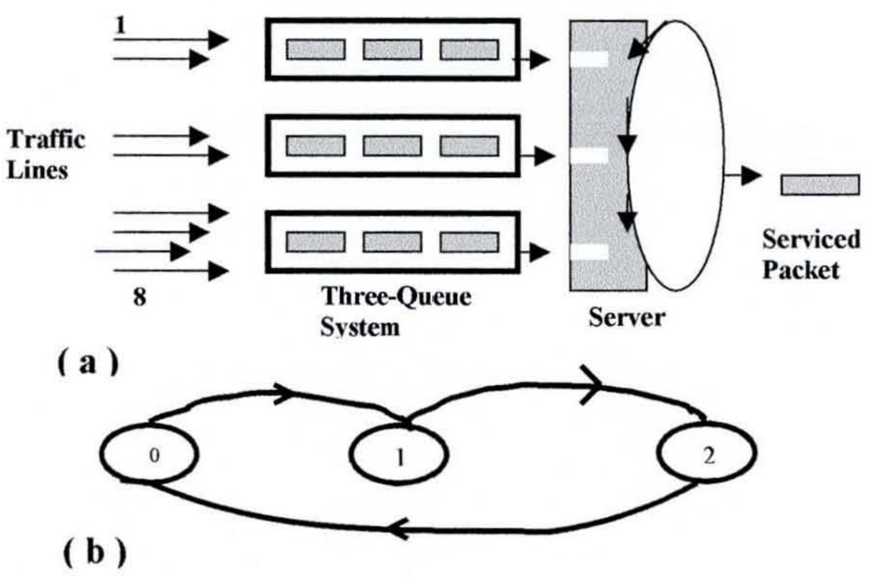


Figure 7.5: (a) Three-Queue System Servicing Arrangement
(b) Server's Queue State Transition Diagram

The server queue state transition matrix is defined as:

$$B = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

The *Euclidean Norm* of matrix B is given by the relationship that follows:

$$||B|| = \sqrt{\sum_{J=1}^{n} \sum_{k=1}^{n} c_{jk}^{2}} = 1.732$$

The work wasted as a result of non-conserving nature of the queue has a hypothetical nominal absolute-value of 1.732, which shows some increase in wasted work by the server, compared with the two-queue system.

7.2.3 Analysis of Results for the Four-Queue System

The simulation results for this queuing system as shown in the global delay and throughput charts of Figure 6.2 and Table 7.1 reveal that there is little improvement in the end-to-end delay for traffic class 1 when compared with the three-queue system. The results revealed that other traffic classes yield lower performance compared with the three-queue system. Throughput for traffic class 1 is the same for the four-queue and the three-queue systems, while for the other traffic classes, the results showed that throughputs are lower in the four-queue system compared with the three-queue system. The four-queue system performs better than higher-queue systems for traffic classes 1 to 4 with a few exceptions as shown in the global delay charts in Section 6.2 and global throughput matrix in table 7.1. Traffic classes 5 to 8 perform better in some of the higher-queue systems than in the four-queue system. The trends in changes in performances of applications in relation to the queue systems are summarised in the global *delay* and throughput row vectors.

From the heuristic analysis, we noted that it was only traffic classes 1 and 2 that have their *limiting quantum queue molecular lengths* reduced by changing from the three-queue system to the four-queue system. The heuristic analysis prediction is in line with the results of the simulation except that the heuristic analysis could not predict the degrading performance of some of the traffic classes as commented earlier. Figures 7.6 (a) and (b) are used to illustrate the service arrangement of the four-queue system and the server's queue transition diagram.

The server queue state transition matrix is defined as:

$$C = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

The *Euclidean Norm* of matrix C is given by the relationship that follows:

$$||C|| = \sqrt{\sum_{J=1}^{n} \sum_{k=1}^{n} c_{jk}^{2}} = 2$$

The work wasted as a result of non-conserving nature of the queue has a hypothetical nominal absolute-value of 2. The pattern of the results is beginning to show the larger this value the less efficient the related multiple queuing system becomes.

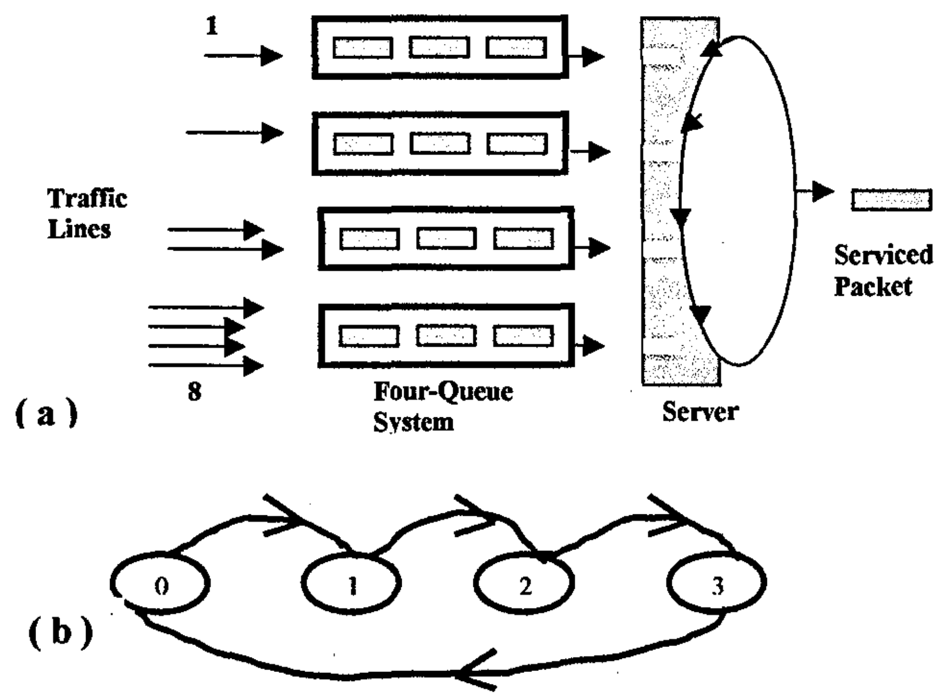


Figure 7.6: (a) Service Arrangement of Four-Queue System
(b) Server Queue State Transition Diagram

7.2.4 Analysis of Results for the Five-Queue System

Simulation results (see Sections 6.2.2 to 6.2.9 and Table 7.1) for this queuing system showed there are improvements in the end-to-end delay performances for traffic classes 4 to 8 when compared with the four-queue system. The results revealed slight improvements in the performances of traffic classes 1 and 2 and a degraded performance of traffic class 3 when compared with the four-queue system. Comparing performances of the five-queue system with higher-queue systems, the results revealed end-to-end delay degraded performances for all traffic classes on all higher-queue systems, except traffic class 6 in all higher-queue systems and traffic class 5, 7 and 8 on the six-queue system. Throughput for traffic classes 3, 5 and 7 also show some improvement in the five-queue system compared with the four-queue system. Throughput results for the five-queue system are generally better or almost the same for all traffic classes when compared with higher-queue systems except traffic class 6 in all higher-queue systems and traffic class 5 on the six-queue system where the throughput results show better performances than the five-queue system.

From the heuristic analysis, we noted that, traffic classes 3 and 4 have their *limiting quantum queue molecular length* reduced by changing from the four-queue system to the five-queue system. The results as highlighted above are close to the anticipated results from heuristic analysis.

Figure 7.7 (a) and (b) are used to illustrate the service arrangement of the five-queue system and the server's queue state transition diagram.

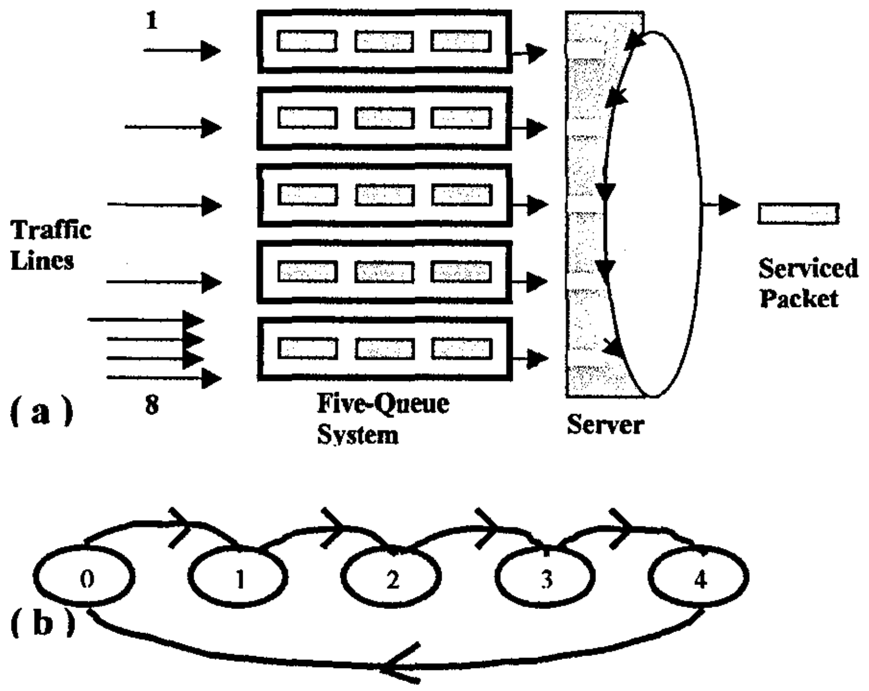


Figure 7.7: (a) Service Arrangement of Five-Queue System
(b) Server Queue State Transition Diagram

The server queue state transition matrix is defined as:

$$D = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The *Euclidean Norm* of matrix D is given by the following relation:

$$||D|| = \sqrt{\sum_{J=1}^{n} \sum_{k=1}^{n} c_{jk}^{2}} = 2.236$$

The work wasted as a result of non-conserving nature of the queue has a hypothetical nominal absolute value of 2.236. The pattern of the results shows that, the efficiency of the multiple queuing system reduces as this value increases.

7.2.5 Analysis of Results for the Six-Queue System

Simulation results (see Sections 6.2.2 to 6.2.9 and Table 7.1) for the six-queue system showed that, there are improvements in the end-to-end delay for traffic classes 5 to 8, and degraded performances for traffic classes 1 to 4 when compared with the five-queue system. Comparing the six-queue system with higher-queue system on end-to-end delay, the results showed degraded performances for higher-queue systems on all traffic classes except traffic class 6. The results also revealed that the six-queue system has improvement in throughput performances for traffic classes 5 and 6 and almost equal performances in throughput for traffic class 1, but degraded performances in others for throughput when compared with the five-queue system. Compared with higher-queue systems, the six-queue system shows better performance in throughput results in all traffic classes except traffic class 6.
From the heuristic analysis, it can be seen that, traffic classes 5, 6 7 and 8 have their *limiting quantum queue molecular length* reduced by changing from the five-queue system to the six-queue system. Thus the results as highlighted above are in line with the anticipated results from the heuristic analysis.
Figure 7.8 (a) and (b) are used to illustrate the service arrangement of the six-queue system and the server's queue state transition diagram.

The server queue state transition matrix is defined as:

$$E = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The *Euclidean Norm* of matrix E is given by the following relationship:

$$||E|| = \sqrt{\sum_{J=1}^{n}\sum_{k=1}^{n} c_{jk}^2} = 2.449$$

The work wasted as a result of the non-conserving nature of the queue has a hypothetical nominal absolute value of 2.449. The inefficiency in the multiple queuing system with this value is shown in the degrading performances of the inelastic applications.

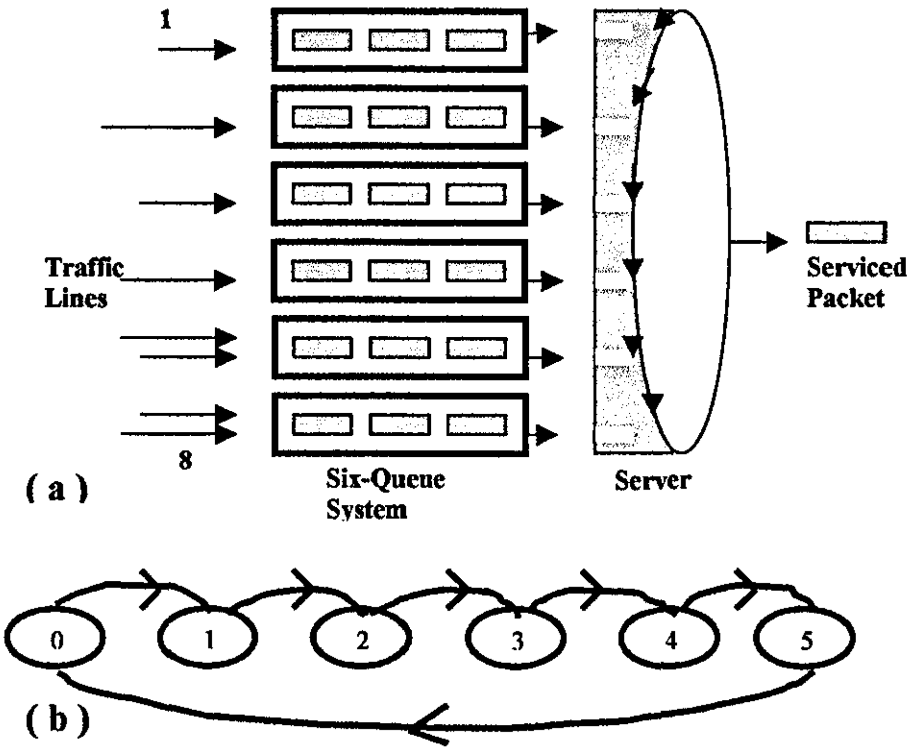


Figure 7.8: (a) Service Arrangement of the Six-Queue System
(b) Server Queue State Transition Diagram

7.2.6 Analysis of Results for the Seven-Queue System

Simulation results for the seven-queue system showed that, there are improvements in both the end-to-end delay and throughput for only traffic class 6 when changing from the six-queue system to the seven-queue system. Other traffic classes recorded degradation of performance in the seven-queue system compared with the six-queue system in both end-to-end delay and throughput. The seven-queue system performs poorly compared with the eight-queue system for all traffic classes on end-to-end delay, except traffic classes 1 and 6. Results for the seven-queue system show little improvement in throughput for traffic classes 1 and 8, equal performance in traffic class 6 and degraded performances in others when compared with the eight-queue system.

The results as shown above are very close to the anticipated results from the heuristic analysis.

Figure 7.9 (a) and (b) are used to illustrate the service arrangement of the seven-queue system and the server's queue state transition diagram.

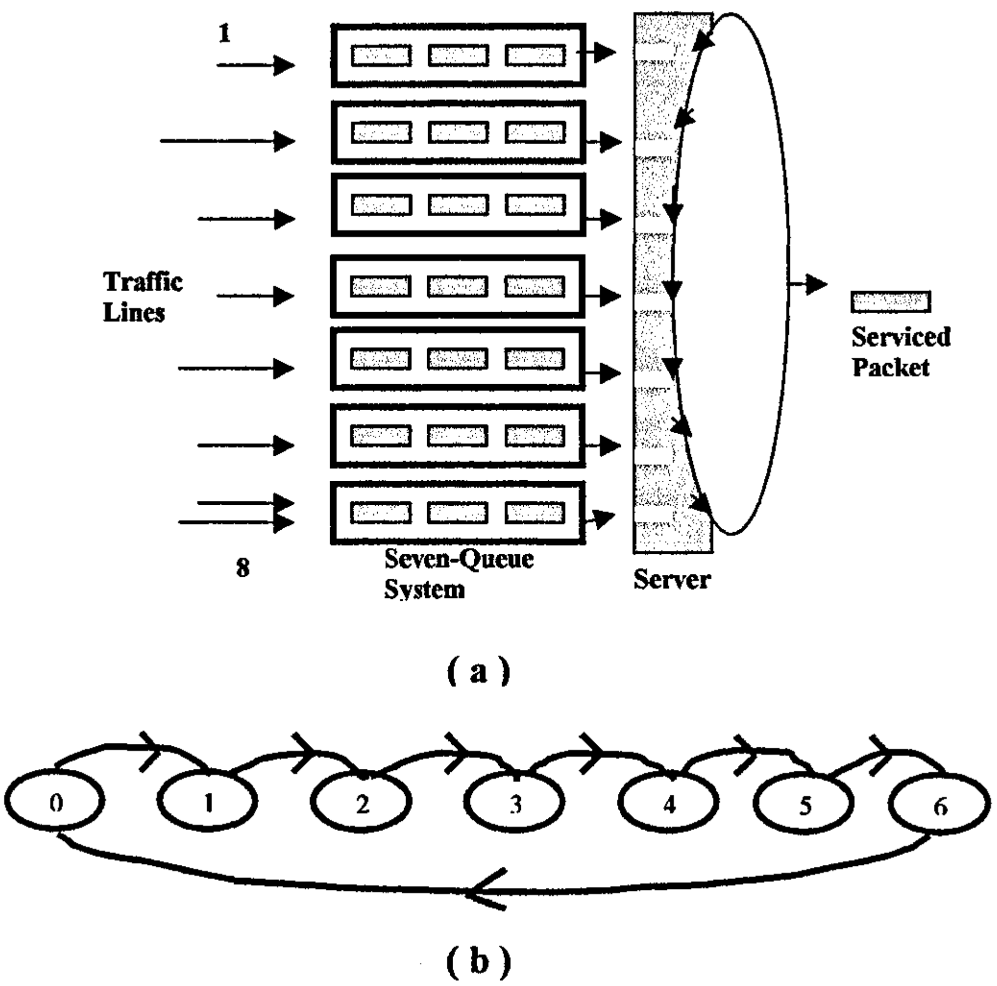


Figure 7.9: (a) Service Arrangement of Seven-Queuing System
(b) Server Queue State Transition Diagram

The server queue state transition matrix is defined as:

$$F = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The *Euclidean Norm* of matrix F is given by the relationship that follows:

$$||F|| = \sqrt{\sum_{J=1}^{n} \sum_{k=1}^{n} c_{jk}^2} = 2.645$$

The work wasted as a result of the non-conserving nature of the queue has a hypothetical nominal absolute value of 2.645. The inefficiency in the seven-queue system, having this value as the server's wasted work, is manifested in the degrading performances of most all the traffic classes.

7.2.7 Analysis of Results for the Eight-Queue System

Simulation results for the eight-queue system showed there are improvements in the end-to-end delay performances for traffic classes 2, 3, 4, 5, 7 and 8, when compared with the seven-queue system. Throughput improvement performances in the eight-queue system are also recorded in traffic classes 2, 3, 4, 5 and 7, equal performance in traffic class 6, but slightly lower performances in traffic classes 1 and 8 when compared with seven-queue system.

The results are not far from the anticipated result from the heuristic analysis. The difference is that, the heuristic analysis did not define degraded performances.

Figure 7.10 (a) and (b) are used to illustrate the server's queue state transition diagram and the service arrangement of the eight-queue system.

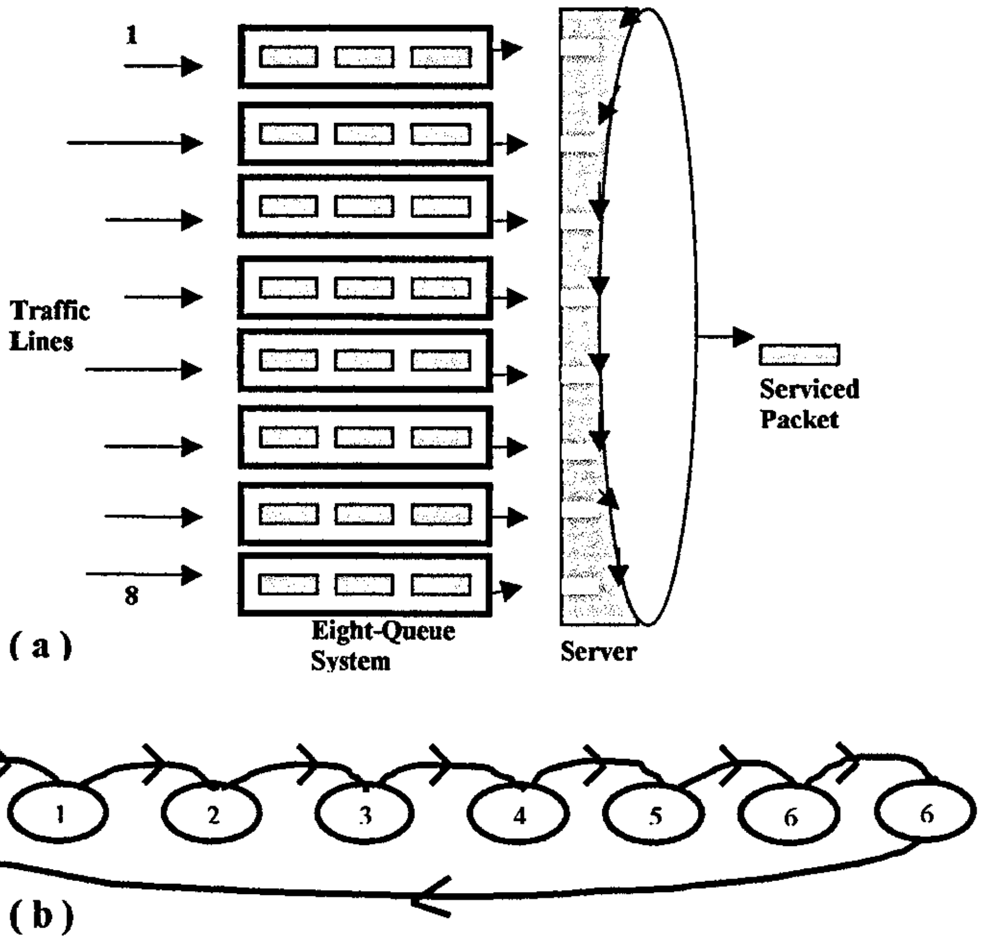


Figure 7.10: (a) Service Arrangement of the Eight-Queue System
(b) Server Queue State Transition Diagram

The serve queue state transition matrix is defined as:

$$G = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The *Euclidean Norm* of matrix G is given by the relationship that follows:

$$||G|| = \sqrt{\sum_{J=1}^{n} \sum_{k=1}^{n} c_{jk}^2} = 2.828$$

The work wasted as a result of the non-work conserving nature of the queue has a hypothetical nominal absolute value of 2.828. The inefficiency in the eight-queue system having this value as the server's wasted effort is revealed in the degrading performances of most of the traffic classes.

7.3 Combined Optimisation Objective

The optimisation objectives for this empirical investigation are:

- Minimise end-to-end delay (Globally)
- Maximise throughput

We now derive a method for combining the optimisation objective functions into one objective criterion to select the *optimum number of the multiple queue system*.

Let X represent a functional system to be minimised.

The following are true mathematically.

Maximise $-X$ = Minimise X
Maximise X^{-1} = Minimise X

If our optimisation objective functions are:
Minimise X and
Maximise Y then,
The optimisation objective functions could be combined into one objective function as follows:
(a) $X^{-1} + Y$ or Y / X for single maximisation criterion.
(b) $X + Y^{-1}$ or X / Y for single minimisation criterion

We now apply the optimisation procedure of (a) to the global *mean-mean* delay row vector and the global *mean-mean* throughput row vector to obtain a single criterion for selecting the optimum number of multiple queue system to support Integrated Services.

	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Mean	5.384	4.399	6.818	6.417	5.616	11.308	9.712

	Q2	Q3	Q4	Q5	Q6	Q7	Q8
GlobMnThrput	49517	49534	43857	43929	41112	18731	18876

Single criterion maximisation row-vector is as follows:

Q2	Q3	Q4	Q5	Q6	Q7	Q8
9197.07	11260.29	6432.53	6845.72	7320.51	1656.44	1943.57

From the single criterion maximisation row-vector, the highest value in the *vector's component (the vector norm)* indicates the *optimum number* of the traffic queuing classes. **The three-queuing system has this value, and thus it is our candidate for the optimum number of traffic queuing classes to support Integrated Services.**

Summary

Analysis of the results of the simulation experiment, to determine the optimum number of traffic priority queuing classes that will best support Integrated Services has been presented in this chapter. The presentation includes comparing the results of the various queuing systems with one another to ascertain superior performances among them. The results show that, the three-queue multiple queuing system, has the best global performances in terms of both end-to-end delay and throughput. Thus it is our candidate for the optimum number of traffic priority queuing classes to support IP convergence.

CHAPTER 8

Predeterministic Distributed Event Response Resource Management (PDERRM)— a novel IP QoS Architecture

The need for a new simple, elegant and robust IP QoS architecture was highlighted in Chapter 1 Section 1.7.2. Having discussed "Generic Components of QoS Architecture" in Chapter 3, and work on its prime member— "Queuing and Scheduling Discipline, which was presented in Chapters 4 through Chapter 7, we now focus on the new IP QoS architecture in this chapter. Thus this chapter is concerned with the discussion and presentation of the novel IP QoS architecture— *Predeterministic Distributed Event Response Resource Management* (PDERRM) which is an improved QoS architecture for QoS delivery in multiservice IP Networks. The motivation for its work, the design and its framework will be presented in this chapter.

The discussion in Section 8.1 will be focused on the motivation for the novel IP QoS architecture in relation to the complexities of the emergent standard IP QoS architectures which are *Integrated Services* (IntServ) and *Differentiated Services* (DiffServ) architectures. The PDERRM architecture and its components will be presented in Section 8.2. The requirements of a *sending and receiving host* are briefly presented in Section 8.3, and the requirements of a *forwarding node* are covered in Section 8.4.

8.1 Motivation for PDERRM Project

Although most of the discussions in this section are covered in Chapter 1, it is necessary to emphasise the points here for the sake of completeness.

Internet Protocol (IP) Quality of Service (QoS) has been a subject of active research and standardization during the past two decades. IP convergence— the convergence of circuit-switched, packet-switched and other multimedia networks has very great appeal. The advantages are numerous. It offers cost savings through technological exploitation and consolidation and cost savings in industrial growth through creation of new services. An effective and functional IP QoS architecture has been identified as the key driving force for IP convergence. Despite the existence of the two standard QoS architectures and the large volume of IP QoS mechanisms that abounds in Communication Engineering literature, end-to-end deployment of QoS is still elusive in the Internet **[Gio et al.03]**.

IntServ **[RFC 1633]** and DiffServ **[RFC 2475]** are the two IETF developed IP QoS architectures that have emerged as standards. IntServ with its signalling protocol—RSVP **[RFC 2205, RFC 2210]**, support per-flow explicit signalling of applications QoS requirements to the network, and per-flow resource allocation on *Network Elements* (NE) along the path of flow end-to-end. It has been widely reported that the per-flow resource-allocation paradigm of IntServ-RSVP architecture could create scalability problems in core Internet routers with gigabit traffic flows. The problem is associated with the complexities involved in maintaining *per-flow states* for very large number of flows in the routers concerned. Beside the scalability problems in core Internet routers, recent experiments carried out at Loughborough University showed that forwarding nodes in *edge (domain) network* degrade in performance for all round traffic classes as the number of resource partitioned classes increased, and resource allocations increasingly tend towards per-flow. Thus the complexities and scalability problems of IntServ-RSVP per-flow paradigm do not reside only in core routers but also in *leave nodes* that handle medium large number of per-flow resource allocation.

The DiffServ architecture was designed to serve as amelioration to the complexities and scalability problems of IntServ-RSVP paradigm. Thus DiffServ has simplification built into its flow aggregation resource allocation mechanism compared with the per-flow end-to-end resource allocation of the IntServ-RSVP paradigm. Despite the heralded simplicity and flexibility of the DiffServ paradigm, it still has its own touch of complexity. Complex traffic conditioning processes have to take place at the edges of each network domain before packets are forwarded. The complex traffic classifications, metering, policing and traffic rate limiting processes required of the DiffServ architecture may not be seamlessly practicable in reality throughout the diverse network nodes processing capabilities of the diverse conglomeration of networks that make up of IP Networks.

The phenomenal success of the Internet has been attributed to it technological structure in which complexities reside in the *end-nodes* and the network is relatively simpler. This is in contrast to the traditional public switched telephone network

(PSTN) in which complexities reside in the network while *end-sets* are relatively simpler. Any extension to the service model of the Internet in the form of QoS architecture and protocols must take the factor of simplicity into consideration for reasons of scalability and ease of deployment. We observed that many aspects of the *generic components* of QoS architectures discussed in Chapter 3 could be manipulated and exploited in order to evolve an IP QoS architecture that will be simple, elegant and flexible for wide spread implementation and deployment. PDERRM has been designed with simplicity, elegance and flexibility in mind.

8.2 PDERRM Architecture and Its Components

In this section, the following will be discussed:

- Principle of Operation
- Features of the Architecture
- The Architecture
- Components of the Architecture

8.2.1 Principle of Operation

PDERRM QoS architecture is simply a mechanism in which every *host* (end node) in the network has a *Traffic Controller* and every *forwarding node* has a *QoS Manger* that dynamically reacts to network loading based on pre-deterministic sharing of resources in the network. The network resource sharing parameters will have values with local and global significance that are consistently understood by all grades of nodes in the network. The nature of the reaction to network loading by a QoS Manager (or Traffic Controller) in a node would be determined by events relating to traffic flows utilisation of resources within the node and traffic loading in the network. Traffic *sources* rate-limit traffic injected into the network in accordance to the traffic classes resource share or quota. *Forwarding Devices* (FD) employ a novel *Stochastic-Gap Jumping Window* (SGJW) algorithm whose distribution is a reflection of *traffic bursts* in the network, in scanning traffic in transit for *admission and policing* purposes. This process ensures traffic flows comply with either local or global resource sharing parameters adopted in the network and enhance throughput plus ensure scalability. PDERRM has flexible network resource federation management, which allows for dynamic shifting of resource allocation quota of traffic classes, based on realities of real-time network loading. The simple basic concept of the QoS architecture resides in the functionality of the QoS Managers in each FD that ensures an equilibrium between network resource capacity of the node and traffic resource utilisation within the node never tilts toward a *deficit*. The simple diction of the architecture is "*cut your coat according to your size*". The resource federation management in the nodes has *agents* that dynamically investigate the actual resource requirements of each class of applications. Resources are then allocated equitably among the various classes of applications contending for network resources in accordance with their QoS requirements, and within the resource capacity of the node and the network. Nodes periodically probe the network to determine if a *congestion spot* exists with minimal overhead before injecting traffic to the network or admitting and forwarding traffic. Each network-forwarding device has a *maximum fixed number of concurrent flows* in each *measuring window* that can be admitted for *each class of application traffic* based on its resource capacity. When the QoS Manager in a node discovers that the maximum fixed number of concurrent flows for a class of application traffic has been reached, it will cause the node to multicast a *traffic control message* to neighbouring nodes informing them of its current state of resource utilisation. As long as the *maximum resource utilisation state* of the *traffic class* exists in the node, neighbouring nodes will not admit traffic that belongs to that class which must be routed through the node that generated the traffic control message. As soon as the situation changes and the number of concurrent flows for the traffic class is less than the maximum allowed for the window, the node concerned will generate a multicast message to neighbouring nodes to provide an update of the situation. It must be understood here that the control functionality of the QoS Manager operates on *quantitative bounded control* of traffic flow, it does not bring flow down or push it up. It ensures an equilibrate between traffic flow and its assigned flow quota.

Resources are generally allocated on a per-class basis, employing a value of DiffServ Codepoint (DSCP) for implicit signalling of the resource requirements. When there is the need for per-flow resource allocation, a DSCP value is still used for implicit signalling of the resource requirements, but in this case, the DSCP value is mapped or hashed into the corresponding value in the node's *QoS Parameter Database* (QPD) or *Repository*. The *Queuing and Scheduling Discipline* (QSD) of the node queries the QPD in allocating resources for the traffic flows.

8.2.2 Features of the Architecture

PDERRM has good features, which make it elegant and robust for multiservice QoS in IP Networks, in view of diverse processing capabilities of IP *network elements* (NEs). The various features include the followings:

- Hybrid functionality in terms of aggregate flow and per-flow resource allocations.
- Adaptation functions regarding meeting the dynamic QoS requirements of flows.
- Mutational or flexible configuration designed to adapt to diverse processing capability of *network elements.*
- Ability to shed processing functions to the simplest level of QoS requirements of applications.
- Through administrative control, a node can elect to operate on either *local* or *global* resource brokerage.
- Use of implicit signalling for per-flow resource allocation.

We now briefly discuss each of these features.

8.2.2.1 Hybrid Functionality

PDERRM could function as a DiffServ architecture, in view of its support for aggregate flow resource allocations. But unlike the DiffServ architecture where there is the need for complex traffic conditioning processes at the edges of the network, PDERRM adopts a simpler approach where the QoS Managers of nodes at the network edges simply ensure that there is a balance or surplus budget between resource capacity and resource utilisation.

PDERRM also could function as an IntServ architecture, but unlike IntServ where RSVP generates *explicit signalling* for per-flow resource reservation for application QoS needs to the network, PDERRM adopts *implicit signalling* using a value of DSCP to signal application's QoS needs to the network. The DSCP value is then mapped into the corresponding value in the QPD in the node concerned for resource allocation. There is no need for routers to keep large amounts of *flow-state* information. That functionality is taken over by QPD, which contains a wide range of *token bucket parameters*, which can meet the diverse QoS requirements of applications.

8.2.2.2 Adaptability Functions in Resource Allocation

On initialisation, each node's QoS manager allocates resources to each class of traffic based on *static resource brokerage*, which may be derived from several factors including administrative policy. During flows, the QoS Manager's background processes monitor traffic loading for each traffic class and also estimate resource utilisation for each of the traffic classes. The QoS Manager could be configured to shift the static resource percentage share of each traffic class to reflect the dynamic network loading of the traffic class if it will not be detrimental to the QoS needs of other traffic classes. Thus the resource federation management of each node in the network has built-in functionality to dynamically adapt the resource quota of each traffic class to ensure equitable resource allocation that will meet the needs of the various classes of application contending for network resources.

8.2.2.3 Mutational or Flexible Configuration

In view of the fact that IP Networks are made from a global conglomeration of networks with diverse *network elements* (NE), the network nodes will have various degrees of processing power and diverse abilities to cope with complex processing. Resource availability in terms of buffer space and bandwidth will vary from one network domain to another. In some geographical regions, many of the network domains may still be operating on *legacy devices.* In order to meet the diverse needs of the conglomeration of networks that comprise IP Networks, PDERRM is designed to be independently operative at different degrees of processing power ranging from simple low level processing power to high level sophisticated processing power. A node with PDERRM-aware for example, could be configured for *coarse granular resource allocation* as found in DiffServ aggregate flow resource allocations, when the QoS needs of generality of all traffic classes are concerned. On the other hand, a PDERRM-aware node could be configured *for fine granular resource allocation* as found in IntServ per-flow resource allocation when the QoS needs of a particular flow is of interest.

8.2.2.4 Ability to Shed Processing Functions

In both aggregate flow resource allocation mode and per-flow resource allocation mode, the PDERRM processing level can be reduced to a minimum based on the capacity of the node and the QoS level required. The QoS level required might be a little-better-than-best-effort and the node involved may be a legacy node with very little processing power, in either case, PDERRM can be configured for reduced resource processing and operate independently of other available modes.

8.2.2.5 Adaptive Resource Brokerage Policies

The resource quota brokerage parameters for each class of traffic may have local significance or global significance. PDERRM has characteristics that made it amenable for easy shift between the two contexts of resource share brokerage parameters.

8.2.2.6 Implicit QoS Signalling

One of the heralded problems of the IntServ-RSVP paradigm lies with the message overhead of the explicit per-flow QoS signalling of the applications needs in the network. The message overhead is removed by PDERRM with the use of a value of DSCP, which is then mapped to a set of token bucket parameters in the node's QPD to indicate the resource reservation, which the application needs. The admission of a flow by all NEs along the flow-path indicates the reservations are accepted.

8.2.3 PDERRM Architecture

The PDERRM architecture, just as in other IP QoS architectures consists of component building blocks, which are partitioned into two groups— the *control plane and data path plane.* The control plane consists of background processes or daemons which co-ordinate resource allocation and utilisation in the node, while the data path plane form the executive wing which translate QoS resource policies into outwardly perceived QoS performance behaviour of applications. Basically, the QoS Manager and the Traffic Controller are considered equivalent, thus the term *traffic controller* will be left out in further consideration of the architecture. The component building blocks include the following:

- QoS Manager
- Classification, Resource Capacity & Quota Brokerage
- Traffic Load and Session Measurement

Control Plane

- Traffic Load Signalling
- QoS Parameter Database

} Control Plane Conts.

- Traffic Classifier and Admission Control
- Traffic Shaper
- Queuing and Scheduling Discipline

} Data Path Plane

The PDERRM architecture is illustrated with Figure 8.1. The structure shown in Figure 8.1 is typical, some components may be absent or vary in their level of processing, depending whether the node is a host or forwarding device. The components of the control plane are made of background processes, which control the functionality of the objects of the data path plane. QoS policies of the architecture are formulated within the control plane and used to parameterise the functionality of the data path plane.

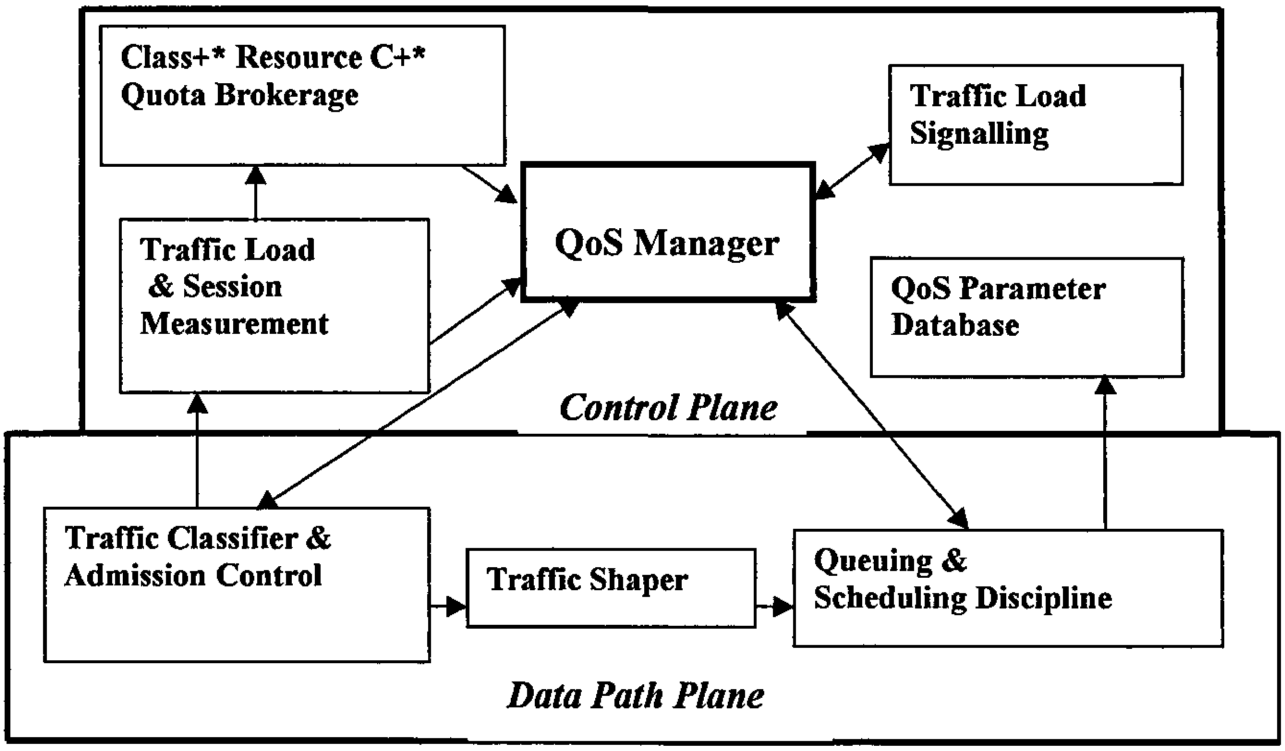

Class+* Resource C+* Quota Brokerage
Traffic Load Signalling
QoS Manager
Traffic Load & Session Measurement
QoS Parameter Database
Control Plane
Traffic Classifier & Admission Control
Traffic Shaper
Queuing & Scheduling Discipline
Data Path Plane

Figure 8.1: Pre-deterministic Distributed Impulse Responce Resource Management (PDERRM) architecture.

"" *In the Figure, **Class+** to be read as **Classification** and **C+** to be read as **Capacity**""

The data path plane serves as the executive branch of the QoS architecture where QoS policies are translated into feasibly observed behaviour of application QoS performances.

Upon initialisation of a node, the *Classification Resource* Capacity and *Quota Brokerage* module of the background processes will pass on information on pre-determined resources quota values of application classes to the QoS Manager. The information will include the value of static percentage share of resources for each class of application, the priority of the application class, and policies on conditions for allowed dynamic shift on static percentage share of resources for each class of application. The QoS Manager in turn will cause the *indication_primitives* of the initial static *quota_resource_brokerage* of each of the application classes to be generated and sent to the Queuing and Scheduling module in the data path plane. The initial operation of the Queuing and Scheduling module will be based on the pre-determined percentage share of resources for each class of applications. While this mode of operation is on, and traffic flows are in progress, the *Traffic Load and Session Measurement* module, which is a daemon process, will perform two related functions. First, it will continue to scan traffic input to the node to determine traffic loading, intensities and bursts. Secondly, the module monitors the flow sessions by recording the start and end of each flow session, and taking the cumulative number of flows of each flow class in each *active window*. The active window is the length of time in which intensities of flows for each of the traffic classes are computed or measured. The active window is spatially distributed in time and the measurement therein forms the basis on which the admission control decision is made. The information on traffic loading, intensities and bursts are sent to the QoS Manager on a regular basis.

Depending on the intensities of traffic loading of each class of application in the node, and the policies guiding the condition for shift in resource quota of each class of application, the QoS Manager could dynamically pass instructions to the Queuing and Scheduling module to vary the resource quota for each class of application. This would be the case if such a shift will support equitable resource allocation to each of the application classes and also support acceptable QoS performances of all the application classes. The Session Measurement processes are very simple. They will be described in a later section.

When a particular class of traffic has exceeded its own resource quota, the QoS Manager would cause the *Signalling* module to multicast the message, *class_i_ exceed_quota* ($0 \leq i \leq N$ = *number of classes*) to neighbouring nodes. This will prevent the neighbouring nodes from sending traffic in that class to the node that generated the message during the time the message is valid. As soon as the situation is reversed, the node will generate a counter message, *class_i_within_quota* to the neighbouring nodes.

Applications are classified into groups— the *inelastic* (real-time applications), *inelastic tolerant* and *elastic* (non-real-time applications). A DSCP value will then be assigned to each group of applications. Thus application traffic classification and identification will be based on DSCP values. Since DSCP values could be grouped for effective management, we propose a DSCP group value with hierarchical section values that can be used to classify and identify application traffic for various PDERRM adaptations QoS processing needs. Each member of a class of application traffic may require coarse granular (aggregate flow) resource allocations or fine granular (per-flow) resource allocations as the case may be. A DSCP value with hierarchical value context could be used to determine which class to which the application traffic belongs and whether the application traffic requires aggregate flow resource allocation or per-flow resource allocation.

Admission Control performs the normal function of limiting traffic flows within the resource capacity of the network. In the design of the admission control, we have adopted a novel strategy in order to achieve scalability. This will be discussed in a later section.

The *Traffic Shaper* ensures the flows temporal profiles will not temporally over-flood or over-stretch the *buffer* and *bandwidth* resource capacities.

Currently PDERRM can make use of any of the available *Queuing and Scheduling* disciplines that are suitable for multiservice operation such as the variants of Class Based Queueing (CBQ) and Weighted Fair Queuing (WFQ) disciplines. The main experiments made use of Weighted Round Robin (WRR) and Deficit Weighted Round Robin (WDRR) queuing-scheduling disciplines, which are variants of CBQ. Research could continue on finding *optimum queuing discipline* that will best support IP convergence.

8.2.4 Components Description

As shown in the architecture, the main components of PDERRM are; Traffic Classification Resource Capacity and Quota Brokerage Agent, Traffic Load and Session Measurement, QoS Parameter Database, Traffic Load-limit Signalling, QoS Manager, Flow Identification and Admission Control, Traffic Shaper and Queuing / Scheduling Discipline. Some of these components are optional in some nodes, and some have functional mechanisms that are well known. Those whose functional mechanisms are new, and their interface with other components also are new, will now be discussed.

8.2.4.1 Traffic Classification Resource Capacity and Quota Brokerage Agent

The processes of defining and specifying the *Flow Classification* section of this module will have similarity with such processes as in the DiffServ architecture. It will be based on allocation of DSCP values to each class of traffic and the process of marking packets with the corresponding DSCP values. It will take input mainly from administrative policy control and its output will be forwarded to the QoS Manager.

The *Resource Capacity and Sharing Agent* will contain object's parameter values, such as the node resource capacity parameters (*CPU speed, buffer space, link bandwidth etc*) and load intensities of each traffic class. Relational operations will be performed on the two sets of parameters. It will take inputs from the *Traffic Load and Session Measurement* module in determining traffic intensities and the resource utilisation of each class of traffic. It will relate the traffic loading with node resource capacity, and based on policy guidelines will specify the *percentage share* of resources for each of the classes of applications. The percentage share of resources for each class of flow will be passed to the QoS Manager as an initial static resource quota for each class of application. The processes of this module are scheduled or repeated at an interval dictated by administrative policy control.

8.2.4.2 Traffic Load and Session Measurement Agent

The ***Traffic Load Measurement*** section of this module is responsible for measuring traffic intensities of each class of flows going through the node. By measuring the rate at which each traffic class arrived at the node or by measuring the quantum of cumulative flow of each traffic class in a window, it would be possible to detect if any class of flows exceed their resource quota. Also through this measurement, we can compute the value of resources consumed by each class of traffic in a predefined time interval. The traffic load monitoring module could be achieved in a number of ways. These include: (1) The use of the *Traffic Network Analyser* which could provide values for traffic loading of the node for each class of traffic, at predetermined interval of times. (2) Adaptation of Loughborough University Network Monitoring Systems. The High Speed Network (HSN) research group of the Department. of Electronic and Electrical Engineering of Loughborough University has extensive experience in network monitoring systems. Some of their packages could be adapted for traffic load measuring tools for the PDERRM architecture. (3) The use of Neural Network Algorithm (NNA) and Fuzzy Logic Management (FLM) have found extensive application in many network functional mechanisms. We hope it could find application in our present purpose. The later and the formers would need empirical investigation to verify their applicability or suitability for the present purpose. (4) Direct Computation, which involves scanning the input to the node, and measuring the values of cumulating flows for each class of traffic in each *window*. This was the method used in the experiment to be presented.

The ***Flow Session Measurement*** section of this module has a number of novel mechanisms and it is optional in a node. It has been designed with adaptation features, which make it suitable for wide ranges of nodes with diverse flow processing power. Its functionality is mainly suitable for real-time classes of traffic, but could be used for non real-time traffic as well. It has been structurally designed with self-contained modular processes to meet the diverse needs of nodes. A node will only need to implement one of the two available options. The options are:

- Flow Session Measurement and Control with Flow Session State Table (FSST).
- Flow Session Measurement and Control without FSST.

The main function of the module is to measure the arrival rate of each flow session in each traffic class and to determine if the flows conform to their agreed temporal characteristics and are within their resource quota. We will now discuss the new optional mechanisms.

8.2.4.2.1 Flow Session Measurement and Control with FSST

This sub-module has a defined interface with the QoS Manager that is used to obtain the values of three sets of parameters, and also to obtain instruction on the interval of time to suspend operation. The three sets of parameters are, the number of traffic classes available, the maximum number of concurrent flows allowed for each traffic class in each *widow* (the predetermined interval of time for measurement), and the maximum allowed traffic rate for each flow. Through the same interface, the sub-module will send the measured value of arrival rate of each *flow session* in each of the traffic classes to the QoS Manager. Other functions include, identifying the class and flow session a packet belongs to, computing the cumulative value of concurrent flows for each class of traffic in a window and maintaining a *Flow Session State Table* (FSST). If per-flow resource allocation is required for a flow, this sub-module also interacts with the *QoS Parameter Database* (QPD) to identify the flow session with its corresponding token bucket parameters. It also interacts with the routing module to determine the next hop for the flow, the address of the next hop is then entered against the flow entry in the column provided in the FSST. The functional procedure is as follows:

Upon initialisation, the sub-module establishes connection with the QoS Manager and obtains the necessary parameters for its operations. The *source* of a flow session tagged the first packet of the flow session with a value "*flow_session_start*" before it is released to the network. At the input of each *Forwarding Device* (FD) that accepts the packet, a copy of the packet is made and the copy forwarded to this module. On receipt of the packet, this module increments the value of cumulative concurrent flows for the class of traffic the flow belong to in the current window by 1, and enters the source address, destination address and the port number (the triple) of the packet in its flow session state table. This module will then start with the measurement of arrival rate for the flow with the first packet. Other packets of the same flow session will not carry the "*flow_session_start*" tag, but will be copied and the copy forwarded to this module for the purpose of computing the cumulative arrival rate for the flow in the current window. This process will continue until the end of the flow. When a window ends, the numbers of concurrent flow for each class of traffic are carried over to the next window. Measurements relating to rate of arrival of each flow are reset to zero when a window ends, and measurement starts afresh at the new window. The last packet of the flow will carry the tag with value "*flow_session_end*" from its *source*. Every forwarding device that has an entry for the flow which the packet belongs to will decrement the cumulative concurrent flow value of the flow class in the current window by 1, and delete its entry from the FSST. The flow arrival rate measurement simply involves taking either the cumulative bit or byte count as the packets arrived instantaneously and compare with allowed value in the current window. When any flow exceed its allowed rate a flag is sent to the QoS Manager, which then take corrective measure.

8.2.4.2.2 Flow Session Measurement and Control without FSST

The operation of this sub-module is similar to the previously described sub-module except that it does not keep FSST and may not operate on a flow session basis. It also receives three sets of parameters from the QoS Manager, which are the number of available traffic classes, the maximum number of concurrent flows for each traffic class in each window and the maximum value of traffic rate intensities allowed for each traffic class in each window.

It receives copies of flows from the node-input driver and increments the number of concurrent flow for each traffic class by 1 when a new flow is detected for each traffic class. It sets a flag signal to the QoS Manager when a traffic class exceeds it concurrent flow number. It computes cumulative arrival rates for each class of traffic in each window and compares it with the allowed value to determine if traffic stayed within quota. It signals violation flags to the QoS Manager for any traffic class that does not stay within its traffic rate limit.

8.2.4.3 Traffic Load-limit Signalling Agent

The Traffic Load-limit Signalling Agent takes input from the QoS Manager to generate traffic rate-limit control messages. Depending on the indication received from the QoS Manager, the message could be:

- The multicast message—*class_i_limit_reached* or *class_i_limi_exceed*, ($0 \leq i \leq N$ = number of traffic classes) that would be sent to neighbouring nodes when any of the traffic classes reached or exceed their allowed maximum input flow rate to the node. Neighbouring nodes will not forward flows in the traffic-class concern to the node that generate the message, during the period the message remain valid.
- The multicast message— *class_i_within_limit* would be sent to neighbouring nodes when the situation above is reversed. That is the input flow rate of the traffic class that caused the message above to be generated is now within allowed value and neighbouring nodes could now forward traffic in that class to the node.
- The unicast message—*src_limit_flow_port_i* (*i* = source port number) generated and sent to the source of flow that exceed its allowed flow rate limit to the network.

The Traffic Load-limit Signalling Agent also receives messages from the network in line with the messages highlighted above when generated by other nodes. The messages are forwarded to the QoS Manager which instructs the relevant modules to take necessary action. For example when the message *class_i_limit_reached* is received from a node, the QoS Manager instructs the admission control not to admit packet of the class concerned if they must be routed through the node that generated the message.

8.2.4.4 QoS Parameter Database (QPD)

The QPD is the repository for a wide range of IntServ *Token Bucket Parameters* (TBP) that are designed to meet the QoS needs of diverse applications. Its entries must be globally standardised and its index and corresponding contents must be consistent for all grades of nodes. When a particular application requires per-flow resource allocation, the DSCP value of the application has a section that indicates *value index* to the QPD. On receipt of such application, the *scheduler* maps its DSCP value index to the corresponding location in the QPD and uses the TBP in that location to parameterise the scheduling process of the application flow. This mechanism is meant to remove RSVP message overheads and reduce scalability problems when keeping large flow state informations in routers that are inherent in the IntServ-RSVP resource allocation paradigm. It has to be noted that this approach has an advantage over the *Stateless Core*, which proposed to achieve per-flow resource allocation scalability by putting *states* in the packet header **[Stoi00]**. This will increase traffic flow overhead, since header to payload ratio will be increased, thereby reducing the efficiency of the network.

8.2.4.5 The QoS Manager

The QoS Manager is the intelligence of the node in terms of QoS resource management. It co-ordinates the activities of other QoS related modules in ensuring traffic loading and resource utilisation in the node are within the node resource capacity.

It receives value indications of the static percentage resource share of each traffic class from the Resource Quota Brokerage Agent and passes this on to the Scheduler. The Scheduler uses the value to compute and parameterise the *buffer space* and *bandwidth* allocation for each class of traffic.

It also receives dynamic traffic input loading information from the Traffic Load Measurement module to determine the operational rate at which traffic in each class is loading the node. Based on policy guidelines and operational flow intensities of each class of traffic, the QoS Manager could shift the basic percentage share of resources for each class of traffic to reflect the QoS needs of the traffic classes. The policy input to the QoS Manager could be achieved through some management tools; **[SNMP] [COPS] [Dim et al.03] [Mai et al.03] [Kri04]**. The QoS Manager also uses the information from the Traffic Load Measurement module in determining when to instruct the Admission Control to resume or suspend its operation. It receives information when any of the queue-class sizes reaches its red flag threshold. This will cause the QoS Manager to instruct the Admission Control to limit flow-input rate. The traffic control functionality of the QoS Manager ensures that:

$$\left(\left(\sum_{t=0}^{W}\sum_{n=1}^{N}(p_1 \times s_1)_{F_1} + (p_2 \times s_2)_{F_1} + \ldots\ldots\right) + \left(\sum_{t=0}^{W}\sum_{n=1}^{N}(p_1 \times s_1)_{F_2} + (p_2 \times s_2)_{F_2} + \ldots\ldots\right) + \ldots\ldots\ldots\ldots + \left(\sum_{t=0}^{W}\sum_{n=1}^{N}(p_1 \times s_1)_{F_K} + (p_2 \times s_2)_{F_K} + \ldots\ldots\right)\right) \leq C_{tot}$$

Where W is the measuring window size in seconds, N is the number of concurrent flow allowed for a flow class F_i, $(1 \leq i \leq K)$, K is the number of flow classes available, p is the packets that arrived for each flow class and s is the packet sizes. The number of packets that arrived in a measuring window for a flow class is not known in advance, thus we use ellipses to represent the unknown terms in the cumulative packet summation. C_{tot} is the node's total resource capacity threshold mark-point for traffic limit control action.

The QoS Manager also controls the functionality of the Signalling mechanism module in generating traffic load control messages to the network, and receives load control messages from the network through the Signalling mechanism. Also the QoS Manager may cause the signalling mechanism to generate traffic rate limits to affected nodes.

The QoS manager will normally run as a *Finite State Machine* (FSM), making transitions from one state to another depending on process threading.

8.2.4.6 Admission Control Module

The Admission Control mechanism is novel and simple. It has a random scattered operational time interval, which accounts for its name— *Stochastic-Gap Jumping Window* (SGJW) algorithm. The window width and position (spatial displacements) in time seek the distribution of the traffic bursts or high intensities in the network. Its operation is different from that of Triggered Jumping Window (TJW) algorithm used in ATM networks for policing functions. TJW action is triggered when the first *cell* (ATM packet) arrived, **[Woo96] [DouSin99]**. In the case of SGJW, it periodically seeks information on flows interarrival distribution from the Traffic Load Measuring agent, based on the burst distribution of traffic flows and instruction from the QoS Manager, SGJW could implement simple window policing algorithm or a hybrid of the leaky bucket and windowing algorithm. The mechanism of windowing algorithm involves instantaneously taking cumulative bit or byte count of flows within a measuring window and comparing this with allowed values to determine if flows are in or out of profile. This could be combined with leaky bucket to take care of very bursty traffic. If a flow is out of profile, a number of actions could follow. The flow may be dropped, or marked for eventual action, or shaped. Flow control messages are sent to the source of the flow to limit rate through co-ordination of the QoS Manager, which will direct the *traffic rate-limit-signalling* agent to take action when necessary.

The operation of the admission control could be denoted as:

$$\left[\left(\sum_{i=1}^{K}\sum_{t=0}^{W}\sum_{n=1}^{N}(p_1 \times s_1)_{F_i} + (p_2 \times s_2)_{F_i} + \ldots\right) \leq C_{tot}\right] = \begin{cases} \text{True, Traffic load} \geq C_{tot} \\ \text{False, Traffic load} \leq C_{tot} \end{cases}$$

Where K is number of flow classes, W is the window size, N is number of concurrent flows allowed in flow class F_i $(1 \leq i \leq K)$, p and s are packet and its size, C_{tot} is node's total resource capacity threshold for traffic control action.

8.2.4.7 Traffic Shaper

The Traffic Shaper is conventional and could make use of the *leaky bucket* algorithm. See Section 3.5.2 for information on leaky bucket algorithm.

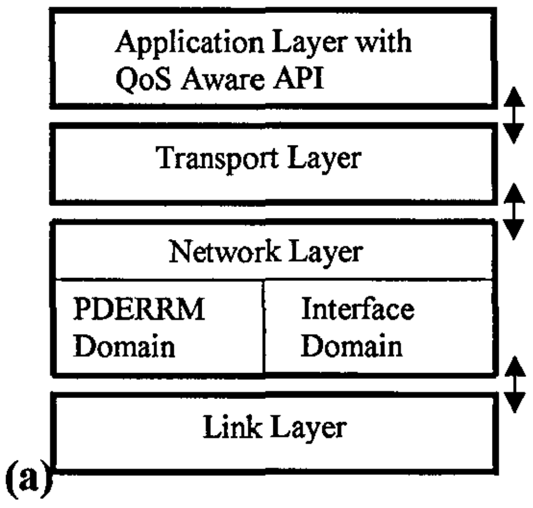
8.3.4.8 The Queuing and Scheduling Discipline

Currently any of the queuing disciplines suitable for multiservice operation could be used with minor modifications. These include CBQ with its variants and WFQ with its variants. The present system made use of the former. The modification required involves the need for the Scheduler to interface with the QPD. Further research needs to be carried out on designing or determining optimum Scheduler for IP QoS.

8.3 PDERRM Host Requirements

PDERRM *End Host* component requirements will now briefly be described. The layer structure is illustrated with Figure 8.2 (a). As shown in the figure, the only addition to normal typical Host is the PDERRM process block at the Network Layer and the need to incorporate QoS aware mechanism into the API at the Application Layer. The function of the added PDERRM module at the Network Layer is to make sure that the rates at which traffic-flows are received do not over-flood the node. And also to ensure that traffic flows are injected to the network in accordance with the traffic rate limit allowed for each class of traffic.

Figure 8.2(b) is the process flow chart that provides illustration of the process flow for QoS related traffic control in an End Host. As depicted in the figure, the *Flow Identifier* identifies which application has the flow and then passes the flow to *Acceptance Control,* which receives input from the *Flow Controller*.



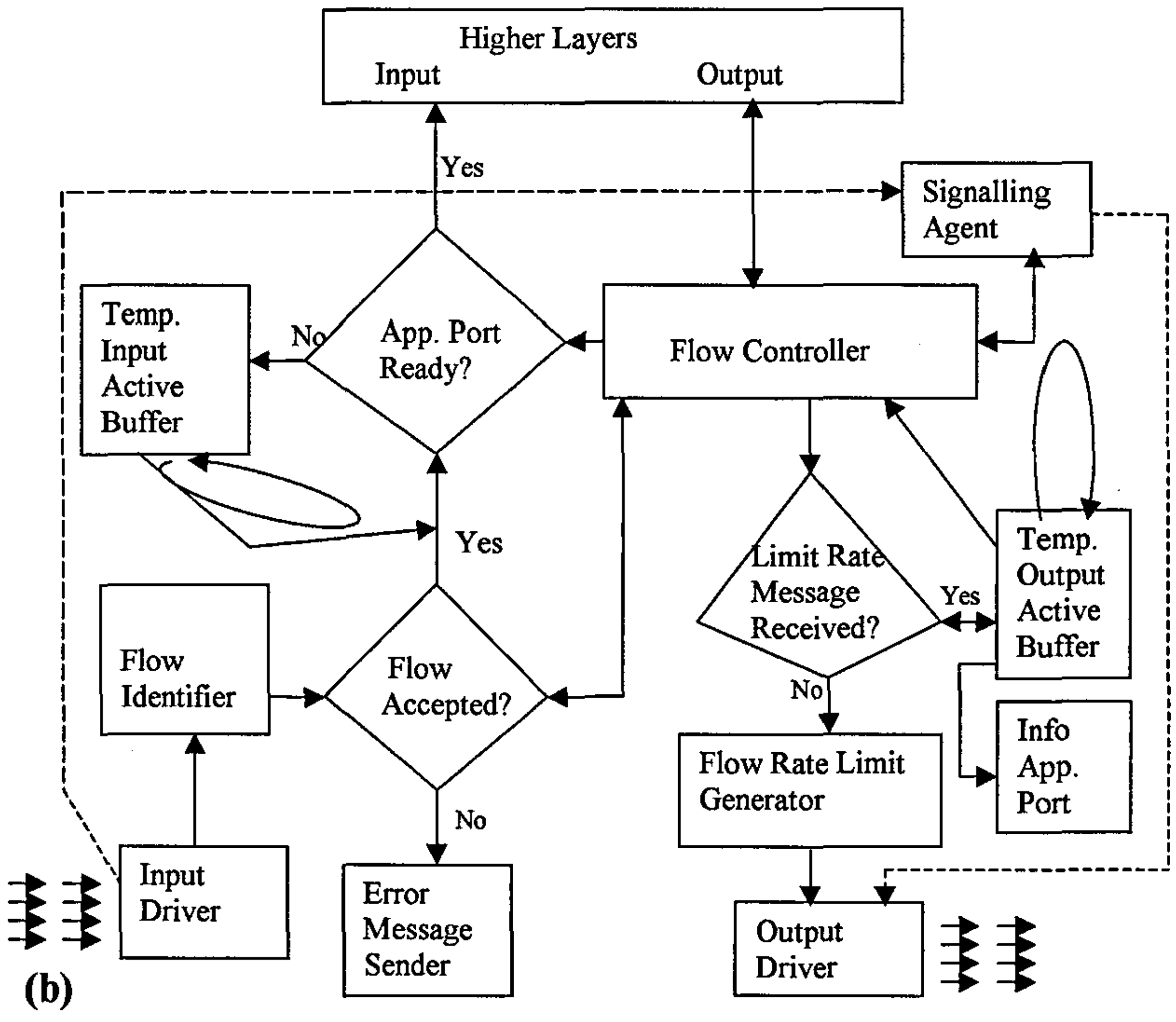


Figure 8.2: PDERRM-Aware Host (a) Layer Structure (b) Process Flow Chart

When a flow cannot be accepted or its flow rate is out of profile, an error message will be sent to the *sender*. The Flow Controller is aware of the node's input traffic processing capability and also has information on which application port is *busy* and which is *ready*. An accepted flow that found its destination application port busy, would be sent temporarily to *Input Active Buffer*. The Input Active Buffer seeks when the busy ports will be free and then send the buffered flows to their corresponding ports.

Application traffic flowing out of the node will pass through the Flow Controller. The Flow Controller interacts with *signalling generator* to determine if the generator has received the message—*flow limit reached* or *exceed*, for any of the traffic classes. In the case that the message has been received, the traffic class concerned will not be released to the network until a counter message—*flow limit within*, is received. The *Traffic Rate-limit Generator* ensures that traffic is released to the network in accordance with the allowed flow rate. When an application generates traffic at the time the traffic cannot be released to the network or the rate at which it generates traffic is higher than allowed, its flows are temporarily stored in the *output active buffer*. The output active buffer will then notify the application to limit its rate of traffic generation.

8.4 PDERRM Aware Router or Forwarding Device

The functionality of a *Router* or any *Forwarding Device* (FD) differs from that of an End Host. Figure 8.3(a) and (b) provides graphical abstraction of the main layer block and the process flow block for a PDERRM-aware router or any FD. The subtle differences between an End Host and a Forwarding Device could be found by comparing Figure 8.2(a) and (b) with Figure 8.3(a) and (b).

Figure 8.3(a) provides illustration for a high level abstraction of the layer block structure of a PDERRM-aware router in which PDERRM section is shown as additional process modules in the network layer of a typical router.

The process flow chart is illustrated with Figure 8.3(b). The QoS Manager serves as the brain of the node in terms of QoS performance of application flow. It co-ordinates the activities of all other flow QoS related process block in the node. Essentially it ensures flows are admitted and processed through the node only if there are resources within the node to support their acceptable QoS performances.

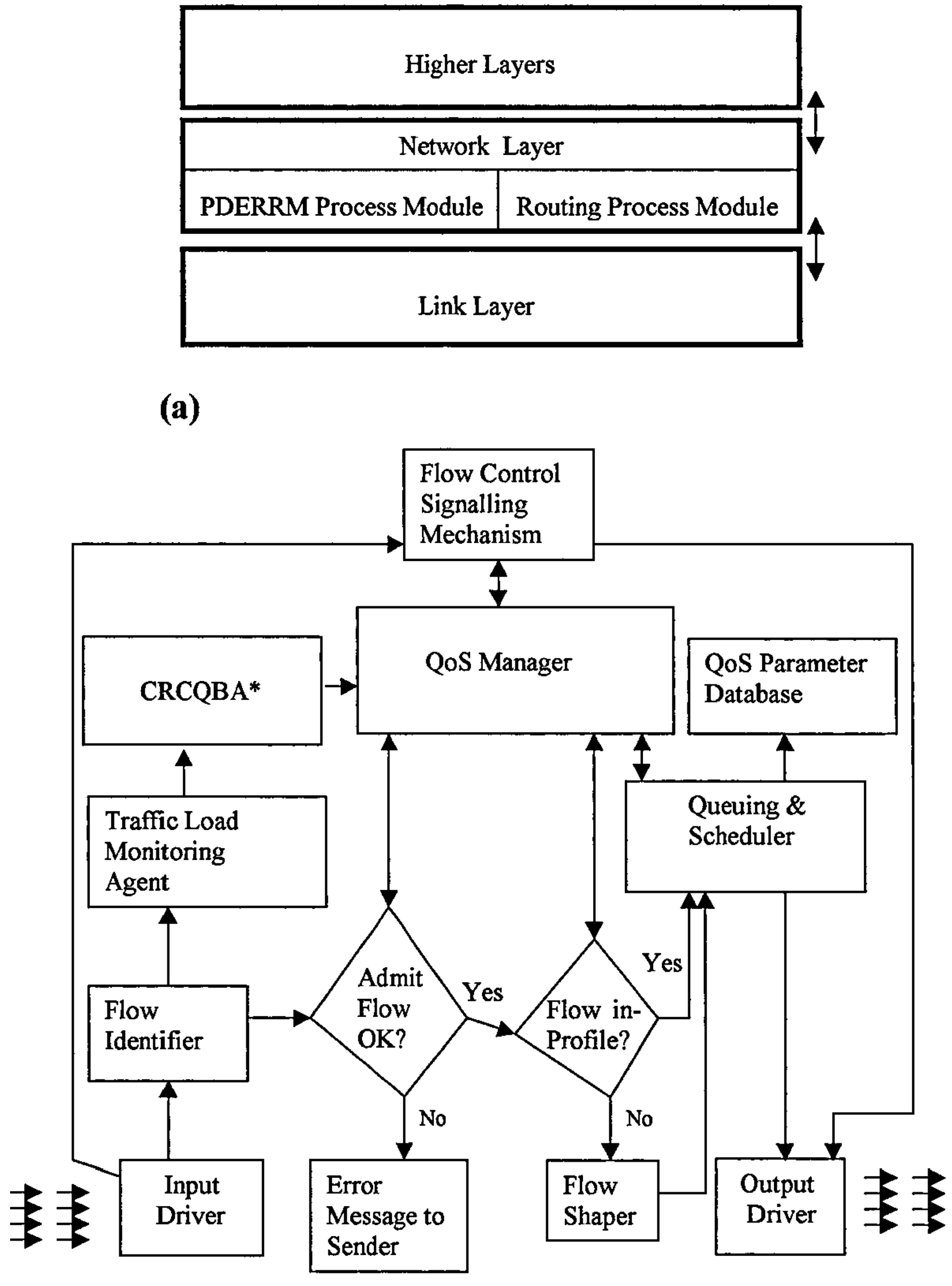


Figure 8.3: PDERRM-Aware Router **(a)** High Level Abstraction Layer Structure
(b) Process Flow Chart

"" *In the Figure, **CRCQBA** to be read as **Classification, Resource Capacity and Quota Brokerage Agent**""

Thus it enforces the loading situations in which the highest resource utilisation can only occur when there is equilibrium or a balance between resource utilisation and resource capacity within the node. It refrains from deficit budgeting between resource utilisation and resource capacity. Compared with the End Host, the object that performs similar functions is called the Flow Controller, (see Section 8.3 for its functionality).

The QoS Manager receives status information on the node resource capacities and policy guide lines on percentage share of resources for each class of traffic from the Classification Resource Capacity & Quota Brokerage Agent (CRCQBA). The CRCQBA takes input from the system kernel and administrative guidelines. The information on each of the traffic classes resource quota are passed to the Scheduler, and the Scheduler uses the information to parameterise its scheduling functions. The Scheduler also interacts with QPD to obtain parameters for per-flow resource allocation when the need arise. The Traffic Load Measuring Agent provides information on the values of traffic load intensities in the node to the QoS Manager, which then correlates the values to the resource capacity values in the node. If traffic load resource utilisation gets to the red-alert threshold, the QoS Manager invokes the Admission Control to limit input traffic to the node. The QoS Manager ensures traffic class resource quota allocations are flexible and reflect dynamic traffic loading in the node and their QoS requirements.

Summary

The PDERRM QoS architecture has been discussed and presented in this chapter. The motivation for the work on PDERRM concerns RSVP complexity and complex traffic conditioning operations required for the DiffServ architecture, and for the fact that QoS deployment at commercial scales is still elusive in the Internet. PDERRM has been presented as a hybrid QoS architecture with mutational (adaptability) features. The principle of operation is very simple— the PDERRM process model simply ensures that there is equilibrium between traffic load resource utilisation and resource capacity in a node. A *host* makes use of the *Traffic Controller* for co-ordinations of transmit and receive traffic. A *router* makes use of the *QoS Manager* for co-ordination of the node resources in admitting and forwarding traffic.

CHAPTER 9

Performance Evaluation of PDERRM QoS Architecture

The presentation in this chapter concerns the various approaches adopted to investigate the performance of the PDERRM QoS architecture. This includes, the experimental procedures employed to determine the effectiveness and strength of its support for QoS provisioning in IP Networks, the results of simulation experiments, and the analysis of the results. Developmental work on PDERRM is on going process, the work described in this chapter is focused on its basic functionality.

In Section 9.1, we present the procedure and method adopted for the simulation investigation of the PDERRM performance evaluation. This includes the technical description of the network topology that we believe is generic with regard to investigation of QoS architecture performance in a network. Section 9.2 deals with the simulation scenario to test PDERRM fundamental functionality— that is ensuring equilibrium between traffic load and resource capacity. The next simulation scenario which is the focus of Section 9.3 examined another functionality of PDERRM, which is based on *source-control of traffic* injected to the network, and the network adopt admission control moratorium at interval of times depending on situations in the network. In Section 9.4, we discuss the simulation experiment based on allowing sources to inject traffic freely to the network, and the network employing PDERRM admission control at the edges to limit traffic load within it's resource capacity. The simulation experiment used to compare PDERRM with a standard QoS architecture in order to determine the extent of its strength is described in Section 9.5. The focus in Section 9.6 is on discussion of simulation experiments used to test PDERRM scalability.

9.1 The Procedure and Method for PDERRM Performance Evaluation

The objective of the simulation experiments was to discover PDERRM strengths and weaknesses from its early primitive design stages and to allow the architecture to evolve in line with the necessary corrections that were needed to meet its design objectives. Thus we have adopted an incremental simulation approach, which sets out to test its features, and where weaknesses were detected, action would be taken on making amendments that would ensure the architecture meets its performance target. The OPNET simulation package was used for the experiment.

In designing the simulation experiments, the main principle of operation of the architecture was of key importance. Thus the notion that the cumulative highest intensities of flows allowed through a *traffic-forwarding node* should adhere to the operational rule that states there should be *"equilibrium between in-load resource utilisation and resource capacity"*. This was the focus in the initial simulation design strategy. Simulations were run with this basic structural functionality of the architecture and the results noted. Other features and components of the architecture were added to the basic architecture in incremental stages for other simulation scenarios of the experiments. The improvements or weaknesses they add to the strength of the architecture were noted. The simulation work carried out so far is grouped into five *Simulations Action Domain* (SAD), each of which will be discussed in the subsequent sections. For the moment we will technically discuss the generic network topology used in four of the five SAD experiments.

9.1.1 The Generic Topology (the Basic Network)

The topology of the basic network used to investigate the performance evaluation of PDERRM as shown in Figure 9.3 is similar to Figure 5.1 used in Section 5.1.1. It has equivalence in structural orientation with the Figure 5.1, but the number of component *nodes* and *links* are at variance.

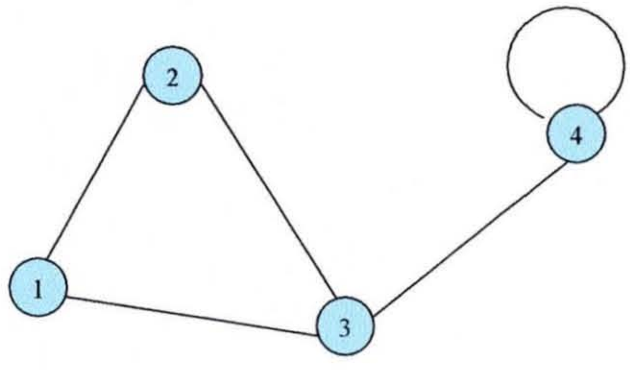
In order to extract the facts of relevance in using the simple basic network shown in Figure 9.3 for modelling the performance evaluation of PDERRM, the description of the network in terms of its technicalities and structural orientation require attention. To describe the basic network technically, we need to make some formal definitions and also to define some new terms. These include definition for *graphs, digraphs, networks,* etc and few of the related terms used with them when they are employed for modelling practical problems. Graphs and digraphs are used in a wide variety of contexts in Engineering, Sciences, Economics, etc to model operations and to model problem solutions. Our concern here is for communication networks.

In general, a *graph* consists of points called ***vertices*** and lines connecting the vertices called ***edges***. Graph models usually represent the entities of a problem as vertices, and the relationships or connections between the entities are represented by connecting edges.

Formal definition: A ***graph* G** consists of two finite sets (sets having finitely many elements), a set V(G) of points called ***vertices*** and a set E(G) of connecting lines called ***edges*** such that each edge connects two vertices called the ***endpoints*** of the edge. Graph G is denoted by

G = (V, E)

Vertices are denoted by letters or by numbers (u, v, ------, or 1, 2, ------). Edges also can be denoted by letters (e_1, e_2 -------), or by their two endpoints, e_1 = (i, j) where i and j are the notation for the endpoints vertices. If an edge links a vertex to itself, then the edge is termed a ***loop***. A ***path*** is a succession of edges, which connect one vertex to another. A ***cycle*** or ***circuit*** is a ***closed*** path where the origin and destination are coincident. The number of edges in a path is referred to as the path ***cardinality*** **[Mol89]**.



A graph G with
Vertex set V(G) = {1, 2, 3, 4} and
Edge set E(G) = {1,2 1,3 2,3 3,4, 4, 4}
A loop edge = 4, 4
A path *p* = 1, 3, 4
p cardinality = 2
A cycle *c* = 1, 2, 3, 1

Figure 9.1 A graph G with n = 4 vertices and n = 5 edges

Formal definition: A ***digraph*** or ***directed graph*** **D_G = (V, E)** is a graph in which each edge e = (i, j), has a direction from its *initial endpoint* i to its *terminal endpoint* j. Two edges connecting the same two endpoints i, j will have opposite directions, i.e. they are, (i, j) and (j, i), e.g. (1, 2) and (2, 1).

Formal definition: A ***tree* T** is a graph that is connected and has no cycle. "Connected" means that, there is a path from any vertex in T to any other vertex in T (figure 9.2(a)).

New definition: A ***ditree*** or ***directed tree*** $_{\mathbf{d}}\mathbf{T}$ is a digraph that is connected and has no cycle (figure 9.2(b)).

New definition: A ***bipartite digraph*** $_{\mathbf{d}}\mathbf{D}_{\mathbf{G}} = \mathbf{(V, E)}$ is a digraph in which the vertex set $V(_{b}D_{G})$ is a union of two subsets V(S) and V(D). Each of the two subsets form a ditree and an edge connect the two ditrees together such that the edge has one of its endpoint incident on a vertex in V(S) ditree and the other endpoint incident on a vertex in V(D) ditree. See Figure 9.2(b).

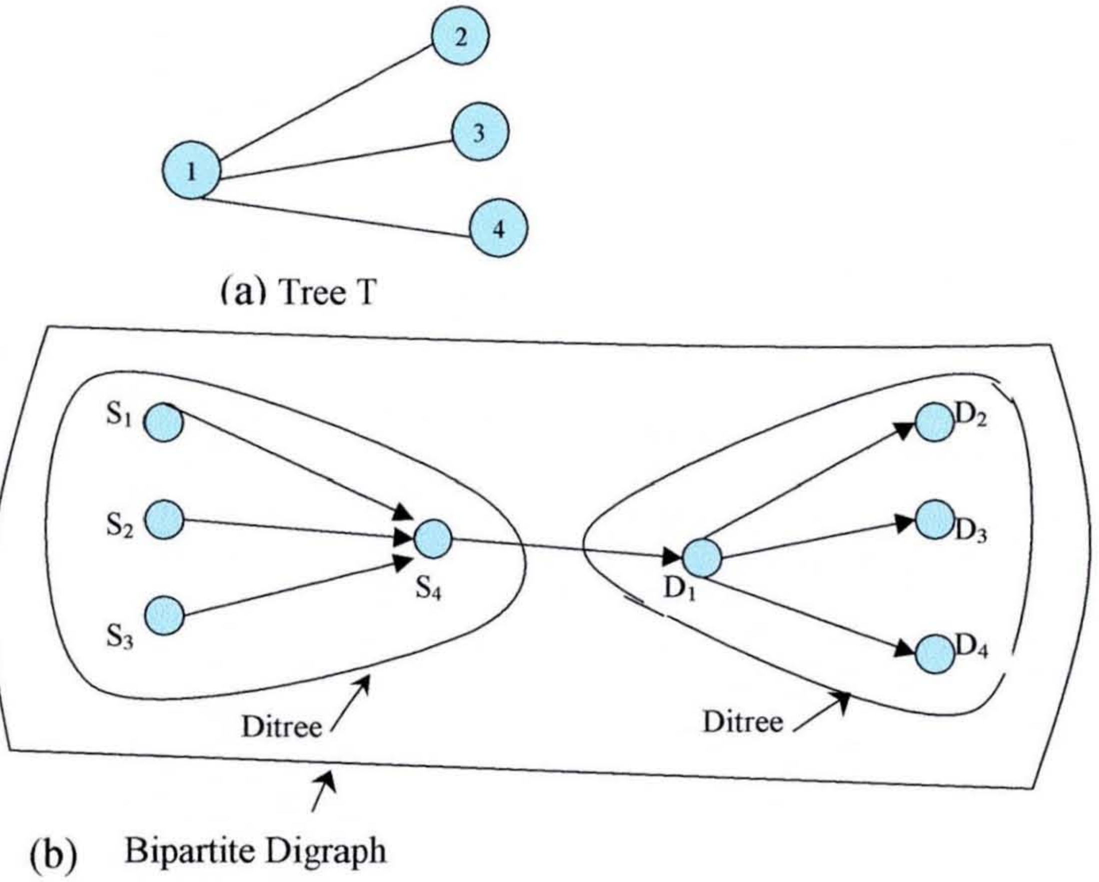


Figure 9.2 (a) A Tree T and (b) A Bipartite digraph with two ditrees having opposing directed edge orientation, and connected root to root.

Formal definition: A ***network*** is a digraph $\mathbf{D_G = (V, E)}$ in which each edge e = (i, j) called a *link* has an assigned capacity $c_{ij} > 0$ [= maximum possible flow along (i, j)], and the vertices are commonly called *nodes*. At one node S, called the *source*, a flow is produced that flows along the links to another node, D, called the *destination* or *target* or *sink* where the flow become useful or disappear **[Kre93]**.

9.1.2 The use of a Basic Network as Tool for Comparison of two or more Resource Allocation Mechanisms

The basic network shown in Figure 9.3 is a bipartite digraph in which each flow has a *path cardinality* of 3. *This is described as a three legged relay flow systems.* The first ditree (subdigraph) as illustrated in Figure 9.3 represents the *source domain*. It has a concave (convergent) flow orientation that has its focal point at the convergent node S_C (Figure 9.3). Flows emerging from node S_C towards node D_C, on the connecting bottleneck link (S_C, D_C) were subjected to the constraint of maximum link capacity utilisation as against low link capacity utilisation that exist in other links of the network. The other ditree, which is the *destination domain*, has convex (divergent) flow orientation. The convex focal-point node D_C distributes the flows divergently to their respective links towards their destination nodes D_i.

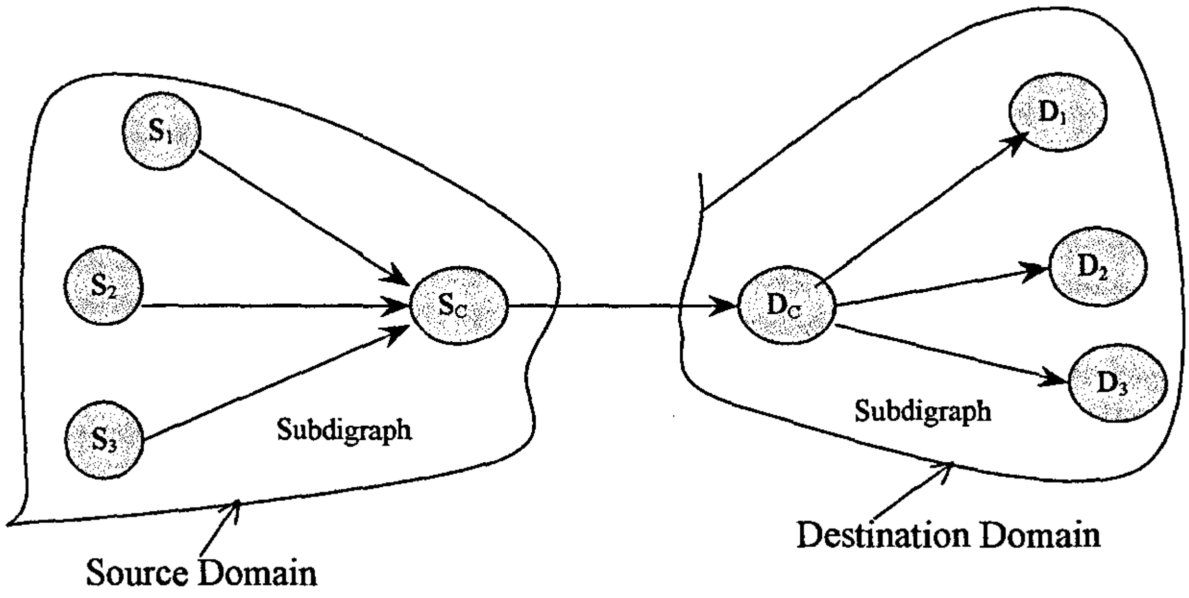


Figure 9.3: The basic network, technically referred to as bipartite digraph.

A flow emerges from its source in the source domain node Si and is transmitted through the three legged relay flow system to its destination node D_i in the destination domain. The flow would suffer flow-constraint at the *bottleneck link*, which would be overloaded, since it would be the convergent link for all other links. At the flow convergent node S_C heavy resource contention would take place between different flow classes. The environment of high resource contention in node S_C or node D_C provides a suitable condition for comparing the performance of various resource allocation mechanisms. Thus two or more flow service paradigms could be compared in terms of their effectiveness on resource contention resolution among different traffic classes when each of the simulation scenarios used in the comparison has identical flow profiles. This forms the basis of our performance evaluation of PDERRM. A best–effort service model (no multiservice QoS) was compared with PDERRM on a number of the later basic features to see if it supports multiservice QoS using the basic network. Also a standard QoS architecture— IntServ-RSVP was similarly compared with PDERRM in order to identify PDERRM strengths. The experimental investigation will be covered in subsequent sections.

9.2 Simulation Action Domain (SAD) with PDERRM First Basic Functionality (Scheme A)

The first simulation experiment on PDERRM performance evaluation was focused on its basic functionality— the process model that ensures the equilibrium between input traffic load resource utilisation and resource capacity in a node does not shift towards a negative imbalance with respect to the node's resource capacity.

QoS performance metrics on each of the application classes are deduced on the following relational formula in connection with the basic network of the Figure 9.4.

QoS Performance Metric:

(1) Latency and jitters condition

$$\text{Average}\left[\sum_{i=1}^{n}(T_{1fi} - T_{2fi})\right] = \begin{cases}\text{min} & (\text{low latency \& jitters})\\ \text{max} & (\text{high latency \& jitters})\end{cases}$$

Where T_1 was the time flow f_i ($1 \le i \le n$ = number of traffic classes) got to node S_C (figure 9.4) and T_2 was the time same flow f_i emerged from node D_C ;

(2) Obeying Kirchhoff's law on throughput

$$\underbrace{\sum f_{SiSc}}_{\substack{S_{cf}\\ \text{Inflow to node } S_C}} - \underbrace{\sum f_{DiDc}}_{\substack{D_{cf}\\ \text{Outflow from node } D_C}} = \begin{cases}0 & (\text{maximum throughput})\\ f & (\text{flow loss recorded})\end{cases}$$

Where S_{cf} was the cumulative sum of flows that entered node S_C (f_{SiSc} means flow Source i into node S_C), and D_{cf} was the cumulative flows that emerged from node D_C (f_{DiDc} means flow i towards destination D emerging from node Dc) (see Fig.9.4).

9.2.1 PDERRM-Aware Host Process

Each of the ***hosts*** (traffic source node and traffic sink node) has four main *process modules*— the traffic or application generator module, traffic sink module, the traffic rate control module and the link layer module. The OPNET simulation package, together with its enhanced utilities provides the platform, facilities and necessary environment for specification and definition of the behavioural logic of the modules. The traffic generator module included processes that determined the packet format, packet size and initial rate of generation. It also included processes for traffic generation statistics. The generator module also included a QoS enhanced API sub-module which applications used to indicate the traffic class to which they belonged, their QoS needs in the network, and the node destination address of the application. The sink module contained processes that received the traffic, took statistics of received traffic and destroyed the traffic.

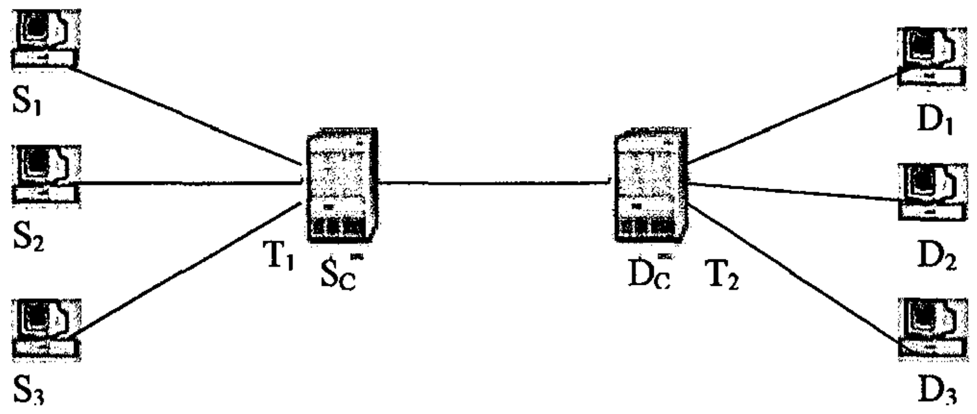


Figure 9.4: Basic Network

The traffic control module main functionality was to control the rate at which traffic was injected into the network in accordance with either local or global parameters. The module also included the signalling mechanism which received information on traffic loading in the network and passed the information to the traffic control section which used it to control its functionality. Some of its processes or functionality might be operated in an *off* and *on* basis depending on events in the network. Its functionality will be described in more detail in the next section where it will be seen that its functionality is the main determinant of the outcome of the experiment in that SAD. The link layer module included the transmitter and receiver sub-modules together with the *pipeline* processes responsible for transmitting and receiving packets.

9.2.2 PDERRM-Aware Router Process

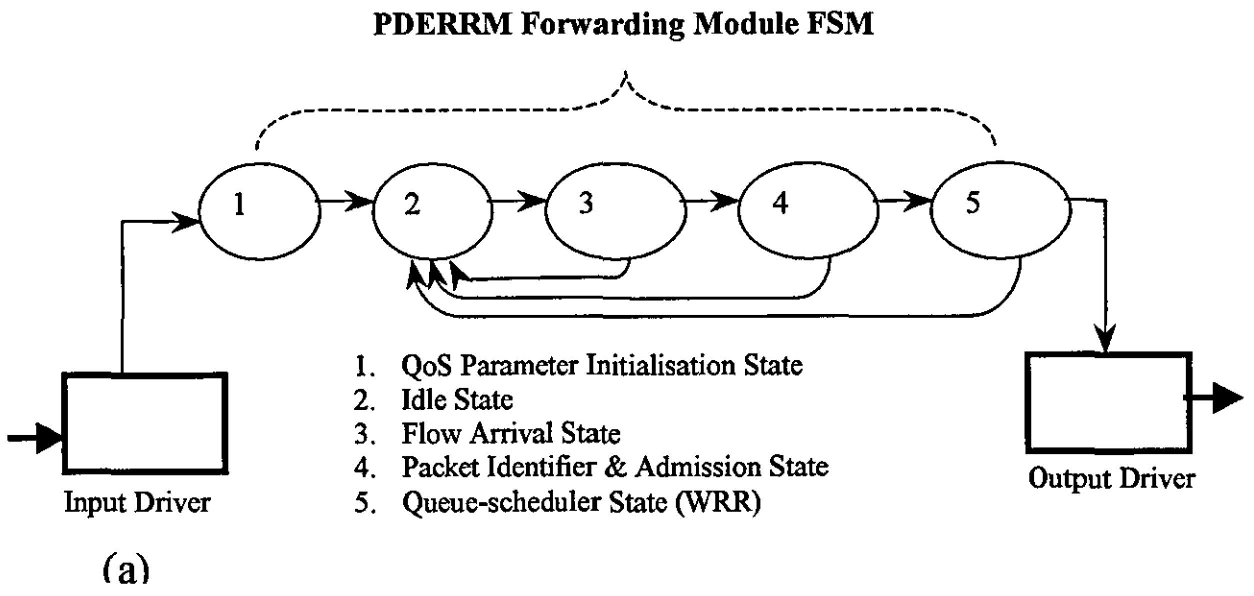
The ***router*** or the ***forwarding device*** in its basic form was made up of three main modules— the input driver module, the QoS enhanced forwarding module and the output driver module. The input driver module was basically the receiver module with its pipeline stages. The forwarding module was made up of the QoS parameter specification sub-module, packet identifier together with PDERRM admission sub-module and queue-scheduler sub-module. The output driver module was basically made up of the transmitter and its pipeline stages.

The functionality of the forwarding module served as the determinant of the QoS outcome of the experiment in this SAD. The *Block and State Transition Diagram* (BSTD) of the service model of basic PDERRM-aware router is shown in Figure 9.5(a). The figure shows the process flow of the forwarding module and its *Finite State Machine* (FSM). Figure 9.5(b) illustrates best-effort service model BSTD as comparison.

The QoS parameter specification sub-module contained information about traffic classes and their resource quota. This information was used by the admission control module in limiting traffic input to the node to fit within the resource quota of the traffic class concerned, and for inserting traffic classes to their appropriate queues. The admission control operated on a *flow-measurement moving window* mechanism. The operation simply took cumulative *bit* or *byte count* of flows as they arrived over the lifetime of a measuring window and correlated the value of the measurement with a computed allowed value. Mathematically, this was a discrete integration or summation, which could be represented notationally as:

$$\text{Admit the packet of Flow-class\_}i\text{ if, } \sum_{t=0}^{W} \sum_{j=1}^{N} (P_{i\_j\_k} \times S_{i\_j\_k}) \le M_{\_i}$$

Where M_*i* is the computed maximum limit of quantum packets allowed for Flow-class_*i*, ($1 \le i \le n$ = number of traffic classes) in the current *window* with size W. $P_{i\_j\_k}$ is the *k*th, packet for flow *j* that arrive in the current window for Flow-class_*i*, (k = 1, 2, ...). $S_{i\_i\_k}$ is its packer size in bits or bytes, and N is the number of concurrent flows allowed for Flow class_*i*.


PDERRM Forwarding Module FSM
1
2
3
4
5
1. QoS Parameter Initialisation State
2. Idle State
3. Flow Arrival State
4. Packet Identifier & Admission State
5. Queue-scheduler State (WRR)
Input Driver
Output Driver
(a)

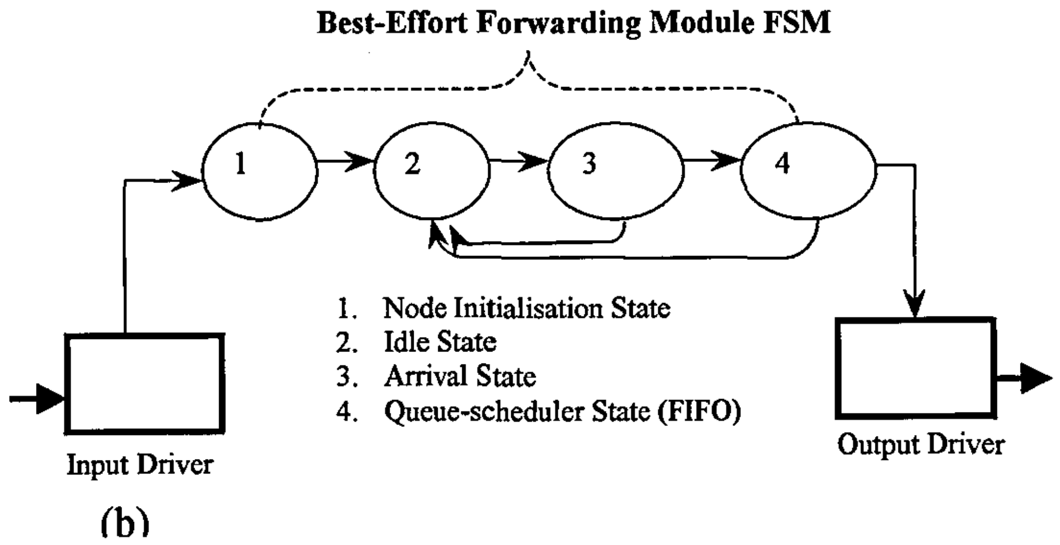

Best-Effort Forwarding Module FSM
1
2
3
4
1. Node Initialisation State
2. Idle State
3. Arrival State
4. Queue-scheduler State (FIFO)
Input Driver
Output Driver
(b)

Figure 9.5 (a) Block and State Transition Diagram of basic PDERRM-aware router. (b) Block and State Transition Diagram of normal router

At the end of every *window*, all measurements were reset to zero. The functionality of the admission control was configured to be a continuous process in the forwarding nodes during all simulation time in this experiment.

The schedulers serviced each traffic class according to information implicitly built into the DSCP value in the packet header. The scheduler has knowledge of the interpretation of DSCP values in terms of resource allocation. The scheduler serviced each flow according to the resource allocation encoded in the DSCP value and assigned the flow to its corresponding output interface, while the output driver module transmitted the flow on its link. The queue-scheduler contained processes that generated different types of queue statistics such as queue sizes and packet queue delays.

9.2.3 Simulation Experiment

For the purpose of this experiment, three broad classes of traffic were defined. These represent *inelastic* (real-time) type of applications, *inelastic tolerant* type of applications and *elastic* (non real-time) type of applications. They are simply represented here as class 1, class 2 and class 3 respectively. A wide range of traffic parameters for each of the traffic classes was used in generating various traffic profiles for the experiment. Table 9.1 represents the various parameters of traffic used in the experiment.

Table 9.1

(a) Traffic Class 1

	Low values	High value	Range
Packet sizes in bytes	32, .., 48, ...	128	96
Packet rate in kb/s	56, .., 64, ...	128	62
Burst time in seconds	0.25,.., 0.35, ...	2.0	1.75
Idle time in seconds	0.45, .., 0.65, ...	3.0	2.55
Inter-packet-arrival distribution	Constant and Exponential		

(b) Traffic Class 2

	Low values	High value	Range
Packet sizes in bytes	64, ...	512	448
Pkt Inter-arrival time	0.03, ...	0.9	0.87
Packet Inter-arrival distribution	Constant and Exponential		

(c) Traffic Class 3

	Low values	High value	Range
Packet sizes in bytes	105, ...	1024	919
Packet rate in Kb/s	32, ...	448	416
Packet Inter-arrival distribution	Constant and Exponential		

The network topology used for the simulation experiment is illustrated in Figure 9.4. After the node process models of the modules had been specified and defined, the modules programme logic were then constructed. The traffic and the statistics were configured for the simulation run. The bottleneck link was driven at maximum capacity. The first simulation scenario was for *baseline* in which the *forwarding device* operated on the basis of best-effort service model. In this case the scheduler would operate the FIFO queuing discipline. The simulation was run and statistics collected. The forwarding device was then reconfigured to operate on the PDERRM service model. Identical traffic input profiles that were used for the baseline were repeated for the PDERRM simulation.

9.2.4 Latency Results and Comments

The emphasis in the results is based on *latency* and *jitter*. The comparison results, which show the first performance evaluation of PDERRM, are shown in Figure 9.6. The performance of PDERRM in terms of end-to-end delay and jitter was highly impressive. This is illustrated in the figure, which shows very low latency compared with best-effort services. Whilst the results for the best-effort service model shows comparatively higher delay with its associated higher jitter, results for PDERRM service model reveal low delay and corresponding low jitter.

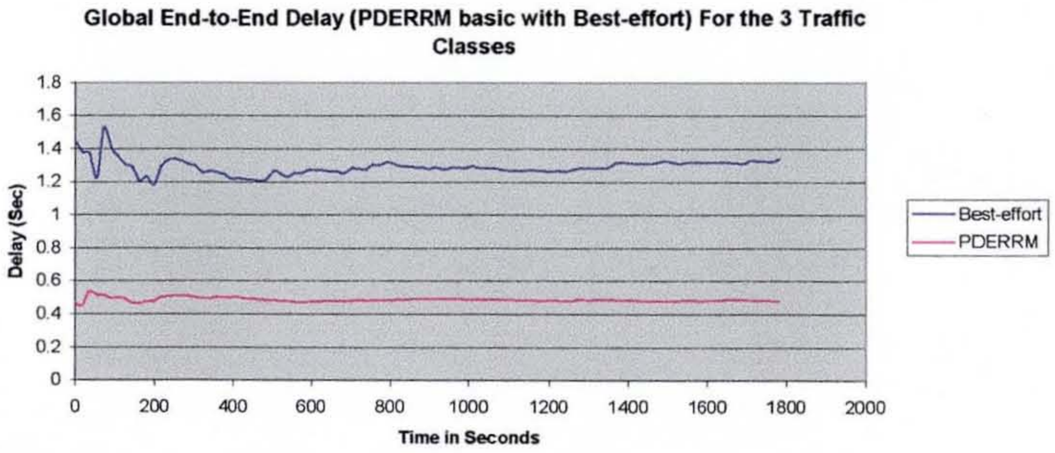


Figure 9.6: Chart Showing average Global End-to-End Delay for the three Traffic Classes used in the Simulation Scenario.

9.2.5 Throughput Results and Comments

As shown in Figure 9.7, the results of the experiment showed better performance of the best-effort service model on *throughput* than PDERRM basic service model. We have discovered that this was the case with most QoS architectures that involve continuous admission control processing. Low throughput is a price that has to be paid for very good latency and jitter performance. This was revealed in the performance of RSVP compared with best-effort that would be shown in Section 9.5.4. However PDERRM have a scheme to improve throughput performance.

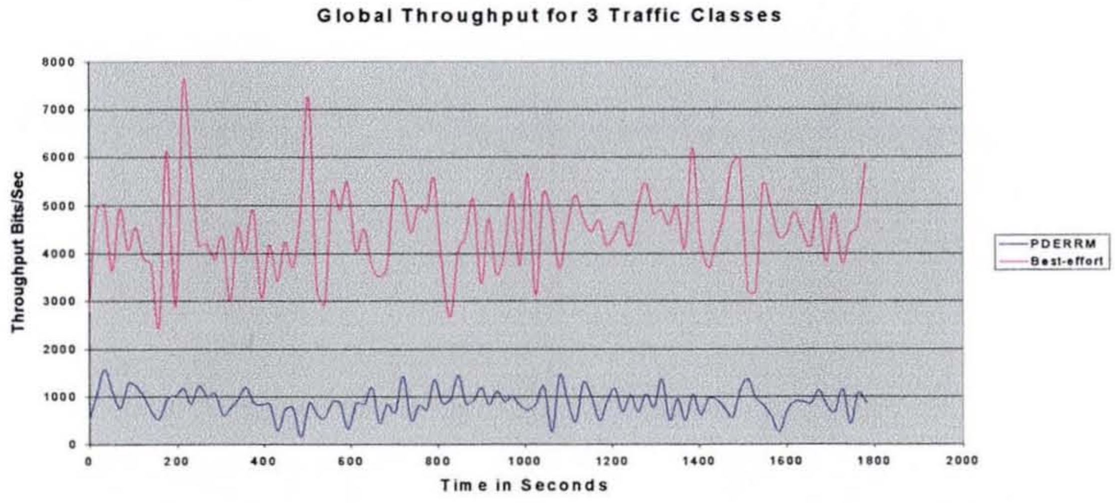


Figure 9.7: Chart showing Global throughput for PDERRM basic compared with Best-effort for the 3 Traffic classes used in the Simulation Scenario.

9.3 SAD with PDERRM Flow Source Control and Moratorium on Admission Control in the Network (Scheme B)

The simulation experiment on PDERRM performance evaluation in this SAD was based on shifting most of the flow rate control from the network to the sources of flows. The underlying functionality specified a necessary condition that all sources of flows should adhere to global parameters on flow rates, in injecting flows to the network. Thus sources of flows compute flow rates for each traffic class employing global parameters specified for such class of application in releasing the flows to the network. The network on the other hand adopts an admission control moratorium on traffic flows, for unequal periodic time intervals depending on the traffic load situation in the network. Each source of flows (host) had well defined global parameters used for regulating flow rates to the network.

Application traffic was generated by the application process module in the host, and the resulting traffic-flows forwarded to the traffic control module, which ensured that the flows were released to the network in accordance with the global parameters. The network nodes would simply allocate resources to each traffic class on the basis of the resource quota for each traffic class without continuous admission control and believing that the sources of flows stayed within their agreed flow rates. The network nodes would monitor resource utilisation and if a threshold was reached, send control signals to source of flows. ***The strategy here was to ensure that the network was relatively simple and complexities resided in the end nodes.*** This was the basic factor that was responsible for the phenomenal success of the Internet.

The underlying function of the **resource management** has its thrust in the functionality that ensured the cumulative traffic flows released to the network stayed within the resource capacity of the network. Here, that functionality resided in the end host and could be achieved practically through global standardisation of flow rate parameters and inventory of the global parameters made available to network operators.

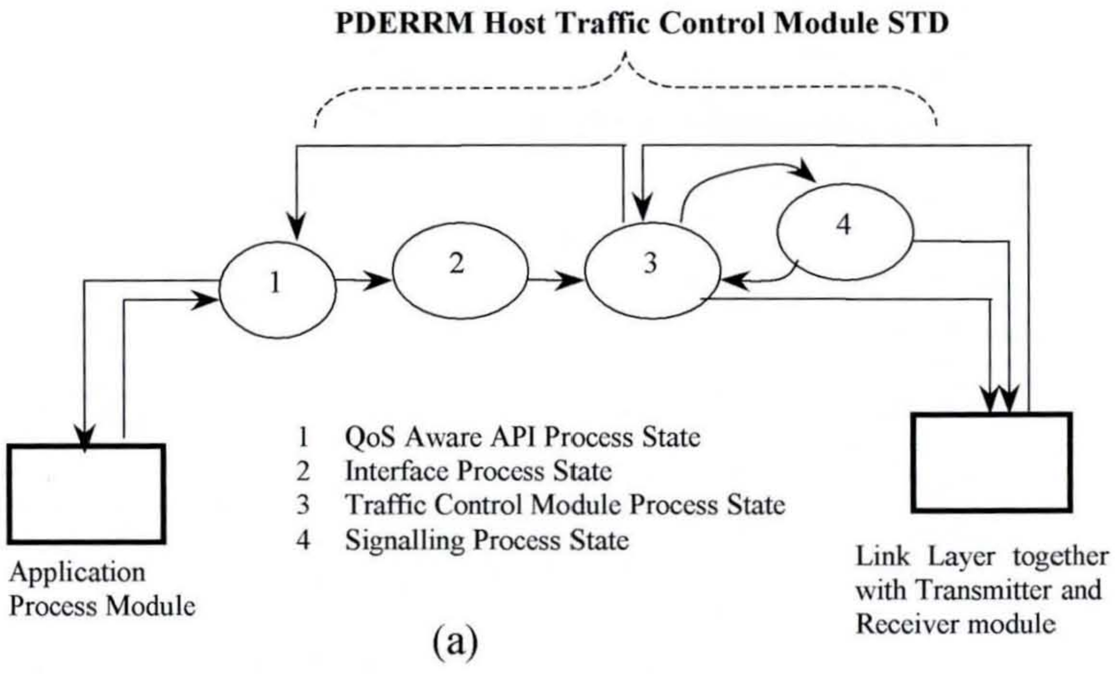
The functionality of the traffic control module in the end host served as the determinant for the outcome of QoS management in this experiment.

The *Block and State Transition Diagram* (BSTD) of traffic process flow in PDERRM-aware host is shown in Figure 9.8(a). The figure illustrates the process flow of traffic as it emerged from the application process module, through intervening modules to the transmitter, where it would be released to the network. Input traffic would retrace the path in the opposite direction from the receiver process module to the application process module. Figure 9.8(b) illustrates the best-effort service traffic process flow's BSTD as a comparison.

The traffic control module regulates flow rate into the network using simply either a *leaky* or *token* bucket algorithm and employing global parameters to parameterise the algorithm.

9.3.1 Simulation Experiment

As in the previous simulation experiment, three broad classes of traffic were defined and used in the simulation. These were simply termed as class 1, class 2 and class 3 flows. The same network topology (Figure 9.4) as used in the previous section was also used in this simulation. In running this simulation, the bottleneck link was also driven at maximum capacity. The first simulation scenario, which was for the baseline, was configured on the basis of the traditional Internet service model. The simulation was run and statistics collected. The PDERRM simulation scenario was reconfigured such that the end host rate limited traffic flow to the network in accordance with global parameters.


PDERRM Host Traffic Control Module STD
1
2
3
4
1 QoS Aware API Process State
2 Interface Process State
3 Traffic Control Module Process State
4 Signalling Process State
Application Process Module
Link Layer together with Transmitter and Receiver module
(a)

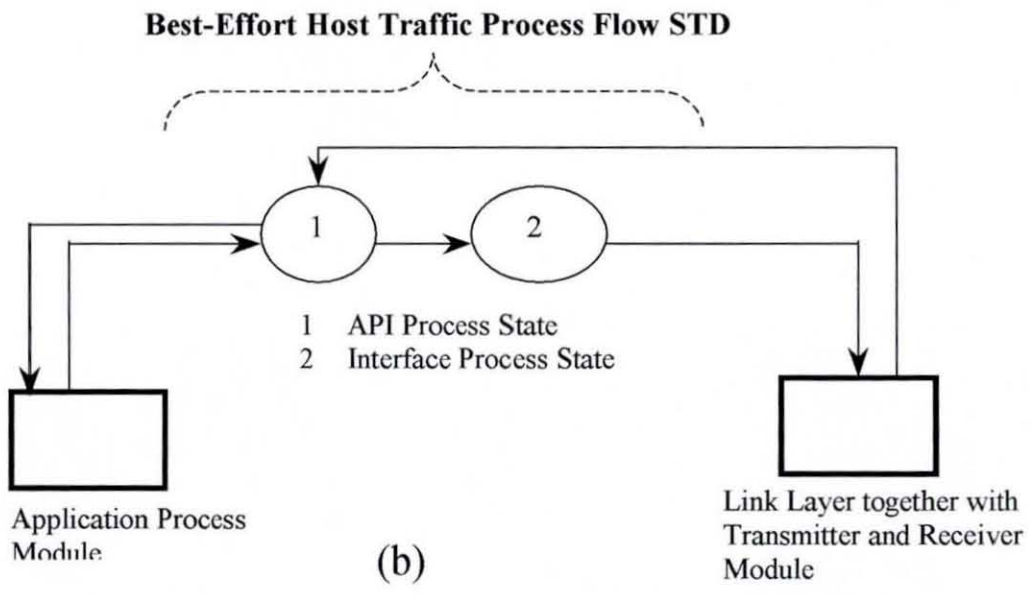

Best-Effort Host Traffic Process Flow STD
1
2
1 API Process State
2 Interface Process State
Application Process Module
Link Layer together with Transmitter and Receiver Module
(b)

Figure 9.8 (a) Block and State Transition Diagram of basic PDERRM-aware host
(b) Block and State Transition Diagram of basic normal Host.

The network on the other hand allocated resources, monitored resource utilisation and took necessary action depending on events in the network. The results were displayed in comparison with the baseline results.

9.3.2 Results and Comments

The comparison results are shown in the graph of Figure 9.9. As illustrated in the figure, the results are very impressive for PDERRM. The global end-to-end delay and jitter for PDERRM was remarkably very low compared with best-effort services. The PDERRM *source traffic control scheme* is an attractive scheme to achieve QoS resource management.

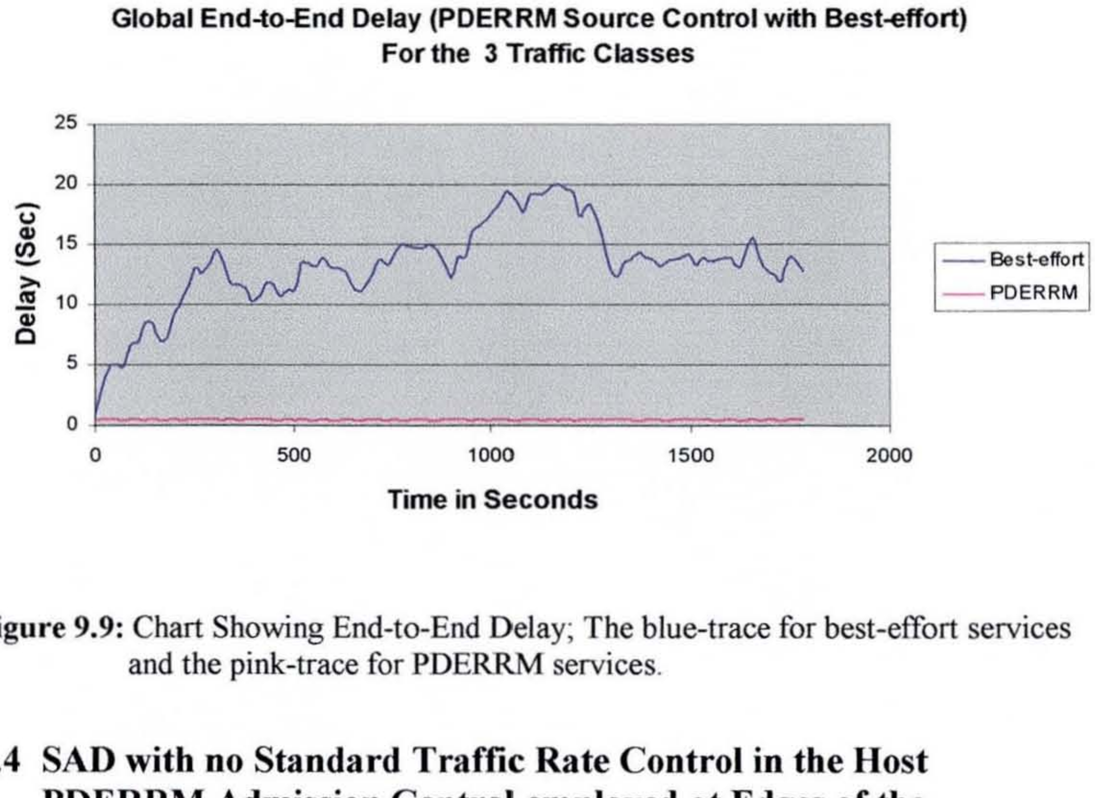


Figure 9.9: Chart Showing End-to-End Delay; The blue-trace for best-effort services and the pink-trace for PDERRM services.

9.4 SAD with no Standard Traffic Rate Control in the Host PDERRM Admission Control employed at Edges of the Network (Scheme C)

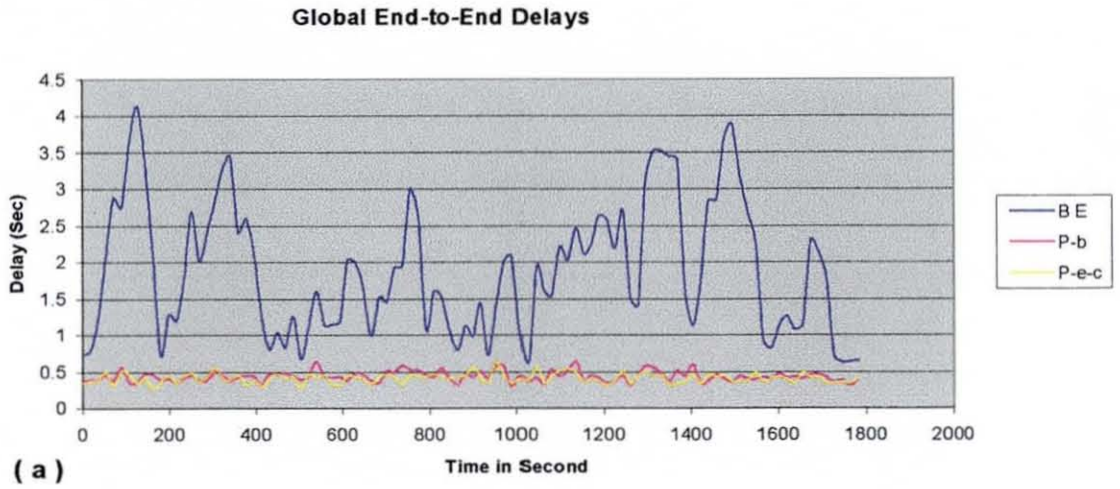
The simulation experiment for PDERRM performance evaluation in this SAD was based on the strategy of limiting the over-all process *time* and *resources* used for the processes of traffic control in order to enhance QoS performance of applications. Here sources of flows were initially allowed to freely inject traffic into the network. In this case, the *host traffic control signal-reception mechanism* listened to the network, in order to receive information on network loading conditions. Any forwarding device or router that received flows at a rate higher than the permitted range will send a signal to the offending source to limit its traffic rate. The source of a flow causing concern will then reduce its flow rate at a percentage based on the standard in operation. (The rate reduction would be a stepwise or piecewise function that would be formulated on percentage of flow rate to be reduced at each step in a sequence of steps that could be standardised (similar to CSMA/CD standard)).

Flow control in the network was based on *edge control*. At the first hop where the packet of a flow entered the network, PDERRM admission control would be applied on the packet to determine if its temporal characteristics complied with specified values. If the packet was in-profile, it would be given a label to indicate this and no further admission control action would be applied in other intermediate hops until it got to its destination. Interior nodes in the network simply allocate the packet to their appropriate queues and service the packet according to its bandwidth requirement. The thrust of the **resource management** operation here places the main functionality of traffic control at the edges of the network, a little traffic control at the host and the interior of the network relatively simple. The objective of this experiment was to find out if the latency performance would remain the same as in previous simulations and to determine if there would be an improvement on throughput.

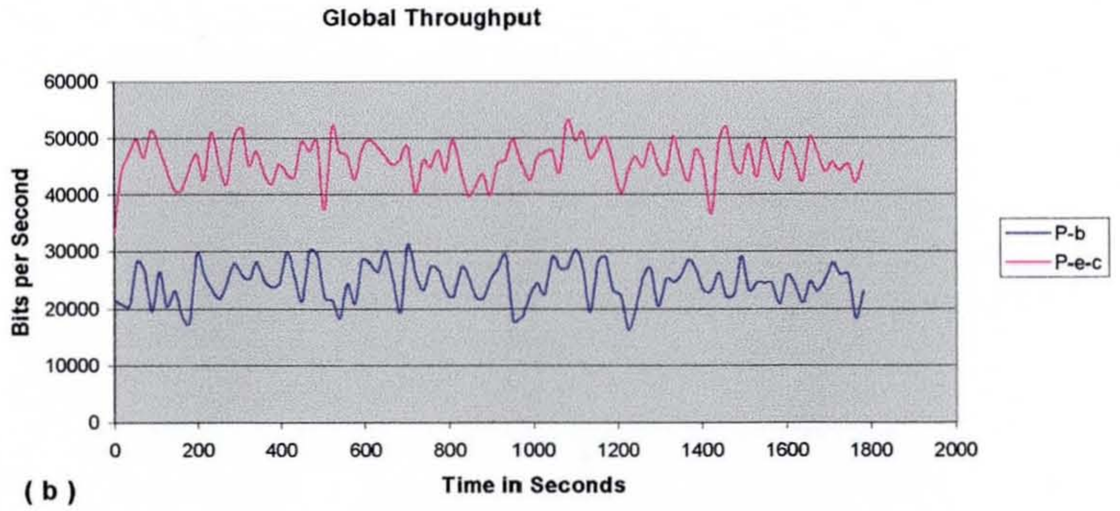
9.4.1 Simulation Experiment

For the experiment in this SAD, four broad classes of traffic were used. These represented *inelastic* (real-time) types of application, *inelastic tolerant* types of application, *mission critical* types of application and traditional *best-effort* types of application. They are simply represented here as class 1, class 2, class 3 and class 4 applications respectively. As in the previous experiments, a wide range of traffic parameters for each of the traffic classes was used in generating various traffic profiles as input to the network used in the simulation. The structure of the network topology used in this experiment was similar to that of Figure 9.4, but the number of host used was eight instead of six.

In running this simulation, the bottleneck link was also driven at maximum capacity as in previous sections. The first simulation scenario, which was for baseline, was configured on the basis of the traditional Internet service model. The simulation was run and statistics collected. The PDERRM simulation scenario was reconfigured such that edge nodes applied PDERRM admission control on flows as they passed through the first hop into the network. Subsequent QoS processes simply concentrated on allocating buffer and bandwidth resources to the flow as it passed through the network.



(a)



(b)

Key: BE = best-effort, P-b = PDERRM-basic, P-e-c = PDERRM-edge-control

Figure 9.10: (a) Showing End-to-End Delay for best-effort, PDERRM-basic and PDERRM-edge-control (b) Showing Throughput for PDERRM-basic and PDERRM-edge-control

9.4.2 Results and Comments

The results are shown in the graphs of Figure 9.10 (a) and (b). As shown in Figure 9.10 (a), the result of end-to-end delay is good and compared favourably with the results of the previous sections in terms of low delay, low jitter. Throughput results show tremendous improvement for PDERRM services with edge traffic control only, when compared with the continuous admission control found in PDERRM basic services.

The throughput results are shown in Figure 9.10 (b), and clearly show that, edge only traffic control has better performance than continuous traffic control on throughput. The results as shown in Figure 9.10 (a) and (b) revealed that, edge only policing is a better scheme than continuous policing in the network. The improvement with edge only traffic policing would be attributed to the lower process threads percentage allocated for traffic control in the network.

9.5 SAD in which PDERRM Performance was compared with that of IntServ-RSVP model (Scheme D)

In this SAD, the experimental focus was to compare the performance of PDERRM with that of a standard IP QoS architecture— the IntServ-RSVP paradigm. OPNET (OPNET Technology Inc, USA) software simulation package has a *plug and play* experimental implementation of the RSVP process model, which was used for the experiment. The procedure for the performance comparison was as follows:

- A simple network of identical topology as shown in Figure 9.12 was used for both RSVP and PDERRM simulation scenarios.
- Identical traffic profiles were used for both simulation scenarios.
- Identical simulation time were used for both simulation scenarios.
- Identical statistics were defined and collected for both simulation scenarios.
- The result of the PDERRM simulation scenario was compared with that of RSVP simulation scenario to make conclusions about performances.

9.5.1 Brief Summary of Operation of RSVP Model

Resource reSerVation Protocol (RSVP)— [RFC 2205] is a network layer signalling protocol that allows applications to reserve network resources for their flows. The key functionalities of the RSVP model are:

- Transmitting applications describe their data-flow characteristics to nodes along the path of flows using the *Path Message* [RFC 2209].
- Receiving applications describe their Quality of service (QoS) requirements using *Resv Messages* [RFC 2209] and retrace the path created by the Path Message in opposite direction.

- Routers deliver the specified QoS request to application traffic along the path of flow.

RSVP treats data flows from receiver to sender as logically independent from the flows from sender to receiver. Accordingly, a reservation for data from sender to receiver is independent from a reservation from receiver to sender. Thus RSVP establishes a reservation for simplex flows.

The following steps show the event sequence for RSVP resource reservation operation:

1 The transmitting sender host typically knows the characteristics of the traffic it generates (described as token bucket parameters) which will be packaged into the Path Message, then will be transmitted hop-by-hop from the sender to the receiver.

2 The Path Message creates a Path State in each router that is traversed from sender to receiver. Through the path setup mechanism, all devices along the path become aware of their adjacent RSVP nodes for data flow.

3 When the Path Message gets to the receiver, the local RSVP module notifies the receiver's host application and the receiver's host application makes the decision whether or not to reserve the resource.

4 Once it is decided to request network resource reservation, the host application sends a request to the local RSVP module to assist in the reservation setup.

5 The local RSVP module in the receiver uses the RSVP protocol to carry the request as Resv Messages to all nodes, hop-by-hop along the data path created by the Path Message and moving in the reverse direction to the sender's data direction up to the sender. Each intermediate node checks if it has sufficient resources when deciding to either grant or reject the reservation request. If the reservation is granted, Resv State is created, and the reservation request is forwarded to the upstream previous hop in the data path.

6 The receiver may request a notification about the reservation status. In such a case, once the sender receives the Resv Message originating from the receiver, it sends a Resv Confirmation message back to the receiver.

7 If the receiver sends any data, it will start sending Path Message to the sender. In this case, steps 1 to 6 are repeated with the receiver acting as the sender and the sender acting as the receiver.

8 Once reservation is established, data will continue to flow and use the reserved resources.

RSVP adopts the "*soft state*" approach in managing Path State and Resv State in routers and hosts. An RSVP soft state is created and periodically refreshed by Path and Resv Messages.

9.5.2 Comparison of PDERRM Operation with RSVP

We will briefly compare and contrast PDERRM basic operation with that of RSVP as highlighted above.

- Unlike RSVP which describes it data-flow characteristics explicitly to the network on a per-flow basis, PDERRM uses DSCP hierarchical encodement to implicitly describe its data-flow characteristics to the network. Each set of application class will have standard set of global parameters consistently understood by all grades of nodes that describe the temporal characteristics of their flow as they are being transmitted through the network. Each sender host simply indicates which class its application traffic belongs by using a DSCP value and the network will understand the application traffic characteristics by decoding the DSCP value.
- PDERRM unlike RSVP has no Path Message. PDERRM's similar operation to RSVP Path Message involves labelling the first packet of a flow with "*flow_session_start*". When the packet gets to each router along the path of flow, and a per-flow resource allocation is required, the packet DSCP value is mapped to relevant index in the *QoS Parameter Database* (QPD) of the node. The destination address and the next hop IP address are entered against its index in the column provided in the QPD. When subsequent packets of the same flow get to each router the destination address will be mapped to its value in the QPD where the next hop IP address for the packet is discovered. The current packet will then be routed such that it follows the same path as the previous packets.
- PDERRM also unlike RSVP has no Resv Message. Each router has a maximum number of independent flows for each class of traffic that it can accommodate based on its resource capacity. Thus the resource allocation mechanism here operates on provisioning unlike the reservation method used in the RSVP paradigm. When a new flow arrives, the router simply checks if the maximum

number of independent flow for the class of the flow has been reached or not in making its decision on the flow admission. The flow can only be accepted if the maximum number of independent flow for its class has not been reached. Thus PDERRM does not incur Resv State overhead.

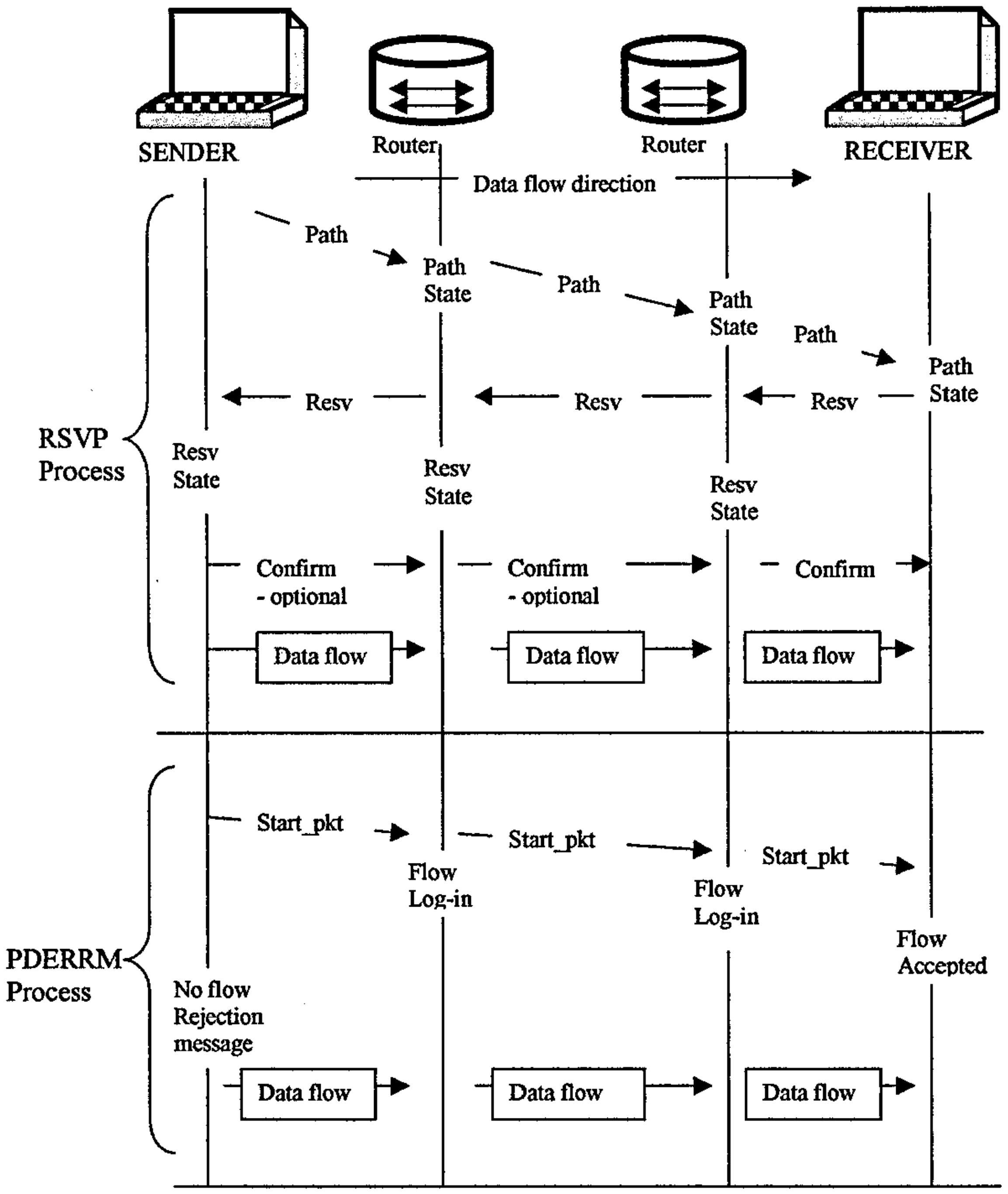


Figure 9.11: Comparison of RSVP processes with that of PDERRM

- The sender waits for a period of time equal to *Round Trip Time* (RTT) after sending the first packet before the next packet is sent. This is to determine if the flow is rejected in which case, the sender gets a rejection message and the flow suspended or re-entered later.

- The sender labels the last packet of a flow with the label "*flow_session_end*". Each router on receiving a packet with the label "*flow_session_end*" checks its flow entry in the QPD and when found deletes it.

As shown in Figure 9.11, the overhead flow processing of RSVP is far higher than that of PDERRM and also the resource allocation processes are simpler in PDERRM than in RSVP.

As highlighted in Chapter 2, there are two types of services defined for IntServ-RSVP model— *Controlled Load* and *Guaranteed Quality of Services.* At the time of this experiment, the OPNET RSVP implementation was the Controlled Load Service, and it was the one used for this experiment. PDERRM implementation used in comparison was the basic service model of PDERRM.

OPNET Documentation was consulted on the work presented in this section. Some of the ideals used—the RSVP part, were borrowed from OPNET Documentation **[OpnetDoc03]**.

9.5.3 Simulation Experiment

The simulation experiment made use of two sources of application traffic-flows as shown in Figure 9.12. In the RSVP scenario, two applications were competing for the same resources. Each of the two *sender hosts* initiated and carried out one application flow session with its *receiver host.* One of the sender hosts made use of RSVP resource reservation and allocation for its traffic-flows while the other source made use of best-effort services. In the PDERRM scenario, the same topology and traffic setup as in RSVP were used. One of the sender hosts made use of the PDERRM resource allocation while the other sender host made use of best-effort. The simulations were run and statistics collected. The results of the experiments were displayed for comparison with RSVP, best-effort and PDERRM services.

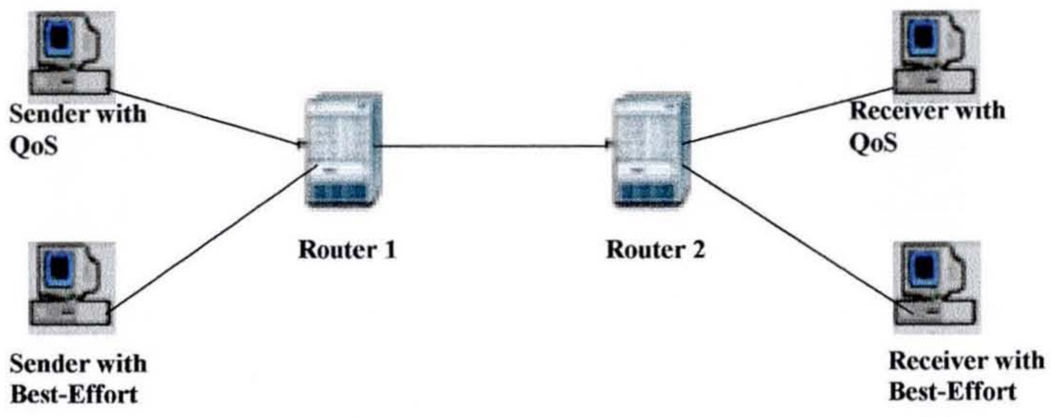
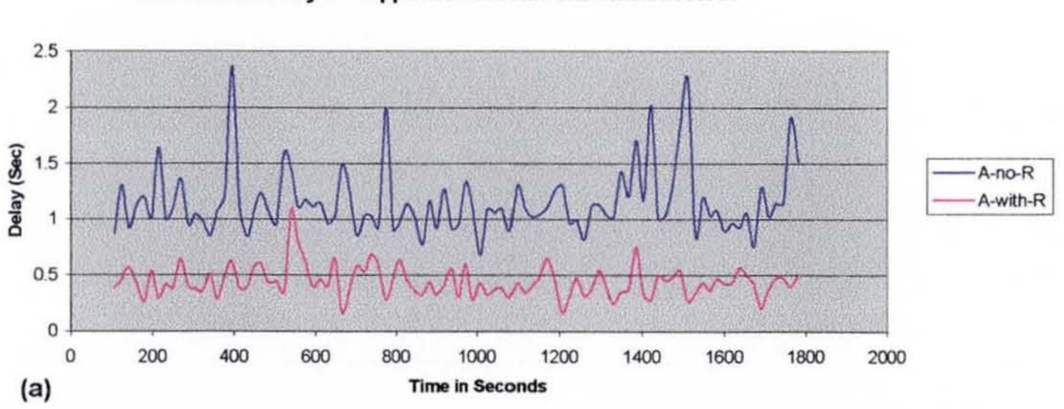


Figure 9.12: Showing network topology for comparing RSVP with Best-effort and comparing RSVP with PDERRM.

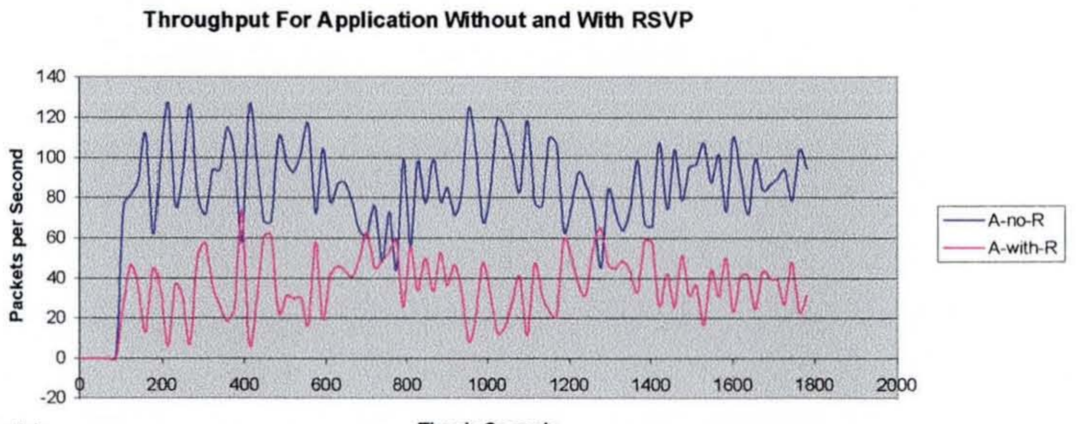
9.5.4 Results and Comments

The results are shown in Figures 9.13 (a) to (d). Figure 9.13(a) shows the results of the experiment to compare RSVP services with best-effort services. As shown in the graph, the flow that made use of RSVP performed better than the flow that made use of best-effort services for end-to-end delay. Figure 9.13(b) shows the results for comparing throughput for application using RSVP with application using best-effort. The results reveal that throughput is higher with best-effort than with RSVP. We believe this would be due to the RSVP overhead and high process threads involved in the resource allocation with RSVP. Figure 9.13(c) is used to compare end-to-end delay for application that made use of RSVP with application that made use PDERRM. As shown in the graph, flows that made use of PDERRM performed far better than flows that made use of RSVP.

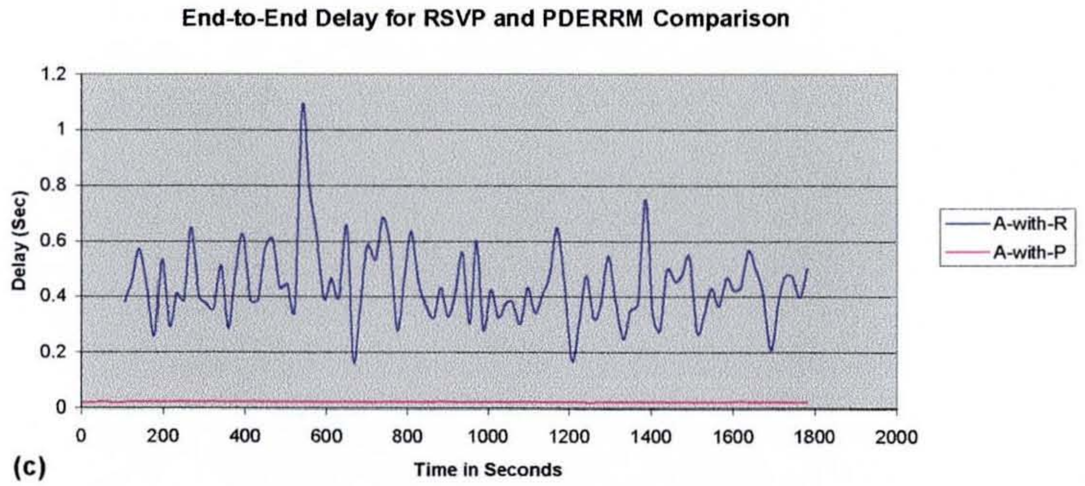


(a)

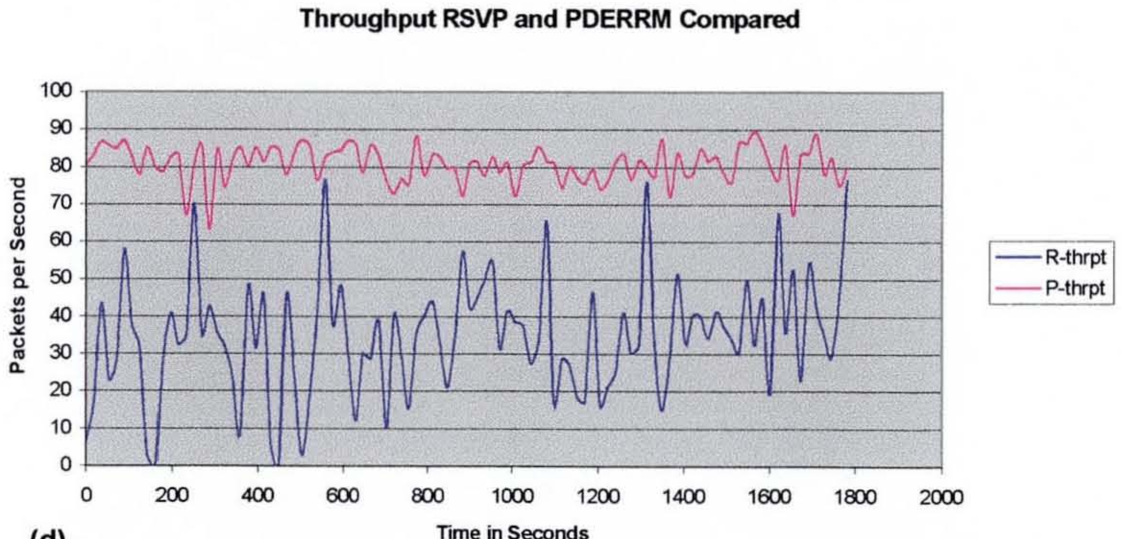
Key: A-no-R = Application-no-RSVP, A-with-R = Application-with-RSVP



(b)



(c)



(d)

Key: A = Application, R = RSVP, P = PDERRM, thrpt = throughput

Figure 9.13: (a) RSVP compared with Best-effort on Delay. (b) RSVP compared with Best-effort on Throughput. (c) RSVP compared with PDERRM on Delay. (d) RSVP compared with PDERRM on Throughput.

Figure 9.13(d) is used to compare flow that made use of PDERRM with flow that made use of RSVP on throughput. The results show PDERRM out performs RSVP also on throughput. This is simply due to the fact that, PDERRM incurs very low overhead.

9.6 Simulation Action Domain (SAD) In Which PDERRM Was Test For Scalability (Scheme E)

PDERRM performance evaluation under this SAD was focused on simulation experiments to test the PDERRM basic service model for scalability. The objective was to test if the PDERRM basic algorithm would be elastic and thus scale for large network use.

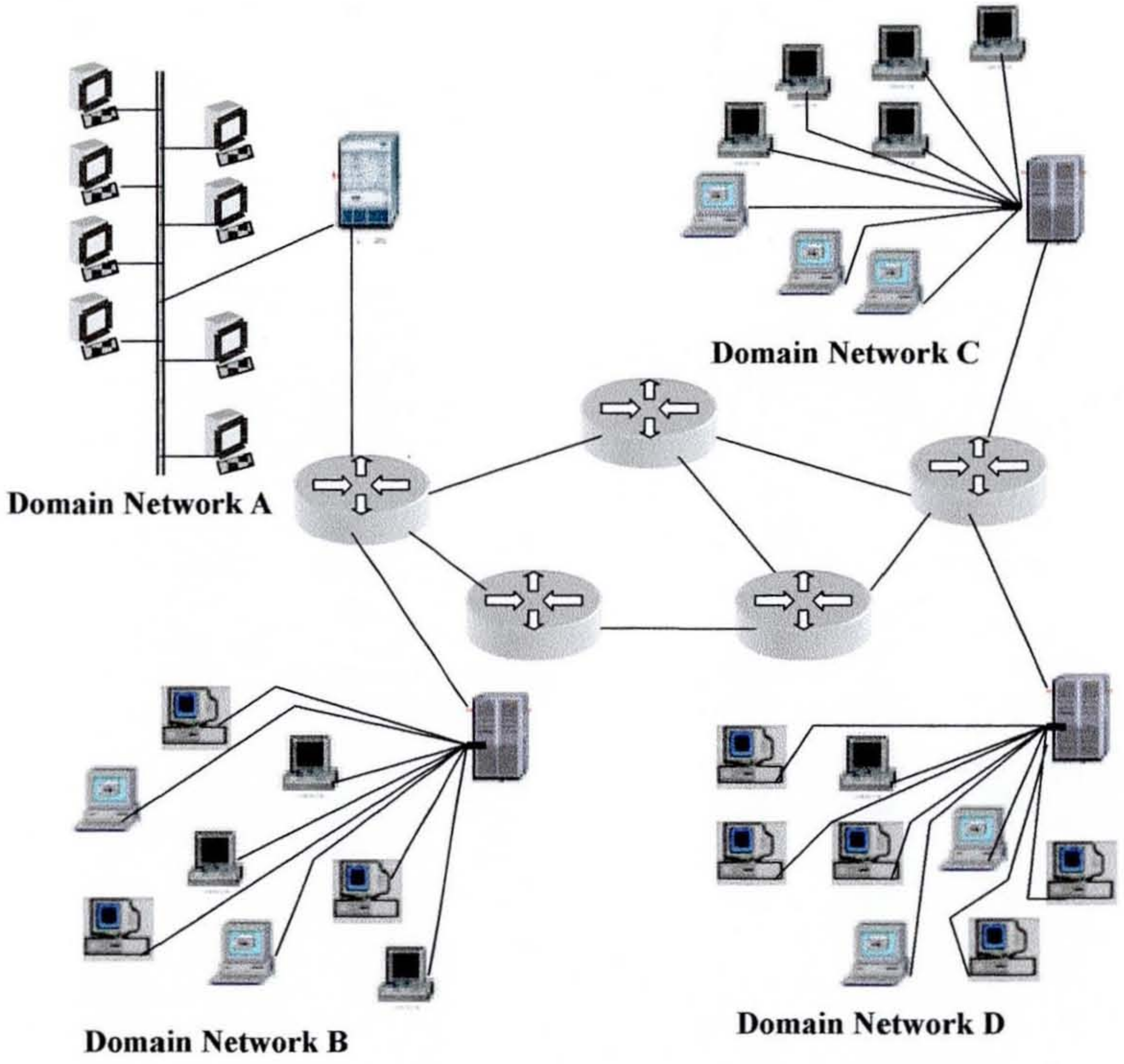


Figure 9.14: Large Network used to test PDERRM for Scalability.

9.6.1 Simulation Experiment

As in the experiment on Section 9.4.1, four classes of traffic were used. Also as in previous experiments, wide range of traffic parameters for each of the traffic classes were used in generating various traffic profiles as input to the network in the experiment. The network topology used for the experiment is shown in Figure 9.14. In the figure, the edges of the network are made up of a *number of hosts* in either a point-to-point centralised switched topology or Ethernet centralised switched topology. Each edge network is labelled as *Domain Network X*, where $A \leq X \leq D$, and are connected through the *core network* as illustrated in the Figure 9.14. In the experiment, each host in each of the Domain Network would send or receive traffic such that traffic-flows were evenly distributed in the network. There were two basic simulation scenarios. The first simulation scenario, which served as the *baseline*, was designed and configured such that the network was operated on the best-effort service mode. The second simulation scenario was designed and configured such that the network was operated on PDERRM basic service mode.

9.6.2 Results and Comments

The results of the simulations are shown in Figure 9.15. The end-to-end global delay results are the results shown in Figure 9.15. The delay results show that PDERRM maintains its performance as a very good QoS architecture in large network, and thus scales very well.

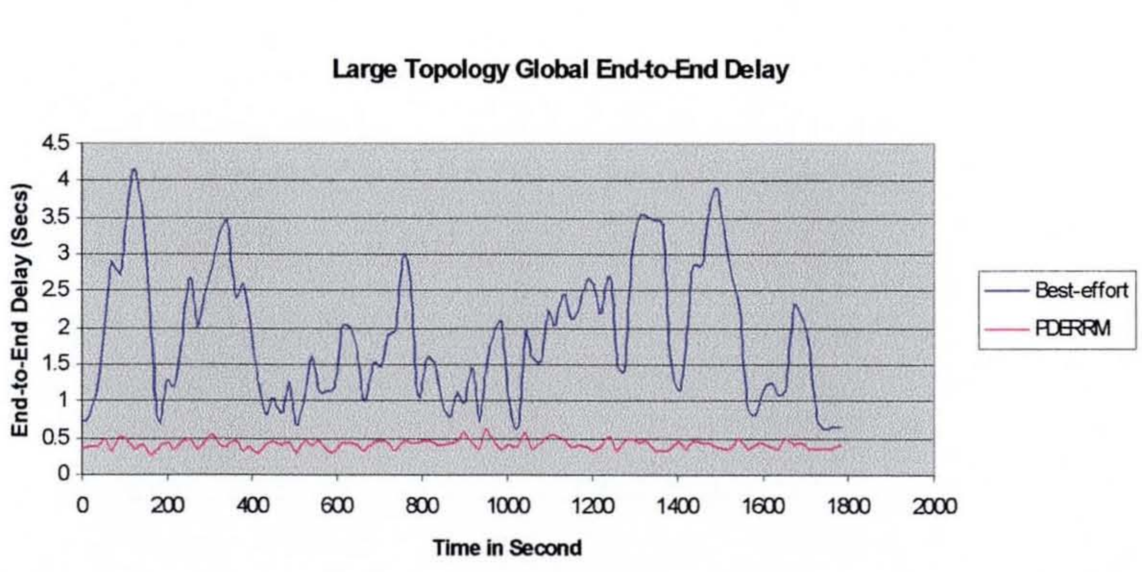


Figure 9.15: Showing End-to-End Delay for Best-effort and PDERRM.

The good scalability results as shown in the Figure 9.15 is a consequence of the fact that PDERRM is a distributive QoS architecture. Unlike the centralised resource brokerage systems in which each node must communicate with a centralised server before making its resource allocation decision, PDERRM-aware nodes make independent decisions on resource allocation based on information available in PDERRM agents within the nodes. In the centralised systems, QoS related communication in the network increases as the topology increases. The servers in the network will have to cope with large volume of QoS related messages. The reverse is the case in the PDERRM architecture, the number of QoS related messages in relation to each node is fixed no matter the increase in the network topology.

Summary

The simulation experiments for performance evaluation of PDERRM has been carried out in this chapter. Five main simulation experiments have been performed to determine PDERRM strengths for support of multiservice QoS. The first-three of the five main experiments were schemes designed to test some basic features of the PDERRM QoS architecture. The results of the three experiments were very impressive and show the PDERRM QoS architecture as a promising candidate for IP convergence. The fourth main experiment was used to compare RSVP with PDERRM, and PDERRM was found to outperform RSVP in both end-to-end delay and throughput. The last experiment was used to test PDERRM for scalability, and PDERRM found to be scalable.

CHAPTER 10

Conclusion and Discussion on Future Work

In this last chapter, the work covered in this thesis in relation to conclusive viewpoints and concepts will be summarised. In addition, future research work on the main empirical investigation work presented in this thesis will be highlighted. The chapter will focus on conclusions drawn from the various results of the investigations and the body of knowledge that the work has added to the engineering and technological environment.

In Section 10.1, we will briefly summarise Internet Technology in relation to IP QoS architecture as presented in this thesis. A concise abstraction of the work on Optimum Number of Traffic Queuing Classes (ONTQC) to support IP convergence will be presented in Section 10.2. The presentation will include conclusions drawn from the results of the experiments and the addition to knowledge, which the work has provided. The coverage in Section 10.3 will be on the novel IP QoS architecture— *Pre-deterministic Distributed Event Response Resource Management* (PDERRM). The discussion in the Section 10.3 will include summary of the features of the architecture, conclusion from the experimental results on performance evaluation of the architecture, and the novelty in its features. In Section 10.4, the discussion will be based on future work on ONTQC, and in Section 10.5 the focus and coverage will be on future work on PDERRM.

10.1 Summary of Internet Technology in Relation to IP QoS

The evolution of the Internet, its ubiquity and success has been phenomenal in the last two decades. These have been attributed to the flexibility of its technology and simplicity of its network. The power of the Internet cannot be overemphasised. Its ubiquity cuts across every aspect of our daily life. The nomination of IP Networks for convergence of all other telecommunication services is as a result of the flexibility of its technology. The Internet's traditional best-effort service model becomes inadequate as the ever-growing end-users increased exponentially and various new applications emerged. The inadequacy of the best-effort service model resulted into application performance degradation that manifested as network vulgaries such as packet delivery high latency, jitter and loss. QoS architectures were designed to improve applications performance with support for integrated and differentiated services in the heterogeneous application environment found in the Internet. The QoS architectures will normally consist of different service models for different classes of application and should include service models that support predictability and assurance in packet delivery time as against variability and uncertainty in packet delivery time under best-effort service model. Predictability in packet delivery time is especially important for time sensitive applications. ***Conclusively, the Internet is one of the greatest inventions in the last century.***

10.2 Summary and Conclusion on ONTQC work

It has been widely accepted within the Internet research community that multiple queuing systems are required in an environment of multiservice networks to provide partitioning of resources that are necessary for differentiated services. There have been various proposals / specifications on the number of traffic queuing classes that will support IP convergence, but none has supported the suggestions with verification in the form of analytical or empirical investigation. The work on the empirical investigation to determine the Optimum Number of Traffic Queuing Classes (ONTQC) to support IP convergence and its results has helped to fill the knowledge vacuum on this engineering quest.

The work on ONTQC provides a number of novel ideas. These include:

- The simple algorithm known as *Fission of the Rightmost Block First* (FRBF) that was used to decompose the root queues to its component queue classes based on priority of the queue components (see Section 5.3.5). The algorithm can be implemented either manually or automatically. It will find useful application in a situation where a job is required to be broken down to its component parts and attention given to the component parts in a sequential manner that reflect priority attached to the component parts.
- The meta-heuristic analysis method used to predict simulation outcome. The operation involved the introduction of new techniques to analyse queue contents and queue operations in packet switched network (see Section 5.4.1).

- The use of matrices and vectors to analyse work wasted in a *non-work conserving* queue (see Section 7.2.1 to Section 7.2.7).

It has been generally assumed that fine granular resource allocation increases as the number of traffic queuing classes increases, and that, the finer the granularity of resource allocation, the better the QoS performances of the applications concerned. This is based on the intuitive consideration that the more you break a broad class of applications down to their individual classes in terms of a mono queuing system to a multiple queuing system, the more you provide for individual application class segregation which will allow individual application class QoS requirements to be met.

Contrary to the general assumption as mentioned above, the results of the ONTQC simulation provide a better insight to the operation of queue disciplines in packet switched networks. It is now clear that, as the number-of-queues in a multiple queue system exceeds a certain threshold, the multiple queue system becomes less efficient. This is due to the fact that the multiple queue system is a *non-work conserving* system. Work is wasted by the *server* making transitions from one queue to another.

Summary of the results

As we increase from a one-queue simulation implementation to an eight-queue simulation implementation the results are summarised in Table 10.1

Table 10.1

Traffic Class	**Performance Metrics**	**Queue Systems for Best Performance**
Class-1 Traffic	End-to-end Delay	Four-queue & Five-queue
	Throughput	Four-queue & Five-queue
Class-2 Traffic	End-to-end Delay	Three-queue
	Throughput	Three-queue
Class-3 Traffic	End-to-end Delay	Three-queue
	Throughput	Three-queue
Class-4 Traffic	End-to-end Delay	Five-queue
	Throughput	Three-queue

Table 10.1 continues

Traffic Class	**Performance Metrics**	**Queue Systems for Best Performance**
Class-5 Traffic	End-to-end Delay	Six-queue
	Throughput	Six-queue
Class-6 Traffic	End-to-end Delay	Seven-queue & Eight-queue
	Throughput	Seven-queue & Eight-queue
Class-7 Traffic	End-to-end Delay	Two-queue
	Throughput	Two-queue
Class-8 Traffic	End-to-end Delay	Two-queue
	Throughput	Two-queue

The combined overall global results revealed that a **three-queue** simulation implementation gives the best overall performance for both ***end-to-end delay*** and ***throughput. Thus three-queue system is the best candidate for ONTQC to support integrated services.*** For the global overall performance, the results showed that a **two-queue** implementation takes the second position as the **second best** overall performances in both ***end-to-end delay*** and ***throughput***.

The results as shown above reveal that, contrary to general assumptions, the QoS metrics performance of applications do not improve linearly as the number of multiple queue increases. **The results will be useful to the Internet engineering community and will serve as a reference point for would be implementers of multiple queue system in packet switching networks.**

In view of the interesting results obtained, the work on ONTQC presented in this thesis has proffered very good answer to the research question posed on ONTQC to support IP convergence.

10.3 Summary and Conclusion on the Work of PDERRM

The Predeterministic Distributed Event Response Resource Management (PDERRM) QoS architecture has interesting features, which could support and enhance wide spread deployment of QoS in the Internet. It is a QoS architecture designed with simplicity, elegance and robustness in mind.

The development of PDERRM followed three main work phases— problem formulation phase, experimental design phase and performance evaluation phase. We will now briefly summarise each of these phases and then make our conclusion.

10.3.1 Problem Formulation

The problem formulation phase concerns the motivation and drive for the PDERRM work. Despite many years of research work by IETF that resulted in the development of two standard IP QoS architectures— IntServ and DiffServ, and the contribution of many IP QoS mechanisms made by others in the research community, end-to-end deployment of IP QoS still remains elusive in the Internet. The problems are multifarious. One aspect of the problem may not be unconnected with the need for simplicity in IP QoS architectures since the Internet's phenomenal success has its root in its *simple network concept,* which has scaled very well. By simple network we mean the Internet technological framework in which complexities lies at the end nodes while the network nodes are relatively simple. A simple IP QoS architecture that has built-in simple and flexible deployment processes will enhance wide spread deployment of QoS in the Internet. Thus the problem formulation that resulted in the empirical work on PDERRM revolved round the notion of simplicity, elegance, flexibility and robustness in the IP QoS architecture.

10.3.2 Brief Overview of PDERRM Design

PDERRM was designed such that its process models are simple and elegant for its purpose. In view of the diversities in processing capacities of various nodes and the variance in bandwidth that exists in the Internet— a conglomeration of networks, PDERRM has been designed with the notion of flexibility and robustness, seen as of prime concern in its design objective. The simple and flexible QoS architecture could enhance wide spread deployment of QoS in the Internet.

Basically the PDERRM architecture consist of the following:

1. A process model that determined the magnitude of network resources capacity of a node. Another process model that shares the network resources in a node

among different traffic classes contending for network resources according to their QoS need. Initially the values of resource quota for each traffic class are determined on static basis. The static quota values can be shifted based on dynamic traffic loading in the node and the permissible ranges of values allowed.

2. The process models that ensure each traffic class complies with its resource quota in the resource utilisation in the node.
3. The process models that queue traffic-flows according to their classes and schedule the flows in accordance with the resource quota allocated to each traffic class.
4. The QoS Manager in the node that co-ordinate the activities of the basic processes highlighted above.

The abstracted block diagram of the architecture of PDERRM is shown in Figure 10.1

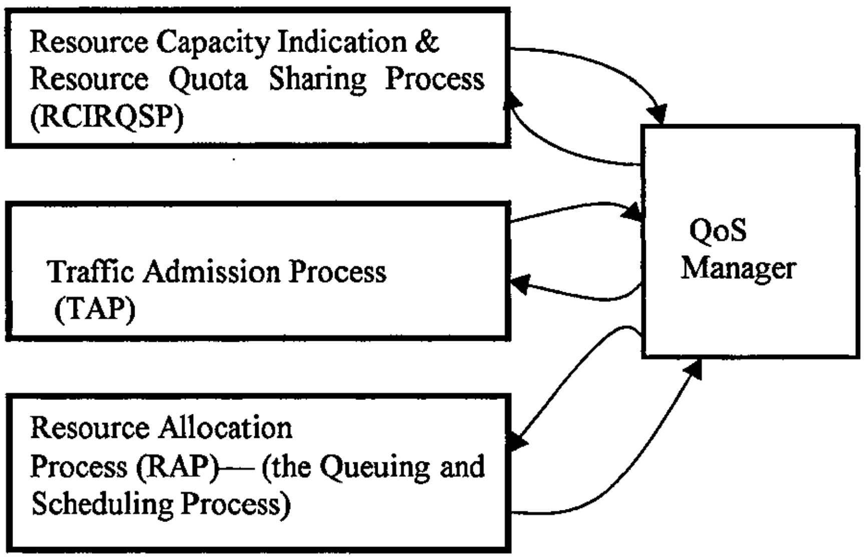

Resource Capacity Indication & Resource Quota Sharing Process (RCIRQSP)
Traffic Admission Process (TAP)
Resource Allocation Process (RAP)— (the Queuing and Scheduling Process)
QoS Manager

Figure 10.1 Abstracted Key Action Block Diagram of PDERRM

As illustrated in the figure, simply the QoS manager gets information on values of node resource capacity and sharing of resource from the RCIRQSP and uses the values to parameterise and control the functionality of the TAP and RAP. The TAP ensure that traffic-flows loading in the node do not exceed the node capacity, and

RAP— i.e. the queuing and scheduling discipline ensure that each traffic class receive its resource quota allocation.

PDERRM has very interesting features, which include the following:

- Simplicity— the basic processes are simple and elegant for their purpose.
- Hybrid functionality— it is possible to operate on the basis of aggregate flow resource allocation as found in the DiffServ architecture or on per-flow resource allocation as found in IntServ architecture.
- Mutational functionality— ability to adapt its process model to processing capability of a node.
- Flexibility in functionality— contains stand-alone options depending on QoS operation desirable in a node.

The architecture operates fully as a distributed protocol, which enhance scalability.

10.3.3 Brief Overview of PDERRM Performance Evaluation

The results of simulation on performance evaluation of PDERRM are very impressive. Five main simulation experiments were carried out to test the performance of its basic features. All the results showed that PDERRM is a very good QoS architecture. The results are summarily rehearsed as follows:

- Simulation **Scheme-A,** (Chapter 9, Section 9.2) that compared PDERRM basic functionality with best-effort service model has the results shown in the ***Delay Vector* A.**
- Simulation **Scheme-B,** (Chapter 9, Section 9.3) that compared the PDERRM *source control* with best-effort service model has the results shown in the ***Delay Vector* B.**
- Simulation **Scheme-C,** (Chapter 9, Section 9.4) that compared PDERRM *edge control* with best-effort service model has the results shown in the ***Delay Vector* C.**
- Simulation **Scheme-D,** (Chapter 9, Section 9.5) that compared PDERRM basic functionality with IntServ-RSVP service model has the results shown in the ***Delay Vector* D.**
- Simulation **Scheme-E,** (Chapter 9, Section 9.6) that tested PDERRM for scalability has the results shown in the ***Delay Vector* E.**

***Delay Vector A*:**

	Flow-class 1 (*mean delay*)	Flow-class 2 (*mean delay*)	Flow-class 3 (*mean delay*)	Global (*mean delay*)
Best-effort	[0.922	1.262	1.980	1.291]
PDERRM	[0.375	0.444	0.717	0.489]

***Delay Vector B*:**

	Flow-class 1 (*mean delay*)	Flow-class 2 (*mean delay*)	Flow-class 3 (*mean delay*)	Global (*mean delay*)
Best-effort	[13.131	13.289	13.441	13.235]
PDERRM	[0.295	0.420	1.085	0.502]

***Delay Vector C*:**

	Best-effort (*mean delay*)	PDERRM-basic (*mean delay*)	PDERRM-edge-control (*mean delay*)
Global	[1.886	0.437	0.413]

***Delay Vector D*:**

	RSVP	PDERRM
Mean Delay	[0.415	0.023]

***Delay Vector E*:**

	Best-effort	PDERRM
Global mean delay	[1.886	0.413]

As shown above the results are very impressive. It is particularly interesting to note that PDERRM gave superior performance compared with the standard IntServ-RSVP paradigm. The performance metric indicators have shown that PDERRM is an improved IP QoS architecture compared with the standard architecture.

Thus the work on PDERRM presented in this thesis has successively proposed effective solutions to the research problems posed concerning designing a simple, elegant and robust IP QoS architecture.

10.4 Future Research Work on ONTQC

Further to the work presented in this thesis on ONTQC, the consideration of the *Virtual Queuing System* (VQS) and *Optimum Queuing Discipline* (OQD) for IP convergence require attention.

Theoretically and intuitively, VQS provides a flexible way to implement multiple queuing system. Since its *soft state* makes it possible to install and remove at will any multiple queuing structure that has been implemented for multiservice in network nodes. Under this consideration for future study and research, we will look at the reason why VQS needs to be considered.

There have been a plethora of publications that accepted that Class Based Queuing (CBQ) and Weighted Fair Queuing (WFQ) together with their variants are the most suitable queuing and scheduling discipline for multiservice in packet switched networks. P Callinan et al concluded in their simulation experiment that CBQ might be more advantageous to continuous media (audio & video) than WFQ, although WFQ might be suitable for data traffic **[Cal et al.00]**, and Fayaz A. Shaikh et al indicated in their experiment that a hybrid of CBQ and WFQ gave better performance than others **[Sha et al.02]**. Judging from the text of the literature review, there has not been assertive statement on Optimum Queuing Discipline (OQD) for IP convergence. Thus we will discuss briefly on the OQD and VQS under this umbrella of Future Research Work on ONTQC.

10.4.1 Virtual Queuing System (VQS)

The OPNET software simulation package contains process models for implementation of all standard protocols and algorithms used in packet switched networks. These include process models for implementation of all standard IP QoS mechanisms. OPNET implements the process models of the standard IP QoS queuing disciplines as global functions, which any node in the network being simulated could call at will. In effect the nodes would make use of the IP QoS queuing disciplines as Virtual Queues. In the course of the experiment on ONTQC, OPNET standard models for CBQ were sometimes used. In one particular experiment, which involved eight simulation scenarios in which the author used OPNET standard models, the first scenario was configured such that the forwarding nodes made use of the FIFO queuing discipline. The remaining seven scenarios were configured such that the forwarding nodes made use of multiple queuing (CBQ) disciplines. Simulation scenario-2 made use of a two-queue system, simulation scenario-3 made use of a three-queue system, etc. The results of the experiment showed that application performances were far better in the two-queue multiple queuing system than in the mono-queue system, but the results for all the multiple queuing systems (two-queue to eight-queue) were the same, i.e. no differences were seen. This was strange, consequently the author designed his own CBQ process model, but instead of designing this as a virtual CBQ, it was designed for hard state CBQ. The results of simulations carried out with the author's CBQ process models were very impressive. Each of the multiple queue results was different, unlike the OPNET implementation of CBQ. It was then concluded that, either there was a bug in OPNET process models of the CBQ or it was the peculiarity of the Virtual Multiple Queue systems.

In view of the operational problem highlighted above, there is a need to carry out future work on the investigation of Virtual Multiple Queues systems in order to understand its strengths and weaknesses. It is pertinent to note here that Srisanka Kunniyur and R. Srikant **[KunSri01]** in their work on VQS did not address the issue of performance of *virtual queue* implementation compared with *hard state queue* implementation. The future studies should look into the comparison performance of Virtual Multiple Queues systems with Hard State Multiple Queues systems in relation to their suitability for multiservice network.

10.4.2 Optimum Queuing Discipline (OQD) for IP Convergence

As highlighted at the beginning of this section, the CBQ and WFQ disciplines are the most suitable for multiservice networks based on consensus from the research community. The work of P. Callinan et al **[Cal et al.00]** lacks a conclusive assertive statement on OQD for IP convergence. Also the work of Fayaz A. Shaikh et al **[Sha et al.02]** was not emphatic on the issue. There is a need for further investigation to identify between CBQ and WFQ, which would be more suitable for multiservice network. Both CBQ and WFQ have their pros and cons. A study on hybrid of the two queuing disciplines would be a worthwhile venture. The future study could include the development of a new queuing discipline that could support integrated services.

10.5 Future Work on PDERRM

Generally for any IP QoS architecture to evolve to full maturity, it would require years of collective study and investigation from the research community. The developmental processes may require modification and additions to the original design. For example the IntServ and DiffServ architectures required the attentions of IETF and other researchers in the Internet research community for many years before the IP QoS architectures become standards. PDERRM cannot be an exception.

At the present stage of PDERRM development and conceptual understanding, the areas that call for further study and investigation include the following:

- The best method or procedure to determine resource utilisation for each of the traffic classes in order to arrive at the best value for a *static resource quota* for each traffic class.
- PDERRM feature for per-flow resource allocation and the use of the QoS Parameter Database.
- PDERRM interworking with other standard IP QoS architectures.
- PDERRM development as an Internet protocol suitable for real life implementation and deployment.

Each of these will now briefly be discussed.

10.5.1 Best method to determine resource utilisation for each of the traffic classes

As stated in Section 8.3.4.2, there are various methods for measuring traffic intensities of each class of flows going through a node. Through measurement, we can compute the value of resources consumed by each class of traffic in a predefined time interval. It was suggested in the section mentioned above that, the number of options available for measuring traffic intensities and resource utilisation include: (1) The use of a *Traffic Network Analyser* which could provide values for traffic loading of the node for each class of traffic, at predetermined interval of times. (2) Adaptation of Loughborough University Network Monitoring Systems. (3) The use of a Neural Network Algorithm (NNA) and Fussy Logic Management (FLM). (4) Direct Computation, which involves scanning the input to the node, and measuring the values of cumulating flows for each class of traffic in each *window*.

There is a need to carry out investigation on each of the options listed above; relate one to another and work out a collective comparative performance for the options, from which the best suitable option can be deduced for PDERRM operation.

10.5.2 PDERRM feature for per-flow resource allocation

PDERRM has a built-in mechanism for per-flow resource allocation. The idea that is being exploited here has to do with the DSCP value and establishment of the QoS Parameter Database. The DSCP value has a hierarchical value interpretation in which a section of its value will represent the class to which the traffic belongs and the other section of its value will represent the value that has to be mapped to a corresponding value in the QoS Parameter Database. The QoS Parameter Database has entries for wide range of resource requirements of traffic flows in the form of *IntServ Token Bucket Parameters*. This per-flow resource allocation feature of PDERRM requires further work. It needs to be studied and investigated and if need be, refined.

10.5.3 PDERRM interworking with other standard IP QoS architectures

PDERRM inter-operation with other IP QoS architectures is a necessity, since it will need to co-exist in the Internet with other IP QoS architectures. The work will include how PDERRM service classes will map into other IP QoS architecture's service classes and how PDERRM processes and other IP QoS architecture's processes will handle the mappings. Also the work will include PDERRM protocol interaction with other IP QoS architectures, especially in the area of interpretation of each other's messages.

10.5.4 PDERRM development to Internet protocol application level

The development of PDERRM to a level ready for real life implementation and deployment will be an attractive work. The work will be attractive in view of the impressive results obtained from PDERRM performance evaluation. Consequently, it is recommends that further work on PDERRM development as an Internet protocol be carried out as soon as possible.

Summary

The summary and conclusions on the work presented in this thesis are covered in this chapter. The Internet technology is flexible and this accounts for its nomination for IP convergence. IP QoS architectures help to improve the performances of applications placed across the networks. The results of empirical investigation on ONTQC to support integrated services showed that *three-queue* multiple queue system gave the best global (over all) performance in both end-to-end delay and throughput. The results of performance evaluation of PDERRM QoS architecture were impressive. PDERRM outperform RSVP in both end-to-end delay and throughput.

Future work could be considered on Virtual Queue Systems (VQS) and Optimum Queue Discipline (OQD) for IP convergence. Also it is recommended that future work be carried out on development of PDERRM to full fledge Internet protocol suitable for real life deployment.

References

The References are divided into three sections: *Section 1 is concerned with Internet (IETF) documents, Section 2 contains organisational documents, and Section 3 has references in the general format.*

Section 1

RFCs (Request for Comments) are typically Internet standard documents accepted for information purposes on Internet technology.

Internet Drafts are Internet Engineering Task Force (IETF) documents that are works in progress, which are not yet standards. Such documents are periodically updated and could become standards.

[RFC 768] Postel J., *User Datagram Protocol,* August 1980.

[RFC 791] *Internet Protocol,* STD 5 (or ISI), September 1981.

[RFC 793] *Transmission Control Protocol* ISI, September 1981.

[RFC 950] Mogul J. and Postel J., *Internet Standard Subnetting Procedure*, August 1985.

[RFC 1122] Braden R., *Requirements for Internet Hosts—Communication Layers*. STD 3, RFC 1122. (October 1989).

[RFC 1349] Almquist P., *Type of Service in the Internet Protocol Suite*, July 1992.

[RFC 1363] Partridge C., *A Proposed Flow Specification,* Network Working Group, September 1992.

[RFC 1633] Braden R., Clark D., Shenker S., *Integrated Services in the Internet Architecture: an Overview*, June 1994.

[IntServ94]	*Integrated Service (IntServ) Charter*, 29th IETF Meeting, March 1994 in Seattle. (http://www.ietf.cnri.reston.va.us/proceedings/94mar/charters/intserv-charter.html), March 1994.
[RFC 1889]	Schulzrinne H. Casner S., Frederick R., Jacobson V., *RTP: A Transport Protocol for Real-Time Applications*, Audio-Video Transport Working Group, January 1996.
[RFC 2205]	Braden R., Zhang L., Berson S., Herzog S., Jamin S. *Resource reSerVation Protocol (RSVP) Version 1 Specifications*, September 1997.
[RFC 2209]	Braden R., Zhang L., *Resource reSerVation Protocol (RSVP) – Version 1 Message Processing Rules*, September 1997.
[RFC 2210]	Wronclawski J., *The Use of RSVP with IETF Integrated Services* September 1997.
[RFC 2211]	Wroclawski J., Specification *of Controlled Load Network Element Service*, Internet Draft, draf-ietf-intserv-ctrl-load-svc-02.txt, June 1996.
[RFC 2212]	Shenker S, Patridge C., Guerin R., *Specification of Guaranteed Quality of Service*, Internet Draft, draft-ietf-intserv-guaranteed-svc-05.txt, July 1996.
[RFC 2215]	Shenkers S., Wroclawski J., *General Characterization Parameters for Integrated Service Network Elements*, September 1997.
[PREDV-SRVC]	Shenker S., Partridge C., Davie B., Breslau L., *Specification of Predictive Quality of Service*, Internet Draft, draft-ietf-intserv-predictive-svc-01.txt, 1995.

[DSFWK]	Bernet Y., Binder J, Blake S, Carlson M., Carpenter E., Keshav S., Davies E., Ohlman B., Verma, D. Wang Z, and Weiss W., *A Framework for Differentiated Services*, < draft-ietf-differv-framework-02.txt> February, 1999.
[RFC 2474]	Nichols K, Blake S., Baker, Black D., *Definition of Differentiated Services Field* (DS Field), December 1998.
[RCF 2475]	Blake et al., An *Architecture for Differentiated Services*, IETF, December 1998.
[RFC 2597]	Heinanen J., Baker F., Weiss W., and Wroclawski J., *Assured Forwarding PHB Group*, June 1999.
[RFC 2598]	Jacobson V., Nichols K., and Poduri K., *An Expedited Forwarding PHB*, June 1999.
[RFC 2814]	Yavatkar R., Hoffman D., Bernet Y., Baker F, Speer M, *SBM (Subnet Bandwidth Manager): A Protocol for RSVP-based Admission Control over IEEE 802-style of LANs*, May 2000.
[MPLSfrwk]	Callon R., Doolan P., Feldman N., Fredette A., Swallow G., Viswanathan A., A *Framework for Multi-Protocol Label Switching, Internet* Draft, draft-iet-mpls-05-.txt, September 1999.
[MPLSarch]	Rosen C., Viswanathan A., & Callow R., "Multi-Protocol Label Switching Architecture," Internet Draft, draft-ietf-mpls-arch-07.txt, July 2000.
[COPS]	Boyle J., Cohen R., Durham D., Herzog S., Rajan R., Sastry A., *The Common Open Policy Service (COPS) Protocol*, Internet Draft, draft-ietf-rap-cops-04.txt, December 1998.
[COPS-usage]	Reichmeyer F. et al., *COPS Usage for Policy Provisioning*, Internet Draft (work in progress) draft-ietf-rap-pr-01.txt October 1999.

[ISSLL-802] Ghanwani A., Wayne Pace J., Srinivasan V., Smith A, Seaman M., *A Framework for Providing Integrated Service Over Shared and Switched IEEE 802 LAN Technologies,* Internet draft, draft-ietf-issll-framework-05.txt, May 1998.

[SignalPro] Brunner M. Ed., *Requirements for Signalling Protocols,* IETF draft, work in progress, http://www.ietf.org/internet-drafts/draft-ietf-nsis-reg-07.txt, March 2003.

[SNMP] Rose M.T. and McCloghriek., *How to Manage Your Network Using SNMP (RFCs 1157, 1441 to 1452).* Englewood Cliffs, NJ; Prentice Hall, 1995.

Section 2

[3ComPro&Srv] 3 Com Product & Services, "*3 Com. Corporation Unveils Next Generation Total Control multiservice Access Platform at Supercom 2000.*" http://www.3com.com/solutions/svprovider/iptelephony2/ip_press_releases.html.

[eWeek99] "Cisco Launches Voice/Data Quality of Service Effort" (http://www.zdnet.com/eweek/stories/general/.html).

[802.1D&p Yr.93] IEEE 802.1D and p Standards "Information Technology Telecommunication and Information Exchange between Systems- Local and Metropolitan area Networks- Common Specifications-Part 3: Media Access Control (MAC) Bridges: P802.1D/D 1993.

[802.1D&pYr.98-05] IEEE 802.1D and p Standards. "Information Technology Telecommunication and Information Exchange between Systems- Local and Metropolitan area Networks- Common Specifications-Part 3: Media Access Control (MAC)

Bridges: P802.1D/D, 1998-2005.

[NSDoc00] Kevin Fall and Kannan Varadhan (Editors) *NS Notes and Documentation,* The VINT Project— A Collaboration between researchers at UC Berkeley LBL, USC/ISI and Xerox PARC February 2000.

[NSMan00] NS—Network Simulator (Version 2) *Manual pages* http://www.isi.edu/nsnam/ns/ns-man.html.

[OpnetDoc03] OPNET Software Documentation, MIL 3, Inc., 3400, International Drive, N.W, Washington D. C. 20008 2003.

Section 3

[Abe et al.06] Abendroth, Dirk et al., *Solving the trade-off between fairness and throughput: Token bucket and leaky bucket-based weighted fair queuing schedulers,* International Journal of Electronics and Communications, v 60, n 5, May 2, 2006, p 404-407.

[Aky et al.03] Akyildiz, I.F. et al., A *new traffic engineering manager for DiffServ/MPLS networks: Design and implementation on an IP QoS testbed,* Computer Communications, v 26, n 4, Mar 1, 2003, p 388-403.

[And et al.00] Andersen Niels et al, *Applying QoS Control through Integration of IP and ATM,* IEEE Communications Magazine, July 2000, p130-136.

[Arm00] Armitage Grenville (2000), *Quality of Service in IP Networks: Foundations for a Multi-Service Internet,* Pearson Education Publisher 2000. ISBN 1-57870-189-9.

[Arm00-art]	Armitage Grenville (2000), *MPLS: The Magic Behind the Myths*, IEEE Communications Magazine, January 2000, p124-131.
[AutKir02]	Autenrieth Achim and Kirstädter Andreas, *Engineering End-to-End IP Resilience Using Resilience-Differentiated QoS*, IEEE Communications Magazine, January 2002, p50-57.
[Bak et al.03]	Bak et al., A framework for providing differentiated QoS guarantees in IP-based Networks, Computer Communications 26 (2003) 327–337.
[Bak98]	Baker Fred, *IP QoS*, Business Communications Review, March 1998. Vol. 28, Iss. 3; pg.28, 4pgs.
[Bay72]	Bayes J., *A Minimum Variance Sampling Technique for Simulation Models*, Journal of the Association for Computing Machinery, Vol. 19, No. 4, October 1972, pp. 734-741.
[BhaCro00]	Bhatti, Saleem N. and Crowcroft, Jon *QoS-sensitive flows: issues in IP packet handling, IEEE* Internet Computing, v 4, n 4, Jul 2000, p 48-57.
[Bia et al.02]	Bianchi G. et al., *Per-flow QoS support over a stateless Differentiated Services IP domain*, Computer Networks 40 (2002) 73–87.
[Bia99]	Biagi Susan, *Quest for QoS drives multiple approaches*, Telephony, Chicago: Jan 18, 1999. Vol. 236, Iss. 3; pg 38, 1pgs.
[Bos et al.04]	Bosco et al., *An Innovative Solution for Dynamic Bandwidth Engineering in IP/MPLS Networks with QoS Support*, Photonic Network Communications, 7:1, 37--2, 2004.

[Bou et al.03] Boutaba, Raouf et al., *A Multi-Agent Architecture for QoS Management in Multimedia Networks*, Journal of Network and Systems Management, v 11, n 1, March, 2003, E-Business Management, p 83-107.

[Bouras et al.03] Bouras, Christos et al., *QoS and SLA aspects across multiple management domains: The SEQUIN approach* Future Generation Computer Systems, v 19, n 2, February, 2003, p 313-326.

[Cal et al.00] Callinan Phyllis, Witwit Mehdi and Ball Frank, *A Comparative Evaluation of Sorted Priority Algorithms and Class Based Queueing Using Simulation.* The Management of Multi-service Networks seminar Kausnor UK July 2000, 8pp.

[Car et al.02] Carter S F et al., *Techniques for the study of QoS in IP networks*, BT Technology Journal, Vol. 20 No 3 July 2002, p100-115.

[Car86] Carlson A. Bruce, *Communication Systems: An Introduction to Signals and Noise in Electrical Communication 3rd. ed.*, McGraw-Hill Book Company 1986. ISBN 0-07-100560-9.

[Cha et al.00] Chang, Ruay-Shiung et al., *Design of a multiple leaky buckets shaper*, Computer Communications, v 23, n 13, Jul, 2000, p 1307-1318.

[Che et al.05] Cheng Yu et al., *Efficient Resource Allocation for China's 3G/4G Wireless Networks*, IEEE Communication Magazine, January 2005, p76-82.

[Chen et al.03] Chen Yang, Qiao Chunming, Hamdi Mounir and Tsang Danny H. K., *Proportional Differentiation: A Scalable QoS Approach*, IEEE Communications Magazine, June 2003, p52-58.

[Cho et al.05]	Choi Yong-Hoon et al., *A Framework for Elastic QoS Provisioning in the cdma200 1xEV-DV Packet Core Network*, IEEE Communication Magazine, April 2005, p82-88.
[ChrLie03]	Christin Nicolas and Liebeherr Jörg, *A QoS Architecture for Quantitative Service Differentiation*, IEEE Communications Magazine, June 2003, p38-45.
[Cisco02]	Cisco Systems, Inc. *White Paper: MPLS and IP Quality of Service In Service Provider ATM Networks*, Posted: Thu. Nov 7 10:56:38 PST 2002.
[Coh et al.82]	Cohen J.W. et al., *STRUCTURED MODELING*, Proceedings of the 1982 Winter Simulation Conference, p253-258.
[Com01]	Comer, D.E ed. (2001) *Computer Networks and Internets: With Internet Applications*. Prentice Hall, Upper Saddle River, N.J. London.
[Cor et al.03]	Cortese Giovanni et al., *CADENUS: Creation and deployment of end-user services in premium IP networks*, IEEE Communications Magazine, v 41, n 1, January, 2003, p 54-60.
[DeM et al.00]	Hermann De Meer et al., *Programmable Agents for Flexible QoS Management in IP Networks*, IEEE Journal on Selected Areas in Communication, Vol. 18, No. 2, February 2000, p256-267.
[Dem et al.89]	Demers Alan, Keshav Srinivasan and Shenker Scott, (1989) *Analysis and Simulation of a Fair Queueing Algorithm*, Proc. ACM SIGCOMM", Page 3-12; 1989.
[Dim et al.03]	Dimopoulou Lila et al., *QM Tool: An XMl-Based Management Platform for QoS-Aware IP Networks*, IEEE Networks, May/June 2003, p8-14.

[DouSin99]	Douligeris Christos and Singh Kumar Brajesh, *Analysis of neural-network-based congestion control algorithms for ATM networks*, Engineering Applications of Artificial Intelligence, v 12, n 4, Aug, 1999, p 453.
[DurYav99]	Durham David and Yavatkar Raj, *Inside the Internet's Resource reSerVation Protocol: Foundations for Quality of Service*, John Wiley & Sons, Inc. 1999. ISBN 0-471-32214-8.
[Eng et al.03]	Engel Thomas et al., *AQUILA: Adaptive Resource Control for QoS Using an IP-Based Layered Architecture*, IEEE Communications Magazine, January 2003, p46-53.
[Fin02]	Fineberg Victoria, *A Practical Architecture for Implementing End-to-End QoS in an IP Networks*, IEEE Communications Magazine, January 2002, p122-130.
[FloJac95]	Floyd S. and Jacobson V., *Link-Sharing and Resource Management Models for Packet Networks,* IEEE ACM Transactions on Networking, Vol. 3, No. 4, August 1995, p365-386.
[Fod et al.03]	Fodor Gabor et al., *Providing Quality of Service in Always Best Connected Networks*, IEEE Communication Magazine, July 2003, p154-163.
[GanMcK99]	Gan, D. and McKenzie S., *Traffic policing in ATM networks with multimedia traffic: The super leaky bucket,* Computer Communications, v 22, n 5, Apr, 1999, p 439-450.
[Gio et al.03]	Giordano Silvia et al., *Advanced QoS Provisioning in IP Networks: The European Premium IP Projects*, IEEE Communications Magazine, January 2003, p30-36.

[Goz et al.03] Gozdecki J. et al., *Quality of Service Terminology in IP Networks*, IEEE Communications Magazine, March 2003, p153-159.

[GuiDup02] Guillemin Fabrice and Dupuis Alain, *Basic requirement for the policing functions in ATM networks*, Computer Networks and ISDN Systems, v 24, n 4, May 15, 1992, p 311-320.

[GunRua99] Gung-Chou Lai and Ruay-Shiung Chang, *Support QoS in IP over ATM*, Computer Communications 22 (1999) 411–418.

[Hal98] Hall Eric, *Implementing prioritization on IP networks*, Network Computing. Manhasset: August 15, 1998. Vol. 9, Iss. 15; pg. 76, 3pgs.

[Hei97] Heidelberg, 18 March 1997, EURESCOM P605 - JUPITER Public Seminars on Joint Usability, Performance and Interoperability Trials in Europe. http://www.eurescom.de/~puplic-seminars/1007/ATM/p605/ts1d011.htm.

[Hov et al.05] Hovell Peter, Briscoe, Bob & Corliano Gabriele, *Guaranteed QoS synthesis - An example of a scalable core IP quality of service solution*, BT Technology Journal, v 23, n 2, April, 2005, p 160-170.

[HsiSiv05] Hsieh Hung-Yun and Sivakumar Raghupathy, *Parallel Transport: A New Transport Layer Paradigm for Enabling Internet Quality of Service*, IEEE Communication Magazine, April 2005, p114-121.

[HuiIgo03] Hui-Lan Lu and Igor Faynberg, *An Architectural Framework for Support of Quality of Service in Packet Networks*, IEEE Communications Magazine, June 2003, p98-105.

[Hus00] Huston Goeff, *Internet Performance Survival Guide: QoS Strategies for Multiservice Networks*, New York, Chichester: Wiley 2000. ISBN 0471378089.

[Kle1,76] Kleinrock Leonard, *Queueing Systems Volume 1: Theory*, John Wiley & Sons, Inc. 1976, ISBN 0-471-49110-1.

[Kle11,76] Kleinrock Leonard, *Queueing Systems Volume 11: Computer Applications*, John Wiley & Sons, Inc. 1976, ISBN 0-471-49111-X.

[Kli et al.02] Klincewicz John G., Schmitt James A. and Wong Richard T., *Incorporating QoS into IP Enterprise Network Design*, Telecommunication Systems 20:1,2, 81–106, 2002. Kluwer Academic Publishers.

[Krap00] Krapf Eric, *IP QOS vs. ATM integrated access*, Business Communication Review. Hinsdale: Apr 2000. Vol. 30, Iss. 4, pg. 6, 1pgs.

[Kre93] Kreyszig Erwin, *7th Ed. Advanced Engineering Mathematics*, John Wiley & Sons, Inc. 1993.

[Kri04] Krief Francine, *Self-aware management of IP networks with QoS guarantees,* International Journal of Network Management; Int. J. Network Mgmt 2004; 14: 351–364 (DOI: 10.1002/nem.532).

[KunSri01] Kunniyur S and Srikant R, *Analysis and Design of an Adaptive Virtual Queue (AVQ) Algorithm for Active Queue Management*, ACM SIGCOMM'01, August 27-31, 2001, San Diego, California, USA, p123-134.

[LeoMas03] Leon-Garcia Alberto and Mason Lorne G. *Virtual Network Resource Management for Next-Generation Networks*, IEEE Communication Magazine, July 2003, p102-109.

[LinDeV99]	Lindstrom Annie and DeVeaux Paul, *Follow the QoS road,* America's Network Duluth: Jun 1, 1999. Vol. 103, Iss. 9; pg. 53, 5pgs.
[Mae03]	Maeda Yoichi, *QoS Standards for IP-Based Networks,* IEEE Communication Magazine, June 2003, pg80, 1pg.
[Mai et al.03]	Mai Thi et al., *COPS-SLS Usage for Dynamic Policy-Based QoS Management over Heterogeneous IP Networks,* IEEE Network May/June 2003, p44-50.
[MalRog]	Malaney Robert & Rogers Glynn Tutorial, *Network Elements that Determine QoS Parameters,* CSIRO Telecommunications & Industrial Physics. http://www.atnf.csiro.au/~rmalaney/tutslides/tutorial.
[Man et al.04]	Maniatis Sotiris I., Nikolouzou Eugenia G., Venieris, Iakovos S., *End-to-end QoS specification issues in the converged all-IP wired and wireless environment,* IEEE Communications Magazine, v 42, n 6, June, 2004, p 80-86.
[Mer et al.91]	Merayo Luis A. et al., *A Microprogram-based hardware implementation of the Leaky Bucket algorithm,* Microprocessing and Microprogramming, v 33, n 2, Nov, 1991, p 91-99.
[Mol et al.05]	Molinaro Antonella et al., *A Scalable Framework for End-to-End QoS Assurance in IP-Oriented Terrestrial-GEO Satellite Networks,* IEEE Communication Magazine, April 2005, p130-137.
[Mol89]	Mole R.H., *BASIC graph and network algorithms,* Butterworth & Co. Limited, 1989. ISBN 0-408-01262-5.

[MooSil03]	Moore, Sean S.B and Siller Curtis A., *Packet sequencing: A deterministic protocol for QoS in IP networks* IEEE Communications Magazine, v 41, n 10, October, 2003, p 98-107.
[Myk et al.03]	Mykoniati Eleni et al., *Admission Control for Providing QoS in DiffServ IP Networks: The TEQUILA Approach*, IEEE Communications Magazine, January 2003, p38-44.
[Pao04]	Paolo Valente, *Exact GPS Simulation with Logarithmic Complexity, and its Application to an Optimally Fair Scheduler*, ACM SIGCOMM'04, Aug. 30ñSept. 3, 2004, Portland, Oregon, USA, p269-280.
[Par92]	Parekh A., *A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks*, PhD. Dissertation, Massachusetts Institute of Technology, (MIT Laboratory for Information and decision systems, Report LIDS-TH-2089), February 1992.
[ParGal94]	Parekh A. K.. and Gallager R.G., *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, p137-150, April 1994.
[Peu99]	Peuhkuri Markus, *IP Quality of Service*, Helsinki University of Technology, Laboratory of Telecommunications Technology, May 1999.
[Rod et al.03]	Rudolf Roth et al., *IP QoS Across Multiple Management Domains: Practical Experiences from Pan-European Experiments*, IEEE Communications Magazine, Jan. 2003, p62-69.

[SchWin04]	Schollmeier Gero and Winkler Christian, *Providing Sustainable QoS in Next-Generation Networks,* IEEE Communications Magazine, June 2004, p102-107.
[Sei03]	Seitz Neal, *ITU-T QoS Standards for IP-Based Networks,* IEEE Communications Magazine, June 2003, p82-89.
[Sha et al.02]	Shaikh Fayaz A. et al., *End-to-End Testing of IP QoS Mechanisms,* IEEE Computers May 2002, p80-87.
[ShrVar95]	Shreedhar M. and Varghese G., *Efficient Fair Queuing Using Deficit Round Robin.* Computer Communication Review Vol. 25, No. 4, October 1995, p 231.
[SikTei]	Sikora J. and Teitelbaum B., *Differentiated Services for Internet,* Internet Draft, Internet2 QoS working group.
[Sol et al.04]	Soldatos John et al., *Enforcing Effective Rates for Packet-Level QoS Control in IP Networks: Theory and Validation Based on Real Traffic Data,* Kluwer Academic Publishers, Telecommunication Systems 27:1, 9–31, 2004.
[Ste00]	Stevens W. Richard, *TCP/IP Illustrated Volume 1: The Protocols,* Addison Wesley Longman, Inc. April 2000.
[Ste98]	Stephenson Ashley, *Diffserv and MPLS: A quality choice*, Data Communications. New York: Nov 21, 1998. Vol. 27, Iss. 17; pg.73, 5pgs.
[Stoi00]	Stoica I., *Stateless Core: A Scalable Approach for Quality of Service in the Internet,* PhD thesis Carnegie Melton Univ. Pittsburgh PA Dec. 2000.
[Tak et al.04]	Takahashi Akira, Yoshino Hideaki and Kitawaki Nobuhiko, *Perceptual QoS Assessment Technologies for VoIP*, IEEE Communications Magazine, July 2004, p28-34.

[Tan96] Tanenbaum A. S., *Computer Networks,* Prentice Hall, Upper Saddle River, N.J. London, 1996.

[Tur86] Turner J., *New Directions in Communications (or Which Way to the Information Age?),* Proc. Zurich seminar on Digital Communication, March 1986, pp. 25-32.

[Veg03] Vegesna Srinivas, *IP Quality of Service: The complete resource for understanding and deploying IP quality of service for Cisco networks,* Cisco Press, December 2003. ISBN 1-57870-116-3.

[Vei00] Veil Mark, *The solution for IP service quality,* Telephony, Chicago: Jan 24, 2000. Vol. 238, Iss. 4; pg. 54, 4pgs.

[Wan01] Wang Zheng, *Internet QoS: Architectures and Mechanisms for Quality of Service,* Morgan Kaufmann Publishers 2001. ISBN 1-55860-608-4.

[Wel et al.03] Welzl Michael, Franzens Leopold and Mühlhäuser Max, *Scalability and Quality of Service: A Trade-off?* IEEE Communications Magazine, June 2003, p32-36.

[Woo93] Woodward Michael E., *Communication and Computer Networks: Modelling with discrete-time queues,* Pentech Press Limited, London 1993. ISBN 0-7273-0410-0.

[Woo96] Woodward M. E *Lecture Notes on Communication Networks: Policing Mechanisms.* Digital Communication Systems, Dept. of Electronic & Electrical Engineering, Loughborough University, December 1996.

[Yan et al.04] Yang Xu, Westhead Martin, Baker Fred, *An Investigation of Multilevel Service Provision for Voice over IP Under Catastrophic Congestion,* IEEE Communications Magazine, June 2004, p94-100.

[Yen et al.01] Yener Bulent, Su Gong and Gabber Eran, *Smart box Architecture: a hybrid solution for IP QoS provisioning*, Computer Networks 36 (2001) 357-357; Elservier Science.

[Zha et al.03] Zhang, Jin-Yu et al., *Quantitative QoS management implement mechanism in IP-DiffServ*, Journal of Computer Science and Technology, v 20, n 6, November, 2005, p831- 835

Bibliography

Ammeraal L. (2000), *C++ For Programmers*. John Wiley Chichester.

Araniti Giuseppe, Iera Antonio and Modafferi Antonio, *QoS guarantees in heterogeneous systems consisting of IP core networks with satellite access* Mobile Networks and Applications, v 9, n 3, June, 2004, p 175-184.

Armitage, Grenville J. (2003), *Revisiting IP QoS: Why do we care, what have we learned?* ACM SIGCOMM 2003 RIPQOS workshop report, Computer Communication Review, v 33, n 5, October, 2003, p 81-88.

Azar Yossi, and Richter Yossi, *Management of multi-queue switches in QoS networks* Algorithmica (New York), v 43, n 1-2, July, 2005, p 81-96.

Basturk E., Birman A., Delp G., Guerin R., Haas R., Kamat S., Kandlur D., Pan P., Pendarakis D., Peris V., Rajan R., Saha D., Williams D., *Design and implementation of a QoS capable switch-router* Computer Networks, v 31, n 1-2, 14 Jan, 1998, p 19-32.

Berg H. van den et al. *QoS-aware bandwidth provisioning for IP network links.* Computer Networks 50 (2006) 631–647.

Bernet Y., *The Complementary Roles of RSVP and Differentiated Services in the Full-Service QoS Network*, IEEE Communication Magazine, February 2000.

Bernet Y., Yavatkar R., Ford P., Baker F., Zhang L., Nichols K., Speers M., *A Framework for Use of RSVP with DiffServ Network*, Internet Draft, draft-ietf-diffserv-rsvp-01.txt, November 1998.

Bianchi G., Borgonovo F., Capone A., Fratta L., Petrioli C., *Endpoint Admission Control with delay variation measurements for QoS in IP networks* Computer Communication Review, v 32, n 2, April, 2002, p 61-69.

Blakey K., Gregson S., Mulvey M., *Unified IP networks* BT Technology Journal, v 18, n 2, Apr, 2000, p 44-56.

Braden R., and Hoffman D., *An RSVP Application Programming Interface*, Internet Draft, draft-ietf-rsvp-rapi-01.ps, February1998.

Briscoe, Bob; Rudkin, Steve *Commercial models for IP quality of service interconnect* BT Technology Journal, v 23, n 2, April, 2005, p 171-195.

Burgstahler L., Dolzer K., Hauser C., Jahnert J., Junghans S., Macian C., Payer, W. S, *Beyond Technology: The Missing Pieces for QoS Success* Proceedings of the ACM SIGCOMM Workshops, Proceedings of the ACM SIGCOMM Workshops, 2003, p 121-130.

Clark D. and Wroclawski J., *An Approach to Service Allocation in the Internet*, <draft-clark-diff-svc-alloc-00.txt>, August, 1997.

Cook Nigel P., *Introductory Digital Electronics*, Prentice-Hall Inc., New Jersey 1998.

Dowdy Shirley and Wearden Stanley Statistics, *Statistics for Research*, John Wiley & Sons Inc., New York 1991.

Everitt B. S., *Introduction to Optimization Methods and their Application in Statistics*, Chapman and Hall, London, New York, 1987.

Gamage Manodha, Hayasaka Mitsuo and Miki Tetsuya, *A connection-oriented network architecture with guaranteed QoS for future real-time applications over the Internet* Computer Networks, v 50, n 8 SPEC. ISS., Jun 6, 2006, p 1130-1144.

Hanly J. R. and Koffman E. B. (1999) *Problem Solving and Program Design in C.*, Addison-Wesley Longman Inc., USA, 1999.

Hanly, Jeri R. (1997) *Essential C++ For Engineers and Scientists,* Addison Wesley Pub. Co., Harlow.

Harrison M. and Michael M (1998) *Effective Tcl/Tk Programming : Writing Better Programs with Tcl and Tk,* Addison-Wesley Harlow.

Herzog S., *Preemption Priority Policy Element,* Internet Draft, draft-ietf-rap signaled-priority-00.txt, November 1998.

Hunt G and Arden P, *QoS requirements for a voice-over-IP PSTN.* BT Technology Journal Vol. 23 No. 2 April 2005.

IEEE Standards for Local and Metropolitan Area Networks: For Virtual Bridged Local Area Networks, (IEEE 802.1Q), IEEE Draft Standard P802.1Q/D11, July 1998.

Johnson Darren M., QoS *control versus generous dimensioning,* BT Technology Journal, v 23, n 2, April, 2005, p 81-96.

Lee S. S., Das S., Yu H., Yamada K., Pau G., Gerla M, *Practical QoS network system with fault tolerance* Computer Communications, v 26, n 15, Sep 22, 2003, p 1764-1774.

Maniatis Sotiris I., Nikolouzou Eugenia G., Venieris Iakovos S., *End-to-end QoS specification issues in the converged all-IP wired and wireless environment* IEEE Communications Magazine, v 42, n 6, June, 2004, p 80-86.

Margaliot Michael and Langholz Gideon (2000) *New Approaches to Fuzzy Modeling and Control: Design and Analysis.* World Scientific Publisher, Singapore, 2000.

Michael C. Fu, *A Tutorial Review of Techniques for Simulation* Optimisation. Proceedings of the 1994 Winter Simulation Conference.

Moore Sean S. B., Siller Jr., Curtis A., *Availability of end-to-end ideal QoS in IP packet networks* Computer Communications, v 28, n 18, Nov 1, 2005, Current Areas of Interest in End-to-End QoS, p 2047-2057.

Nyhoff Larry R., *C++ An Introduction to Data Structures*, Prentice-Hall Inc. New Jersey 1999.

Oualline, S. (1995) *Practical C++ Programming.* O'Reilly & Associates Inc., USA, 1995.

Ousterhout, J.K. (1994) *Tcl and the Tk Toolkit,* Addison-Wesley Wokingham.

RFC 1812. Baker F., *Requirements for IP version 4 Routers,* June 1995.

RFC 2460. Deering S. and Hinden R., *Internet Protocol, version 6 (IPv6) Specification,* December 1998.

Robert Mandeville, David Newman, *Traffic tuners: Striking the right note?* Data Communications. New York: Nov 21, 1998. Vol. 27, Iss. 17; pg. 51, 1pgs

Saaty Thomas L., *Element of Queueing Theory with Applications,* McGraw-Hill Book Company Inc. New York 1961.

Sven Ubik et al., *Low-Cost Precise QoS Measurement Tool.* CESNET technical report number 7/2001.

Tanenbaum A. S. (2001), *Modern Operating Systems.* Upper Saddle River, N.J: Prentice Hall.

Tassiulas Leandros, Chung Hung Yao, and. Panwar Shivendra S., *Optimal Buffer Control During Congestion in an ATM Network Node,* IEEE/ACM Transactions on Networking, Vol. 2, No. 4, August 1994.

Trecordi Vittorio, Verticale Giacomo, *QoS support for per-flow services: POS vs. IP-over-ATM* IEEE Internet Computing, v 4, n 4, Jul, 2000, p 58-64.

Tsern-Huei Lee and Kuen-Chu Lai, *Characterization of Delay-Sensitive Traffic,* IEEE/ACM Transactions on Networking, Vol. 6, No. 4, August 1998 p499.

Vince Vittore, *IP encryption, QoS provide difference among carriers.* Telephony Chicago: Oct 12, 1998. Vol. 235, Iss. 15; pg. 62 1pgs.

Walrand Jean, *Introduction to Queueing Networks*, Prentice-Hall International, Inc. 1988.

Wille E.C.G. et al., *Algorithms for IP network design with end-to-end QoS constraints.* Computer Networks 50 (2006) 1086–1103.

William C. Thompson, *The Application of Simulation in Computer System Design and Optimisation*, W.C. Thompson—the Vice President, Software Products Corporation, Falls Church, Virginia.

Woodward, Michael E et al. ed. (1994) *Computer and Telecommunication Systems Performance Engineering*: 9th UK Performance Engineering Workshop for Computer and Telecommunication Systems held at Loughborough University of Technology July 1993. London: Pentech

Yavatkar R., Pendarakis D., Guerin R., *A Framework for Policy-based Admission Control,* Internet Draft, draft-ieft-rap-framework-01.txt, November 1998.

Zhang Runtong, Phillis Yannis A. and Kouikoglou Vassilis S., *Fuzzy Control of Queuing Systems*, Springer-Verlag, USA 2005.

APPENDIX A

Brief Overview of Network Simulator (NS)

A.0 Introduction

Network Simulator (NS) is the nickname for VINT (Virtual InterNetwork Testbed) project which is a DARPA-funded research project whose aim is to build a network simulator that will allow the study of communication networks and protocol interaction in the context of current and future network protocols. The VINT is a collaborative project involving USC/ISI, Xerox PARC, LBNL, and UC Berkeley **[NS Doc00]**.

A.1 Overview of NS

The NS is an event-driven network simulator. It is an object oriented simulator written in C++ with an Object Tool Command Language (OTcl) interpreter as a frontend. The Simulator is an extensible simulation engine implemented in C++ that uses MIT's OTcl (an object oriented version of Tool Command Language (Tcl)) as the command and configuration interface. NS version 1 is a previous version of the simulator that used the Tcl as the configuration language. The current version still supports simulation scripts written in Tcl meant for the NS version 1 simulator.

A.1.1 Brief Code Overview

The Simulator supports object class hierarchy in C++ (referred to as compiled hierarchy in the documentation), and a similar object class hierarchy within the OTcl interpreter (referred to as interpreted hierarchy in the documentation). The two hierarchies are closely related to each other, from user's perspective, there is a one-to-one correspondence between a class in the interpreted hierarchy and one in compiled hierarchy. As shown in the Figure A1, the root of the hierarchy is the class TclObject. Users create new simulator objects through the interpreter, these objects are instantiated within the interpreter and are closely imaged and linked to the corresponding object in the compiled hierarchy. The interpreted class hierarchy is automatically established through methods defined in the class TclClass (see

Figure A1). User instantiated objects in the interpreted hierarchy are mirrored through methods defined in the class TclObject. The code for the Simulator is quite extensive, there are other hierarchies in the C++ code and Otcl scripts, which do not share the same mirrored alliance in the manner of TclClass with TclObject **[NSMan00].**

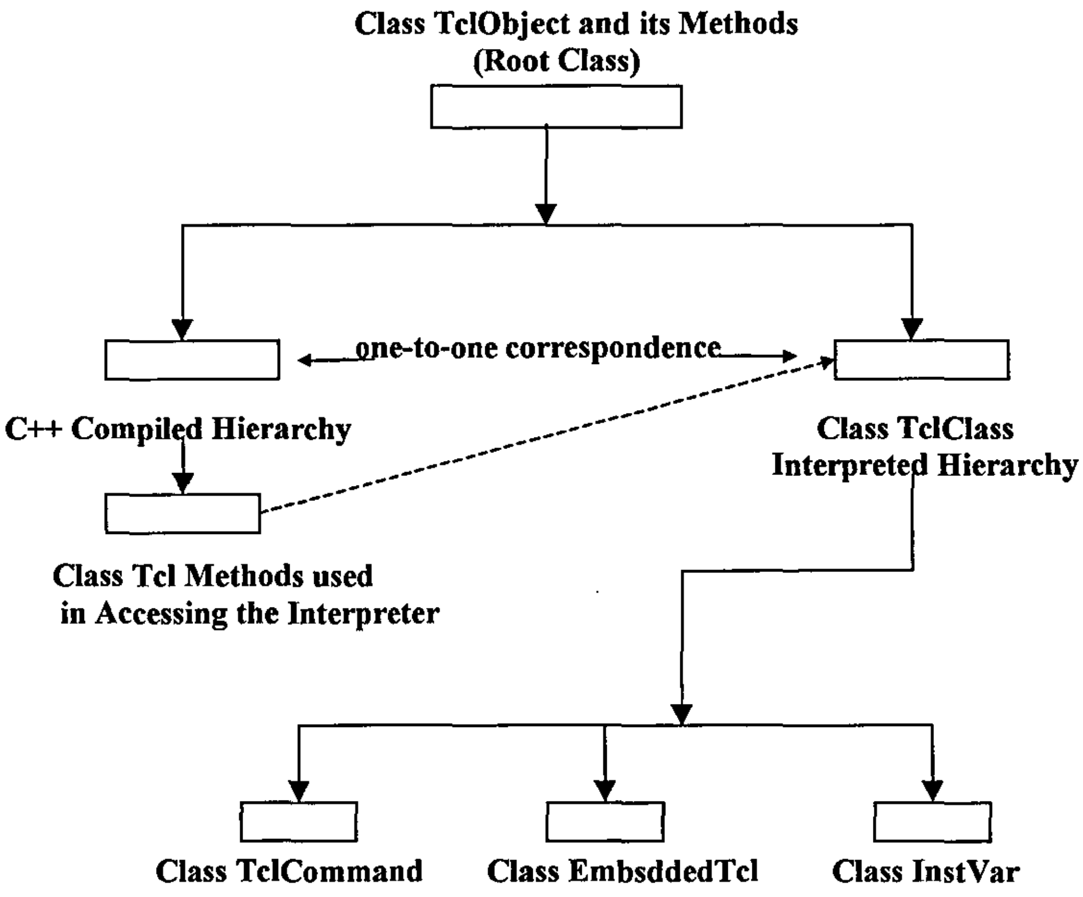


Figure A1: Object Class hierarchy in NS

The NS code is divided into two main directories—the code to interface with the interpreter resides in one directory–the *tclcl directory* and the rest of the simulator code resides in the other directory–the *ns-2 directory.* There are a number of classes defined in *~tclcl* directory, only six of these are concerned with the Simulator. The Class *Tcl* contains the methods that C++ code will use to access the interpreter. As mentioned above, the class *TclObject* is the base class for all simulator objects that are also mirrored in the compiled hierarchy. The class *TclClass* defines the interpreted class hierarchy and the methods to permit the user to instantiate TclObjects. The class *TclCommand* is used to define simple global interpreter commands. The class *EmbeddedTcl* contains the methods to load higher level built-in commands that make simulation configuration easier. Finally, the class *InstVar* contains methods to access C++ member variables as OTcl instance variables **[NSDoc00]**. These are illustrated in Figure A1. Chapter 3 of NS Documentation contains more details on NS code overview.

A.1.2 The Use of Two Languages

The NS adopt two languages because the Simulator has two different kinds of simulation processes to address. On one hand, detailed simulation of protocols requires a system language, which can efficiently manipulate network dynamic objects such as packets, timers, etc, and implement algorithms that run over large data sets. For these tasks run-time speed is important but operation and maintenance time is less important.

On the other hand, a large part of network research involves slightly varying parameters or configurations, which involve quickly exploring a number of scenarios. In these cases iteration time (change the model and re-run) is more important. In view of the fact that configuration run once (at the beginning of the simulation), run time of this part of the task is less important **[NSDoc00].**

The NS meets both of these needs with two languages, C++ and Otcl. C++ is fast to run but slower to change, making it suitable for detailed protocol implementation. OTcl runs much slower but can be changed very quickly (and interactively), making it ideal for simulation configuration. NS (via tclcl) provides glue to make objects and variables appear on both languages **[NSDoc00].** See the documentation for more information on this.

A.2 Configuring and Running a Simulation

A simulation is defined by an OTcl script using a text editor such *VI* or *Emas*. The scripts use the *Simulator Class* as the principal interface to the simulation engine. Using the methods defined in the Simulator Class, a network topology is defined, traffic sources and sinks are configured, the simulation is invoked, and the statistics are collected. The simulator is invoked via the *NS* interpreter, which is an extension of the vanilla *otclsh* command shell. By building upon a fully functional language, offered by OTcl, arbitrary actions can be programmed into the simulation configuration **[NSMan00]**.

The first step in the simulation is to acquire an instance of the Simulator Class. Instances of objects in classes are created and destroyed in NS using the *new* and *delete* methods. For example, an instance of the Simulator object is created by the following command:

set ns [new Simulator]

A network topology is realised using three primitive building objects: nodes, links, and agents. The Simulator Class has methods to create or configure each of these building blocks. The nodes are created with the *node* Simulator method that automatically assigns a unique address to each node. The links are created between nodes to form a network topology with the *simplex-link* or *duplex-link* methods that set up unidirectional and bi-directional links respectively. The agents are the objects that actively drive the simulation. The *agents* can be thought of as the processes and/or transport entities that run on *nodes* that may be end hosts or routers. Traffic sources and sinks, dynamic routing modules and the various protocol modules are all examples of agents. Agents are created by instantiating objects in the subclass of class Agent i.e., *Agent/type* where type specifies the nature of the agent. For example, a TCP agent is created using the command:

set tcp [new Agent/TCP]

Once the agents are created, they are attached to nodes with the *attach-agent* Simulator method. Each agent is automatically assigned a port number unique across all agents on a given node (analogous to a tcp or udp port). Some types of agents may have sources attached to them while others may generate their own data. For example, you can attach ``ftp" and ``telnet" sources to ``tcp" agents but ``constant bit-rate" agents generate their own data. Sources are attached to agents using the *attach-source* and *attachtraffic* agent methods **[NSMan00]**.

Each object has some configuration parameters associated with it that can be modified. Configuration parameters are instance variables of the object. These parameters are initialised during startup to default values that can simply be read from the instance variables of the object. For example, *$tcp set window_* returns the default window size for the tcp object. The default values for that object can be explicitly overridden by simple assignment either before a simulation begins, or dynamically, while the simulation is in progress. For example the window-size for a particular TCP session can be changed in the following manner:

```
$tcp set window_ 25
```

The default values for the configuration parameters of all the class objects subsequently created can also be changed by simple assignment. For example, we can say,

```
Agent/TCP set window_ 30
```

to make all future tcp agent creations default to a window size of 30 **[NSMan00].**

Events are scheduled in NS using the *at* Simulator method that allows OTcl procedures to be invoked at arbitrary points in the simulation time. These OTcl callbacks provide a flexible simulation mechanism— they can be used to start or stop sources, dump statistics, instantiate link failures, reconfigure the network topology etc. The simulation is started via the *run* method and continues until there are no more events to be processed. At this time, the original invocation of the *run* command returns and the Tcl script can exit or invoke another simulation run after possible reconfiguration. Alternatively, the simulation can be prematurely halted by invoking the *stop* command or by exiting the script with Tcl's standard *exit* command **[NSMan00].**

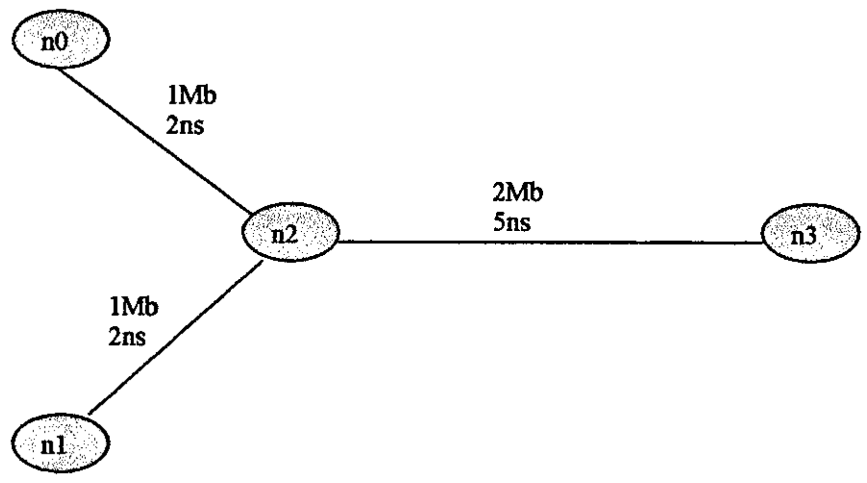
Packets are forwarded along the shortest path route from a source to a destination, where the distance metric is the sum of costs of the links traversed from the source to the destination. The cost of a link is 1 by default; the distance metric is simply the hop count in this case. The cost of a link can be changed with the *cost* Simulator method. A static topology model is used as the default in NS in which the states of nodes/links do not change during the course of a simulation. Network Dynamics could be specified using methods described in NS documentation under *Network Dynamic Methods* section. Also static unicast routing is the default in which the routes are pre-computed over the entire topology once prior to starting the simulation. Methods to enable and configure dynamic unicast and multicast routing are described also in NS Documentation under the *Unicast Routing Methods* and *Multicast Routing Methods* sections respectively **[NSDoc00].**

A.3 NS Basic Script for Simple Network Simulation

Resource like Marc Greis's tutorial on web pages (at http://titan.cs.uni-bonn.de/-greis/ns/ns.html) is best place to learn NS scriptive programming. The information here serves only as introductory knowledge.

The first step in running a simulation as stated before is to acquire an instance of the Simulator class that has methods to configure and run the simulation. This is achieved with the *new* method of the Simulation Class as follows:

```
set  ns  [new Simulator]

# Next you create file to write simulation traces, i.e. to write events of the
#  simulation such as packet sent time, receive time, packet loss, etc.

set trace-file [open out .tr w]
$ns trace-all $trace-file

# Also open file to store animation of simulation objects

set nam-trace [open out.nam w]
$ns namtrace-all $nam-trace

# Create four nodes

set n0 [$ns node]
set nl [$ns node]
set n2 [$ns node]
set n3 [$ns node]
```



```
#
#
#
#
#
#
#
#
#
#
#
#
#
# Figure A2: Topology of the Simulated Network
# Connect the nodes with bi-directional links with link parameters as shown in the
#  Figure A2. Note that, Mb is the Bandwidth in Mb/s, ns is propagation delay in
#  Nanosecond and DropTail is the FIFO queue discipline.

$ns duplex-link $n0 $n2 1Mb 2ns DropTail
$ns duplex-link $n1 $n2 1Mb 2ns DropTail
$ns duplex-link $n2 $n3 2Mb 5ns DropTail

# Create agents and attach agents to nodes and also create traffic sources and attach
# traffic sources to the agents. Also specify traffic sources parameters.

set udp0 [new Agent/UDP]
$ns attach-agent $n0 $udp0
set cbr0 [new Application/Traffic/CBR]
$cbr0 attach-agent $udp0
$cbr0 set packetSize_ 48
$cbr0 set interval_ 0.005
$udp0 set class_ 0

set null0 [new Agent/Null]
$ns attach-agent $n3 $null0

# Connect source agent to sink agent

$ns connect $udp0 $null0
$ns at 1.0 "$cbr0 start"

set tcp [new Agent/TCP]
$tcp set class_ 1
$ns attach-agent $n1 $tcp

set sink [new Agent/TCPSink]
$ns attach-agent $n3 $sink

set ftp [new Application/FTP]
$ftp attach-agent $tcp
$ns connect $tcp $sink
$ns at 1.2 "$ftp start"

# Define a finish procedure to close files in the simulation and execute the network
# animation file

proc finish { } {
        global ns trace-file nam-trace
        $ns flush-trace
        close $trace-file
        close $nam-trace
        exec nam out.nam &
        exit 0
}
```

```
# Execute the proc finish (finish procedure) at 10 minutes of simulation time to close
# the simulation.

$ns at 10.0 ‘finish’

# Run the simulation

$ns run
```

The code above simulates a network of four nodes. Node n0 and n1 are the sources of traffic (See Figure A2). They send their traffic through node n2 to node n3. Node n0 start to send traffic at 1.0 second of simulation time and continue to send until the end of the simulation, while node n1 start to send its own traffic at 1.2 second of the simulation time and also continue to send until the end of the simulation. The simulation runs for 10.0 seconds.

The NS has extensive commands to create and configure different types of nodes, links and agents that are the building blocks of a network topology in the Simulator. OTcl being a programmable language used by the interpreter (*otclsh*) provides platform for extensible features that can be programmed into the simulation to meet most investigation requirements of network performance evaluations. Where the existing capabilities in NS do not meet a particular requirement of network simulation, such as simulating a new protocol or algorithm, then there will be need to define a new class and its methods with C++ to create the base compiled class and the necessary commands. The corresponding interpreted class and its procedure will have to be defined with OTcl as well. Marc Greis tutorial part IV is a good place to learn this approach.

The NS object hierarchy and methods are fully documented in NS documentation. Readers should please consult the documentation for necessary further information.

APPENDIX B

Sample Code Used in NS

Two sample codes are presented in this appendix, and they represent example of key programs written within NS environment on the work of Optimum Number of Traffic Queuing Classes (ONTQC) to support Multservice QoS in IP Networks.

The codes are written in Tool Command Language (TCL) and its Object oriented extension OTCL, which are scriptive languages used in NS. The codes are used to model the implementation of the topology shown in Figure B1. The topology is technically referred to a *Bipartite Digraph* (see Section 9.1.1), as illustrated in the figure, it is a two tree-like star topology interconnected by a single link. The code model traffic generation, processing and transmission from the source hosts of the network, through the bottleneck link to the traffic destination hosts.

The code presented in Section B.1 implement FIFO queue discipline in the bottleneck link and the one presented in Section B.2 implement Multiple queue discipline in the same bottleneck link. In view of limitation of space, other voluminous and complex code used in the work could not be chosen for presentation here.

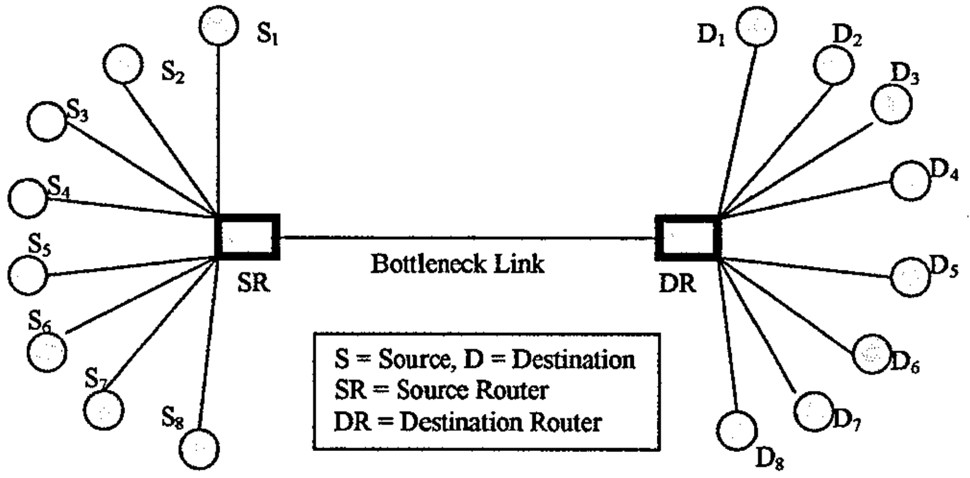


Figure B1: Topology Implemented by Code in Section 1 and 2

B.1 Code for Base Scenario (FIFO Queue at Bottleneck Link)

```
# NS Basic Code for Single Queue (FIFO)
# Simulation with eight classes of traffic employing single queue in
# congested link between node(8-SR) and node(9-DR)

# Create a Simulator Object

set ns [new Simulator]

# Set colour to differentiate between processes in the simulation
# objects.

$ns color 1 Blue
$ns color 2 Red
$ns color 3 Green
$ns color 4 Yellow
$ns color 5 Orange
$ns color 6 Grey
$ns color 7 Violet
$ns color 8 Black
set qlim 1000

set nf [open outp2T8.nam w]
$ns namtrace-all $nf

set tf [open tracep2T8.tr w]
$ns trace-all $tf

# Define a 'finish' procedure
proc finish {} {
     global ns nf tf
     $ns flush-trace
     #Close the trace file
     close $nf
     close $tf
     #Execute nam on the trace file
     exec nam outp2T8.nam &
     exec tr tracep2T8.tr &
     exit 0
}
# Create eighteen nodes, one of them as router

for {set i 0} {$i < 18} {incr i} {
      set n($i) [$ns node]
}

# Create duplex links between the nodes and the nodes implementing
# FIFO queue discipline

$ns duplex-link $n(0) $n(8) 1Mb 5ms DropTail
$ns duplex-link $n(1) $n(8) 1Mb 5ms DropTail
$ns duplex-link $n(2) $n(8) 1Mb 5ms DropTail
$ns duplex-link $n(3) $n(8) 1Mb 5ms DropTail
$ns duplex-link $n(4) $n(8) 1Mb 5ms DropTail
$ns duplex-link $n(5) $n(8) 1Mb 5ms DropTail
$ns duplex-link $n(6) $n(8) 1Mb 5ms DropTail
$ns duplex-link $n(7) $n(8) 1Mb 5ms DropTail
$ns duplex-link $n(8) $n(9) 1.5Mb 10ms DropTail
$ns duplex-link $n(9) $n(10) 1Mb 5ms DropTail
$ns duplex-link $n(9) $n(11) 1Mb 5ms DropTail
```

```
$ns duplex-link $n(9) $n(12) 1Mb 5ms DropTail
$ns duplex-link $n(9) $n(13) 1Mb 5ms DropTail
$ns duplex-link $n(9) $n(14) 1Mb 5ms DropTail
$ns duplex-link $n(9) $n(15) 1Mb 5ms DropTail
$ns duplex-link $n(9) $n(16) 1Mb 5ms DropTail
$ns duplex-link $n(9) $n(17) 1Mb 5ms DropTail

# Arrange the topology such that it produces two stars
# interconnected by a single link as shown in Figure B1.

$ns duplex-link-op $n(8) $n(0) orient left-up
$ns duplex-link-op $n(8) $n(1) orient left
$ns duplex-link-op $n(8) $n(2) orient left-down
$ns duplex-link-op $n(8) $n(3) orient down
$ns duplex-link-op $n(8) $n(4) orient right-up
$ns duplex-link-op $n(8) $n(5) orient up
$ns duplex-link-op $n(8) $n(6) orient right-down
$ns duplex-link-op $n(8) $n(7) orient 20
$ns duplex-link-op $n(8) $n(9) orient right
$ns duplex-link-op $n(9) $n(10) orient up
$ns duplex-link-op $n(9) $n(11) orient right-up
$ns duplex-link-op $n(9) $n(12) orient right
$ns duplex-link-op $n(9) $n(13) orient right-down
$ns duplex-link-op $n(9) $n(14) orient down
$ns duplex-link-op $n(9) $n(15) orient left-down
$ns duplex-link-op $n(9) $n(16) orient left-up
$ns duplex-link-op $n(9) $n(17) orient left

$ns queue-limit $n(8) $n(9) $qlim
$ns duplex-link-op $n(8) $n(9) queuePos 0.5

# create Agents and attach them to Nodes

set udp0 [new Agent/UDP]
$ns attach-agent $n(0) $udp0
$udp0 set packetSize_ 210
$udp0 set fid_ 1
set cbr0 [new Application/Traffic/CBR]
$cbr0 set interval_ 0.02
$cbr0 set random_ 1
$cbr0 attach-agent $udp0

set null0 [new Agent/Null]
$ns attach-agent $n(10) $null0
$ns connect $udp0 $null0

set udp1 [new Agent/UDP]
$ns attach-agent $n(1) $udp1
$udp1 set packetSize_ 48
$udp1 set fid_ 2
set exp0 [new Application/Traffic/Exponential]
$exp0 set burst_time_ 1s
$exp0 set idle_time_ 100ms
$exp0 set rate_ 64k
$exp0 attach-agent $udp1

set null1 [new Agent/Null]
$ns attach-agent $n(11) $null1
$ns connect $udp1 $null1
```

```
set udp2 [new Agent/UDP]
$ns attach-agent $n(2) $udp2
$udp2 set packetSize_ 156
$udp2 set fid_ 3
set exp1 [new Application/Traffic/Exponential]
$exp1 set burst_time_ 3s
$exp1 set idle_time_ 100ms
$exp1 set rate_ 128k
$exp1 attach-agent $udp2

set null2 [new Agent/Null]
$ns attach-agent $n(12) $null2
$ns connect $udp2 $null2

set udp3 [new Agent/UDP]
$ns attach-agent $n(3) $udp3
$udp3 set packetSize_ 200
$udp3 set fid_ 4
set paret [new Application/Traffic/Pareto]
$paret set burst_time_ 2s
$paret set idle_time_ 100ms
$paret set rate_ 200k
$paret set shape_ 1.5
$paret attach-agent $udp3

set null3 [new Agent/Null]
$ns attach-agent $n(13) $null3
$ns connect $udp3 $null3

set tcp0 [new Agent/TCP]
$ns attach-agent $n(4) $tcp0
$tcp0 set packetSize_ 625
$tcp0 set fid_ 5
set telnet0 [new Application/Telnet]
$telnet0 set interval_ 0.005
$telnet0 attach-agent $tcp0

set sink0 [new Agent/TCPSink]
$ns attach-agent $n(14) $sink0
$ns connect $tcp0 $sink0

set tcp1 [new Agent/TCP]
$ns attach-agent $n(5) $tcp1
$tcp1 set packetSize_ 800
$tcp1 set fid_ 6
set telnet1 [new Application/Telnet]
$telnet1 set interval_ 0.005
$telnet1 attach-agent $tcp1

set sink1 [new Agent/TCPSink]
$ns attach-agent $n(15) $sink1
$ns connect $tcp1 $sink1

set tcp2 [new Agent/TCP]
$ns attach-agent $n(6) $tcp2
$tcp2 set packetSize_ 700
$tcp2 set fid_ 7
set ftp0 [new Application/FTP]
$ftp0 attach-agent $tcp2
set sink2 [new Agent/TCPSink]
$ns attach-agent $n(16) $sink2
```

```
$ns connect $tcp2 $sink2

set tcp3 [new Agent/TCP]
$ns attach-agent $n(7) $tcp3
$tcp3 set packetSize_ 1000
$tcp3 set fid_ 8
set ftp1 [new Application/FTP]
$ftp1 attach-agent $tcp3
set sink3 [new Agent/TCPSink]
$ns attach-agent $n(17) $sink3
$ns connect $tcp3 $sink3

# set the start time and stop time
$ns at 0.0 "$cbr0 start"
$ns at 0.0 "$exp0 start"
$ns at 0.0 "$exp1 start"
$ns at 0.0 "$paret start"
$ns at 0.0 "$telnet0 start"
$ns at 0.0 "$telnet1 start"
$ns at 0.0 "$ftp0 start"
$ns at 0.0 "$ftp1 start"
$ns at 55.0 "$cbr0 stop"
$ns at 55.0 "$exp0 stop"
$ns at 55.0 "$exp1 stop"
$ns at 55.0 "$paret stop"
$ns at 55.0 "$telnet0 stop"
$ns at 55.0 "$telnet1 stop"
$ns at 55.0 "$ftp0 stop"
$ns at 55.0 "$ftp1 stop"


$ns at 56.0 "finish"
$ns run
```

B.2 Code for Multiple Queue Scenario (CBQ at Bottleneck Link)

```
# NS Multiple Queue implementation in Investigation of ONTQC
# Simulation with multiple queue in the congested link
# (Eight Queue implementation)

set ns [new Simulator]

# Set colour to differentiate between processes in the simulation
# objects.

$ns color 1 Blue
$ns color 2 Red
$ns color 3 Green
$ns color 4 Yellow
$ns color 5 Orange
$ns color 6 Violet
$ns color 7 Grey
$ns color 8 Black

set rng [new RNG]
$rng seed 0

set f1 [open traceRandCl.tr w]
$ns trace-all $f1
```

```
set nf [open outRandCl.nam w]
$ns namtrace-all $nf

# Define a 'finish' procedure

proc finish {} {
     global ns f1 nf
     $ns flush-trace
     close $f1
     close $nf
     exec tr traceRandCl.tr &
     exec nam outRandCl.nam &
     exit 0
}

# Create nineteen nodes

for {set i 0} {$i < 18} {incr i} {
        set n($i) [$ns node]
}

# Connect the nodes to have topology orientation as shown in Figure
# B1

for {set i 0} {$i < 8} {incr i} {
        $ns duplex-link $n($i) $n(8) 1Mb 5ms DropTail
        $ns duplex-link-op $n(8) $n($i) orient left
}

$ns simplex-link $n(8) $n(9) 1.5Mb 10ms CBQ/WRR
$ns simplex-link $n(9) $n(8) 1.5Mb 10ms DropTail
$ns duplex-link-op $n(8) $n(9) orient right

for {set i 10} {$i < 18} {incr i} {
        $ns duplex-link $n(9) $n($i) 1Mb 10ms DropTail
        $ns duplex-link-op $n(9) $n($i) orient right
}
# Create CBQClasses and insert them into the CBQLink

set cbqlink [$ns link $n(8) $n(9)]
set topclass [new CBQClass]
$topclass setparams none 0 1.0 auto 8 2 0

set voiceClass [new CBQClass]
set voiceQueue [new Queue/DropTail]
$voiceQueue set limit_ 50
$voiceClass install-queue $voiceQueue
$voiceClass setparams $topclass true 0.05 auto 0 1 0

set audioClass [new CBQClass]
set audioQueue [new Queue/DropTail]
$audioQueue set limit_ 100
$AudioClass install-queue $audioQueue
$audioClass setparams $topclass true 0.1 auto 1 1 0

set videoClass0 [new CBQClass]
set videoQueue0 [new Queue/DropTail]
$videoQueue0  set limit_ 250
$videoClass0 install-queue $videoQueue0
$videoClass0 setparams $topclass true 0.12 auto 2 1 0
```

```
set videoClass1 [new CBQClass]
set videoQueue1 [new Queue/DropTail]
$videoQueue1  set limit_ 250
$videoClass1 install-queue $videoQueue1
$videoClass1 setparams $topclass true 0.13 auto 3 1 0

set dataClass0 [new CBQClass]
set dataQueue0 [new Queue/DropTail]
$dataQueue0 set limit_ 600
$dataClass0 install-queue $dataQueue0
$dataClass0 setparams $topclass true 0.13 auto 4 1 0

set dataClass1 [new CBQClass]
set dataQueue1 [new Queue/DropTail]
$dataQueue1 set limit_ 600
$dataClass1 install-queue $dataQueue1
$dataClass1 setparams $topclass true 0.13 auto 5 1 0

set dataClass2 [new CBQClass]
set dataQueue2 [new Queue/DropTail]
$dataQueue2 set limit_ 600
$dataClass2 install-queue $dataQueue2
$dataClass2 setparams $topclass true 0.17 auto 6 1 0

set dataClass3 [new CBQClass]
set dataQueue3 [new Queue/DropTail]
$dataQueue3 set limit_ 600
$dataClass3 install-queue $dataQueue3
$dataClass3 setparams $topclass true 0.17 auto 7 1 0

$cbqlink insert $topclass
$cbqlink insert $voiceClass
$cbqlink insert $audioClass
$cbqlink insert $videoClass1
$cbqlink insert $videoClass0
$cbqlink insert $dataClass0
$cbqlink insert $dataClass1
$cbqlink insert $dataClass2


$cbqlink bind $voiceClass 1
$cbqlink bind $audioClass 2
$cbqlink bind $videoClass1 3
$cbqlink bind $videoClass0 4
$cbqlink bind $dataClass0 5
$cbqlink bind $dataClass1 6
$cbqlink bind $dataClass2 7
$cbqlink bind $dataClass2 8


set src0 [$ns create-connection UDP $n(0) LossMonitor $n(10) 1]
set expoo0 [new Application/Traffic/Exponential]
set pkts0 [expr 48 + [$rng integer 152]]
$expoo0 set packetSize_ $pkts0
$expoo0 set burst_time_ 1.0s
$expoo0 set idle_time_ 0.5s
set r0 [expr 100 + [$rng integer 100]]kb
$expoo0 set rate_ $r0
$expoo0 attach-agent $src0
```

```
set src1 [$ns create-connection UDP $n(1) LossMonitor $n(11) 2]
set cbr [new Application/Traffic/CBR]
set pkts1 [expr 105 + [$rng integer 105]]
$cbr set packetSize_ $pkts1
set int [$rng uniform 0.001 0.05]
$cbr set interval_ $int
$cbr attach-agent $src1

set src2 [$ns create-connection UDP $n(2) LossMonitor $n(12) 3]
set expoo1 [new Application/Traffic/Exponential]
set pkts2 [expr 48 + [$rng integer 100]]
$expoo1 set packetSize_ pkts2
$expoo1 set burst_time_ 1.5s
$expoo1 set idle_time_ 0.5s
set r1 [expr 150 + [$rng integer 150]]k
$expoo1 set rate_ $r1
$expoo1 attach-agent $src2

set src3 [$ns create-connection UDP $n(3) LossMonitor $n(13) 4]
set pareto [new Application/Traffic/Pareto]
set pkts3 [expr 70 + [$rng integer 140]]
$pareto set packetSize_ $pkts3
$pareto set burst_time_ 0.5s
$pareto set idle_time_ 0.5s
set r2 [expr 200 + [$rng integer 150]]k
$pareto set rate_ $r2
$pareto set shape_ 1.5
$pareto attach-agent $src3

set src4 [$ns create-connection TCP $n(4) TCPSink $n(14) 5]
set telnet0 [$src4 attach-app Telnet]
set pkts4 [expr 500 + [$rng integer 500]]
$src4 set packetSize_ $pkts4
set tint0 [$rng uniform 0.001 0.005]
$telnet0 set interval_ $tint0

set src5 [$ns create-connection TCP $n(5) TCPSink $n(15) 6]
set telnet1 [$src5 attach-app Telnet]
set pkts5 [expr 500 + [$rng integer 300]]
$src5 set packetSize_ $pkts5
set tint1 [$rng uniform 0.001 0.004]
$telnet1 set interval__ $tint1

set src6 [$ns create-connection TCP $n(6) TCPSink $n(16) 7]
set ftp0 [$src6 attach-app FTP]
set pkts6 [expr 1000 + [$rng integer 500]]
$src6 set packetSize_ $pkts6

set src7 [$ns create-connection TCP $n(7) TCPSink $n(17) 8]
set ftp1 [$src7 attach-app FTP]
set pkts7 [expr 750 + [$rng integer 500]]
$src7 set packetSize_ $pkts7

puts "\n"
puts "Exponential(0) packet size = $pkts0 and  rate_ = $r0  \n"
puts ""
puts "CBR packet size = $pkts1 and interval = $int\n"
puts ""
puts "Exponential(1) packet size = $pkts2 and rate = $r1\n"
puts ""
puts "Pareto packet size = $pkts3 and rate = $r2\n"
puts ""
puts "Telnet(0) packet size = $pkts4 and interval = $tint0\n"
puts ""
puts "Telnet(1) packet size = $pkts5 and interval = $tint1\n"
puts ""
puts "Ftp(0) packet size = $pkts6\n"
puts ""
puts "Ftp(1) packet size = $pkts7\n"

# set the start time and stop time
$ns at 0.0 "make_2cbqclass"
$ns at 0.0 "$expoo0 start"
$ns at 0.0 "$cbr start"
$ns at 0.0 "$expoo1 start"
$ns at 0.0 "$pareto start"
$ns at 0.0 "$telnet0 start"
$ns at 0.0 "$telnet1 start"
$ns at 0.0 "$ftp0 start"
$ns at 0.0 "$ftp1 start"
$ns at 55.0 "$expoo0 stop"
$ns at 55.0 "$cbr stop"
$ns at 55.0 "$expoo1 stop"
$ns at 55.0 "$pareto stop"
$ns at 55.0 "$telnet0 stop"
$ns at 55.0 "$telnet1 stop"
$ns at 55.0 "$ftp0 stop"
$ns at 55.0 "$ftp1 stop"

$ns at 55.0 "finish"

$ns run
```

APPENDIX C

Overview of OPNET Software Modelling Package

This appendix provides a brief overview of the capabilities made available by OPNET software modelling simulation package for communication network designers, practitioners, researchers, students, etc. Materials presented here are derived solely from OPNET Documentation. Readers are referred to the documentation for more detailed information on OPNET software modelling paradigm.

The OPNET software package is a powerful modelling-simulation tool, which is invaluable to system modellers and communication system designers in their effort on system performance measures and behavioural analysis of existing or proposed systems. Its power on system modelling can be captured from the few introductory statements of the Modelling Overview chapter of OPNET Documentation **[OpnetDoc03]**. I quote verbatim—"OPNET provides a comprehensive development environment supporting the modelling of communication networks and distributed systems. Both behaviour and performance of modelled systems can be analysed by performing discrete event simulations. The OPNET environment incorporates tools for all phases of a study, including model design, simulation, data collection, and data analysis".

The OPNET Modeller is designed for two groups of people: (1) those who study system behaviours and performances and (2), those who deliver modelling environments to (design models for) "end users".

C.1 Main System Features

The OPNET is a vast software package with an extensive set of features designed to support general network modelling and to provide specific support for particular types of network simulation projects. A brief enumeration of some of the most important capabilities of OPNET is as follows:

- Object orientation—Systems specified in OPNET consist of objects, each with configurable sets of attributes. Objects-classes orientations are adopted for specifications and definitions of object characteristics and behaviours.
- Specialised in communication networks and information systems—OPNET is specifically suitable for many constructs relating to communications and information processing.
- Hierarchical models—Models are hierarchical, naturally paralleling the structure of actual communication networks.
- Graphical specification—In most cases, models are entered via graphical editors. These editors provide an intuitive mapping from the modelled system to the model specification in OPNET environment.
- Flexibility to develop detailed custom models—OPNET provides a flexible, high-level programming language with extensive support for communications and distributed systems. This environment allows realistic modelling of all communications protocols, algorithms, and transmission technologies.
- Automatic generation of simulations—Model specifications are compiled automatically into executable, efficient, discrete-event simulations implemented in the C programming language. Advanced simulation construction and configuration techniques minimise compilation requirements.
- Application-specific statistics—OPNET provides built-in performance statistics that can be collected automatically during simulations. Modellers can also augment this set with new application-specific statistics that are computed by user-defined processes.
- Integrated post-simulation analysis tools—OPNET includes a sophisticated tool for graphical presentation and processing of simulation output.
- Interactive analysis—All OPNET simulations automatically incorporate support for analysis via a sophisticated interactive debugger.
- Animation—Simulation runs can be configured to automatically generate animations of the modelled system at various levels of detail and can include animation of statistics as they change over time. Extensive support for developing customised animations is also provided.

- Cosimulation—You can connect OPNET with one or more other simulators so that you can see how the models in those simulators interact with OPNET models. The external models can represent anything from network hardware to end-user behaviour patterns.
- Application programs interface (API)—As an alternative to graphical specification, OPNET models and data files may be specified via a programmatic interface. This is useful for automatic generation of models or to allow OPNET to be tightly integrated with other tools.

C.2 Basic Operation in OPNET

Modelling task in OPNET consists of three basic steps or phases and these are:

- Specification
- Data Assembly and Simulation
- Result Analysis

These phases are normally performed in sequence. They generally form a cycle, with a return to Specification after Result Analysis. Specification is actually divided into two parts—Initial Specification and Re-Specification, with only the latter belonging to the cycle, as illustrated in Figure C1.

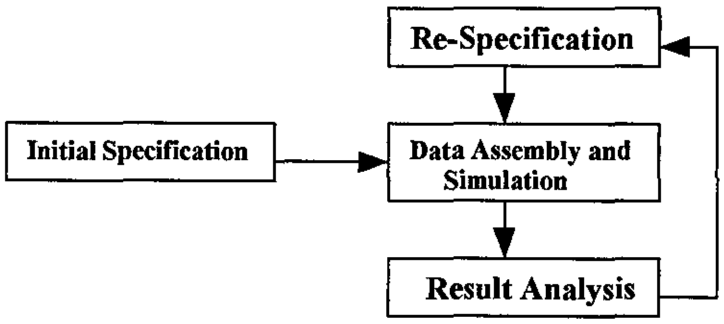


Figure C1: Modelling Simulation Project Cycle.

Model specification is the task of developing a representation of the system that is to be studied. The OPNET supports the concept of model reuse so that most models are based on primitive lower level models developed beforehand and stored in model libraries. The library models can be used for any applicable simulation scenario.

C.3 Main Components of OPNET Modeller Software Architecture

The principal modelling tools for simulation of systems made available to users by the OPNET software package includes:

- Extensive Model Library
- Extensive Kernel Procedure Library and Library of Functions and Utilities
- Animation Utility Program
- OPNET Modelling Editors

We will now briefly summarise information on the principal elements of the tools listed above which OPNET provides to make communication network system modelling and simulation very interesting.

C.3.1 Overview Model Library

The OPNET provides an extensive library of models that can be used to build networks. These models fall into three categories: the *standard models*, the *specialised models and* the *contributed models*. The standard models are made available for users to use at will. The specialised models support the needs of users with particular interests in emerging or vendor-specific technologies. These are like the standard models, but an additional license is needed to use these models in a simulation. Contributed models are models designed by some users and made available to the user community at no charge.

Most users work primarily with objects from the standard model library. A user may decide to design his or her own model taking advantage of the available extensive library of functions.

The standard model library consists of the following types of objects—Communication Devices (Node), Links, LANs and Clouds, and Utility objects.

LAN and Cloud Models: The OPNET abstract local area network infrastructure into one object, and called it a LAN object. The LAN object models many users on the same LAN, and allows for a server within the LAN as well. This paradigm dramatically reduce the amount of configuration you need to do to represent your internetwork of LANs.

In a similar manner to the use of LAN objects, parts of wide area network (WAN) infrastructure can be abstracted into a Cloud model to represent the WAN infrastructure. The Cloud model provides high-level characteristics used to simulate the behaviour of the WAN networks. ATM, Frame Relay, and IP model suites can all be included in a single Cloud object model.

Utility Objects: Objects that don't correspond to actual physical infrastructure but also used to construct network models are grouped as utility objects in the model library. In general, these represent logical function in the network, such as configuration of network resources on a global level.

C.3.2 Kernel Procedure—the Library of Functions

The OPNET provides extensive library of functions for communication network and distributed system modelling. The packages known as Kernel Procedures (KPs) represent services provided by the Simulation Kernel for modelling communication networks and distributed systems. Simulation services are accessed through the KPs, which are procedures that can be called from within process models, Transceiver Pipeline stages, C/C++ functions that have been scheduled as interrupts, or simply C/C++ functions that are directly or indirectly invoked from one of these contexts.

The KPs are categorised by primary function, based on the types of objects they attempt to manipulate. The collection of the KPs within a category is called a package, and the KPs within the same package share a common package keyword in their procedure names. For instance, a large number of KPs are concerned with manipulating packets; these are grouped together in the packet package and use the "pk" keyword. Detailed information on the use of KP and the means of accessing them can be found in the *Discrete Event Simulation API Reference Manual* of OPNET Documentation. OPNET also has other extensive library of function to make modelling construct interesting.

C.3.3 Animation Utility Programs

Animations of simulations provide useful support for analysing, verifying, and troubleshooting dynamic models. Simulations developed within OPNET's modelling editors offer animations for investigating the behaviour and performance of dynamic models. Information on Animation System Architecture can be found on Utility Programs Reference Manual of OPNET Documentation.

C.3.4 OPNET Modelling Editors

The OPNET supports model specification with a number of tools, called editors, which have built-in capabilities to capture the characteristics of a modelled system's behaviour. The suite of editors made it easier to address different aspects of a model, and offer specific capabilities to address the diverse issues encountered in networks and distributed systems. These editors present the model developer with an intuitive interface required for modelling information in a manner that is parallel to the structure of real network systems. The model-specification editors are organised hierarchically. Model specifications performed in the Project Editor rely on elements specified in the Node Editor; in turn, when working in the Node Editor, process models developed in the Process Editor and External System Editor will be used. The remaining editors are used to define various data models, typically tables of values that are later referenced by process- or node-level models. This organisation is depicted in the following list:

- Project Editor—Develop network models. Network models are made up of topology, subnet and node models. This editor also includes basic simulation and analysis capabilities.
- Node Editor—Develop node models. Node models are objects in a network model. Node models are made up of modules, which is made up of process models. Modules may also include parameter models.
- Process Editor—Develop process models. Process models control module behaviour and may reference parameter models.
- External System Editor—Develop external system definitions. External system definitions are necessary for cosimulation.
- Link Model Editor—Create, edit, and view link models.
- Packet Format Editor—Develop packet formats models. Packet formats dictate the structure and order of information stored in a packet.

- ICI Editor—Create, edit, and view interface control information (ICI) formats. ICIs are used to communicate control information between processes.
- PDF Editor—Create, edit, and view probability density functions (PDFs). PDFs can be used to control certain events, such as the frequency of packet generation in a source module.

There are other editors, which are not included in the list above.

As highlighted above, OPNET Models are structured hierarchically, in a manner that parallels real network systems. Specialised editors address issues at different levels of the hierarchy. This provides an intuitive modelling environment and also permits re-use of lower level models. All the model-specification editors present a graphical interface in which the user manipulates objects representing the model components and structure. Each editor has its own specific set of objects and operations that are correct for the modelling task on which it is focused. For instance, the Project Editor uses node and link objects; the Node Editor provides processors, queues, transmitters, and receivers; and the Process Editor is based on states and transitions.
The core modelling and simulation operations carried out in the OPNET software environment are centred on three principal editors, which are, the Project Editor, the Node Editor, and the Process Editor. The key functionality in system model specifications and definitions are carried out making use of these three editors, and their environments are referred to as *modelling domains*. Modelling operations involving the three domains (network domain, node domain and process domain) has a hierarchical flow orientation. This hierarchical modelling domain paradigm adopted by the OPNET has structural orientation equivalence to what is obtainable in real systems. The work flow is: process models specified and defined in the process domain are used in building a module, modules specified and defined in the node domain are used to build a node model and node models are interconnected to build a communication network. This hierarchy is illustrated with Figure C2.
It is worthwhile to summarily have a view of the modelling operations carried out in these key-modelling environments.

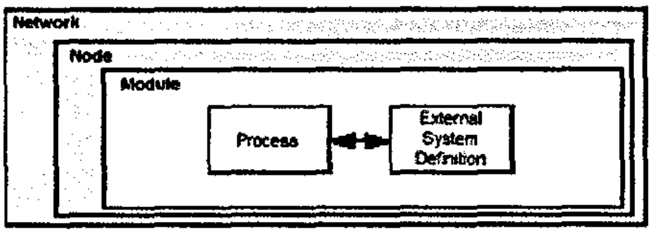


Figure C2: Relationship of Hierarchical Levels in OPNET Models

C.3.4.1 The Project Editor—Network Domain

The Project Editor is the workplace with built-in capabilities to model communication networks and it is referred to as *Network Domain.* The specific real life communication network whose performance analysis is of interest will be called *network model* in the environment of the Project Editor. The Network Domain's role is to define the topology of a communication network. The communicating entities are called *nodes* and are interconnected by communication *links*. The specific capability of each node is defined by designating its model from available *node models*. The node models are developed using the Node Editor. Within one network model, there may be many nodes that are based on the same node model. The term *node instance* is used to refer to an individual node to distinguish it from the class of nodes sharing the same model.

Network models are composed of the following main building blocks: *subnetworks, communication nodes,* and *communication links.* These objects, either singly or as a whole, may be referred to as a *site*. A subnetwork encapsulates other network level objects. Communication nodes model network objects with definable internal structure. Communication links provide a mechanism to transport information between communication nodes.

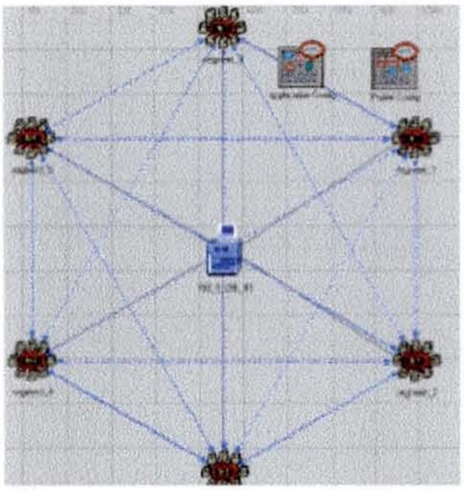
A network model defines the overall scope of a system to be simulated. It is a high-level description of the objects contained in the system. The network model specifies the objects in the system, as well as their physical locations, interconnections and configurations. The size and scope of the networks modelled can range from simple to complex topology. A network model may contain one node, or one subnetwork, or many interconnected nodes and subnetworks, because the structure and complexity of a network model typically follows those of the system to be modelled.

A network model may make use of any number of node models. Modellers can develop their own library of customised node models, implementing any functionality they require. The Project Editor can provide a geographic context for network model development. Modellers can choose locations on world or country maps for the elements of wide-area networks and can use dimensioned areas for local-area networks. Different types of links—point-to-point, bus and wireless include objects made available in the Project Editor.

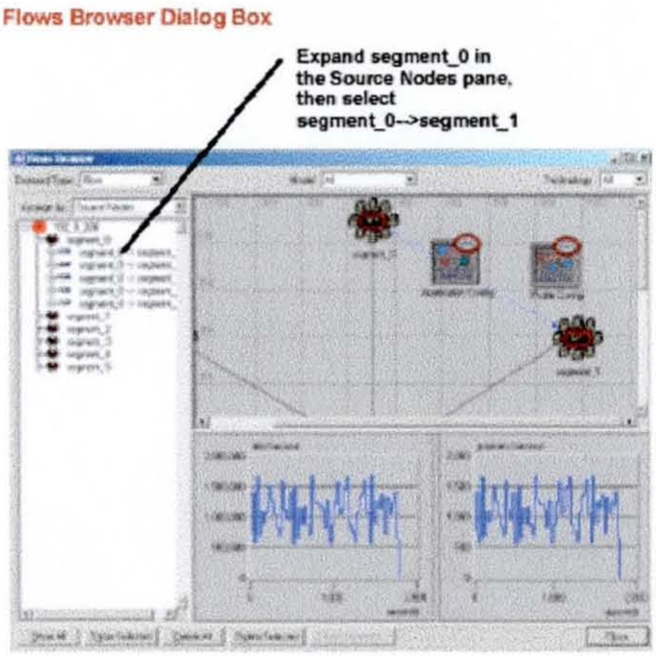
The concept of subnetwork objects in fixed, mobile, and satellite topology in the network domain is to provide hierarchy in the network model, and are used to break down complexity into multiple levels. Subnets can contain various combinations of nodes, links, and other subnets, and can be nested to any depth.

The Project Editor provides interface for modellers to control the characteristics and behaviour of objects in the Network Domain. Thus the characteristics and behaviour of communication subnetworks, communication nodes and communication links can be controlled through the appropriate changes to parameters made available in their attribute interface.

Figure C3 (a) and (b) are used to illustrate the modelling and simulation capabilities made available to users in the Project Editor. Figure C3 (a) shows an example of the use of the Project Editor, in which the topology of a corporate network that consists of six **LAN subnetworks**, interconnected by a **single switch** has been modelled. The corporate network model includes two utility nodes, which are used for configuration of applications used in the network and for configuration of application user's profiles. Figure C3 (b) shows that within the project editor you can define statistics to collect, run simulation and display results after simulation. In the Figure C3 (b) the user displayed the throughput for background traffic from LAN segment_0 to LAN segment_1 in bits per second and packets per second.



(a)


Flows Browser Dialog Box
Expand segment_0 in the Source Nodes pane, then select segment_0-->segment_1

(b)

Figure C3: (a) A corporate network topology, which consists of six LAN, a switch and two utility nodes. (b) The use of Project Editor to display the results of simulation.

C.3.4.2 The Node Editor —Node Domain

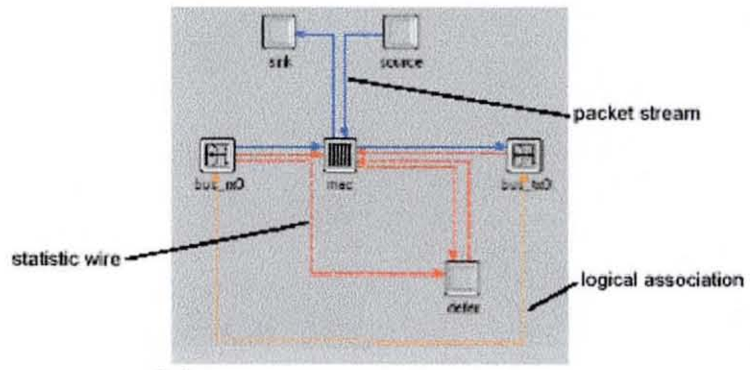
The next in hierarchy of workplace when working from top down on communication networks modelling after the Network Domain is the Node Domain. This approach is the paradigm employed in the OPNET software-modelling environment since in real life systems, you decompose communication networks to its constituent nodes. The Node Editor, whose workplace is referred to as *Node Domain*, provides tools for the modelling of communication devices that can be deployed and interconnected at the network level. In OPNET terms, these devices are called *nodes*, and in the real world they may correspond to various types of computing and communicating equipment such as routers, bridges, workstations, terminals, mainframe computers, file servers, fast packet switches, satellites, and so on.

Node models are developed in the Node Editor and are specified in terms of smaller building blocks called *modules*. Some modules offer capability that is substantially predefined and can only be configured through a set of built-in parameters. These include various *transmitters* and *receivers* allowing a node to be attached to communication links in the network domain. Other modules, called *processors*, *queues*, and *external systems*, are highly programmable, their behaviour being prescribed by an assigned *process model*. Process models are developed using the Process Editor.

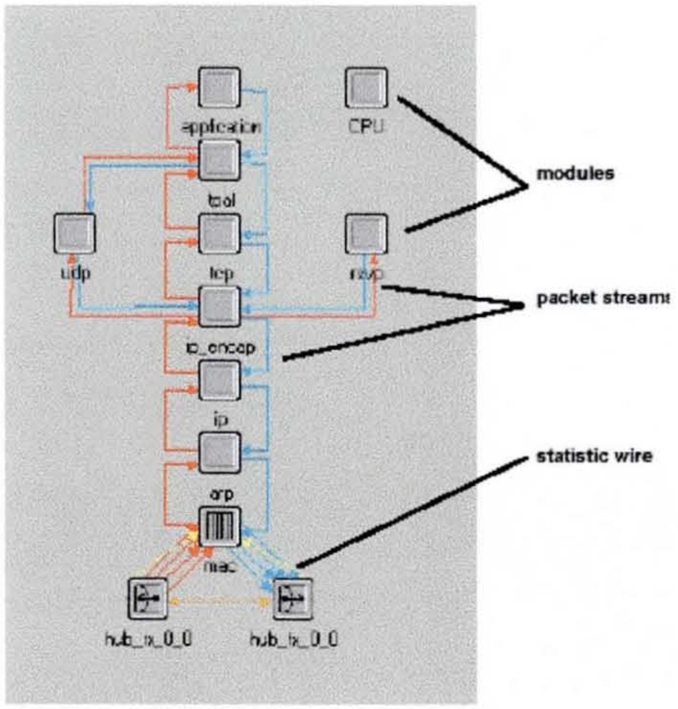
A node model can consist of any number of modules of different types. Three types of connections are provided to support interaction between modules. These are called *packet streams*, *statistic wires* and *logical associations*. Packet streams allow *packets* to be conveyed from one module to another. Statistic wires convey simple numeric signals or control information between modules, and are typically used when one module needs to monitor the performance or state of another. Both packet streams and statistic wires have parameters that may be set to configure some aspects of their behaviour. Logical associations identify a binding between modules. Currently, they are allowed only between transmitters and receivers to indicate that they should be used as a pair when attaching the node to a link in the Network Domain.

The modelling paradigm selected for the Node Domain was designed to support general modelling of high-level communication devices. It is particularly well suited for modelling arrangements of stacked or layered communication protocols. In the Node Editor, a device that relies on a particular stack of protocols can be modelled by creating a processor object for each layer of those stack and defining packet streams between neighbouring layers.

Figure C4 (a) shows a typical node model developed in the Node Editor that includes the three types of connections. And Figure C4 (b) shows a node model which employ the TCP/IP protocol stacks, in this case the Ethernet Server node model.



(a)



(b)

Figure C4: (a) Node model employing *packet streams*, *statistic wires*, and *logical associations* (b) Ethernet server node model.

C.3.4.3 The Process Editor—Process Domain

As highlighted in the previous section, each node can be decomposed into its constituent modules. Thus the next workplace in OPNET software environment after the Node Domain is the *Process Domain* which is the workplace environment of the Process Editor. The Process Editor is used to develop Process models, which are used in specifying and defining module behaviours.

Queue and processor modules are user-programmable elements that are key elements of communication nodes. The tasks that these modules execute are called *processes*. A process is similar to an executing software program, since it has a set of instructions and maintains state memory. Processes in OPNET are based on process models that are defined in the Process Editor. The relationship between process model and process is similar to the relationship between a program and a particular session of that program running as a task. Just as nodes created in the Project Editor are instances of node models defined with the Node Editor, each process that executes in a queue or processor module is an instance of a particular process model.

The process modelling paradigm of OPNET supports the concepts of *process groups*. A process group consists of multiple processes that execute within the same processor or queue. When a simulation begins, each module has only one process, termed the *root process*. This process can later create new processes, which can in turn create others as well, etc (no limits to the number of processes that may be created in a particular processor or queue module). When a process creates another one, it is termed the new process' *parent*; the new process is called the *child* of the process that created it. Processes that are created during the simulation are referred to as *dynamic processes*. Processes may be created and destroyed based on dynamic conditions that are analysed by the logic of the executing processes. This paradigm provides a very natural framework for modelling many common systems. In particular, multitasking operating systems where the root process represents the operating system itself and the dynamically created processes correspond to new tasks. Or in multi-context protocols where the root process represents a session

manager, for example, and each new session that is requested is modelled by creating a new process of the correct type.
Only one process can be executing at any time. A process is considered to be executing when it is progressing through new instructions that are part of its process model. When a process begins execution it is said to be invoked. A process that is currently executing can invoke another process in its process group to cause it to begin executing. When this happens, the invoking process is temporarily suspended until the invoked process blocks. A process blocks by indicating that it has completed its processing for its current invocation. After the invoked process has blocked, the invoking process resumes execution where it had left off, in a manner similar to the procedure-call mechanism in a programming language such as C.

Processes respond to interrupts, which indicate that events of interest have occurred such as the arrival of a message or the expiration of a timer. When a process is interrupted, it takes actions in response and then blocks, awaiting a new interrupt. It may also invoke another process; its execution is suspended until the invoked process blocks. Interrupts and / invocations may be generated by sources external to a process group, by other members of a process group, or by a process for itself.

The OPNET's Process Editor specifies and defines process models in a language called Proto-C, which is specifically designed to support development of protocols and algorithms. Proto-C is based on a combination of state transition diagrams (STDs), a library of high-level commands known as *Kernel Procedures*, and the general facilities of the C or C++ programming language. A process model's STD defines a set of primary modes or *states* that the process can enter and, for each state, the conditions that would cause the process to move to another state. The condition needed for a particular change in state to occur and the associated Destination State are called a *transition*. **Proto-C** models allow actions to be specified at various points in the finite state machine (FSM).

The state transition diagram representation of **Proto-C** is well suited to the specification of an interrupt-driven system because it methodically decomposes the states of the system and the processing that should take place at each interrupt.

In a process model, parameters can be defined, which are called *attributes*. These are used when the process model is instantiated as a process to customise aspects of its behaviour. This technique fosters reuse of process models for various purposes by avoiding hardwired specification where possible. For instance, a process model that performs window-based flow control may be defined with the window size as an attribute, so that it is reusable in different situations requiring different values of the window size.

The following Figure C5, taken from the Process Editor, shows example of a process model's STD.

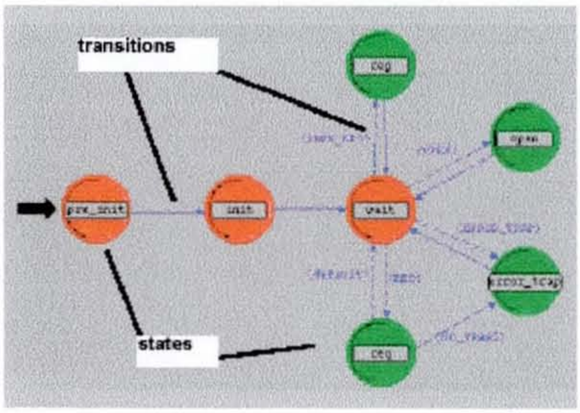


Figure C5: Showing example of a Process Model from the Process Editor.

APPENDIX D

OPNET Proto-C Sample Code for Traffic Generator and Sink in a Host

The work on performance evaluation of PDERRM was centred on modelling and simulation of a number of communication network scenarios in which network nodes made use of PDERRM process models. This involved extensive code generation. In view of limitation of space, the presentations in this appendix and the next two appendixes consist only limited sample code to represent few key operations.

Essentially, there were five basic *modules* used for building an *End-Host* node model in the work on performance evaluation of PDERRM. These are; *traffic generator, traffic sink, traffic processor, transmitter and receiver* modules. The transmitter and the receiver modules were predefined by OPNET, users could only configure them through their parameter attribute interface. The sample code for the process model of traffic generator and traffic sink will be presented in this appendix. Section D.1 presents sample code for traffic generator process model, while Section D.2 presents the process model for traffic sink.

D.1 Process Model for Traffic Generator

```
/* The Process Model for Host Traffic Generator */
/* Process model C form file: pre_src_gen.pr.c */
/* Portions of this file copyright 1992-2003 by OPNET Technologies,
Inc. */

/* This variable carries the header into the object file */
const char pre_src_gen_pr_c [] = "MIL_3_Tfile_Hdr_ 100A 30A
op_runsim 7 43088516 43088516 1 initial model 0 0 none none 0 0 none
0 0 0 0 0 0 0 0 90b 2";

#include <string.h>

/* OPNET system definitions */
#include <opnet.h>

/* Header Block */

/* Include files.                              */
#include     <oms_dist_support.h>

/* Special attribute values.            */
#define            SSC_INFINITE_TIME          -1.0
```

```
/* Interrupt code values.                 */
#define           SSC_START                     0
#define           SSC_GENERATE                  1
#define           SSC_STOP                      2

/* Node configuration constants.    */
#define           SSC_STRM_TO_LOW               0

/* Macro definitions for state            */
/* transitions.                                 */
#define           START                   (intrpt_code == SSC_START)
#define           DISABLED                (intrpt_code == SSC_STOP)
#define           STOP                    (intrpt_code == SSC_STOP)
#define           PACKET_GENERATE         (intrpt_code ==
SSC_GENERATE)

/* Function prototypes.                   */
static void             ss_packet_generate (void);

/* End of Header Block */

/* OPNET predefine code block */
#if !defined (VOSD_NO_FIN)
#undef      BIN
#undef      BOUT
#define     BIN         FIN_LOCAL_FIELD(_op_last_line_passed) =
__LINE__ - _op_block_origin;
#define     BOUT  BIN
#define     BINIT FIN_LOCAL_FIELD(_op_last_line_passed) = 0;
_op_block_origin = __LINE__;
#else
#define     BINIT
#endif /* #if !defined (VOSD_NO_FIN) */


/* State variable definitions */
typedef struct
      {
      /* Internal state tracking for FSM */
      FSM_SYS_STATE
      /* State Variables */
      Objid                               own_id;
      char                                format_str [64];
      double                              start_time;
      double                              stop_time;
      OmsT_Dist_Handle                    interarrival_dist_ptr;
      OmsT_Dist_Handle                    pksize_dist_ptr;
      Boolean                             generate_unformatted;
      Evhandle                            next_pk_evh;
      double                              next_intarr_time;
      Stathandle                          bits_sent_hndl;
      Stathandle                          packets_sent_hndl;
      Stathandle                          packet_size_hndl;
      Stathandle                          interarrivals_hndl;
      } pre_src_gen_state;

#define pr_state_ptr                      ((pre_src_gen_state*)
(OP_SIM_CONTEXT_PTR->mod_state_ptr))
#define own_id                            pr_state_ptr->own_id
#define format_str                        pr_state_ptr->format_str
```

```
#define start_time                              pr_state_ptr->start_time
#define stop_time                               pr_state_ptr->stop_time
#define interarrival_dist_ptr                   pr_state_ptr-
>interarrival_dist_ptr
#define pksize_dist_ptr                         pr_state_ptr-
>pksize_dist_ptr
#define generate_unformatted                    pr_state_ptr-
>generate_unformatted
#define next_pk_evh                             pr_state_ptr->next_pk_evh
#define next_intarr_time                        pr_state_ptr-
>next_intarr_time
#define bits_sent_hndl                          pr_state_ptr-
>bits_sent_hndl
#define packets_sent_hndl                       pr_state_ptr-
>packets_sent_hndl
#define packet_size_hndl                        pr_state_ptr-
>packet_size_hndl
#define interarrivals_hndl                      pr_state_ptr-
>interarrivals_hndl

/* These macro definitions will define a local variable called    */
/* "op_sv_ptr" in each function containing a FIN statement. */
/* This variable points to the state variable data structure,     */
/* and can be used from a C debugger to display their values.     */
#undef FIN_PREAMBLE_DEC
#undef FIN_PREAMBLE_CODE
#if defined (OPD_PARALLEL)
#  define FIN_PREAMBLE_DEC     pre_src_gen_state *op_sv_ptr;
OpT_Sim_Context * tcontext_ptr;
#  define FIN_PREAMBLE_CODE
if (VosS_Mt_Perform_Lock) \
    VOS_THREAD_SPECIFIC_DATA_GET
(VosI_Globals.simi_mt_context_data_key, tcontext_ptr,
SimT_Context*);

else tcontext_ptr = VosI_Globals.simi_sequential_context_ptr;
op_sv_ptr = ((pre_src_gen_state *)(tcontext_ptr->mod_state_ptr));

#else

#define FIN_PREAMBLE_DEC      pre_src_gen_state *op_sv_ptr;
#define FIN_PREAMBLE_CODE     op_sv_ptr = pr_state_ptr;
#endif

/* Function Block */

#if !defined (VOSD_NO_FIN)
enum { _op_block_origin = __LINE__ };
#endif
static void ss_packet_generate (void)
      {
      Packet*                     pkptr;
      double                      pksize;

      /** This function creates a packet based on the packet
generation specifications of the source model and sends it to the
lower layer.      **/
      FIN (ss_packet_generate ());

      /* Generate a packet size outcome.                              */
      pksize = (double) ceil (oms_dist_outcome (pksize_dist_ptr));
      /* Create a packet of specified format and size.      */
      if (generate_unformatted == OPC_TRUE)
            {
            /* We produce unformatted packets. Create one.  */
            pkptr = op_pk_create (pksize);
            }
      else
            {
            /* Create a packet with the specified format.   */
            pkptr = op_pk_create_fmt (format_str);
            op_pk_total_size_set (pkptr, pksize);
            }

      /* Update the packet generation statistics.                     */
      op_stat_write (packets_sent_hndl, 1.0);
      op_stat_write (packets_sent_hndl, 0.0);
      op_stat_write (bits_sent_hndl, (double) pksize);
      op_stat_write (bits_sent_hndl, 0.0);
      op_stat_write (packet_size_hndl, (double) pksize);
      op_stat_write (interarrivals_hndl, next_intarr_time);

      /* Send the packet via the stream to the lower layer. */
      op_pk_send (pkptr, SSC_STRM_TO_LOW);

      FOUT;
      }

/* End of Function Block */

/* Tracing used for debug purposes */
/* Undefine optional tracing in FIN/FOUT/FRET */
/* The FSM has its own tracing code and the other */
/* functions should not have any tracing.         */
#undef FIN_TRACING
#define FIN_TRACING

#undef FOUTRET_TRACING
#define FOUTRET_TRACING

#if defined (__cplusplus)
extern "C" {
#endif
      void pre_src_gen (OP_SIM_CONTEXT_ARG_OPT);
      VosT_Obtype pre_src_gen_init (int * init_block_ptr);
      VosT_Address pre_src_gen_alloc (VOS_THREAD_INDEX_ARG_COMMA
VosT_Obtype, int);
      void pre_src_gen_diag (OP_SIM_CONTEXT_ARG_OPT);
      void pre_src_gen_terminate (OP_SIM_CONTEXT_ARG_OPT);
      void pre_src_gen_svar (void *, const char *, void **);


      VosT_Fun_Status Vos_Define_Object (VosT_Obtype * _op_obst_ptr,
const char * _op_name, unsigned int _op_size, unsigned int
_op_init_obs, unsigned int _op_inc_obs);
      VosT_Address Vos_Alloc_Object_MT (VOS_THREAD_INDEX_ARG_COMMA
VosT_Obtype _op_ob_hndl);
      VosT_Fun_Status Vos_Poolmem_Dealloc_MT
(VOS_THREAD_INDEX_ARG_COMMA VosT_Address _op_ob_ptr);
#if defined (__cplusplus)
}
```

```
/* end of 'extern "C"' */
#endif

/* Process model interrupt handling procedure */

void pre_src_gen (OP_SIM_CONTEXT_ARG_OPT)
      {

#if !defined (VOSD_NO_FIN)
      int _op_block_origin = 0;
#endif
      FIN_MT (pre_src_gen ());
      if (1)
            {
            /* Variables used in the "init" state.          */
            char         interarrival_str [128];
            char         size_str [128];
            Prg_List*    pk_format_names_lptr;
            char*        found_format_str;
            int               low, high;
            Boolean           format_found;
            int               i;

            /* Variables used in state transitions.         */
            int               intrpt_code;


            FSM_ENTER ("pre_src_gen")

FSM_BLOCK_SWITCH
            {

            /** state (init) enter executives **/
            FSM_STATE_ENTER_UNFORCED_NOLABEL (0, "init",
"pre_src_gen
            [init enter execs]")
            FSM_PROFILE_SECTION_IN ("pre_src_gen [init enter
execs]",
            state0_enter_exec)
            {
            /* At this initial state, we read the values of source
            attributes and schedule a set interrupt that will
            indicate our start time for packet generation. Obtain
the
            object id of the surrounding module. */

           own_id = op_id_self ();

         /* Read the values of the packet generation parameters,
i.e.
         the attribute values of the surrounding module. */

         op_ima_obj_attr_get (own_id, "Packet Interarrival Time",\
         interarrival_str);
         op_ima_obj_attr_get (own_id, "Packet Size", size_str);
         op_ima_obj_attr_get (own_id, "Packet Format", format_str);
         op_ima_obj_attr_get (own_id, "Start Time",  &start_time);
         op_ima_obj_attr_get (own_id, "Stop Time", &stop_time);

      /* Load the PDFs that will be used in computing the packet */
      /* interarrival times and packet sizes.                    */
      interarrival_dist_ptr =\
      oms_dist_load_from_string(interarrival_str);
      pksize_dist_ptr        = oms_dist_load_from_string (size_str);

      /* Verify the existence of the packet format to be used for */
      /* generated packets.                                        */
      if (strcmp (format_str, "NONE") == 0)
        {
        /* We will generate unformatted packets. Set the flag. */
        generate_unformatted = OPC_TRUE;
         }
      else
      {
      /* We will generate formatted packets. Turn off the flag. */
      generate_unformatted = OPC_FALSE;

      /* Get the list of all available packet formats.      */
      pk_format_names_lptr = prg_tfile_name_list_get\
       (PrgC_Tfile_Type_Packet_Format);

      /* Search the list for the requested packet format.   */
      format_found = OPC_FALSE;
      for (i = prg_list_size (pk_format_names_lptr);\
           ((format_found == OPC_FALSE) && (i > 0)); i--)
       {
      /* Access the next format name and compare with requested   */
      /* format name.                                              */
      found_format_str = (char *) prg_list_access\
      (pk_format_names_lptr, i - 1);
      if (strcmp (found_format_str, format_str) == 0)
      format_found = OPC_TRUE;
      }

     if (format_found == OPC_FALSE)
      {
      /* The requested format does not exist. Generate unformatted
       packets. */

      generate_unformatted = OPC_TRUE;

      /* Display an appropriate warning.        */
      op_prg_odb_print_major ("Warning from simple packet generator
      model (simple_source):", "The specified packet format",
      format_str, "is not found. Generating unformatted packets
      instead.", OPC_NIL);
      }
      /* Destroy the lits and its elements since we don't need it
      anymore. */
      prg_list_free (pk_format_names_lptr);
      prg_mem_free  (pk_format_names_lptr);
      }
/* Make sure we have valid start and stop times, i.e. stop time is
not earlier than start time. */

if ((stop_time <= start_time) && (stop_time != SSC_INFINITE_TIME))
   {
   /* Stop time is earlier than start time. Disable the source. */
   start_time = SSC_INFINITE_TIME;

  /* Display an appropriate warning. */
  op_prg_odb_print_major ("Warning from simple packet generator
model
  (simple_source):", "Although the generator is not disabled (start
time is set to a finite value),", "a stop time that is not later
than the start time is specified.", "Disabling the generator.",
OPC_NIL);
}
/* Schedule a self interrupt that will indicate our start time for
*//* packet generation activities. If the source is disabled, */
/* schedule it at current time with the appropriate code value. */
if (start_time == SSC_INFINITE_TIME)
   {
   op_intrpt_schedule_self (op_sim_time (), SSC_STOP);
   }
 else
  {
  op_intrpt_schedule_self (start_time, SSC_START);

 /* In this case, also schedule the interrupt when we will stop */
 /* generating packets, unless we are configured to run until */
 /* the end of the simulation.      */

if (stop_time != SSC_INFINITE_TIME)
   {
   op_intrpt_schedule_self (stop_time, SSC_STOP);
 . }
   next_intarr_time = oms_dist_outcome (interarrival_dist_ptr);

/* Make sure that interarrival time is not negative. In that case
it\
 will be set to 0. */
if (next_intarr_time <0)
{
 next_intarr_time = 0.0;
}

}

/* Register the statistics that will be maintained by this model. */
bits_sent_hndl  = op_stat_reg ("Generator.Traffic Sent (bits/sec)",\
                  OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);

packets_sent_hndl  = op_stat_reg ("Generator.Traffic Sent \
(packets/sec)",   OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);

packet_size_hndl  = op_stat_reg ("Generator.Packet Size (bits)",
OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);

interarrivals_hndl  = op_stat_reg ("Generator.Packet Interarrival
Time (secs)", OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);

}
FSM_PROFILE_SECTION_OUT (state0_enter_exec)

/** blocking after enter executives of unforced state. **/
FSM_EXIT (1,"pre_src_gen")


/** state (init) exit executives **/
FSM_STATE_EXIT_UNFORCED (0, "init", "pre_src_gen [init exit execs]")
```

```
FSM_PROFILE_SECTION_IN ("pre_src_gen [init exit execs]",
state0_exit_exec)
            {
/*Determine the code of the interrupt, which is used in evaluating
*/
/* state transition conditions. */

intrpt_code = op_intrpt_code ();
}
FSM_PROFILE_SECTION_OUT (state0_exit_exec)

/** state (init) transition processing **/
FSM_PROFILE_SECTION_IN ("pre_src_gen [init trans conditions]",
state0_trans_conds)
FSM_INIT_COND (START)
FSM_TEST_COND (DISABLED)
FSM_TEST_LOGIC ("init")
FSM_PROFILE_SECTION_OUT (state0_trans_conds)

FSM_TRANSIT_SWITCH
{
FSM_CASE_TRANSIT (0, 1, state1_enter_exec, ss_packet_generate();,
"START", "ss_packet_generate()", "init", "generate")
FSM_CASE_TRANSIT (1, 2, state2_enter_exec, ;, "DISABLED", "",
"init", "stop")
}

/** state (generate) enter executives **/
FSM_STATE_ENTER_UNFORCED (1, "generate", state1_enter_exec,
"pre_src_gen [generate enter execs]")
FSM_PROFILE_SECTION_IN ("pre_src_gen [generate enter execs]",
state1_enter_exec)
{
/* At the enter execs of the "generate" state we schedule the */
/* arrival of the next packet. */
next_intarr_time = oms_dist_outcome (interarrival_dist_ptr);

/* Make sure that interarrival time is not negative. In that case it
will be set to 0. */
if (next_intarr_time <0)
   {
    next_intarr_time = 0;
    }

   next_pk_evh      = op_intrpt_schedule_self (op_sim_time () +
   next_intarr_time, SSC_GENERATE);

   }
   FSM_PROFILE_SECTION_OUT (state1_enter_exec)

  /** blocking after enter executives of unforced state. **/
   FSM_EXIT (3,"pre_src_gen")

  /** state (generate) exit executives **/
  FSM_STATE_EXIT_UNFORCED (1, "generate", "pre_src_gen [generate
exit
  execs]")
  FSM_PROFILE_SECTION_IN ("pre_src_gen [generate exit execs]",
  state1_exit_exec)
  {
```

```
  /* Determine the code of the interrupt, which is used in
evaluating
    state transition conditions. */
    intrpt_code = op_intrpt_code ();
   }
  FSM_PROFILE_SECTION_OUT (state1_exit_exec)

 /** state (generate) transition processing **/
 FSM_PROFILE_SECTION_IN ("pre_src_gen [generate trans conditions]",
 state1_trans_conds)
 FSM_INIT_COND (STOP)
 FSM_TEST_COND (PACKET_GENERATE)
 FSM_DFLT_COND
 FSM_TEST_LOGIC ("generate")
 FSM_PROFILE_SECTION_OUT (state1_trans_conds)

 FSM_TRANSIT_SWITCH
   {
    FSM_CASE_TRANSIT (0, 2, state2_enter_exec, ;, "STOP", "",
    "generate", "stop")
    FSM_CASE_TRANSIT (1, 1, state1_enter_exec,
ss_packet_generate();,
   "PACKET_GENERATE", "ss_packet_generate()", "generate",
"generate")
  FSM_CASE_TRANSIT (2, 1, state1_enter_exec, ;, "default", "",
   "generate", "generate")
  }


/** state (stop) enter executives **/
FSM_STATE_ENTER_UNFORCED (2, "stop", state2_enter_exec, "pre_src_gen
[stop enter execs]")
FSM_PROFILE_SECTION_IN ("pre_src_gen [stop enter execs]",
state2_enter_exec)
{
/* When we enter into the "stop" state, it is the time for us to  */
/* stop generating traffic. We simply cancel the generation of the
*/
/* next packet and go into a silent mode by not scheduling anything
else. */
if (op_ev_valid (next_pk_evh) == OPC_TRUE)
   {
    op_ev_cancel (next_pk_evh);
   }

       }
       FSM_PROFILE_SECTION_OUT (state2_enter_exec)

       /** blocking after enter executives of unforced state. **/
       FSM_EXIT (5,"pre_src_gen")

       /** state (stop) exit executives **/
       FSM_STATE_EXIT_UNFORCED (2, "stop", "pre_src_gen [stop exit
       execs]")
       FSM_PROFILE_SECTION_IN ("pre_src_gen [stop exit execs]",
       state2_exit_exec)
       {
       }
       FSM_PROFILE_SECTION_OUT (state2_exit_exec)

       /** state (stop) transition processing **/
```

```
      FSM_TRANSIT_MISSING ("stop")
      }
      FSM_EXIT (0,"pre_src_gen")
      }
      }
      void
      pre_src_gen_diag (OP_SIM_CONTEXT_ARG_OPT)
      {
      /* No Diagnostic Block */
      }
      void
      pre_src_gen_terminate (OP_SIM_CONTEXT_ARG_OPT)
      {
      #if !defined (VOSD_NO_FIN)
      int _op_block_origin = __LINE__;
      #endif

      FIN_MT (pre_src_gen_terminate ())

      Vos_Poolmem_Dealloc_MT (OP_SIM_CONTEXT_THREAD_INDEX_COMMA
      pr_state_ptr);

      FOUT
      }

/* Undefine shortcuts to state variables to avoid */
/* syntax error in direct access to fields of */
/* local variable prs_ptr in pre_src_gen_svar function. */
#undef own_id
#undef format_str
#undef start_time
#undef stop_time
#undef interarrival_dist_ptr
#undef pksize_dist_ptr
#undef generate_unformatted
#undef next_pk_evh
#undef next_intarr_time
#undef bits_sent_hndl
#undef packets_sent_hndl
#undef packet_size_hndl
#undef interarrivals_hndl

#undef FIN_PREAMBLE_DEC
#undef FIN_PREAMBLE_CODE
#define FIN_PREAMBLE_DEC
#define FIN_PREAMBLE_CODE

VosT_Obtype
pre_src_gen_init (int * init_block_ptr)
      {

#if !defined (VOSD_NO_FIN)
      int _op_block_origin = 0;
#endif
      VosT_Obtype obtype = OPC_NIL;
      FIN_MT (pre_src_gen_init (init_block_ptr))

      Vos_Define_Object (&obtype, "proc state vars (pre_src_gen)",
            sizeof (pre_src_gen_state), 0, 20);
      *init_block_ptr = 0;
```

```
	FRET (obtype)
	}

VosT_Address
pre_src_gen_alloc (VOS_THREAD_INDEX_ARG_COMMA VosT_Obtype obtype,
int init_block)
	{
#if !defined (VOSD_NO_FIN)
	int _op_block_origin = 0;
#endif
	pre_src_gen_state * ptr;
	FIN_MT (pre_src_gen_alloc (obtype))

	ptr = (pre_src_gen_state *)Vos_Alloc_Object_MT
(VOS_THREAD_INDEX_COMMA obtype);
	if (ptr != OPC_NIL)
		ptr->_op_current_block = init_block;
	FRET ((VosT_Address)ptr)
	}
void
pre_src_gen_svar (void * gen_ptr, const char * var_name, void **
var_p_ptr)
	{
	pre_src_gen_state		*prs_ptr;

	FIN_MT (pre_src_gen_svar (gen_ptr, var_name, var_p_ptr))

	if (var_name == OPC_NIL)
		{
		*var_p_ptr = (void *)OPC_NIL;
		FOUT
		}
	prs_ptr = (pre_src_gen_state *)gen_ptr;

	if (strcmp ("own_id" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->own_id);
		FOUT
		}
	if (strcmp ("format_str" , var_name) == 0)
		{
		*var_p_ptr = (void *) (prs_ptr->format_str);
		FOUT
		}
	if (strcmp ("start_time" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->start_time);
		FOUT
		}
	if (strcmp ("stop_time" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->stop_time);
		FOUT
		}
	if (strcmp ("interarrival_dist_ptr" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->interarrival_dist_ptr);
		FOUT
		}
	if (strcmp ("pksize_dist_ptr" , var_name) == 0)
		{
```

```
            *var_p_ptr = (void *) (&prs_ptr->pksize_dist_ptr);
            FOUT
            }
      if (strcmp ("generate_unformatted" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->generate_unformatted);
            FOUT
            }
      if (strcmp ("next_pk_evh" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->next_pk_evh);
            FOUT
            }
      if (strcmp ("next_intarr_time" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->next_intarr_time);
            FOUT
            }
      if (strcmp ("bits_sent_hndl" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->bits_sent_hndl);
            FOUT
            }
      if (strcmp ("packets_sent_hndl" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->packets_sent_hndl);
            FOUT
            }
      if (strcmp ("packet_size_hndl" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->packet_size_hndl);
            FOUT
            }
      if (strcmp ("interarrivals_hndl" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->interarrivals_hndl);
            FOUT
            }
      *var_p_ptr = (void *)OPC_NIL;

      FOUT
      }
```

D.2 Process Model for Traffic Sink

```
/* The Process Model for Host Traffic Sink */
/* Process model C form file: pre_rcv_proc.pr.c */
/* Portions of this file copyright 1992-2003 by OPNET Technologies,
Inc. */

/* This variable carries the header into the object file */
const char pre_rcv_proc_pr_c [] = "MIL_3_Tfile_Hdr_ 100A 30A
op_runsim 7 43088517 43088517 1 initial model 0 0 none none 0 0 none
0 0 0 0 0 0 0 0 0 90b 2";

#include <string.h>

/* OPNET system definitions */
#include <opnet.h>
```

```
/* Header Block */

#define ARRIVAL_RCV  (op_intrpt_type() == OPC_INTRPT_STRM)

/* End of Header Block */
/* OPNET predefine code block */
#if !defined (VOSD_NO_FIN)
#undef      BIN
#undef      BOUT
#define     BIN         FIN_LOCAL_FIELD(_op_last_line_passed) =
__LINE__ - _op_block_origin;
#define     BOUT  BIN
#define     BINIT FIN_LOCAL_FIELD(_op_last_line_passed) = 0;
_op_block_origin = __LINE__;
#else
#define     BINIT
#endif /* #if !defined (VOSD_NO_FIN) */

/* State variable definitions */
typedef struct
      {
      /* Internal state tracking for FSM */
      FSM_SYS_STATE
      /* State Variables */
      Stathandle                        bits_rcvd_stathandle;
      Stathandle                        bitssec_rcvd_stathandle;
      Stathandle                        pkts_rcvd_stathandle;
      Stathandle                        pktssec_rcvd_stathandle;
      Stathandle                        ete_delay_stathandle;
      Stathandle                        bits_rcvd_gstathandle;
      Stathandle                        bitssec_rcvd_gstathandle;
      Stathandle                        pkts_rcvd_gstathandle;
      Stathandle                        pktssec_rcvd_gstathandle;
      Stathandle                        ete_delay_gstathandle;
      } pre_rcv_proc_state;

#define pr_state_ptr                    ((pre_rcv_proc_state*)
(OP_SIM_CONTEXT_PTR->mod_state_ptr))
#define bits_rcvd_stathandle            pr_state_ptr-
>bits_rcvd_stathandle
#define bitssec_rcvd_stathandle         pr_state_ptr-
>bitssec_rcvd_stathandle
#define pkts_rcvd_stathandle            pr_state_ptr-
>pkts_rcvd_stathandle
#define pktssec_rcvd_stathandle         pr_state_ptr-
>pktssec_rcvd_stathandle
#define ete_delay_stathandle            pr_state_ptr-
>ete_delay_stathandle
#define bits_rcvd_gstathandle           pr_state_ptr-
>bits_rcvd_gstathandle
#define bitssec_rcvd_gstathandle        pr_state_ptr-
>bitssec_rcvd_gstathandle
#define pkts_rcvd_gstathandle           pr_state_ptr-
>pkts_rcvd_gstathandle
#define pktssec_rcvd_gstathandle        pr_state_ptr-
>pktssec_rcvd_gstathandle
#define ete_delay_gstathandle           pr_state_ptr-
>ete_delay_gstathandle

/* These macro definitions will define a local variable called    */
/* "op_sv_ptr" in each function containing a FIN statement. */
/* This variable points to the state variable data structure,     */
/* and can be used from a C debugger to display their values.     */
#undef FIN_PREAMBLE_DEC
#undef FIN_PREAMBLE_CODE
#if defined (OPD_PARALLEL)
#  define FIN_PREAMBLE_DEC    pre_rcv_proc_state *op_sv_ptr;
OpT_Sim_Context * tcontext_ptr;
#  define FIN_PREAMBLE_CODE   \
            if (VosS_Mt_Perform_Lock) \
                  VOS_THREAD_SPECIFIC_DATA_GET
(VosI_Globals.simi_mt_context_data_key, tcontext_ptr, SimT_Context
*); \
            else \
                  tcontext_ptr =
VosI_Globals.simi_sequential_context_ptr; \
            op_sv_ptr = ((pre_rcv_proc_state *)(tcontext_ptr-
>mod_state_ptr));
#else
#  define FIN_PREAMBLE_DEC    pre_rcv_proc_state *op_sv_ptr;
#  define FIN_PREAMBLE_CODE   op_sv_ptr = pr_state_ptr;
#endif

/* No Function Block */

/* OPNET predefined code block */
#if !defined (VOSD_NO_FIN)
enum { _op_block_origin = __LINE__ };
#endif

/* Undefine optional tracing in FIN/FOUT/FRET */
/* The FSM has its own tracing code and the other */
/* functions should not have any tracing.         */
#undef FIN_TRACING
#define FIN_TRACING

#undef FOUTRET_TRACING
#define FOUTRET_TRACING

#if defined (__cplusplus)
extern "C" {
#endif
      void pre_rcv_proc (OP_SIM_CONTEXT_ARG_OPT);
      VosT_Obtype pre_rcv_proc_init (int * init_block_ptr);
      VosT_Address pre_rcv_proc_alloc (VOS_THREAD_INDEX_ARG_COMMA
VosT_Obtype, int);
      void pre_rcv_proc_diag (OP_SIM_CONTEXT_ARG_OPT);
      void pre_rcv_proc_terminate (OP_SIM_CONTEXT_ARG_OPT);
      void pre_rcv_proc_svar (void *, const char *, void **);

      VosT_Fun_Status Vos_Define_Object (VosT_Obtype * _op_obst_ptr,
const char * _op_name, unsigned int _op_size, unsigned int
_op_init_obs, unsigned int _op_inc_obs);
      VosT_Address Vos_Alloc_Object_MT (VOS_THREAD_INDEX_ARG_COMMA
VosT_Obtype _op_ob_hndl);
      VosT_Fun_Status Vos_Poolmem_Dealloc_MT
(VOS_THREAD_INDEX_ARG_COMMA VosT_Address _op_ob_ptr);
#if defined (__cplusplus)
} /* end of 'extern "C"' */
#endif
```

```
/* Process model interrupt handling procedure */

void pre_rcv_proc (OP_SIM_CONTEXT_ARG_OPT)
      {
      #if !defined (VOSD_NO_FIN)
      int _op_block_origin = 0;
      #endif
      FIN_MT (pre_rcv_proc ());
      if (1)
            {
            Packet*           pkptr;
            double            pk_size;
            double            ete_delay;

            FSM_ENTER ("pre_rcv_proc")

            FSM_BLOCK_SWITCH
                  {
/*---------------------------------------------------------*/
                  /** state (DISCARD) enter executives **/
FSM_STATE_ENTER_FORCED (0, "DISCARD", state0_enter_exec,
"pre_rcv_proc [DISCARD enter execs]")
FSM_PROFILE_SECTION_IN ("pre_rcv_proc [DISCARD enter execs]",
state0_enter_exec)
{
 /* Obtain the incoming packet.     */
  pkptr = op_pk_get (op_intrpt_strm ());

 /* Caclulate metrics to be updated.            */
  pk_size = (double) op_pk_total_size_get (pkptr);
  ete_delay = op_sim_time () - op_pk_creation_time_get(pkptr);
 /* Update local statistics.                    */
 op_stat_write (bits_rcvd_stathandle, pk_size);
 op_stat_write (pkts_rcvd_stathandle, 1.0);
 op_stat_write (ete_delay_stathandle, ete_delay);

 op_stat_write (bitssec_rcvd_stathandle, pk_size);
 op_stat_write (bitssec_rcvd_stathandle, 0.0);
 op_stat_write (pktssec_rcvd_stathandle, 1.0);
 op_stat_write (pktssec_rcvd_stathandle, 0.0);

 /* Update global statistics. */
 op_stat_write (bits_rcvd_gstathandle, pk_size);
 op_stat_write (pkts_rcvd_gstathandle,    1.0);
 op_stat_write (ete_delay_gstathandle, ete_delay);

 op_stat_write (bitssec_rcvd_gstathandle, pk_size);
 op_stat_write (bitssec_rcvd_gstathandle, 0.0);
 op_stat_write (pktssec_rcvd_gstathandle, 1.0);
 op_stat_write (pktssec_rcvd_gstathandle, 0.0);

 /* Destroy the received packet.    */
 op_pk_destroy (pkptr);
 }

FSM_PROFILE_SECTION_OUT (state0_enter_exec)

 /** state (DISCARD) exit executives **/
 FSM_STATE_EXIT_FORCED (0, "DISCARD", "pre_rcv_proc [DISCARD exit
execs]")
```

```
 FSM_PROFILE_SECTION_IN ("pre_rcv_proc [DISCARD exit execs]",
state0_exit_exec)
      {
      }
FSM_PROFILE_SECTION_OUT (state0_exit_exec)

/** state (DISCARD) transition processing **/
FSM_TRANSIT_FORCE (2, state2_enter_exec, ;, "default", "",
"DISCARD", "st_12")
/*--------------------------------------------------------------*/

/** state (INIT) enter executives **/
FSM_STATE_ENTER_FORCED_NOLABEL (1, "INIT", "pre_rcv_proc [INIT enter
execs]")
FSM_PROFILE_SECTION_IN ("pre_rcv_proc [INIT enter execs]",
state1_enter_exec)
  {
  /* Initilaize the statistic handles to keep track of traffic
sinked by this process. */
bits_rcvd_stathandle = op_stat_reg ("Traffic Sink.Traffic Received
(bits)", OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);

bitssec_rcvd_stathandle = op_stat_reg ("Traffic Sink.Traffic
Received (bits/sec)", OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);

pkts_rcvd_stathandle = op_stat_reg ("Traffic Sink.Traffic Received
(packets)",OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);

pktssec_rcvd_stathandle = op_stat_reg ("Traffic Sink.Traffic
Received (packets/sec)",      OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);

ete_delay_stathandle = op_stat_reg ("Traffic Sink.End-to-End Delay
(seconds)", OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);

bits_rcvd_gstathandle = op_stat_reg ("Traffic Sink.Traffic Received
(bits)", OPC_STAT_INDEX_NONE, OPC_STAT_GLOBAL);

bitssec_rcvd_gstathandle = op_stat_reg ("Traffic Sink.Traffic
Received (bits/sec)", OPC_STAT_INDEX_NONE, OPC_STAT_GLOBAL);

pkts_rcvd_gstathandle = op_stat_reg ("Traffic Sink.Traffic Received
(packets)",OPC_STAT_INDEX_NONE, OPC_STAT_GLOBAL);

pktssec_rcvd_gstathandle = op_stat_reg ("Traffic Sink.Traffic
Received (packets/sec)", OPC_STAT_INDEX_NONE, OPC_STAT_GLOBAL);

ete_delay_gstathandle = op_stat_reg ("Traffic Sink.End-to-End Delay
(seconds)",OPC_STAT_INDEX_NONE, OPC_STAT_GLOBAL);
}

FSM_PROFILE_SECTION_OUT (state1_enter_exec)

/** state (INIT) exit executives **/
FSM_STATE_EXIT_FORCED (1, "INIT", "pre_rcv_proc [INIT exit execs]")
FSM_PROFILE_SECTION_IN ("pre_rcv_proc [INIT exit execs]",
state1_exit_exec)
{
}
FSM_PROFILE_SECTION_OUT (state1_exit_exec)

/** state (INIT) transition processing **/
```

```
FSM_TRANSIT_FORCE (2, state2_enter_exec, ;, "default", "", "INIT",
"st_12")
/*---------------------------------------------------------*/

/** state (st_12) enter executives **/
FSM_STATE_ENTER_UNFORCED (2, "st_12", state2_enter_exec,
"pre_rcv_proc [st_12 enter execs]")

FSM_PROFILE_SECTION_IN ("pre_rcv_proc [st_12 enter execs]",
state2_enter_exec)
      {
      }
      FSM_PROFILE_SECTION_OUT (state2_enter_exec)

      /** blocking after enter executives of unforced state. **/
      FSM_EXIT (5,"pre_rcv_proc")

      /** state (st_12) exit executives **/
FSM_STATE_EXIT_UNFORCED (2, "st_12", "pre_rcv_proc [st_12 exit
execs]")
FSM_PROFILE_SECTION_IN ("pre_rcv_proc [st_12 exit execs]",
state2_exit_exec)
    {
    }
FSM_PROFILE_SECTION_OUT (state2_exit_exec)

/** state (st_12) transition processing **/
FSM_PROFILE_SECTION_IN ("pre_rcv_proc [st_12 trans conditions]",
state2_trans_conds)
FSM_INIT_COND (ARRIVAL_RCV)
FSM_DFLT_COND
FSM_TEST_LOGIC ("st_12")
FSM_PROFILE_SECTION_OUT (state2_trans_conds)

FSM_TRANSIT_SWITCH
   {
FSM_CASE_TRANSIT (0, 0, state0_enter_exec, ;, "ARRIVAL_RCV", "",
"st_12", "DISCARD")
FSM_CASE_TRANSIT (1, 2, state2_enter_exec, ;, "default", "",
"st_12", "st_12")
   }
/*---------------------------------------------------------*/

  }
 FSM_EXIT (1,"pre_rcv_proc")
  }
  }
 void pre_rcv_proc_diag (OP_SIM_CONTEXT_ARG_OPT)
      {
      /* No Diagnostic Block */
      }
 void pre_rcv_proc_terminate (OP_SIM_CONTEXT_ARG_OPT)
      {
  #if !defined (VOSD_NO_FIN)
 int _op_block_origin = __LINE__;
 #endif

 FIN_MT (pre_rcv_proc_terminate ())

 Vos_Poolmem_Dealloc_MT (OP_SIM_CONTEXT_THREAD_INDEX_COMMA
pr_state_ptr);
```

```
      FOUT
      }

/* Undefine shortcuts to state variables to avoid */
/* syntax error in direct access to fields of */
/* local variable prs_ptr in pre_rcv_proc_svar function. */
#undef bits_rcvd_stathandle
#undef bitssec_rcvd_stathandle
#undef pkts_rcvd_stathandle
#undef pktssec_rcvd_stathandle
#undef ete_delay_stathandle
#undef bits_rcvd_gstathandle
#undef bitssec_rcvd_gstathandle
#undef pkts_rcvd_gstathandle
#undef pktssec_rcvd_gstathandle
#undef ete_delay_gstathandle

#undef FIN_PREAMBLE_DEC
#undef FIN_PREAMBLE_CODE

#define FIN_PREAMBLE_DEC
#define FIN_PREAMBLE_CODE

VosT_Obtype
pre_rcv_proc_init (int * init_block_ptr)
      {

#if !defined (VOSD_NO_FIN)
      int _op_block_origin = 0;
#endif
      VosT_Obtype obtype = OPC_NIL;
      FIN_MT (pre_rcv_proc_init (init_block_ptr))

      Vos_Define_Object (&obtype, "proc state vars (pre_rcv_proc)",
            sizeof (pre_rcv_proc_state), 0, 20);
      *init_block_ptr = 2;

      FRET (obtype)
      }

VosT_Address
pre_rcv_proc_alloc (VOS_THREAD_INDEX_ARG_COMMA VosT_Obtype obtype,
int init_block)
      {
#if !defined (VOSD_NO_FIN)
      int _op_block_origin = 0;
#endif
      pre_rcv_proc_state * ptr;
      FIN_MT (pre_rcv_proc_alloc (obtype))

      ptr = (pre_rcv_proc_state *)Vos_Alloc_Object_MT
(VOS_THREAD_INDEX_COMMA obtype);
      if (ptr != OPC_NIL)
            ptr->_op_current_block = init_block;
      FRET ((VosT_Address)ptr)
      }

void pre_rcv_proc_svar (void * gen_ptr, const char * var_name, void
** var_p_ptr)
      {
```

```
      pre_rcv_proc_state             *prs_ptr;

      FIN_MT (pre_rcv_proc_svar (gen_ptr, var_name, var_p_ptr))

      if (var_name == OPC_NIL)
            {
            *var_p_ptr = (void *)OPC_NIL;
            FOUT
            }
      prs_ptr = (pre_rcv_proc_state *)gen_ptr;

      if (strcmp ("bits_rcvd_stathandle" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->bits_rcvd_stathandle);
            FOUT
            }
      if (strcmp ("bitssec_rcvd_stathandle" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr-
>bitssec_rcvd_stathandle);
            FOUT
            }
      if (strcmp ("pkts_rcvd_stathandle" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->pkts_rcvd_stathandle);
            FOUT
            }
      if (strcmp ("pktssec_rcvd_stathandle" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr-
>pktssec_rcvd_stathandle);
            FOUT
            }
      if (strcmp ("ete_delay_stathandle" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->ete_delay_stathandle);
            FOUT
            }
      if (strcmp ("bits_rcvd_gstathandle" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->bits_rcvd_gstathandle);
            FOUT
            }
      if (strcmp ("bitssec_rcvd_gstathandle" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr-
>bitssec_rcvd_gstathandle);
            FOUT
            }
      if (strcmp ("pkts_rcvd_gstathandle" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->pkts_rcvd_gstathandle);
            FOUT
            }
      if (strcmp ("pktssec_rcvd_gstathandle" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr-
>pktssec_rcvd_gstathandle);
            FOUT
            }
      if (strcmp ("ete_delay_gstathandle" , var_name) == 0)
```

```
        {
        *var_p_ptr = (void *) (&prs_ptr->ete_delay_gstathandle);
        FOUT
        }
*var_p_ptr = (void *)OPC_NIL;

FOUT
}
```

APPENDIX E

Sample Code for Traffic Processor and Regulator in a PDERRM-Host

The sample source codes presented in this appendix concern traffic processing that took place in a host after traffic has been generated by the traffic generator module. There are two modes of operation—*basic operation* and *basic operation with traffic rate regulation.*

The basic operation involves traffic programming interface and network interface linkage. The second operation adds on top of the basic operation, the action of traffic rate regulation, which made use of global parameters in injecting traffic to the network.

E.1 Process Model for Traffic Processor

```
/* The Process Model for Host Traffic Processor */
/* Process model C form file: pre_src_proc.pr.c
/* Portions of this file copyright 1992-2003 by OPNET Technologies,
Inc. */

/* This variable carries the header into the object file */
const char pre_src_proc_pr_c [] = "MIL_3_Tfile_Hdr_ 100A 30A
op_runsim 7 43088517 43088517 1 initial model 0 0 none none 0 0 none
0 0 0 0 0 0 0 0 90b 2";

#include <string.h>

/* OPNET system definitions */
#include <opnet.h>

/* Header Block */

/* Define Input Packet Streams */
#define GEN_IN_STRM  0
#define RCV_IN_STRM  1

/* Output Packet Streams */
#define TO_SINK_OUT_STRM 0
#define TO_XMT_OUT_STRM  1

/* OUtput Statistic for Source Adaptive Control */
#define TO_GEN_OUT_STAT 0

/* Macros for in Packet Processing */
#define GEN_ARRVL (op_intrpt_type() == OPC_INTRPT_STRM &&
op_intrpt_strm() == GEN_IN_STRM)
```

```
#define RCV_ARRVL (op_intrpt_type() == OPC_INTRPT_STRM &&
op_intrpt_strm() == RCV_IN_STRM)

/* Global String declaration */
char *reduce_arrvl_rate;

/* End of Header Block */

/* OPNET predefined code block */
#if !defined (VOSD_NO_FIN)
#undef      BIN
#undef      BOUT
#define     BIN         FIN_LOCAL_FIELD(_op_last_line_passed) =
__LINE__ - _op_block_origin;
#define     BOUT  BIN
#define     BINIT FIN_LOCAL_FIELD(_op_last_line_passed) = 0;
_op_block_origin = __LINE__;
#else
#define     BINIT
#endif /* #if !defined (VOSD_NO_FIN) */

/* State variable definitions */
typedef struct
      {
      /* Internal state tracking for FSM */
      FSM_SYS_STATE
      /* State Variables */
      Objid                              proc_id;
      int                                dest_address;
      Objid                              parent_node;
      int                                app_qos_attr;
      } pre_src_proc_state;

#define pr_state_ptr                     ((pre_src_proc_state*)
(OP_SIM_CONTEXT_PTR->mod_state_ptr))
#define proc_id                          pr_state_ptr->proc_id
#define dest_address                     pr_state_ptr->dest_address
#define parent_node                      pr_state_ptr->parent_node
#define app_qos_attr                     pr_state_ptr->app_qos_attr

/* These macro definitions will define a local variable called    */
/* "op_sv_ptr" in each function containing a FIN statement. */
/* This variable points to the state variable data structure,     */
/* and can be used from a C debugger to display their values.     */
#undef FIN_PREAMBLE_DEC
#undef FIN_PREAMBLE_CODE
#if defined (OPD_PARALLEL)
#  define FIN_PREAMBLE_DEC    pre_src_proc_state *op_sv_ptr;
OpT_Sim_Context * tcontext_ptr;
#  define FIN_PREAMBLE_CODE    \
            if (VosS_Mt_Perform_Lock) \
                  VOS_THREAD_SPECIFIC_DATA_GET
(VosI_Globals.simi_mt_context_data_key, tcontext_ptr, SimT_Context
*); \
            else \
                  Tcontext_ptr =
VosI_Globals.simi_sequential_context_ptr; \
            op_sv_ptr = ((pre_src_proc_state *)(tcontext_ptr-
>mod_state_ptr));
#else
#  define FIN_PREAMBLE_DEC    pre_src_proc_state *op_sv_ptr;
#  define FIN_PREAMBLE_CODE   op_sv_ptr = pr_state_ptr;
#endif


/* No Function Block */

#if !defined (VOSD_NO_FIN)
enum { _op_block_origin = __LINE__ };
#endif

/* Undefine optional tracing in FIN/FOUT/FRET */
/* The FSM has its own tracing code and the other */
/* functions should not have any tracing.         */
#undef FIN_TRACING
#define FIN_TRACING

#undef FOUTRET_TRACING
#define FOUTRET_TRACING

#if defined (__cplusplus)
extern "C" {
#endif
void pre_src_proc (OP_SIM_CONTEXT_ARG_OPT);
VosT_Obtype pre_src_proc_init (int * init_block_ptr);
VosT_Address pre_src_proc_alloc (VOS_THREAD_INDEX_ARG_COMMA
VosT_Obtype, int);
void pre_src_proc_diag (OP_SIM_CONTEXT_ARG_OPT);
void pre_src_proc_terminate (OP_SIM_CONTEXT_ARG_OPT);
void pre_src_proc_svar (void *, const char *, void **);

VosT_Fun_Status Vos_Define_Object (VosT_Obtype * _op_obst_ptr, const
char * _op_name, unsigned int _op_size, unsigned int _op_init_obs,
unsigned int _op_inc_obs);
VosT_Address Vos_Alloc_Object_MT (VOS_THREAD_INDEX_ARG_COMMA
VosT_Obtype _op_ob_hndl);
VosT_Fun_Status Vos_Poolmem_Dealloc_MT (VOS_THREAD_INDEX_ARG_COMMA
VosT_Address _op_ob_ptr);

#if defined (__cplusplus)
} /* end of 'extern "C"' */
#endif

/* Process model interrupt handling procedure */

void pre_src_proc (OP_SIM_CONTEXT_ARG_OPT)
      {
#if !defined (VOSD_NO_FIN)  int _op_block_origin = 0;
#endif
FIN_MT (pre_src_proc ());
if (1)
   {
    Packet*  pkptr;

    FSM_ENTER ("pre_src_proc")
   FSM_BLOCK_SWITCH
    {
      /*---------------------------------------------------------*/
      /** state (init) enter executives **/
FSM_STATE_ENTER_FORCED_NOLABEL (0, "init", "pre_src_proc [init enter
execs]")
```

```
FSM_PROFILE_SECTION_IN ("pre_src_proc [init enter execs]",
state0_enter_exec)
 {
 proc_id = op_id_self();
 parent_node = op_topo_parent(proc_id);
 op_ima_obj_attr_get(parent_node, "App_dest_address",
&dest_address);
 op_ima_obj_attr_get(parent_node, "App_qos_attribute",
&app_qos_attr);
 }
 FSM_PROFILE_SECTION_OUT (state0_enter_exec)

/** state (init) exit executives **/
FSM_STATE_EXIT_FORCED (0, "init", "pre_src_proc [init exit execs]")
FSM_PROFILE_SECTION_IN ("pre_src_proc [init exit execs]",
state0_exit_exec)
 {
  }
  FSM_PROFILE_SECTION_OUT (state0_exit_exec)

  /** state (init) transition processing **/
  FSM_TRANSIT_FORCE (1, state1_enter_exec, ;, "default", "", "init",
"idle")
/*-----------------------------------------------------------*/

/** state (idle) enter executives **/
FSM_STATE_ENTER_UNFORCED (1, "idle", state1_enter_exec,
"pre_src_proc [idle enter execs]")
FSM_PROFILE_SECTION_IN ("pre_src_proc [idle enter execs]",
state1_enter_exec)
 {
 }
 FSM_PROFILE_SECTION_OUT (state1_enter_exec)

/** blocking after enter executives of unforced state. **/
FSM_EXIT (3,"pre_src_proc")

/** state (idle) exit executives **/
FSM_STATE_EXIT_UNFORCED (1, "idle", "pre_src_proc [idle exit
execs]")
FSM_PROFILE_SECTION_IN ("pre_src_proc [idle exit execs]",
state1_exit_exec)
 {
  }
 FSM_PROFILE_SECTION_OUT (state1_exit_exec)

/** state (idle) transition processing **/
FSM_PROFILE_SECTION_IN ("pre_src_proc [idle trans conditions]",
state1_trans_conds)
FSM_INIT_COND (RCV_ARRVL)
FSM_TEST_COND (GEN_ARRVL)
FSM_DFLT_COND
FSM_TEST_LOGIC ("idle")
FSM_PROFILE_SECTION_OUT (state1_trans_conds)

FSM_TRANSIT_SWITCH
{
FSM_CASE_TRANSIT (0, 3, state3_enter_exec, ;, "RCV_ARRVL", "",
"idle", "rcv")
FSM_CASE_TRANSIT (1, 2, state2_enter_exec, ;, "GEN_ARRVL", "",
"idle", "xmt")
FSM_CASE_TRANSIT (2, 1, state1_enter_exec, ;, "default", "", "idle",
"idle")
}
/*---------------------------------------------------------------*/

/** state (xmt) enter executives **/
FSM_STATE_ENTER_FORCED (2, "xmt", state2_enter_exec, "pre_src_proc
[xmt enter execs]")
FSM_PROFILE_SECTION_IN ("pre_src_proc [xmt enter execs]",
state2_enter_exec)
{
pkptr = op_pk_get(GEN_IN_STRM);
op_pk_nfd_set(pkptr,"dst_addr", dest_address);
op_pk_nfd_set(pkptr, "qos_num", app_qos_attr);
op_pk_send(pkptr, TO_XMT_OUT_STRM);
}
FSM_PROFILE_SECTION_OUT (state2_enter_exec)

/** state (xmt) exit executives **/
FSM_STATE_EXIT_FORCED (2, "xmt", "pre_src_proc [xmt exit execs]")
FSM_PROFILE_SECTION_IN ("pre_src_proc [xmt exit execs]",
state2_exit_exec)
{
}
FSM_PROFILE_SECTION_OUT (state2_exit_exec)

/** state (xmt) transition processing **/
FSM_TRANSIT_FORCE (1, state1_enter_exec, ;, "default", "", "xmt",
"idle")
/*---------------------------------------------------------------*/

/** state (rcv) enter executives **/
FSM_STATE_ENTER_FORCED (3, "rcv", state3_enter_exec, "pre_src_proc
[rcv enter execs]")
FSM_PROFILE_SECTION_IN ("pre_src_proc [rcv enter execs]",
state3_enter_exec)
{
 pkptr = op_pk_get(RCV_IN_STRM);
 op_pk_send(pkptr, TO_SINK_OUT_STRM);
}
FSM_PROFILE_SECTION_OUT (state3_enter_exec)

/** state (rcv) exit executives **/
FSM_STATE_EXIT_FORCED (3, "rcv", "pre_src_proc [rcv exit execs]")
FSM_PROFILE_SECTION_IN ("pre_src_proc [rcv exit execs]",
state3_exit_exec)
{
}
FSM_PROFILE_SECTION_OUT (state3_exit_exec)

/** state (rcv) transition processing **/
FSM_TRANSIT_FORCE (1, state1_enter_exec, ;, "default", "", "rcv",
"idle")
/*---------------------------------------------------------------*/
}
FSM_EXIT (0,"pre_src_proc")
}
}
void pre_src_proc_diag (OP_SIM_CONTEXT_ARG_OPT)
  {
  /* No Diagnostic Block */
  }

void pre_src_proc_terminate (OP_SIM_CONTEXT_ARG_OPT)
  {
  #if !defined (VOSD_NO_FIN)
  int _op_block_origin = __LINE__;
  #endif
  FIN_MT (pre_src_proc_terminate ())
Vos_Poolmem_Dealloc_MT (OP_SIM_CONTEXT_THREAD_INDEX_COMMA
pr_state_ptr);

 FOUT
  }
/* Undefine shortcuts to state variables to avoid */
/* syntax error in direct access to fields of */
/* local variable prs_ptr in pre_src_proc_svar function. */
#undef proc_id
#undef dest_address
#undef parent_node
#undef app_qos_attr

#undef FIN_PREAMBLE_DEC
#undef FIN_PREAMBLE_CODE

#define FIN_PREAMBLE_DEC
#define FIN_PREAMBLE_CODE

VosT_Obtype
pre_src_proc_init (int * init_block_ptr)
      {
#if !defined (VOSD_NO_FIN)  int _op_block_origin = 0;
#endif
VosT_Obtype obtype = OPC_NIL;
FIN_MT (pre_src_proc_init (init_block_ptr))
Vos_Define_Object (&obtype, "proc state vars (pre_src_proc)",
sizeof (pre_src_proc_state), 0, 20);
*init_block_ptr = 0;

FRET (obtype)
}
VosT_Address
pre_src_proc_alloc (VOS_THREAD_INDEX_ARG_COMMA VosT_Obtype obtype,
int init_block)
      {
#if !defined (VOSD_NO_FIN)
int _op_block_origin = 0;
#endif
pre_src_proc_state * ptr;
FIN_MT (pre_src_proc_alloc (obtype))

ptr = (pre_src_proc_state *)Vos_Alloc_Object_MT
(VOS_THREAD_INDEX_COMMA obtype);
if (ptr != OPC_NIL)
ptr->_op_current_block = init_block;
FRET ((VosT_Address)ptr)
}
void pre_src_proc_svar (void * gen_ptr, const char * var_name, void
** var_p_ptr)
 {
pre_src_proc_state            *prs_ptr;
```

```
FIN_MT (pre_src_proc_svar (gen_ptr, var_name, var_p_ptr))

if (var_name == OPC_NIL)
      {
      *var_p_ptr = (void *)OPC_NIL;
      FOUT
      }
prs_ptr = (pre_src_proc_state *)gen_ptr;

if (strcmp ("proc_id" , var_name) == 0)
  {
   *var_p_ptr = (void *) (&prs_ptr->proc_id);
      FOUT
    }
if (strcmp ("dest_address" , var_name) == 0)
   {
*var_p_ptr = (void *) (&prs_ptr->dest_address);
      FOUT
      }
if (strcmp ("parent_node" , var_name) == 0)
      {
      *var_p_ptr = (void *) (&prs_ptr->parent_node);
      FOUT
      }
if (strcmp ("app_qos_attr" , var_name) == 0)
      {
      *var_p_ptr = (void *) (&prs_ptr->app_qos_attr);
    FOUT
      }
  *var_p_ptr = (void *)OPC_NIL;
   FOUT
   }
```

E.2 Process Model for Traffic Processor and Regulator

```
/* The Process Model for Traffic Processor and Regulator in a Host
*/
/* Process model C form file: src_proc_add.pr.c */
/* Portions of this file copyright 1992-2003 by OPNET Technologies,
Inc. */

/* This variable carries the header into the object file */
const char src_proc_add_pr_c [] = "MIL_3_Tfile_Hdr_ 100A 30A
op_runsim 7 4308C8ED 4308C8ED 1 igbega Asiri 0 0 none none 0 0 none
0 0 0 0 0 0 0 0 90b 2";

#include <string.h>

/* OPNET system definitions */
#include <opnet.h>

/* Header Block */

/* Define Input Packet Streams */
#define GEN_IN_STRM  0
#define RCV_IN_STRM  1

/* Output Packet Streams */
#define TO_SINK_OUT_STRM 0
#define TO_XMT_OUT_STRM  1
```

```
/* OUtput Statistic for Source Adaptive Control */
/* #define TO_GEN_OUT_STAT 0 */

/* Macros for in Packet Processing */
#define GEN_ARRVL (op_intrpt_type() == OPC_INTRPT_STRM &&
op_intrpt_strm() == GEN_IN_STRM)
#define RCV_ARRVL (op_intrpt_type() == OPC_INTRPT_STRM &&
op_intrpt_strm() == RCV_IN_STRM)

/* define control constant */
#define WINDOW_SIZE 1.0

/* Intrrupt code constant */
#define RARRVL_RATE   5

/* Global declaration of arrival rate*/
  double arrvl_rate;

/* End of Header Block */

/* OPNET System predefined block */
#if !defined (VOSD_NO_FIN)
#undef      BIN
#undef      BOUT
#define     BIN         FIN_LOCAL_FIELD(_op_last_line_passed) =
__LINE__ - _op_block_origin;
#define     BOUT  BIN
#define     BINIT FIN_LOCAL_FIELD(_op_last_line_passed) = 0;
_op_block_origin = __LINE__;
#else
#define     BINIT
#endif /* #if !defined (VOSD_NO_FIN) */




/* State variable definitions */
typedef struct
      {
      /* Internal state tracking for FSM */
      FSM_SYS_STATE
      /* State Variables */
      int                                 dest_address;
      Objid                               proc_id;
      Objid                               parent_node;
      int                                 qos_numb;
      double                              resource_allocatn;
      int                                 allowed_bits;
      int                                 total_bits_rcvd;
      int                                 time_ctrl;
      double                              service_rate;
      } src_proc_add_state;

#define pr_state_ptr                      ((src_proc_add_state*)
(OP_SIM_CONTEXT_PTR->mod_state_ptr))
#define dest_address                      pr_state_ptr->dest_address
#define proc_id                           pr_state_ptr->proc_id
#define parent_node                       pr_state_ptr->parent_node
#define qos_numb                          pr_state_ptr->qos_numb
```

```
#define resource_allocatn                         pr_state_ptr-
>resource_allocatn
#define allowed_bits                              pr_state_ptr->allowed_bits
#define total_bits_rcvd                           pr_state_ptr-
>total_bits_rcvd
#define time_ctrl                                 pr_state_ptr->time_ctrl
#define service_rate                              pr_state_ptr->service_rate

/* These macro definitions will define a local variable called    */
/* "op_sv_ptr" in each function containing a FIN statement. */
/* This variable points to the state variable data structure,     */
/* and can be used from a C debugger to display their values.     */
#undef FIN_PREAMBLE_DEC
#undef FIN_PREAMBLE_CODE
#if defined (OPD_PARALLEL)
#  define FIN_PREAMBLE_DEC    src_proc_add_state *op_sv_ptr;
OpT_Sim_Context * tcontext_ptr;
#  define FIN_PREAMBLE_CODE   \
            if (VosS_Mt_Perform_Lock) \
                  VOS_THREAD_SPECIFIC_DATA_GET
(VosI_Globals.simi_mt_context_data_key, tcontext_ptr, SimT_Context
*); \
            else \
                  tcontext_ptr =
VosI_Globals.simi_sequential_context_ptr; \
            op_sv_ptr = ((src_proc_add_state *)(tcontext_ptr-
>mod_state_ptr));
#else
#  define FIN_PREAMBLE_DEC    src_proc_add_state *op_sv_ptr;
#  define FIN_PREAMBLE_CODE   op_sv_ptr = pr_state_ptr;
#endif

/* No Function Block */

#if !defined (VOSD_NO_FIN)
enum { _op_block_origin = __LINE__ };
#endif

/* Undefine optional tracing in FIN/FOUT/FRET */
/* The FSM has its own tracing code and the other */
/* functions should not have any tracing.         */
#undef FIN_TRACING
#define FIN_TRACING

#undef FOUTRET_TRACING
#define FOUTRET_TRACING

#if defined (__cplusplus)
extern "C" {
#endif
      void src_proc_add (OP_SIM_CONTEXT_ARG_OPT);
      VosT_Obtype src_proc_add_init (int * init_block_ptr);
      VosT_Address src_proc_add_alloc (VOS_THREAD_INDEX_ARG_COMMA
VosT_Obtype, int);
      void src_proc_add_diag (OP_SIM_CONTEXT_ARG_OPT);
      void src_proc_add_terminate (OP_SIM_CONTEXT_ARG_OPT);
      void src_proc_add_svar (void *, const char *, void **);

      VosT_Fun_Status Vos_Define_Object (VosT_Obtype * _op_obst_ptr,
const char * _op_name, unsigned int _op_size, unsigned int
_op_init_obs, unsigned int _op_inc_obs);
```

```
      VosT_Address Vos_Alloc_Object_MT (VOS_THREAD_INDEX_ARG_COMMA
VosT_Obtype _op_ob_hndl);
      VosT_Fun_Status Vos_Poolmem_Dealloc_MT
(VOS_THREAD_INDEX_ARG_COMMA VosT_Address _op_ob_ptr);
#if defined (__cplusplus)
} /* end of 'extern "C"' */
#endif

/* Process model interrupt handling procedure */

void src_proc_add (OP_SIM_CONTEXT_ARG_OPT)
      {

#if !defined (VOSD_NO_FIN)
int _op_block_origin = 0;
#endif
      FIN_MT (src_proc_add ());
      if (1)
            {
            Packet*   pkptr;
            int       insert_ok;
            int       pksize;
            int       intrpt_code;
            double    current_time;
            double    start_time;
            double    change_time;
            double    narrvl_rate;

            FSM_ENTER ("src_proc_add")

            FSM_BLOCK_SWITCH
                  {
      /*---------------------------------------------------------*/
                  /** state (init) enter executives **/
FSM_STATE_ENTER_FORCED_NOLABEL (0, "init", "src_proc_add [init enter
execs]")
FSM_PROFILE_SECTION_IN ("src_proc_add [init enter execs]",
state0_enter_exec)
      {
       proc_id = op_id_self();
       parent_node = op_topo_parent(proc_id);
       op_ima_obj_attr_get(proc_id, "Service_rate", &service_rate);
       op_ima_obj_attr_get(proc_id, "Resource_allocation",
       &resource_allocatn);
       op_ima_obj_attr_get(parent_node, "App_dest_address",
       &dest_address);
       op_ima_obj_attr_get(parent_node, "App_qos_numb", &qos_numb);

       time_ctrl = 0;


       allowed_bits = resource_allocatn * WINDOW_SIZE *
service_rate;

       total_bits_rcvd = 0;
       }

FSM_PROFILE_SECTION_OUT (state0_enter_exec)

/** state (init) exit executives **/
FSM_STATE_EXIT_FORCED (0, "init", "src_proc_add [init exit execs]")
```

```
FSM_PROFILE_SECTION_IN ("src_proc_add [init exit execs]",
state0_exit_exec)
      {
      }
FSM_PROFILE_SECTION_OUT (state0_exit_exec)

/** state (init) transition processing **/
FSM_TRANSIT_FORCE (1, state1_enter_exec, ;, "default", "", "init",
"idle")
/*---------------------------------------------------------*/

/** state (idle) enter executives **/
FSM_STATE_ENTER_UNFORCED (1, "idle", state1_enter_exec,
"src_proc_add [idle enter execs]")
FSM_PROFILE_SECTION_IN ("src_proc_add [idle enter execs]",
state1_enter_exec)
      {
      }
FSM_PROFILE_SECTION_OUT (state1_enter_exec)
/** blocking after enter executives of unforced state. **/
 FSM_EXIT (3,"src_proc_add")

/** state (idle) exit executives **/
FSM_STATE_EXIT_UNFORCED (1, "idle", "src_proc_add [idle exit
execs]")
FSM_PROFILE_SECTION_IN ("src_proc_add [idle exit execs]",
state1_exit_exec)
      {
      }
FSM_PROFILE_SECTION_OUT (state1_exit_exec)

/** state (idle) transition processing **/
FSM_PROFILE_SECTION_IN ("src_proc_add [idle trans conditions]",
state1_trans_conds)
      FSM_INIT_COND (GEN_ARRVL)
      FSM_TEST_COND (RCV_ARRVL)
      FSM_DFLT_COND
      FSM_TEST_LOGIC ("idle")
      FSM_PROFILE_SECTION_OUT (state1_trans_conds)

      FSM_TRANSIT_SWITCH
      {
FSM_CASE_TRANSIT (0, 2, state2_enter_exec, ;, "GEN_ARRVL", "",
"idle", "xmt")
FSM_CASE_TRANSIT (1, 3, state3_enter_exec, ;, "RCV_ARRVL", "",
"idle", "rcv")
FSM_CASE_TRANSIT (2, 1, state1_enter_exec, ;, "default", "", "idle",
"idle")
      }
/*---------------------------------------------------------*/

/** state (xmt) enter executives **/
FSM_STATE_ENTER_FORCED (2, "xmt", state2_enter_exec, "src_proc_add
[xmt enter execs]")
FSM_PROFILE_SECTION_IN ("src_proc_add [xmt enter execs]",
state2_enter_exec)
      {
       pkptr = op_pk_get(GEN_IN_STRM);
       op_pk_nfd_set(pkptr,"dst_addr", dest_address);
       op_pk_nfd_set(pkptr,"qos_num", qos_numb);
```

```
        current_time = op_sim_time();

        if (time_ctrl == 0)
        start_time = current_time;

        time_ctrl = 1;

        change_time = start_time + WINDOW_SIZE;

        if (current_time >= change_time)
           {
             time_ctrl = 0;

             total_bits_rcvd = 0;
           }

          pksize = op_pk_total_size_get(pkptr);

          total_bits_rcvd += pksize;

          if ((total_bits_rcvd <= allowed_bits) && (current_time <=
change_time))
          {
          op_pk_send(pkptr, TO_XMT_OUT_STRM);
          }
          else
          {
          /* Compute new arrival rate */
narrvl_rate = 1/((resource_allocatn * service_rate)/pksize);

/* Insert packet into queue and schedule self interrupt */
/* to remove the packet from the queue such that  */
/* packet rate to the network will be reduced */

 op_subq_pk_insert(0, pkptr, OPC_QPOS_TAIL);

 op_intrpt_schedule_self(op_sim_time() + (1.5 * arrvl_rate),
 RARRVL_RATE);
 op_pk_destroy(pkptr);
      }


      }
FSM_PROFILE_SECTION_OUT (state2_enter_exec)

/** state (xmt) exit executives **/
FSM_STATE_EXIT_FORCED (2, "xmt", "src_proc_add [xmt exit execs]")
FSM_PROFILE_SECTION_IN ("src_proc_add [xmt exit execs]",
state2_exit_exec)
      {
      }
FSM_PROFILE_SECTION_OUT (state2_exit_exec)

/** state (xmt) transition processing **/
FSM_TRANSIT_FORCE (1, state1_enter_exec, ;, "default", "", "xmt",
"idle")
/*-------------------------------------------------------------*/

/** state (rcv) enter executives **/
FSM_STATE_ENTER_FORCED (3, "rcv", state3_enter_exec, "src_proc_add
[rcv enter execs]")
```

```
FSM_PROFILE_SECTION_IN ("src_proc_add [rcv enter execs]",
state3_enter_exec)
      {
      pkptr = op_pk_get(RCV_IN_STRM);
      op_pk_send(pkptr, TO_SINK_OUT_STRM);
      }
      FSM_PROFILE_SECTION_OUT (state3_enter_exec)

      /** state (rcv) exit executives **/
      FSM_STATE_EXIT_FORCED (3, "rcv", "src_proc_add [rcv exit
      execs]")
      FSM_PROFILE_SECTION_IN ("src_proc_add [rcv exit execs]",
      state3_exit_exec)
      {
      }
      FSM_PROFILE_SECTION_OUT (state3_exit_exec)

     /** state (rcv) transition processing **/
FSM_TRANSIT_FORCE (1, state1_enter_exec, ;, "default", "", "rcv",
"idle")
/*------------------------------------------------------------*/
      }

FSM_EXIT (0,"src_proc_add")
      }
      }

void src_proc_add_diag (OP_SIM_CONTEXT_ARG_OPT)
      {
      /* No Diagnostic Block */
      }

void src_proc_add_terminate (OP_SIM_CONTEXT_ARG_OPT)
      {
#if !defined (VOSD_NO_FIN) int _op_block_origin = __LINE__;
#endif

FIN_MT (src_proc_add_terminate ())

Vos_Poolmem_Dealloc_MT (OP_SIM_CONTEXT_THREAD_INDEX_COMMA
pr_state_ptr);

      FOUT
      }
/* Undefine shortcuts to state variables to avoid */
/* syntax error in direct access to fields of */
/* local variable prs_ptr in src_proc_add_svar function. */
#undef dest_address
#undef proc_id
#undef parent_node
#undef qos_numb
#undef resource_allocatn
#undef allowed_bits
#undef total_bits_rcvd
#undef time_ctrl
#undef service_rate

#undef FIN_PREAMBLE_DEC
#undef FIN_PREAMBLE_CODE

#define FIN_PREAMBLE_DEC
```

```
#define FIN_PREAMBLE_CODE

VosT_Obtype
src_proc_add_init (int * init_block_ptr)
      {

#if !defined (VOSD_NO_FIN)
      int _op_block_origin = 0;
#endif
      VosT_Obtype obtype = OPC_NIL;
      FIN_MT (src_proc_add_init (init_block_ptr))

      Vos_Define_Object (&obtype, "proc state vars (src_proc_add)",
      sizeof (src_proc_add_state), 0, 20);
      *init_block_ptr = 0;

      FRET (obtype)
      }

VosT_Address
src_proc_add_alloc (VOS_THREAD_INDEX_ARG_COMMA VosT_Obtype obtype,
int init_block)
      {
      #if !defined (VOSD_NO_FIN) int _op_block_origin = 0;
      #endif
      src_proc_add_state * ptr;
      FIN_MT (src_proc_add_alloc (obtype))

      ptr = (src_proc_add_state *)Vos_Alloc_Object_MT
      (VOS_THREAD_INDEX_COMMA obtype);
      if (ptr != OPC_NIL)
            ptr->_op_current_block = init_block;
      FRET ((VosT_Address)ptr)
      }

void src_proc_add_svar (void * gen_ptr, const char * var_name, void
** var_p_ptr)
      {
      src_proc_add_state            *prs_ptr;

      FIN_MT (src_proc_add_svar (gen_ptr, var_name, var_p_ptr))

      if (var_name == OPC_NIL)
            {
            *var_p_ptr = (void *)OPC_NIL;
            FOUT
            }
      prs_ptr = (src_proc_add_state *)gen_ptr;

      if (strcmp ("dest_address" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->dest_address);
            FOUT
            }
      if (strcmp ("proc_id" , var_name) == 0)
            {
            *var_p_ptr = (void *) (&prs_ptr->proc_id);
            FOUT
            }
      if (strcmp ("parent_node" , var_name) == 0)
            {
```

```
      *var_p_ptr = (void *) (&prs_ptr->parent_node);
      FOUT
      }
if (strcmp ("qos_numb" , var_name) == 0)
      {
      *var_p_ptr = (void *) (&prs_ptr->qos_numb);
      FOUT
      }
if (strcmp ("resource_allocatn" , var_name) == 0)
      {
      *var_p_ptr = (void *) (&prs_ptr->resource_allocatn);
      FOUT
      }
if (strcmp ("allowed_bits" , var_name) == 0)
      {
      *var_p_ptr = (void *) (&prs_ptr->allowed_bits);
      FOUT
      }
if (strcmp ("total_bits_rcvd" , var_name) == 0)
      {
      *var_p_ptr = (void *) (&prs_ptr->total_bits_rcvd);
      FOUT
      }
if (strcmp ("time_ctrl" , var_name) == 0)
      {
      *var_p_ptr = (void *) (&prs_ptr->time_ctrl);
      FOUT
      }
if (strcmp ("service_rate" , var_name) == 0)
      {
      *var_p_ptr = (void *) (&prs_ptr->service_rate);
      FOUT
      }
*var_p_ptr = (void *)OPC_NIL;

FOUT
}
```

APPENDIX F

Sample Code for Traffic Forwarding Process in a PDERRM Router

The PDERRM *forwarding device* (router) essentially consists of three modules, which are—*forwarding module, transmitter module and receiver module*. As stated in Appendix C and D, transmitter and receiver modules are predefined by OPNET. The forwarding module could consist of a number of process models, the sample code presented here is based on the basic root process model.

Process Model for Traffic Forwarding Process

```
/* The basic Process Model for Traffic Management and Forwarder in
   a PDERRM Forwarding Device (Router)*/
/* Process model C form file: pre_fwd_proc.pr.c */
/* Portions of this file copyright 1992-2003 by OPNET Technologies,
Inc. */

/* This variable carries the header into the object file */
const char pre_fwd_proc_pr_c [] = "MIL_3_Tfile_Hdr_ 100A 30A
op_runsim 7 43088517 43088517 1 initial model 0 0 none none 0 0 none
0 0 0 0 0 0 0 0 90b 2";

#include <string.h>

/* OPNET system definitions */
#include <opnet.h>

/* Header Block */

/* Define Output Streams */
#define XMT_OUT_STRM_0     0
#define XMT_OUT_STRM_1     1
#define XMT_OUT_STRM_2     2
#define SUBNET_STRM        3

/* Define Macro for Packet Procesing */

#define EMPTY_QUEUE       (op_q_empty())
#define ARRIVAL           (op_intrpt_type() == OPC_INTRPT_STRM)
#define COMPLT_SVC        (op_intrpt_type() == OPC_INTRPT_SELF)

/* Define Resource share for each class */
#define RESOURCE_0    0.2
#define RESOURCE_1    0.3
#define RESOURCE_2    0.5


/* Define Window size for estimation of arrived packets */
#define WINDOW_SIZE   1.0
```

```
/* Define Packet state */
#define PKT_IN_SVC 0

/* Declare functions for packet arrival estimation */
static double time_rf_pt(void);
static int pkt_counter (int pkt_class);

/* End of Header Block */

/* OPNET System predefined Code Block */
#if !defined (VOSD_NO_FIN)
#undef      BIN
#undef      BOUT
#Define     BIN         FIN_LOCAL_FIELD(_op_last_line_passed) =
__LINE__ - _op_block_origin;
#define     BOUT  BIN
#define     BINIT FIN_LOCAL_FIELD(_op_last_line_passed) = 0;
_op_block_origin = __LINE__;
#else
#define     BINIT
#endif /* #if !defined (VOSD_NO_FIN) */



/* State variable definitions */
typedef struct
      {
      /* Internal state tracking for FSM */
      FSM_SYS_STATE
      /* State Variables */
      double                                  service_rate;
      Objid                                   own_id;
      int                                     nd_config;
      int                                     server_busy;
      int                                     allowed_bits0;
      int                                     allowed_bits1;
      int                                     allowed_bits2;
      int                                     t_contrl;
      int                                     pkt_in_bits_rcvd0;
      int                                     pkt_in_bits_rcvd1;
      int                                     pkt_in_bits_rcvd2;
      Objid                                   parent_nd_config;
      } pre_fwd_proc_state;

#define pr_state_ptr                          ((pre_fwd_proc_state*)
(OP_SIM_CONTEXT_PTR->mod_state_ptr))
#define service_rate                          pr_state_ptr->service_rate
#define own_id                                pr_state_ptr->own_id
#define nd_config                             pr_state_ptr->nd_config
#define server_busy                           pr_state_ptr->server_busy
#define allowed_bits0                         pr_state_ptr-
>allowed_bits0
#define allowed_bits1                         pr_state_ptr-
>allowed_bits1
#define allowed_bits2                         pr_state_ptr-
>allowed_bits2
#define t_contrl                              pr_state_ptr->t_contrl
#define pkt_in_bits_rcvd0                     pr_state_ptr-
>pkt_in_bits_rcvd0
```

```
#define pkt_in_bits_rcvd1                    pr_state_ptr-
>pkt_in_bits_rcvd1
#define pkt_in_bits_rcvd2                    pr_state_ptr-
>pkt_in_bits_rcvd2
#define parent_nd_config                     pr_state_ptr-
>parent_nd_config

/* These macro definitions will define a local variable called    */
/* "op_sv_ptr" in each function containing a FIN statement. */
/* This variable points to the state variable data structure,     */
/* and can be used from a C debugger to display their values.     */
#undef FIN_PREAMBLE_DEC
#undef FIN_PREAMBLE_CODE
#if defined (OPD_PARALLEL)
#  define FIN_PREAMBLE_DEC    pre_fwd_proc_state *op_sv_ptr;
OpT_Sim_Context * tcontext_ptr;
#  define FIN_PREAMBLE_CODE   \
            if (VosS_Mt_Perform_Lock) \
                  VOS_THREAD_SPECIFIC_DATA_GET
(VosI_Globals.simi_mt_context_data_key, tcontext_ptr, SimT_Context
*); \
            else \
                  tcontext_ptr =
VosI_Globals.simi_sequential_context_ptr; \
            op_sv_ptr = ((pre_fwd_proc_state *)(tcontext_ptr-
>mod_state_ptr));
#else
#  define FIN_PREAMBLE_DEC    pre_fwd_proc_state *op_sv_ptr;
#  define FIN_PREAMBLE_CODE   op_sv_ptr = pr_state_ptr;
#endif

/* Function Block */

#if !defined (VOSD_NO_FIN)
enum { _op_block_origin = __LINE__ };
#endif

/* The Function Block provides optional service */
/* Procedure to determine time reference for window interval */

static double time_ref_pt(void)
      {
      static int k = 0;
      double future_time;
      static double time_ref;
      double current_time0;

      FIN (time_ref_pt(void));

      current_time0 = op_sim_time();

      if (k == 0) time_ref = current_time0;

      k++;

      future_time = time_ref + WINDOW_SIZE;

      if (current_time0 <= future_time)
          { FRET (time_ref);
            }
      else {
              if (current_time0 > future_time)
                {
                 k = 0;

                 time_ref = 0;

                 FRET (current_time0);
                }
           }
      }

static int pkt_counter (int cnt)
      {
       static int pkt_cnt1 = 0;
       static int pkt_cnt2 = 0;
       static int pkt_cnt3 = 0;
       double current_time;
       double time_interval;
       double start_time;

       FIN (pkt_counter ());

      start_time = time_ref_pt();

      if (cnt == 1)
            {
             pkt_cnt1 += 1;

             current_time = op_sim_time();

             time_interval = current_time - start_time;

             if (time_interval <= WINDOW_SIZE)
                   { FRET (pkt_cnt1);
                   }

             else {
                         if (time_interval > WINDOW_SIZE)
                           { pkt_cnt1 = 1;
                             FRET (pkt_cnt1);
                           }
                  }
            }

      else if (cnt == 2)
                {
                  pkt_cnt2 += 1;

                  current_time = op_sim_time();

                  time_interval = current_time - start_time;

                  if (time_interval <= WINDOW_SIZE)
                        { FRET (pkt_cnt2);
                          }

                else {
                             if (time_interval > WINDOW_SIZE)
                            { pkt_cnt2 = 1;
                              FRET (pkt_cnt2);
                        }
                    }
                  }

      else {
            if (cnt == 3)
                {
                  pkt_cnt3 += 1;

                  current_time = op_sim_time();

                  time_interval = current_time - start_time;

                  if (time_interval <= WINDOW_SIZE)
                        { FRET (pkt_cnt3);
                          }

                  else {
                            if (time_interval > WINDOW_SIZE)
                             { pkt_cnt3 = 1;
                               FRET (pkt_cnt3);
                             }
                        }

                }
              }
      }



/* End of Function Block */

/* OPNET System predefined code block */
/* Undefine optional tracing in FIN/FOUT/FRET */
/* The FSM has its own tracing code and the other */
/* functions should not have any tracing.        */
#undef FIN_TRACING
#define FIN_TRACING

#undef FOUTRET_TRACING
#define FOUTRET_TRACING

#if defined (__cplusplus)
extern "C" {
#endif
      void pre_fwd_proc (OP_SIM_CONTEXT_ARG_OPT);
      VosT_Obtype pre_fwd_proc_init (int * init_block_ptr);
      VosT_Address pre_fwd_proc_alloc (VOS_THREAD_INDEX_ARG_COMMA
VosT_Obtype, int);
      void pre_fwd_proc_diag (OP_SIM_CONTEXT_ARG_OPT);
      void pre_fwd_proc_terminate (OP_SIM_CONTEXT_ARG_OPT);
      void pre_fwd_proc_svar (void *, const char *, void **);

      VosT_Fun_Status Vos_Define_Object (VosT_Obtype * _op_obst_ptr,
const char * _op_name, unsigned int _op_size, unsigned int
_op_init_obs, unsigned int _op_inc_obs);
      VosT_Address Vos_Alloc_Object_MT (VOS_THREAD_INDEX_ARG_COMMA
VosT_Obtype _op_ob_hndl);
      VosT_Fun_Status Vos_Poolmem_Dealloc_MT
(VOS_THREAD_INDEX_ARG_COMMA VosT_Address _op_ob_ptr);
#if defined (__cplusplus)
```

```
} /* end of 'extern "C"' */
#endif

/* Process model interrupt handling procedure */

void pre_fwd_proc (OP_SIM_CONTEXT_ARG_OPT)
      {

#if !defined (VOSD_NO_FIN)
      int _op_block_origin = 0;
#endif
      FIN_MT (pre_fwd_proc ());
      if (1)
            {
            Packet*   pkt_ptr;
            int       dest_address;
            int       pkt_class;
            int       pkt_lent0;
            int       pkt_lent1;
            int       pkt_lent2;
            int       insert_ok;
            double    pkt_svc_time;
            double    current_time;
            double    start_time;
            double    change_time;

            FSM_ENTER ("pre_fwd_proc")

            FSM_BLOCK_SWITCH
                  {
/*---------------------------------------------------------*/
      /** state (init) enter executives **/
FSM_STATE_ENTER_FORCED_NOLABEL (0, "init", "pre_fwd_proc [init enter
execs]")
FSM_PROFILE_SECTION_IN ("pre_fwd_proc [init enter execs]",
state0_enter_exec)
      {
       /* Obtain queue own object identification */
       own_id = op_id_self();

      /* Get assigned value of serve processsing rate */
      op_ima_obj_attr_get(own_id, "service_rate", &service_rate);

      /* Determine the connection configuration of the parent node
*/
      parent_nd_config = op_topo_parent(own_id);
      op_ima_obj_attr_get(parent_nd_config, "Node Configuration",
     &nd_config);

     /* Allocate memory for schedule time array at the Subqueues */
     pkt_shdle_time = (double*) op_prg_mem_alloc(sizeof (double) *
     num_pkts);

      /* Initialise the server to idle state */
      server_busy = 0;

      t_contrl = 0;

      allowed_bits0 = WINDOW_SIZE * RESOURCE_0 * service_rate;

      allowed_bits1 = WINDOW_SIZE * RESOURCE_1 * service_rate;
      allowed_bits2 = WINDOW_SIZE * RESOURCE_2 * service_rate;

      pkt_in_bits_rcvd0 = 0;

      pkt_in_bits_rcvd1 = 0;

      pkt_in_bits_rcvd2 = 0;

      }
FSM_PROFILE_SECTION_OUT (state0_enter_exec)

/** state (init) exit executives **/
FSM_STATE_EXIT_FORCED (0, "init", "pre_fwd_proc [init exit execs]")
FSM_PROFILE_SECTION_IN ("pre_fwd_proc [init exit execs]",
state0_exit_exec)
     {
     }
FSM_PROFILE_SECTION_OUT (state0_exit_exec)

/** state (init) transition processing **/
FSM_PROFILE_SECTION_IN ("pre_fwd_proc [init trans conditions]",
state0_trans_conds)
      FSM_INIT_COND (ARRIVAL)
      FSM_DFLT_COND
      FSM_TEST_LOGIC ("init")
      FSM_PROFILE_SECTION_OUT (state0_trans_conds)

      FSM_TRANSIT_SWITCH
      {
FSM_CASE_TRANSIT (0, 1, state1_enter_exec, ;, "ARRIVAL", "", "init",
"arrlv")
FSM_CASE_TRANSIT (1, 4, state4_enter_exec, ;, "default", "", "init",
"idle")
      }
/*---------------------------------------------------------*/

/** state (arrlv) enter executives **/
FSM_STATE_ENTER_FORCED (1, "arrlv", state1_enter_exec, "pre_fwd_proc
[arrlv enter execs]")
FSM_PROFILE_SECTION_IN ("pre_fwd_proc [arrlv enter execs]",
state1_enter_exec)
      {
       /* Acquire the arriving packets from various input stream */
       pkt_ptr = op_pk_get(op_intrpt_strm());

       current_time = op_sim_time();

       if (t_contrl == 0)
       start_time = current_time;

       t_contrl = 1;

       change_time = start_time + WINDOW_SIZE;

       if (current_time >= change_time)
           {
             t_contrl = 0;

             pkt_in_bits_rcvd0 = 0;
             pkt_in_bits_rcvd1 = 0;
```

```
            pkt_in_bits_rcvd2 = 0;
          }

/* Get the value of packet field for QoS processing */
op_pk_nfd_get(pkt_ptr, "qos_num", &pkt_class);

/* Segregate packets for differential service */

switch (pkt_class)
     {
      case 1:

     /* Calculate the size of the packet */
     pkt_lent0 = op_pk_total_size_get(pkt_ptr);


     pkt_in_bits_rcvd0 += pkt_lent0;

    /* Calculate total number of packets allowed for this class of
     packet in each window */
    pkt_allowed0 = (int) WINDOW_SIZE * RESOURCE_0 *
    service_rate/pkt_lent0;

   /* Get estimate of packet delivered so far */
      estimate    = pkt_counter(pkt_class);

   /* Compare the estimate with allowed and process */
if ((pkt_in_bits_rcvd0 <= allowed_bits0) && (current_time <=
     change_time))
   {
   if (op_subq_pk_insert(0, pkt_ptr, OPC_QPOS_TAIL) != OPC_QINS_OK)
       {
        /* Insertion may fail due to may be full queue, destroy the
         packet */
         op_pk_destroy(pkt_ptr);

         /* Set Flag indicating insertion failed */
         insert_ok = 0;
        }
       else
        {
         /* Insertion successful */
         insert_ok = 1;
        }
     }
  else
   {
    /* If source of the packet exceed its arrival rate, destroy the
    packet */
    op_pk_destroy(pkt_ptr);
    insert_ok = 0;
    }

    break;

    case 2:

   /*Calculate the size of the packet */
    pkt_lent1 = op_pk_total_size_get(pkt_ptr);

    pkt_in_bits_rcvd1 += pkt_lent1;
   /* Calculate total number of packets allowed for this class of
    packet in each window */
    pkt_allowed1 = (int) WINDOW_SIZE * RESOURCE_1 *
    service_rate/pkt_lent1;

    /* Get estimate of packet delivered so far */
     estimate     =  pkt_counter(pkt_class);

    /* Compare the estimate with allowed and process */
  if ((pkt_in_bits_rcvd1 <= allowed_bits1) && (current_time <=
      change_time))
      {
      if (op_subq_pk_insert(1, pkt_ptr, OPC_QPOS_TAIL) !=
      OPC_QINS_OK)
      {
       /* Insertion may fail due to, may be full queue,
       destroy the packet */
       op_pk_destroy(pkt_ptr);

       /* Set Flag indicating insertion failed */
       insert_ok = 0;
      }
      else
      {
      /* Insertion successful */
      insert_ok = 1;
      }
   }

else
  {
   /* If source exceed its allowed arrival rate destroy the packet
*/
      op_pk_destroy(pkt_ptr);
      insert_ok = 0;
  }
break;

case 3:

/* Calculate the size of the packet */
pkt_lent2 = op_pk_total_size_get(pkt_ptr);

pkt_in_bits_rcvd2 += pkt_lent2;

/* Calculate total number of packets allowed
 for this class of packet in each window */

pkt_allowed2 = (int) WINDOW_SIZE * RESOURCE_2 *
service_rate/pkt_lent2;

/* Get estimate of packet delivered so far */
 estimate     =  pkt_counter(pkt_class);

/* Compare the estimate with allowed and process */
if ((pkt_in_bits_rcvd2 <= allowed_bits2) && (current_time <=
change_time))
{
 if (op_subq_pk_insert(2, pkt_ptr, OPC_QPOS_TAIL) != OPC_QINS_OK)
    {
      /* Insertion may fail due to may be full queue,
         destroy the packet */
       op_pk_destroy(pkt_ptr);

       /* Set Flag indicating insertion failed */
       insert_ok = 0;
       }
       else
       {
        /* Insertion successful */
        insert_ok = 1;
       }
    }
   else
    {
 /* If source exceed its allocated arrival rate destroy the packet
*/
    op_pk_destroy(pkt_ptr);
    insert_ok = 0;
    }

    break;
    }
}

FSM_PROFILE_SECTION_OUT (state1_enter_exec)

/** state (arrlv) exit executives **/
FSM_STATE_EXIT_FORCED (1, "arrlv", "pre_fwd_proc [arrlv exit
execs]")
FSM_PROFILE_SECTION_IN ("pre_fwd_proc [arrlv exit execs]",
state1_exit_exec)
    {
      }
FSM_PROFILE_SECTION_OUT (state1_exit_exec)

/** state (arrlv) transition processing **/
FSM_PROFILE_SECTION_IN ("pre_fwd_proc [arrlv trans conditions]",
state1_trans_conds)
FSM_INIT_COND (!server_busy && insert_ok)
        FSM_DFLT_COND
        FSM_TEST_LOGIC ("arrlv")
        FSM_PROFILE_SECTION_OUT (state1_trans_conds)

        FSM_TRANSIT_SWITCH
     {
FSM_CASE_TRANSIT (0, 2, state2_enter_exec, ;, "!server_busy &&
insert_ok", "", "arrlv", "in_svc")
FSM_CASE_TRANSIT (1, 4, state4_enter_exec, ;, "default", "",
"arrlv", "idle")
     }
/*------------------------------------------------------------*/

/** state (in_svc) enter executives **/
FSM_STATE_ENTER_FORCED (2, "in_svc", state2_enter_exec,
"pre_fwd_proc [in_svc enter execs]")
FSM_PROFILE_SECTION_IN ("pre_fwd_proc [in_svc enter execs]",
state2_enter_exec)
    {

     if (!op_subq_empty(0))
```

```
        {
         pkt_ptr = op_subq_pk_access(0, OPC_QPOS_HEAD);

         /* Obtain the size of the packet */
         pkt_lent0 = op_pk_total_size_get(pkt_ptr);

         /* Compute the service time for this packet */
         pkt_svc_time = (double) pkt_lent0/service_rate;

         /* Schedule the packet for the next available time */
         op_intrpt_schedule_self(op_sim_time() + pkt_svc_time, 0);

         /* Flag the present state of the server */
         server_busy = 1;
        }

 else if (!op_subq_empty(1))
      {
       pkt_ptr = op_subq_pk_access(1, OPC_QPOS_HEAD);

       /* Obtain the size of the packet */
       pkt_lent1 = op_pk_total_size_get(pkt_ptr);

        /* Compute the service time for this packet */
        pkt_svc_time = (double) pkt_lent1/service_rate;

        /* Stamp the packet to indicate its present state
         of service */
         op_pk_priority_set(pkt_ptr, PKT_IN_SVC); */

        /* Schedule the packet for the next available time */
        op_intrpt_schedule_self(op_sim_time() + pkt_svc_time, 1);

      /* Flag the present state of the server */
        server_busy = 1;
         }

   else if (!op_subq_empty(2))
         {
          pkt_ptr = op_subq_pk_access(2, OPC_QPOS_HEAD);

          /* Obtain the size of the packet */
          pkt_lent2 = op_pk_total_size_get(pkt_ptr);

          /* Compute the service time for this packet */
          pkt_svc_time = (double) pkt_lent2/service_rate;

          /* Stamp the packet to indicate its present state of
            service */
          op_pk_priority_set(pkt_ptr, PKT_IN_SVC);

          /* Schedule the packet for the next available time */
            op_intrpt_schedule_self(op_sim_time() + pkt_svc_time,
2);

           /* Flag the present state of the server */
           server_busy = 1;
           }

      }
FSM_PROFILE_SECTION_OUT (state2_enter_exec)
```

```
/** state (in_svc) exit executives **/
FSM_STATE_EXIT_FORCED (2, "in_svc", "pre_fwd_proc [in_svc exit
execs]")
FSM_PROFILE_SECTION_IN ("pre_fwd_proc [in_svc exit execs]",
state2_exit_exec)
      {
      }
FSM_PROFILE_SECTION_OUT (state2_exit_exec)

/** state (in_svc) transition processing **/
FSM_TRANSIT_FORCE (4, state4_enter_exec, ;, "default", "", "in_svc",
"idle")
/*-------------------------------------------------------*/

/** state (complt_svc) enter executives **/
FSM_STATE_ENTER_FORCED (3, "complt_svc", state3_enter_exec,
"pre_fwd_proc [complt_svc enter execs]")
FSM_PROFILE_SECTION_IN ("pre_fwd_proc [complt_svc enter execs]",
state3_enter_exec)
{
 if (op_intrpt_code() == 0)
     {
      pkt_ptr = op_subq_pk_remove (0, OPC_QPOS_HEAD);
      if (nd_config == 1)
     {
      /* forward the packet on stream designated as SUBNET_LINE
       causing an immediate interrupt at destination. */
      op_pk_send_forced (pkt_ptr, SUBNET_STRM);

      /* server is idle again. */
      server_busy = 0;
      }

     else
       {  op_pk_nfd_get(pkt_ptr, "dst_addr", &dest_address);
          if (dest_address == 5)
               { op_pk_send_forced (pkt_ptr, XMT_OUT_STRM_0);
                 server_busy = 0;
               }
        else if (dest_address == 6)
               { op_pk_send_forced (pkt_ptr, XMT_OUT_STRM_1);
                 server_busy = 0;
               }
       else if (dest_address == 7)
               { op_pk_send_forced (pkt_ptr, XMT_OUT_STRM_2);
                server_busy = 0;
               }
          }
      }

 else if (op_intrpt_code() == 1)
      {
        pkt_ptr = op_subq_pk_remove (1, OPC_QPOS_HEAD);
        if (nd_config == 1)
      {
       /* forward the packet on stream designated as SUBNET_LINE
         causing an immediate interrupt at destination. */
        op_pk_send_forced (pkt_ptr, SUBNET_STRM);

       /* server is idle again. */
```

```
          server_busy = 0;
         }
     else
        { op_pk_nfd_get(pkt_ptr, "dst_addr", &dest_address);
          if (dest_address == 5)
         { op_pk_send_forced (pkt_ptr, XMT_OUT_STRM_0);
           server_busy = 0;
          }
     else if (dest_address == 6)
              { op_pk_send_forced (pkt_ptr, XMT_OUT_STRM_1);
                server_busy = 0;
              }
     else if (dest_address == 7)
              { op_pk_send_forced (pkt_ptr, XMT_OUT_STRM_2);
                server_busy = 0;
              }
          }
      }
   else if (op_intrpt_code() == 2)
             {
              pkt_ptr = op_subq_pk_remove (2, OPC_QPOS_HEAD);
              if (nd_config == 1)
                  {
                   /* forward the packet on stream designated as
                     SUBNET_LINE causing an immediate interrupt at
                     destination. */
                   op_pk_send_forced (pkt_ptr, SUBNET_STRM);

                    /* server is idle again. */
                    server_busy = 0;
                   }
               else
                 {  op_pk_nfd_get(pkt_ptr, "dst_addr", &dest_address);
                    if (dest_address == 5)
             { op_pk_send_forced (pkt_ptr, XMT_OUT_STRM_0);
               server_busy = 0;
              }
         else if (dest_address == 6)
              { op_pk_send_forced (pkt_ptr, XMT_OUT_STRM_1);
                server_busy = 0;
              }
        else if (dest_address == 7)
              { op_pk_send_forced (pkt_ptr, XMT_OUT_STRM_2);
                server_busy = 0;
              }
          }
        }

     }
   FSM_PROFILE_SECTION_OUT (state3_enter_exec)

   /** state (complt_svc) exit executives **/
   FSM_STATE_EXIT_FORCED (3, "complt_svc", "pre_fwd_proc [complt_svc
   exit execs]")
   FSM_PROFILE_SECTION_IN ("pre_fwd_proc [complt_svc exit execs]",
   state3_exit_exec)
         {
         }
   FSM_PROFILE_SECTION_OUT (state3_exit_exec)

   /** state (complt_svc) transition processing **/
```

```
FSM_PROFILE_SECTION_IN ("pre_fwd_proc [complt_svc trans
conditions]", state3_trans_conds)
      FSM_INIT_COND (!EMPTY_QUEUE)
      FSM_DFLT_COND
      FSM_TEST_LOGIC ("complt_svc")
      FSM_PROFILE_SECTION_OUT (state3_trans_conds)

      FSM_TRANSIT_SWITCH
      {
FSM_CASE_TRANSIT (0, 2, state2_enter_exec, ;, "!EMPTY_QUEUE", "",
"complt_svc", "in_svc")
FSM_CASE_TRANSIT (1, 4, state4_enter_exec, ;, "default", "",
"complt_svc", "idle")
      }
/*---------------------------------------------------------*/

/** state (idle) enter executives **/
FSM_STATE_ENTER_UNFORCED (4, "idle", state4_enter_exec,
"pre_fwd_proc [idle enter execs]")
FSM_PROFILE_SECTION_IN ("pre_fwd_proc [idle enter execs]",
state4_enter_exec)
      {
      }
FSM_PROFILE_SECTION_OUT (state4_enter_exec)

/** blocking after enter executives of unforced state. **/
FSM_EXIT (9,"pre_fwd_proc")


/** state (idle) exit executives **/
FSM_STATE_EXIT_UNFORCED (4, "idle", "pre_fwd_proc [idle exit
execs]")
FSM_PROFILE_SECTION_IN ("pre_fwd_proc [idle exit execs]",
state4_exit_exec)
      {
      }
FSM_PROFILE_SECTION_OUT (state4_exit_exec)

/** state (idle) transition processing **/
FSM_PROFILE_SECTION_IN ("pre_fwd_proc [idle trans conditions]",
state4_trans_conds)
      FSM_INIT_COND (COMPLT_SVC)
      FSM_TEST_COND (ARRIVAL)
      FSM_TEST_LOGIC ("idle")
      FSM_PROFILE_SECTION_OUT (state4_trans_conds)

      FSM_TRANSIT_SWITCH
      {
FSM_CASE_TRANSIT (0, 3, state3_enter_exec, ;, "COMPLT_SVC", "",
"idle", "complt_svc")
FSM_CASE_TRANSIT (1, 1, state1_enter_exec, ;, "ARRIVAL", "", "idle",
"arrlv")
       }
/*---------------------------------------------------------*/
       }
FSM_EXIT (0,"pre_fwd_proc")
       }
     }

void pre_fwd_proc_diag (OP_SIM_CONTEXT_ARG_OPT)
      {
```

```
      /* No Diagnostic Block */
      }
void pre_fwd_proc_terminate (OP_SIM_CONTEXT_ARG_OPT)
      {

#if !defined (VOSD_NO_FIN)  int _op_block_origin = __LINE__;
#endif

FIN_MT (pre_fwd_proc_terminate ())
Vos_Poolmem_Dealloc_MT (OP_SIM_CONTEXT_THREAD_INDEX_COMMA
pr_state_ptr);
 FOUT
   }

/* Undefine shortcuts to state variables to avoid */
/* syntax error in direct access to fields of */
/* local variable prs_ptr in pre_fwd_proc_svar function. */
#undef service_rate
#undef own_id
#undef nd_config
#undef server_busy
#undef allowed_bits0
#undef allowed_bits1
#undef allowed_bits2
#undef t_contrl
#undef pkt_in_bits_rcvd0
#undef pkt_in_bits_rcvd1
#undef pkt_in_bits_rcvd2
#undef parent_nd_config

#undef FIN_PREAMBLE_DEC
#undef FIN_PREAMBLE_CODE

#define FIN_PREAMBLE_DEC
#define FIN_PREAMBLE_CODE

VosT_Obtype pre_fwd_proc_init (int * init_block_ptr)
      {

      #if !defined (VOSD_NO_FIN)
      int _op_block_origin = 0;
      #endif
      VosT_Obtype obtype = OPC_NIL;
      FIN_MT (pre_fwd_proc_init (init_block_ptr))

      Vos_Define_Object (&obtype, "proc state vars (pre_fwd_proc)",
      sizeof (pre_fwd_proc_state), 0, 20);
      *init_block_ptr = 0;

      FRET (obtype)
      }

VosT_Address
pre_fwd_proc_alloc (VOS_THREAD_INDEX_ARG_COMMA VosT_Obtype obtype,
int init_block)
      {

      #if !defined (VOSD_NO_FIN)
      int _op_block_origin = 0;
      #endif
      pre_fwd_proc_state * ptr;
	FIN_MT (pre_fwd_proc_alloc (obtype))

	ptr = (pre_fwd_proc_state *)Vos_Alloc_Object_MT
	(VOS_THREAD_INDEX_COMMA obtype);
	if (ptr != OPC_NIL)
		ptr->_op_current_block = init_block;
	FRET ((VosT_Address)ptr)
	}

void
pre_fwd_proc_svar (void * gen_ptr, const char * var_name, void **
var_p_ptr)
	{
	pre_fwd_proc_state		*prs_ptr;

	FIN_MT (pre_fwd_proc_svar (gen_ptr, var_name, var_p_ptr))

	if (var_name == OPC_NIL)
		{
		*var_p_ptr = (void *)OPC_NIL;
		FOUT
		}
	prs_ptr = (pre_fwd_proc_state *)gen_ptr;

	if (strcmp ("service_rate" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->service_rate);
		FOUT
		}
	if (strcmp ("own_id" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->own_id);
		FOUT
		}

	if (strcmp ("nd_config" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->nd_config);
		FOUT
		}
	if (strcmp ("server_busy" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->server_busy);
		FOUT
		}
	if (strcmp ("allowed_bits0" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->allowed_bits0);
		FOUT
		}
	if (strcmp ("allowed_bits1" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->allowed_bits1);
		FOUT
		}
	if (strcmp ("allowed_bits2" , var_name) == 0)
		{
		*var_p_ptr = (void *) (&prs_ptr->allowed_bits2);
		FOUT
		}
	if (strcmp ("t_contrl" , var_name) == 0)
```

```
      {
      *var_p_ptr = (void *) (&prs_ptr->t_contrl);
      FOUT
      }
if (strcmp ("pkt_in_bits_rcvd0" , var_name) == 0)
      {
      *var_p_ptr = (void *) (&prs_ptr->pkt_in_bits_rcvd0);
      FOUT
      }
if (strcmp ("pkt_in_bits_rcvd1" , var_name) == 0)
      {
      *var_p_ptr = (void *) (&prs_ptr->pkt_in_bits_rcvd1);
      FOUT
      }
if (strcmp ("pkt_in_bits_rcvd2" , var_name) == 0)
      {
      *var_p_ptr = (void *) (&prs_ptr->pkt_in_bits_rcvd2);
      FOUT
      }
if (strcmp ("parent_nd_config" , var_name) == 0)
      {
      *var_p_ptr = (void *) (&prs_ptr->parent_nd_config);
      FOUT
      }
*var_p_ptr = (void *)OPC_NIL;

FOUT
}
```