A NEW BLUEPRINT FOR NETWORK QoS

a thesis submitted to
The University of Kent at Canterbury
in the subject of computer science
for the degree
of doctor of philosophy.

By
David C. Reeve
August 2003

Abstract

In this thesis we present a new approach to the provisioning and delivery of Quality of Service in packet switched data networks. Such networks are endeavoring to support an increasing range of applications. For an application to deliver a successful outcome to its user it requires statistical guarantees of data transport performance from the network. These guarantees must be 'strong' in the sense that they hold over short timescales. With such a statistically strong approach to the planning, provisioning and delivery, it becomes possible to bridge the gap between the performance aspirations of applications (and hence their users) and the operational performance of the network.

This thesis is founded upon the following ontological basis. Finite queues have two degrees of freedom in the three parameters throughput, loss and delay; any QoS scheme must account for this fact at all its stages to be successful. All network elements have a finite capacity and, as they process the load offered to them, introduce quality degradation. They possess an 'intrinsic' quality, which can only be shared among their customers. It is the configuration and management of this sharing that delivers differential quality of service. Ultimately delivering quality of service is assuring a bound on the degradation experienced.

Using this basis we develop a methodology for creating and reasoning about statistically multiplexed data networks. In these networks the operational behavior can be made to mirror the mathematical foundations of queuing theory more closely than is commonly believed. We illustrate how the provisioning, design and implementation process can be approached in a consistent fashion, using new packet handling mechanisms which have evolved from the ontological basis.

We demonstrate the use of this approach by configuring examples of multi-service networks and demonstrating that their target properties are achieved through simulation.

Contents

List of Tables

List of Figures

Glossary

ADSL	Asynchronous Digital Subscriber Line
ARIMA	Autoregressive Integrated Moving-Average
BCMP	Baskett, Chandy, Muntz and Palacios
BGP	Border Gateway Protocol
BOOTP	Boot Protocol
CPR	Constant Packet Rate
DDoS	Distributed Denial of Service
DiffServ	Differentiated Services
DNS	Domain Name System
DRR	Deficit Round Robin
DTD	Data Type Definition
DoS	Denial of Service
FFQ	Fluid-Fair Queueing
FQ	Fair Queueing
GHC	Glasgo Haskell Compiler
GPRS	General Packet Radio Service
GPS	Generalised Processor Sharing
HTTP	Hypertext Transfer Protocol
HOL	Head-of-Line
ICMP	Internet Control Message Protocol
IETF	Internet Engineering Task Force
IMAP	Internet Message Access Protocol
IntServ	Integrated Services
IP	Internet Protocol
IPT	Inter-packet Time
ISDN	Integrated Services Digital Network
ISP	Internet Service Provider
MSN	Multi-Service Network
MSS	Maximum Segment Size

MTU	Maximum Transmission Unit
MVA	Mean Value Analysis
NBMA	Non-Broadcast Multiple Access
NNTP	Network News Transport Protocol
NTP	Network Time Protocol
OSPF	Open Shortest Path First
PGPS	Packetised Generalised Processor Sharing
QDF	Quality Degradator Function
QoS	Quality of Service
RED	Random Early Drop
RTT	Round Trip Time
RTP	Real-Time Protocol
SCED	Service Curve-based Earliest Deadline
SCFQ	Self-Clocked Fair Queueing
SDSL	Synchronous Digital Subscriber Line
SLA	Service Level Agreement
SNMP	Simple Network Management Protocol
SSN	Single Service Network
STFQ	Start-Time Fair Queueing
TCP	Transmission Control Protocol
TDM	Time Devision Multiplexing
ToS	Type of Service
UDP	Unreliable Data Protocol
VC	Virtual Clock
VoIP	Voice over IP
WFQ	Weighted Fair Queueing
WLAN	Wireless LAN
WRR	Weighted Round Robin

Acknowledgements

This thesis has been funded by the Engineering and Physical Sciences Research Council (EPSRC); thank you for your support. I would also like to thank the University of Kent, the Conseil European pour la Recherché Nucleaire (CERN) and U4EA Technologies; to all of whom I owe a debt of gratitude for supporting my work.

I would like to thank Ian Utting and Peter Linington for supervising me at the University of Kent and Gerald Tripp for his helpful comments in my panel. I would also like to thank Bob Dobinson and Brian Martin (for his words of encouragement) at CERN, and the people of U4EA Technologies; all of whom have helped, encouraged and supported me. Thanks also to Paul Thomas for patiently proof reading my work.

On a personal level I would like to thank my friends and family for supporting me throughout my thesis. Thanks to Fred Barnes for his hospitality, support and encouragement. Thanks to Jude for putting up with me on a day to day basis. Finally I would like to thank my mentor Neil Davies, who's support and encouragement on a professional and personal level has much to do with the completion of my thesis.

Dedication

This thesis is dedicated to my mother — without who's tireless support I would never have gone to university, let alone have completed this thesis.

Chapter 1

Introduction

Providing consistent and controllable quality is a prerequisite to the creation of differential services on IP networks. This chapter outlines the issues that are involved in delivering consistent quality to a number of subscribers in the presence of varying load.

1.1 The Current Internet

Today the Internet is becoming a key underpinning of the modern world. Since its conception in the early 80's the number of users has been growing steadily, and this shows no sign of slowing down. It has been estimated that the peak-to-mean ratio of demand for bandwidth is approaching thousands to one [25]; this is placing an enormous strain on the Internet's infrastructure. This is important as providers want high utilisation (as that is what defines their monetary return) and users want a service that has no bottlenecks and is reliable and predictable. The key is exploiting statistical multiplexing without loss of quality to the applications that need it.

1.1.1 Resource Sharing

The Internet can be viewed as one large communications resource that is shared between all its users. It allows any user to exchange data with any connected users. The level of service that you, the user, are likely to receive is highly variable as the Internet is based on "best-effort" service. This means that the network will attempt to deliver data packets to their destination as fast as it can and with as little loss as possible.

Resource sharing on this best-effort basis provides you with no guarantees; this means that you have no way of knowing how long your data will take to arrive at their destination, nor even the

likelihood of their eventual arrival. The level of service that you receive will be highly dependent on the load that other users are placing on the network - amongst other factors.

There are a number of projects underway [47, 46] that aim to support different levels of service on the Internet. The motivation for this is simple; applications place different requirements on the Internet, requiring different levels of service. These ideas are in stark contrast to those of best-effort, where everyone receives the same level of service.

1.1.2 Resource Provisioning

The purpose of resource provisioning is to ensure that you have enough capacity to satisfy your customers' requirements. A classic example of the application of provisioning can be found in the way the telephone system is managed. Here the telephone company attempts to make sure that they have enough trunk line capacity at any given time to satisfy the customer's demands [37, 38]. However at peak times there is a small possibility that some customers will be refused, but this value can be calculated.

On the Internet, the currency of consumption is bandwidth. It is this that backbone providers sell to Internet Service Providers (ISP), and then they sell on to end users. The problem that exists in this area is one of availability of bandwidth. Users of the same ISP contend for resources to reach the core of the ISP's backbone, and from there they must then contend for resources to reach other parts of the network. At peak times these problems become more acute. Another problem is the provision of a fixed bandwidth between two points, which may be hard to sustain at peak times.

As we have already shown telephone networks allocate to their expected peak call loads. They can do this because the peak-to-mean ratio of calls is about 5:1. However on the Internet it has been estimated as being more like 1000:1[1] [25]. Installing a thousand times more bandwidth than the average is clearly uneconomical, and prohibitively expensive. However, as the Internet is predominantly best-effort based, this is not necessarily a problem; the performance a user receives will simply degrade at peak times. The users' experience of the network will be entirely dependent on how badly this drop in performance affects them.

Backbone providers need the ability to guarantee some level of service between two points in their network. They sell this service to companies that want to implement Virtual Private Networks (VPNs), for example to an ISP that wants a guaranteed level of service across a transatlantic link. Currently bandwidth guarantees can only be provided in some circumstances; for example, where the network is underutilised. Even here the service that will be received will still be best-effort; although the bandwidth may be guaranteed, and indeed supplied, the delay and loss characteristics

[1] It is not known if this is the upper bound on the ratio. Installing even more capacity does not bring with it any feelings of confidence that there will be enough resource to go round.

may still be unacceptable and are not guaranteed. If the backbone is highly loaded then the level of service that the end user will receive will be lower than when the network is unloaded.

One approach taken today for guaranteeing quality is to directly measure the quality that a network provides. This involves injecting packets into the network and measuring their loss and delay at the edges of the network. The problem is that accumulating statistically accurate results can take a number of days[2]. Once this has been done, bandwidth can only be allocated if spare capacity has been observed. However, this approach relies on the fact that past behaviour is a good indicator of future behaviour and this may not be the case. If conditions change dramatically on the network during the measurement period, the results are unreliable.

1.1.3 Economic Factors

If the Internet is to become a commodity utility then it must find ways of supporting itself in the long term. This will involve new ways of selling and managing the use of the Internet. We will now explore some of the economic problems in more depth.

Currently bandwidth is the primary measure of capacity on the Internet. It is this that backbone provides sell to ISPs and ISPs sell to end-users. The problem comes in being able to satisfy everyone's short-term demands for bandwidth. This is especially difficult with the massive peak-to-mean traffic loads that are observed on the Internet. With the drive to provide differentiated services the problem is exacerbated.

Each user connected to the Internet purchases a given amount of bandwidth from their provider; however, they do not use all of this bandwidth all of the time. When it comes to peak times this is, of course, different and the load placed on the upstream links increases significantly. It is this increased loading that leads to the degradation of service at peak times. As the measure of quality is currently bandwidth, we cannot ever satisfy the user's demands.

A method of managing peak to mean traffic ratios is required, given the problem outlined above. Traditionally in other businesses, economic methods are used to time-shift the load. For example 7-11 electricity deals, or off-peak phone calls, use economic reward to shift the load to other times of the day where there is less demand for the resource.

Time-shifting could be used as a method of managing peak loads on the Internet. Different time-zones have a small effect, but the US tends to dominate [95]. Either monetary reward for using bandwidth at quiet times, or reducing the quality of connection during peak times, would be possible answers. However, the ability to accurately control the quality that a given subscriber receives will be key in managing this problem.

[2] This is for accurate answers; the measurement period can be lowered where the resolution is 'good enough'.

1.2 Multi-Service Networks

A Multi-Service Network (MSN) is a network designed to carry traffic from more than one application. This is in contrast to Single-Service Networks (SSN), such as the Telephone system, which can only carry, broadly speaking, one application. Although the Internet can be carried over the telephone system, we do not view this as a MSN, as it was not designed with this end in mind.

1.2.1 Why MSNs ?

Today there are a number of large networks in existence; for example, the Internet, the telephone system, the GSM telephone system, and the cable TV network. Most of these networks are able to carry each others traffic. The phone system carries phone calls natively, and can support Internet connectivity through various mechanisms, including modems and leased lines. The Internet can carry voice calls using Voice-over-IP (VoIP). Currently neither is well suited to carrying the other.

It is the author's view that the Internet is the most likely candidate for supporting a MSN. The Internet deals with all data in small units, called packets. This allows it to make decisions at a fine granularity. The phone system, on the other hand, can only deal with connections of a fixed bandwidth between two physical points. Both technologies use statistical multiplexing, one at a packet level and the other at the connection level. If the bandwidth provided by the phone system is not used then it cannot be used by another consumer[3], this represents a lost opportunity for the telephone company.

Conversely, on the Internet, this spare capacity is available to anyone else who requires it. This ability to share resources at the lowest possible level leads to higher utilisation. The difference between these two systems is that one is circuit-switched and the other, the Internet, is packet-switched. The benefit of packet-switched networks is that they are, or should be, able to gain higher rewards from statistical multiplexing; that is using the "spare" capacity of one connection to service other connections.

A correctly implemented MSN can also be more cost effective. Simply put, one network costs[4] less to install and maintain than two. The potential savings for an individual organisation are substantial, let alone the savings that could be made nationally or internationally. These cost savings are made by increased utilisation of the network infrastructure, delivering the maximum return on investment.

[3] This may not be true of an ATM core; however, the statistical multiplexing gains may not be as high.

[4] Cost as used here is an accountancy view, once the 'value' has been written off the equipment 'costs' nothing. It is this zero cost of the existing telephony infrastructure that represents the largest barrier to the converged network; telephony does not have to re-coup the capital charges as they have already been written off.

1.2.2 Flexibility and Expansion

A true MSN is capable of natively supporting a number of different applications; by definition this makes it more flexible. Underpinning this is the ability to provide differing levels of service to different applications. These can even be adjusted through the day or week to meet the needs of the users.

For example, let us consider an IP based MSN carrying predominantly VoIP, e-mail and World Wide Web (WWW) traffic. The amount of resources allocated to each of these applications can be adjusted throughout the day. In the morning most people check their e-mail, making it the most important type of traffic to carry. For the remainder of the day VoIP calls are more important, so they get a higher share. Finally at lunchtime WWW traffic is predominant. Our aim is to keep the network fully utilised at all times, while still providing the required level of service to the users.

Using this style of provisioning we can go some way towards controlling the peak-to-mean ratio of the Internet. This scenario is equally applicable to a single organisation as it is to an ISP or a backbone provider. Changing the performance of the network can be especially powerful if the user is aware of when they are likely to receive a better level of service. We may, for example, choose to lower bulk data transfers during the day, to encourage users to do such tasks at off-peak times such as overnight[5]. This is analogous to 7-11 electricity deals, as introduced previously.

Another important benefit of MSNs is their capacity for expansion. Where there are capacity problems, more can be added simply. The advantage is that this increased capacity can be used by any of the applications carried by the network, depending on the policies. In contrast, adding more capacity to a SSN only increases availability to the application in question, and cannot be redistributed as needed later on.

1.2.3 Support for New Applications

A correctly designed MSN will be able to carry new applications, perhaps not even in existence when the network was conceived. Indeed this is one of the key ideas behind MSNs; invest in one network which you can upgrade later. This is made possible through differentiated servicing and provisioning.

There are a large number of new applications that can be provided by a MSN. These will be covered in later chapters. An example of such an application would be Video conferencing. This is currently possible on the Internet (where it is of highly variable quality) and the telephone network (where it is of high cost). As technological advances are made these applications will become more widespread.

[5]These are all non-interactive processes and can be automated, all that is required is the right software. Why don't they exist? There is currently no economic pressure to create them.

1.3 Meeting Users' Expectations

One of our main aims in creating a MSN is to meet users' expectations. By this we mean creating a consistent experience for the user over any given period. To do this we will view a MSN as a utility. Then we will explore some of the key aspects of meeting the users' expectations.

1.3.1 Utilities and Expectations

What is a utility[6]? A utility is a service upon which you rely to support your day to day business; moreover, it is something that you use and trust in your daily life. This would seem consistent with the way the Internet is evolving. Many people use the WWW today for shopping, managing their banks accounts and entertainment. Should we not therefore view the Internet as a utility?

We have already mentioned a few example uses of the Internet. How do these compare to their high-street equivalent? If you arrived at a shop or a bank during normal office hours it would be quite surprising to find them closed. Similarly you would also be quite surprised if you arrived home from work to find that the TV was not working. However when web-sites don't work we are not all that surprised, but unhappy that we cannot perform the task we set out to do.

When things don't work as we expect we are disappointed; they do not meet our expectations. In the above examples we would probably be inclined to complain, either in person or via the mail. However when it comes to the Internet we accept this as a fact of life. It seems unlikely that this trend will continue indefinitely, especially given the increased reliance of business on the Internet.

One of the first properties that utilities have is predictability. Different utilities have different time-scales over which they operate. They will give the same level of service, or a defined level, when they are operating. Here are a few examples:

Electricity, Gas and Water. We expect these services to be available 24 hours a day, 7 days a week, no matter what[7]. If they fail it is unacceptable, although we accept that from time to time there are technical problems. We also expect to receive the same level of service over the same time-scales. For example it would be unacceptable for the electricity board to deliver you 50% of the normal voltage one day; yet, we accept a 50% reduction in the speed of downloads in peak times on the Internet.

Shops and Banks. We expect shops and banks to be open in normal office hours, and it is unusual for this not to be the case. It is accepted that occasionally they do not have the goods or services

[6] Webster's Revised Unabridged Dictionary defines utility as: "The quality or state of being useful; usefulness; production of good; profitableness to some valuable end; adaptation to satisfy the desires or wants; intrinsic value."

[7] Excluding acts of god like fire, flood, drought, etc.

we require, and that next time they probably will. This is, of course, less of an inconvenience than having no electricity or water.

Pizza delivery. This is probably not what most people would expect to be a utility, students being perhaps an exception to this. When you phone for pizza delivery one of two things could happen. If you live within range you expect your pizza to arrive within about 30 minutes. If you are not in range then you will be told that delivery will not be possible. Both of these outcomes are acceptable, even if we don't receive a pizza.

1.3.2 Networks and Expectations

The first and most important requirement of an ideal Internet connection is that it be available at all times. This means both connection to an ISP and connectivity of the network as a whole. In terms of the telephone we should always be able to connect to the local exchange, as well as any exchange in the world.

The level of service that we receive should be predictable. It is likely that there will be on-peak and off-peak times. During peak times we would expect some service to be degraded, for example bulk data transfers. We may even be able to negotiate a Service Level Agreement (SLA) with our provider to meet our particular needs. In this context we take predictable to mean the users' experience will be the same as it was at a similar point in time.

Managing users' expectations is an important aspect of delivering a service to the user. Users become accustomed to levels of service, and will expect these levels of service to be available in the future. Take for example our pizza delivery utility. If you phoned them up last week and they took 20 minutes to deliver the pizza, you would be quite surprised if this week they took an hour to deliver. Similarly, if a user gets a high-speed down-load one week he is going to be unhappy if he is unable to receive the same the next week.

The provider can overcome this problem by explicitly managing this expectation, lowering connection speeds, or acceptance rates, to some acceptable level. This level is then maintained even if the network is capable of delivering more. This is essentially the opposite of best-effort. The levels can be changed depending on the time of day (time-shifting). For example, bulk file transfers are reduced during peak times and increased during off-peak times. The user then receives a more consistent experience.

It is important that the network make good its commitments to the user. If a particular application is expected to work correctly at a given time, then the user should generally find this to be the case. This is similar to the way we treat shops as a utility, we expect them to be open during office hours and to be capable of providing the goods we require. For example, it might be expected that

VoIP works during office hours. Conversely these applications may be throttled during other times to allow for other services. Again what we are attempting to do is provide a consistent experience to the user.

There will always be more services than we support; this means that some services are going to be unsupported. In this case they are likely to be treated as best-effort, the level of service they will receive is undefined and highly variable. As we have already stated, we need to manage users' expectations. Traffic that is best-effort is therefore likely to be limited to prevent varying levels of service. Unsupported may also mean un-subscribed; for example a user may have a connection with out the provision for VoIP, when VoIP is used it is treated as best-effort and will always get a lower level of service than a subscribed version (it may not work at all depending on the policy of the network). Essentially it is important to ensure that the user is unable to defraud the cost model in use by the network.

Cost will also be an important factor in users' expectations. Price is known to be a key factor in driving the development of broadband Internet access. When the cost is relatively high there are always a number of early adopters. For the Internet to become a ubiquitous utility a low cost, basic rate access is required. This may only provide the user with a best-effort based connection.

It is likely that some services will cost more to support that others. In this case the user should have the possibility to subscribe to additional services to meet their needs. The revenue from these services can then be used to fund the network. This is similar to the way traditional airline companies split the cost of a flight between economy, business and first class, where an economy class seat costs less to the consumer than it does to the airline to provide; however this is balanced by the higher cost of business and first class.

The prerequisite to all of these points is predictability. To meet users' expectations requires availability, price and performance to be carefully controlled. Out of these we believe that performance, in terms of quality, is the key problem.

1.3.3 The Internet and Expectations

How does the current provision of Internet services compare to those outlined above? The answer is that it compares badly, it fails to tackle adequately most of the previously made points.

Connection availability from the point of the user is highly variable, and most definitely is not always on. Modem users are accustomed to busy signals during peak Internet access times. Even ADSL and Cable Modem users find that connection availability is less than 24 hours a day, 7 days a week. If your electricity supply failed as often as you are unable to connect to the Internet you would be unhappy.

Once you are connected to your ISP there is no guarantee that you will be able to connect to all points in the network. Router flap (See section 8.2) is one well know cause of this problem. If you do connect to a remote machine the level of service that you receive will be dependant on the loading of the network, at peak times you are likely to experience degraded performance. Some applications like VoIP are likely to work poorly at best.

As the Internet is predominately best-effort based, it therefore makes no commitments regarding the level of service that you will receive. As such, it is not surprising that some applications function poorly. However at times these applications will work as expected. Unfortunately this does not manage users' expectations of the network, and it is easy to become frustrated as "it worked fine last night". What is more frustrating is that there is no way to upgrade your level of service to meet these demands at the moment. You pay a flat fee and get whatever level of service is available.

As there is no capacity to deliver differing levels of service, managing when users perform certain tasks becomes difficult. Telephone companies make heavy use of time-shifting to lower the peak-to-mean ratio on their network. They do this by offering the user an incentive to use their service at a less loaded time. A similar technique could be adopted on the Internet.

1.4 Internet Infrastructure

This section outlines the current "nature" of the Internet. Firstly we consider the physical equipment that makes up the core infrastructure of the Internet. Then we will outline the protocols used on the Internet, and their current shortcomings. Finally we will look at the applications in use on the Internet.

1.4.1 Connectivity

The Internet is composed of a set of interconnected networks. These networks are composed of switching elements, called routers, connected together with physical links. These links allow the data to be transported between the routers. These data are assembled into variable sized units called packets. Packets contain, at a minimum, information about their source and their intended destination.

Links allow packets to be transported from one switch to another. They always service packets at a fixed rate, called the line speed. When a link is fully utilised it will transfer packets as fast as it possibly can. Each link places a physical limit between the two routers it connects, and this cannot be exceeded. There is also a delay associated with this transfer, called the transmission delay, and a probability of data corruption - the loss probability. These three parameters will limit any traffic that traverses the link.

Packets arrive at the router through its input ports. The packet is then examined to determine its destination, then sent to the appropriate output port. Packets are usually buffered inside a router while a decision is being made; there exists the possibility that there will be insufficient buffer space to contain the packet, in which case it will be discarded. The speed at which a router can forward packets is restricted both by the output link speed, and the speed at which it can make its decisions. This causes packets to experience a delay when traversing the router.

Routers gain their knowledge of where to send packets by two methods. The first is static routing, where an administrator enters a directive telling the router where to send the packet. The second is dynamic routing. Here the router communicates with other routers on the network to gain knowledge of where various addresses can be found. We will cover this in more detail later.

The key points here are as follows. All data are transfered in units called packets. As packets traverses a link or a router they will experience a delay. There is a small, almost negligible, possibility that a packet will be corrupted and therefore lost when traversing a link[8]. Packets are far more likely to be lost due to lack of buffer space caused by congestion. There is a fixed rate at which packets can be sent, imposed by the link speed.

1.4.2 Protocols

The networks that come together to form the Internet have one thing in common; they all use the same protocol suite. The TCP/IP protocol suite is a de-facto standard, defined in the Internet Request for Comments (RFCs). It defines a set of layered protocols which broadly correspond to groups of layers from the OSI seven layer reference model [35].

On the Internet all machines are identified by a globally unique number called an IP address. To communicate with another machine on the Internet all you need to know is its address, as well as having an address of your own. The Internet Protocol (IP) [84] is a connection-less datagram protocol. Its main function is to provide a way of addressing a packet, so that intermediate routers can decide where to send it. IP is not usually used in isolation, as it is simply a way of addressing a packet. However, it provides a carrier for higher protocol layers to use.

There are a large number of protocols that make use of IP as a carrier. The most common of these are the Internet Control Message Protocol (ICMP) [79], the User Datagram Protocol (UDP) [78] and the Transmission Control Protocol (TCP) [83][9]. TCP provides the user with a connection orientated reliable data stream connection, whereas UDP provides the user with a datagram based unreliable connection.

[8] We do not class loss due to contention on a shared link, such as shared Ethernet segments, as corruption; we model this explicitly as part of the service facility of the queue.

[9] This is by no means an exhaustive list, please refer to the Assigned Numbers RFC [88] for a complete list.

Both TCP and UDP make use of port numbers. Port numbers are used to identify the application responsible for handling a particular connection. When establishing a connection to a remote machine (the server), the client chooses an arbitrary source port number with which to communicate (ephemeral port). The port at the remote end is usually a well know number, also defined in [88]. Thus, port numbers can be used to identify well known applications.

The Internet provides a vehicle for supporting a diverse set of applications. We will now take a brief look at some of the more common ones.

As we have already stated the Internet is comprised of a set of interconnected routers. These routers can dynamically update their routes by communicating with each other. To do this they use a number of UDP based protocols, including BGP [87], RIP [48] and OSPF [72]. Each router in the network uses one of these protocols to send information about known destinations to its peers. This is a key idea in Internet technology. By using this method no global knowledge of the network is required to route packets.

Another key part of the architecture is name resolution. The Domain Name Service (DNS) [70] allows natural names to be converted into IP addresses, thus preventing users from having to remember long strings of numbers. Both DNS and the routing protocols are part of the core Internet architecture; without them the Internet will not work correctly.

Next we have some traditional TCP based applications. These include the World Wide Web (WWW) [41], remote sessions (TELNET [81], SSH [106]), file transfer (FTP [82]) and email (SMTP [80], POP3 [73], IMAP [28]). These applications are known to work well in a best-effort environment, but can become slow at peak times.

Finally there are new applications such as voice and video conferencing and streaming. These applications generally have real time constraints that need to be met by the network. As such their performance is heavily dependant on the load of the network. Both TCP and UDP are used to support these applications, and in many cases RTP [90] is used as well. These applications are likely to become more widespread as the popularity of the Internet increases.

1.5 Thesis Aims and Road-map

This thesis was motivated by the lack of a broad and self-consistent framework for reasoning about Quality of Service on packet-switched networks. Our aims during this thesis are as follows:

- Develop fundamental quality axioms that exist on all packet-switched networks, to provide us with a basis for further work.

- Develop a framework for the planning, delivery and management of QoS enabled packet-switched networks.
- Develop an example operational model, based on our framework, which will illustrate how to meet users' requirements.

The thesis is organised as follows: In chapters one and two we outline the area of Quality of Service and look at the previous research in this area. In chapter three we examine how the expression of requirements for QoS systems change depending on your viewpoint. In chapter four we outline some of the fundamental quality constraints of packet-switched networks. In chapters five and six we document, respectively, the queueing theory and simulation system used in the thesis. In chapter seven we produce a comparison between our mathematical predictions and simulation. In chapters eight and nine we expand our work to look at differential quality systems. In chapter ten we look at the areas that this thesis omits. Finally in chapter eleven we conclude our discourse.

Chapter 2

Previous Research

2.1 Introduction

In this chapter we will cover the previous research in the area of Teletraffic engineering. Throughout we will look at this research and its ability to provide us with useful information to help engineer an effective scheme for Quality of Service support.

Any scheme for QoS support should be simple to understand. Without this simplicity wide scale adoption of any QoS scheme becomes unlikely. Ultimately it is wide scale adoption that will determine if a QoS scheme is successful.

Given the vast body of literature we provide a simple classification of where this research sits. To achieve this we use the notion of timescale of operation; figure 2.1 shows this graphically. At different time scales different areas of research become relevant. This classification is meant to be a guide only, it provides us with a simple way of positioning the various research in this area.

Over the timescale of seconds quality delivery and traffic patterns are important. Quality delivery covers any packet handling mechanism, or associated theory, that makes low level decisions on a per packet basis. This, for example, includes Weighted Fair Queueing (WFQ) [75] and its variants. Also at this time scale are traffic patterns, which we take to mean distribution of inter-packet times. In general we do not consider connection arrival rates and connection durations at this level of detail; these are considered as traffic trends.

Over the timescale of minutes, traffic trends, admission control and quality signalling are important. We have already noted that traffic trends encompass information about the rate of arriving connections and their duration. Quality signalling covers protocols that are designed to convey the quality requirements of an application to the network; this includes RSVP [46] and IP ToS fields

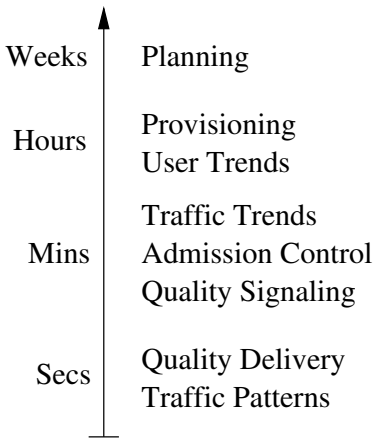


Figure 2.1: A Time Scale of Teletraffic Research

[4]. At this time-scale we also find admission control algorithms, which are responsible for making decisions about acceptance of flows with differing quality constraints.

Admission control and quality signalling are usually done in the context of a framework. The IETF have proposed two such frameworks, DiffServ [47] and IntServ [13], which use IP ToS fields and RSVP respectively to signal their requirements. IntServ works by an application asking each router along a path to reserve a given amount of capacity; if there is sufficient capacity then the router remembers the flow. DiffServ works by marking packets to place them in a predefined class of service (see section 8.6). IntServ is therefore more flexible as it allows the application to specify its requirements more accurately; however, it pays the price for having to maintain more state and therefore does not scale as well. DiffServ on the other hand places flows into pre-defined classes, but does not prescribe an admission control algorithm. It is generally accepted that DiffServ will be used in the core of the Internet with IntServ around the edges for admission control and charging, since the classes are determined on entry.

Over the time scales of hours, weeks and greater, user trends, provisioning and planning become important. User trends capture information about busy periods and peak hours; these are then used to provision and plan for the future. Most of the literature in this area concentrates on empirical measurements of network traffic, and less on detailed methodologies about provisioning or planning.

In this thesis we are mostly concerned with timescales in the seconds and milliseconds; as such our focus here is on traffic patterns and quality delivery methods. We will however make reference to other parts of the literature when necessary. In the rest of this chapter we will look predominately at this area of the literature.

2.2 Traffic Observations

In this section we will review some of the previous literature on traffic observations and modelling. There is a large body of this literature [1, 17, 5, 8, 12, 18, 23, 77, 39, 57, 20, 24, 66, 74, 76, 16, 61, 103, 85, 32, 33, 95, 71, 53] that has measured, classified, and modelled Internet traffic in a wide variety of situations. All of this literature has a common theme; measurements of real traffic have been taken, their statistical properties have been measured and models of the traffic have then been proposed. Simulations have often then been used to demonstrate the validity of the models that have been produced. We start by looking at self-similar traffic which has been observed on many places on the Internet. Next we see that we cannot use Poisson traffic models on the Internet because of the emergence of self-similar traffic. Finally we look at the implications for QoS that self-similar traffic causes.

2.2.1 Self-Similar Traffic

The main findings from the literature in this area, as first highlighted by Leland and Willinger et. al. in [61], is that traffic exhibits self-similar or fractal behaviour. In layman's terms this means that the traffic is "bursty" at all time scales and there is no natural length of a burst; in effect at any instant of time it is impossible to predict if a burst will occur, and if it does, how long that burst will last.

Self-similar traffic has an underlying dependence structure which exhibits long-range dependence (ie., hyperbolic decay of autocorrelations with increasing time separation). This is in contrast to classical traffic models, such as Poisson, which exhibit short-range dependence (ie., exponentially decaying autocorrelations). Self-similar traffic may also exhibit short-range dependence, but this is on it's own insufficient to accurately parametrise the traffic.

A common measure of self-similarity is the Hurst parameter, H. It is essentially a measure of the range divided by the sample standard deviation for a given duration; this is commonly displayed as a variance-time plot. This provides us with a measure of the long-range dependence of a stochastic process. When this parameter is between 0.5 and 1 the traffic is said to be self-similar, that is it exhibits long-range dependence. As H approaches 1 the degree of self-similarity increases. For more information about estimations of the Hurst parameter see [85]. The Hurst parameter was developed by Harold Hurst in 1965 while studying water storage.

Studying self-similar traffic requires models for analytical work and generators for simulation. Having generating algorithms that closely reflect real traffic is important as they allow us to perform simulations that are as close as possible to the real network traffic. Without this the results from simulations would not accurately reflect the results that would be expected in the real world.

There are two common families of self-similar generators: fractional Gaussian noise and fractional ARIMA processes; in the Teletraffic literature the former is more prevalent. Fractional Gaussian noise, as shown in [103], is produced when a number of on-off sources are multiplexed together. Each source is either sending traffic at a constant rate, in the on state, or sending no traffic at all, in the off state. The amount of time spent in each state is heavy tailed; commonly with the Pareto distribution, with finite mean and infinite variance, is used to model this. We shall see later, in Section 2.3, that this explanation for the emergence of self-similar traffic is highly likely.

2.2.2 Poisson Approximations

Poisson modelling has been used in the area of Teletraffic engineering for a number of years. It provides a simple and trivially tractable model for reasoning about network traffic. However, as is apparent by the results of studies of real network traffic, it is a poor approximation. As reported in [77] Poisson models seriously underestimate the burstieness of aggregated network traffic.

Before we proceed we require a better definition of what is bursty and what is smooth. For our purposes we will follow the definition in [71]: If the variance and mean have a linear relationship then the traffic is smooth (ie. Poisson-like), if the variance and mean have a quadratic relation the traffic is bursty (e.g. like Self-similar). The consequence of this is that any analytical model based on smooth traffic will produce incorrect conclusions when used to model bursty traffic. This is why we cannot safely use Poisson based models to investigate real aggregated Internet traffic.

The heavy traffic approximation [57] tells us that waiting times for service will increase exponentially as the load on a queue approaches 100%. Additionally recent research [16], based on measurement, also shows that Internet traffic tends to Poisson when the load on a queue approaches 100%. Both of these pieces of research are appealing as they would allow us to return to simple Poisson models of networks; however, they seem to be at odds with findings of various papers on self-similar traffic.

A unified framework [53] based on Wavelet Models shows us that the behaviour of the traffic is dependent on the load it places on a queue, as well as the period of observation. The bursty nature of self-similar traffic and the smooth nature of traffic at high loads are in fact a continuum of behaviour. The major finding of this work is that traffic tends to be smooth when it is measured over a long time period or the load placed on a queue is high or low. Additionally this work shows that the burstieness is at its worst when the queue is moderately (50%) loaded; which is where most measurements of self-similar traffic have been performed[1].

As a result modelling today's Internet using Poisson traffic is inadequate. While we may be able to model congested queues using Poisson arrivals, we cannot safely model the majority of moderately loaded queues.

[1]Which is no coincidence as network upgrades are performed when nodes become overloaded. As a result measurement are likely to be performed on nodes that are moderately loaded.

2.2.3 Implications for QoS

The presence of self-similar traffic has a profound effect on our ability to provide QoS support on the Internet. The main reason for this is the lack of a simple way of predicting the behaviour of the network. We will now have a brief look at why this behaviour makes QoS support so difficult.

Packets can arrive in unpredictably long bursts; when there is insufficient buffer capacity packets are simply dropped. As these bursts are long, and unpredictably so, it is difficult to predict the loss that an individual flow will experience when multiplexed with a number of sources. The problem arises as the burst is generated by a small number of sources over a short time period, yet the congestion is created by a larger number of sources over a longer time period. Schemes such as RED [62] and ABE [50] have been proposed. RED goes some way towards making loss fair, by weighting loss on queue occupancy. However, if the queue occupancy is highly variable, RED may not perform as well as predicted. ABE attempts to assure low delay by bounding queue length, hence introducing loss; however, the amount of loss, and hence delay, depends on the arrival of packet bursts.

For queues that are moderately loaded the occupancy of the queue tends to oscillate between full and empty; this is due to the unpredictable nature of the bursts. As a result the waiting time, and its variance, will fluctuate wildly. This makes prediction of delay and jitter difficult.

We know that such predictions can be performed when the network is lightly loaded, as the mean-variance relation is linear. However, underutilisation of the network to provide QoS guarantees is not cost effective in the long run. In other circumstances guarantees can be given, but not with any great accuracy. Admission control algorithms based on this concept tend to be very conservative in their nature.

Current networks are based on work-conserving deterministic service. For every packet in a given flow we attempt to keep the loss and delay for every packet below some predefined value. This is contrasted to stochastic service, where we provide a probability of the loss and delay being below some predefined value. The latter is able to better exploit statistical multiplexing, and hence can achieve higher utilisation. The problem with deterministic service is that it is hard to provide accurate guarantees for loss and delay where the network utilisation is high. In [102] a comparison between deterministic and stochastic service for strict priority queues is presented. It highlights the sensitively of deterministic schedulers to bursts, and the effect on end-to-end delay and loss guarantees.

2.3 Application Behaviour

Understanding the behaviour of the Internet is an extremely complex task. As we have already seen some substantial effort has been expended in measuring and modelling aggregated Internet traffic. In

this section we will take a look at traffic observations of some important individual applications. As with research on aggregated Internet traffic, much of this data has been produced by measurement, modelling, simulation and finally comparison. Ultimately this means that simulation results based on parameters from previous research are only valid while the parameters accurately reflect current network behaviour, i.e. if we perform a simulation using parameters from a previous measurement the results are only valid if the parameters are still accurate.

Before looking at the studies in more detail it is important to understand the kind of information that we are likely to obtain. In general most works classify application behaviour using one or more probability distributions. As we have already mentioned the parameters of these distributions have been determined by empirical measurement. For our purposes there are three important measures that are used to capture an application's behaviour: inter-connection time, connection duration and inter-packet time. These measure respectively: how often new connections start, how long a connection lasts for, and within a connection how often packets arrive.

Most of the heuristics that we use here are derived from TCPLib [33]. This provides a set of generator functions for various types of Internet style traffic based on empirical measurement. We also refer to [77] for details about TELNET and FTP traffic and [17, 24] for HTTP traffic. While this is perhaps a small set of literature it is representative of the general findings in this area.

2.3.1 Connection Lengths

When looking at connection lengths we general find that they follow a heavy tailed distribution, such as the Pareto distribution with $0.5 \leq \alpha \leq 1$. This means that there will be a wide variation in the length of a connection, and connections that are longer will tend to be disproportionally long in relation to the mean. This is especially prevalent for HTTP traffic, where the average file size has shown to be heavy-tailed.

2.3.2 Inter-Connection Times

It is important to understand that inter-connection and user activity are different. A number of connections can be generated by a single piece of user activity. A good example of this is HTTP traffic, where a user looking at a web page will usually generate a number of connections to download the various objects in the page. Clearly such an actively would also likely generate DNS requests as well. In this section we concentrate on the aggregated effect of user activity and the inter-connection times that it generates.

When looking at inter-connection times we find that there are broadly three types of distribution: deterministic, Poisson, and heavy-tailed. Deterministic inter-connection times are generated by

applications such as NNTP which use periodic times to send information to their neighbours. Such timer based applications start at times that are not determined by user activity; however, the length of the connection may still be affected by user activity. NNTP for example may send information to its neighbours, but the amount of information it sends is dependent on how much 'news' the users have generated. Poisson distributed inter-connection times are the result of some sorts of user activity. Telnet in [77] has been shown over one hour periods to exhibit this behaviour.

Finally we come to heavy-tailed inter-connection times. Applications such as HTTP and FTP exhibit this behaviour. What we will see is a burst of activity followed by a quiet period (determined by the heavy-tail). This, in the case of HTTP, is due to the number of objects that have to be fetched for a single web-page to be displayed by a browser[2]. FTP data connections have also been shown to exhibit this behaviour.

2.3.3 Inter-Packet Times

Inter-packet times, for a single connection, are perhaps the hardest to quantify and model. The reason for this is predominately due to the inherent feedback mechanisms of protocols such as TCP. To attempt to unravel this a little we will start by looking at some empirical measurements, and then at the intended behaviour of TCP itself. In reality the behaviour of a real application is likely to lie somewhere between the two.

In [77] TELNET traffic is shown to have a heavy-tailed Pareto distribution of inter-packet times. To the eye it appears that bursts of packet arrivals occur, followed by a somewhat larger gap in activity. Intuitively this would seem to correspond to the way people use computers, that is type a clump of data followed by a variable length thinking time. The paper also shows how a Poisson distributed arrival process grossly underestimates the burstieness of this type of traffic.

TELNET is an interesting example because it disables Nagle's algorithm [94], resulting in packets being sent as soon as possible. Other applications, such as HTTP and FTP, are bulk transfer applications from the point of view of TCP; as such they follow the more general rules for TCP behaviour. TCP has been designed to attempt to use as much available capacity in the network as possible. To achieve this it clocks packets, by the use of acknowledgements, into the network at a constant rate. The rate at which it does this is intended to stabilise such that little packet loss occurs, and the buffers in the bottleneck router are fully utilised. Generally most flows on the Internet see only one congested router. TCP adapts to the capacity of this router; as a result some of the other routes may be underutilised.

[2]However this may be harder to model as of HTTP 1.1, which allows more than one object to be retrieved with a single connection.

Ignoring the slow start algorithm[3] TCP can be viewed, at least theoretically, as a constant rate packet sending process. This is perhaps a bit of a gross simplification, as the actual performance of TCP is determined by the level of packet loss that it sustains. Indeed, real measurements of TCP inter-packet times have show that its behaviour is far from deterministic [32].

Other applications such as VoIP, streaming audio and video, are more usually based on UDP. These types of applications are usually rate based senders; that is, they send packets at fixed intervals of time. Where the application has an adaptive codec this rate may change over time to accommodate changing network conditions. Again ultimately we can simplify this type of application to constant rate senders.

2.3.4 Aggregation Effects

If we use our constant rate model of Inter-packet times we find that it is not hard to see how applications such as HTTP would lead to self-similar traffic when aggregated. If TCP is a constant rate source, and there are a number of TCP connections with a long-tailed arrival and duration then the effect is fractional Gaussian noise; which is self-similar. This idea is not new, a clear speculation of this process can be found in [77]. We will use HTTP as our example for the explanation of emergence of self-similar traffic mainly because HTTP traffic is one of the dominant traffic types found on the Internet [24][4].

We know that HTTP connection start times and durations are heavy-tailed, the former being due to the number of objects per page, and the latter to the range of file sizes found on the Internet. We also assume that TCP sends data at a constant rate. As shown in [103] fractional Gaussian noise is produced when a number of on-off sources, which produce constant work while on and have Pareto state transitions, are superimposed.

2.4 TCP Behaviour

In this section we will look at some of the research relevant to TCP. We have already seen that merged TCP streams are one possible cause for the emergence of self-similar traffic. This makes the investigation of TCP important to an understanding how networks will behave. Here we will look at two areas of TCP research. The first is the modelling of TCP throughput where the network conditions, such as round trip time (RTT) and loss rates, as known. The second is models for generating TCP-like traffic which also exhibits behaviour similar to traffic found on the Internet.

[3] Which as noted in [52] is not all that slow, and lasts for only a small fraction of time. However, it is only a small fraction for connections of significant lifetime. Most connections are quite short (just a few packets), and may never get out of slow start [24].

[4] Although this may now have been overtake by peer-to-peer applications.

2.4.1 Models of TCP

Modelling TCP is important as it is used by the majority of applications on the Internet today. In this section we are interested in predicting the behaviour of TCP when the network conditions are well known. There are two aspects of TCP performance that we are interested in: throughput and latency. As TCP is a reliable transport protocol we do not measure the loss performance; however, in TCP loss in the underlying network causes retransmissions which lead to latency and reduced throughput. Loss is not the only factor the effects the throughput, the RTT also has an effect on the speed of data transmission.

TCP is a complex protocol to model; it has many states of operation which, depending on the situation, can effect its performance. In this section we do not explain the behaviour of TCP in any great detail; however, we do assume a familiarity with its operation. For detailed information about TCP see [94].

The majority of TCP models fit into two categories. The first considers long running connections, such as file transfers, where the sender always has data to send. The second considers short lived connections, such as WWW transfers, where the sender has a fixed amount of data to send. In these two cases different aspects of TCP behaviour are important. In the former the congestion avoidance mechanisms of TCP dominate. In the latter the startup three way handshake and slow start tend to dominate.

In [74] a model of TCP throughput as a function of loss rate and RTT is proposed. The sender is assumed to be saturated, and as such always has data to send. This work extends previous works to include the effect of TCP fast retransmit and timeout mechanisms. The authors model TCP's congestion avoidance behaviour in terms of a number of rounds. Each round corresponds to a window of packets to be sent back-to-back; it begins with the transmission of the first packet in the round and ends with the receipt of the first acknowledgement of a packet in this round. Lost packets, detected by duplicate ACKs or by timeout, cause the window size to be modified, thus, adjusting the throughput. This model of TCP assumes that losses are correlated, that is any packet dropped in a round causes all subsequent packets in the same round to be dropped. This assumption is motivated by the drop-tail behaviour of FIFO queues; as packets in a round are sent back-to-back if they arrive at a full FIFO queue they will be dropped in a burst. Clearly this assumption, as the authors note, will not hold when using RED [62] queues.

Within the assumptions made the methods proposed in [74] can predict the throughput of TCP. However, its accuracy is dependent on the distribution of losses, which may not always be bursty in nature. In addition this method does not take into account the actions of fast retransmit and fast recovery. In [5] another model of TCP, using stochastic processes, is presented. This paper models loss in a more generic way, assuming that the loss process is only stationary and erodic. The model requires the user to choose a loss process, for which the authors present deterministic, Poisson,

general renewal and Markovian examples. This allows TCP to be modelled in situations that have very different loss processes, such as tail-drop FIFO and RED. In addition using this method it is possible to calculate the first (average) and second moments of the throughput.

In [105] a model of TCP behaviour over a differentiated services network is presented. This model can predict the throughput of TCP when there is a maximum bandwidth allocation imposed on a TCP flow. The network is assumed to mark packets with a drop precedence, with two (in/out of contract) and three (green/amber/red) levels of marking. At each point in the network RED is used to make drop decisions, this assumptions makes the modelling of loss far easier. The results from this work make it possible to calculate the throughput of TCP given a allocated bandwidth, or expressed another way the ability to calculate the correct bandwidth allocation for a target TCP throughput. This work produced accurate results within the assumptions of the network presented. However, it was noted by the authors that the assumptions about packet loss under RED are only valid when the network is under-subscribed; where the network is over-subscribed loss occurs in bursts due to full buffers, a situation where RED cannot make drop decisions.

Finally, in [18] a model of TCP latency is presented. This model differs from the ones we have presented so far in that it is concerned with the time it takes to transfer a fixed amount of data over a TCP connection; whereas before we were concerned with the steady state throughput. This model extends [74] and incorporates the behaviour of the initial three way handshake and slow-start algorithms. This method again uses the concept of rounds when modelling TCP, where packet loss in a round causes the remaining packets in the round to be lost. When compared to a number of WWW traces this approach proved accurate in approximating the latency of the transfer.

The models that we have presented here are able to calculate the throughput or latency of TCP. These models are limited by two factors: an accurate model of the loss process, and an accurate model of the RTT. The former has been investigated in much more detail as it tends to dominate the behaviour of TCP. However, it should be noted that a highly variable end-to-end delay could also cause inaccuracies. Overall we can see that it is possible to calculate the performance of TCP over a packet switched network so long as we know the loss and delay behaviour of the network. This is a major motivation for developing networks where this behaviour is well understood (predictable).

2.4.2 TCP Traffic Generators

In this section we will take a look at some TCP traffic generators. The purpose of these generators is to produce an arrival pattern of traffic that resembles that which would be found on the Internet, so that they can be used for network simulation. The benefit of having accurate traffic models is that it is possible to measure the performance of the network under a realistic load. Where the loss and delay (or latency) are measured in such simulations it is possible to calculate the resulting performance of individual streams, using the techniques presented in the previous section.

In this area of research HTTP has received the most attention; this is not surprising since it is one of the most common applications in use on the Internet. There are other works such as TCPLib [33] which can model a variety of applications. In this section we will look at two types of HTTP model: the first models the behaviour of a single "web browser" and the second models the behaviour of an aggregated flow of HTTP traffic.

The first approach to modelling HTTP involves capturing the behaviour of an individual web browser. By measuring traces of real web traffic parameters that classify the behaviour can be extracted. These parameters include: request length, reply length, files per request, thinking time and concurrent retrievals. In [66] web traces were collected and then used to create a set of cumulative distribution lookup tables, which were then used to create a traffic generator. A similar approach was also taken by [20]; however, the traces where matched with well known distributions to produce the generator. For example, it has been shown that file sizes tend to be Pareto distributed, and thinking times Poisson distributed. The benefit of this approach is the level of parametrisation, it allow the models to be adapted as browsing patterns change. It also allows individual connections to be simulated, which is useful when exact models for HTTP behaviour is required. When aggregated these approaches resulted in traffic that exhibited self-similar behaviour. A down side of this approach is that when used to create an aggregate flow of traffic the simulator has to maintain state for each of the browsers, increasing the complexity and speed of the simulation.

In [1] a less detailed approach to modelling HTTP behaviour is taken. The concept of a user click is used (which is similar to the notion of a web-request presented in [20]) as a basis for the model. A click results in a given amount of data being transfered, these data can include a number of web pages as well as any objects (such as images) that are included in the pages. Between clicks there is a variable amount of thinking time, which is intended to capture the time taken to read the web-page and any time where the user is performing some other task. This model therefore has two parameters, the distribution of reply size and the distribution of thinking time. To produce synthetic traffic the packet traces were again used to produce a set of lookup tables used by the generator. When aggregated this model again produces traffic that exhibits self-similar properties. This approach is much less complex that the previous two, making it much easier to use for simulation purposes. However, it may suffer from inaccuracies if used to model HTTP transfers rather than the behaviour of the network under HTTP traffic.

The final approach we are going to look at involves generating HTTP traffic that is representative of a number of users. In [8] a model for generating HTTP traffic is presented. Like other models this is based on the behaviour of an individual web browser, using parameters similar to those in [66]. This model is specifically designed to generate a set of aggregated HTTP traffic that is representative of a number of users. While the model uses empirically derived distributions, for the behaviour of an individual browser, it only requires a number of users to simulate. Each user is modelled as a on-off process, which could easily be implemented using threads in a simulator.

The choice of a model for HTTP traffic depends on the level of detail required. For simulations

who's aim is to look at the performance of individual HTTP transfers models such as [66, 20] are appropriate; this is because they model individual browsers accurately. For simulations which create synthetic traffic then [8] would be a good choice. Overall, one of the key requirements of all these models is to generate self-similar traffic; although HTTP is just one application that uses TCP such models are good for generating self-similar traffic for simulation purposes.

2.5 Fluid Flow Models

There is a large body of literature based on what is called "Fluid Flow Approximations". These consider the bandwidth of network traffic, modelling packet switched networks as fluid like flows of data. Here we present two pieces of interesting research in this field. The first is Effective Bandwidth, which gives a statistical measure of the bandwidth used by a number of distributions. The second is bandwidth sharing mechanisms, such as WFQ [36, 75], which divide the available network bandwidth between a number of classes.

2.5.1 Effective Bandwidth

Effective bandwidth is a simple metric that can be used to describe the statistical properties of a wide variety of traffic sources over different time and space scales. The individual sources can, in many situations, be poorly parametrised and still be modelled using effective bandwidth. Kelly [56] provides an excellent summary of all the important findings in this area of research. Effective bandwidth attempts to give a measure of the amount of bandwidth that a given source will use over a given time period; essentially is is a parametrised average bandwidth.

Effective bandwidth is used in a number of areas, namely: network provisioning, connection admission and network charging. For provisioning and admission decisions effective bandwidths can be summed, thus allowing a decision to be made as to whether enough network capacity exists to carry the traffic. Effective bandwidth is also used in the area of network charging. Here it is used to calculate the amount of bandwidth a source will use, and then the source is charged appropriately.

The definition of effective bandwidth, α, is as follows. Let $X[0,t]$ be the amount of work that arrives from a source in the interval $[0,t]$. $E[x]$ is the expected mean of the distribution function x. Assume that $X[0,t]$ has stationary increments. The definition of effective bandwidth is therefore:

$\alpha(s,t) = \frac{1}{st} log E[e^{sX[0,t]}]$

When used for the analysis of a multiplexer that guarantees some level of QoS the parameters s, called the space parameter, and t, called the time parameter, characterise the context of the source. This includes the multiplexer resources (capacity and buffer), scheduling discipline, and QoS. The

parameter s indicates the degree of statistical multiplexing: Large values of s indicate a low degree of statistical multiplexing; such is the case when multiplexed streams with peak rates not much smaller than the link capacity. On the other hand, small values of s indicate a large degree of statistical multiplexing; such is the case when we multiplex streams with peak rates much smaller than the link capacity. When s is set to infinity this corresponds to the case of deterministic multiplexing, where is there is zero probability of buffer overflow. A more mathematical interpretation is the following: over the busy period preceding a buffer overflow the amount of work produced by a stream is exponentially tilted, with tilt parameter s.

The parameter t corresponds to the duration of a buffer busy period prior to overflow. Hence, it indicates the time scales that are important for overflow: A small value of t indicates that small time periods are responsible for buffer overflow, whereas a large value of t indicates that large time periods of congestion are responsible for buffer overflow. Furthermore, parameter t shows the minimum time granularity that traces must have in order to capture the statistical properties that affect buffer overflow.

It is possible to calculate the effective bandwidth for a large number of traffic sources, including but not limited to: periodic sources, Markovian and general on-off sources, normally distributed Gaussian sources, and Levy processes. The effective bandwidth tells us how much bandwidth is required, over a given time period for a given size, to bound the work a source will produce within a given probability. For some sources the effective bandwidth is not dependent on the time period, this covers any source that displays Markovian properties, ie. X[0,t] has independent increments. However for the majority of sources in the literature this is not the case.

An important usage of effective bandwidth is in admission control. For each source arriving at a multiplexing point we have an effective bandwidth; in addition we also have a constraint. This constraint specifies the acceptance region for which this source will receive an acceptable delay and packet loss rate. Using the total effective bandwidth and capacity of the queueing algorithm in terms of effective bandwidth it is possible to calculate if the specified constraints are satisfiable. This is then used as the basis for an admission control decision.

An interesting effect of the time dependent nature of effective bandwidths can be observed by looking at on-off sources. These sources produce work at a fixed rate during the on period, and no work during the off period. The effective bandwidth requirement increases when the time scaling parameter is less than the period of the source or the interval during which the source remains 'on' or 'off'. As the time scaling increases the effective bandwidth tends towards a mean value.

The implication of this is that choosing an appropriated value for the time scaling is important. If you use a short time scale then you are likely to over estimate the bandwidth required in the long term, yet if you use a long time scale you may underestimate the bandwidth required over short time periods (due to bursts). There is then the possibility that you will incorrectly estimate the effective

bandwidth of a on-off source; clearly this has implications for admission control mechanisms. This is of course unsurprising given the bursty nature of on-off sources.

These estimation errors can be removed, but only when the traffic source is not dependent on the time scaling. This restricts the arriving traffic to only those distributions that are memory-less, Poisson being the only distribution with this property. Unfortunately, as we already know, self-similar traffic is very time dependent this would essentially cause the same problems in estimation as that for on-off sources.

Effective bandwidth, as we have shown it here, can be used to perform allocation to peak. Given that we have a number of multiplexed sources we can calculate the effective bandwidth, and hence know the peak. We can then ensure that a contention point can supply enough service such that it does not become too backlogged, which would cause excessive loss and delay. This approach hinges on choosing the correct model for a source, as well as the space and time parameters used to calculate the effective bandwidth; incorrect choices lead to loss of accuracy. The problem with this approach is that choosing a correct source model is hard, as we have seen from the work on self-similar traffic. In addition choosing the correct time and space scaling is also hard, and is usually based upon measurement; this relies on historical data being a good approximation of future behaviour, and this may not necessarily be the case.

2.5.2 Bandwidth Sharing

In this section we will review some of the fundamental packet scheduling mechanisms that share the bandwidth of an outgoing link. These are used to allocate a minimum bandwidth to each flow that crosses a link, as well a providing a delay bound.

Much of this work is based on Generalised-Processor-Sharing (GPS) [75] a general form of the head-of-line processor sharing service discipline. In GPS packets are considered to be infinitely divisible, this is why this is considered a fluid-model. GPS is sometimes called Fluid Fair Queueing (FFQ) as a result. As packets can be serviced in small quanta GPS has ideal fairness and isolation properties. Each flow passing through a GPS server is allocated a minimum bandwidth, the sum of all these allocations being that of the outgoing link. If there is spare capacity then it is shared between the backlogged flows with respect to their weight. Due to the infinitely divisible nature of GPS it is impossible to implement in practice, this has given birth to a large number of queueing disciplines based upon it.

Packet-by-Packet Generalised-Processor-Sharing (PGPS) [75] and Weighted-Fair-Queueing (WFQ) [36] are the approximations of GPS scheduling; they are essentially the same but were developed independently[5]. WFQ does not assume that packets are infinitely divisible, therefore it is possible

[5]We use the term WFQ to refer to this discipline.

to implement in practice. A hypothetical GPS server is used to calculate the virtual departure time of each packet; this is the time when the tail of the packet would have departed in a GPS queue. Packets are then serviced in departure time order, leading to an approximation of GPS. The problem with WFQ is the computational and space overhead. For each packet a departure time is calculated, with respect to GPS, and a sorted list of departures is also maintained. These two combined make WFQ inefficient for real world implementation.

The problems with WFQ have lead to a large number of alternative implementations, all of which attempt to reduce the complexity of WFQ. We do not cover them in any depth here, as we are more interested in their ideal properties. A few notable alternatives are: Self-Clocked Fair Queueing (SCFQ) [43] which uses an alternative algorithm to calculate the departure time, reducing the complexity. Start-Time Fair Queueing (STFQ) [44] considers start and finish times in order to address the short term unfairness of SCFQ. Worst-case Fair Weighted Fair Queueing (WF^2Q) [10] is another approximation to GPS, however, it only considers packets that have started or possibly finished service to reduce complexity.

A simple approximation to GPS is Weighted Round Robin (WRR) [55]. WRR services packets from a flow depending on a weight. If all flows have the same weight then the server acts like a classical round robin server, flows with a higher weighting will be serviced more in a given round. The problem with WRR is that it does not take into account packet sizes, so a flow with larger packet sizes will gain a higher share of the bandwidth. Clearly this is not a problem if all the flows have the same packet size, but this is unlikely. To address this the weighting is normalised by dividing the weight by the packet size for each flow. Unfortunately this requires knowing the packet size in advance, which is unlikely. As a result WRR does not have good fairness properties, although it is simple to implement and maintain.

Deficit Round Robin (DRR) [93] attempts to address the fairness problems of WRR. To do this during a round each flow is allocated a number of tokens, usual measured in bytes or bits, per round. Depending on the weight of a flow more or less tokens are added to each flow. A packet is sent from each flow when there are sufficient tokens to send the packet at the head of its queue. Once a packet has been sent the size of the packet is subtracted from the tokens, or if the packet can not be sent the tokens accumulate for the next round. DRR has good longer term fairness properties, but may be unfair for measurement periods less than a round trip time. DRR is commonly implemented due to its simplicity and good complexity properties.

One final bandwidth sharing mechanism is Virtual Clock (VC) [107]. Instead of emulating GPS, VC emulates time devision multiplexing (TDM). The benefit of doing this is that it makes calculating the virtual departure time much simpler, although a sorted list of departure times still needs to be maintained. VC has comparable fairness properties to WFQ when fully loaded.

2.6 Network Calculi

In this section we will take a look at some of the literature on network calculi. A network calculus is essentially a method for calculating the performance of a network under a given set of conditions. It is able to yield information about the expected performance of the network. A successful calculus should be able to accurately predict the behaviour of the network, within useful bounds. In general all of the approaches that we will cover have a sound mathematical basis, from which the performance can be calculated.

2.6.1 Jackson and BCMP Networks

The first example of a network calculus, commonly referred to as 'Jackson Networks', was presented by Jackson [51] in 1957. This calculus deals with separable or product form networks, which are essentially a set of interconnected queues. It uses, as its mathematical basis, queueing theory; predominately based upon Markovian service facilities. Using this calculus it is possible to calculate the transit time of customers passing through the system. When this is applied to a packet switched network it is possible to calculate the expected delay of packets traversing the network. In Jackson Networks queues are modelled as M/M/n queues (infinite buffering), this allows the queues to be convolved as the departure traffic pattern is still Poisson. One of the restrictions is that the level of demand for each service facility must be less than it's capacity; thus insuring that the infinite buffers do not fill. As a result this calculus is unable to model loss, in fact it avoids the issue by ensuring that it cannot occur in the first place.

In [9] Baskett, Chandy, Muntz and Palacios present a generalisation of Jackson Networks called BCMP Queueing Networks. The theory behind this calculus is much the same as Jackson, but has been extended. BCMP networks can model customers that have differing service requirements; this is the beginning of a model that can support multi-service networks. They also model a number of different queueing disciplines, and not just FIFO as presented in Jackson's work. However, the restriction that demand cannot exceed capacity still remains.

2.6.2 Mean Value Analysis

In [65] a method for calculating the performance of Jackson or BCMP networks is presented. Solving product form networks has traditionally been accomplished by constructing and solving balance equations. The problem with such an approach is that it is computationally inefficient and usually produces more information that is required. To address this problem Mean Value Analysis (MVA) deals directly with the quantities that are required, such as mean transit time and queue length, while being less computationally expensive.

MVA is a method for recursively calculating the performance of a network. We know that the mean time a customer stays at a service facility equals its own mean service time plus the mean backlog on arrival. This simply allow us to calculate the delay a customer experiences over a single service facility. By adding the mean time a customer spends at each of the service facilities together it is possible to calculate the mean time spent in the network. Essentially it adds us the mean delay at each stage in the network.

There have been many [27, 26, 59, 21, 64, 63] refinements of MVA in terms of performance and accuracy. Overall it has been shown that it is possible to calculate average delays across a network using this approach. However, as is common with these approaches the network of queues is seen as lossless.

2.6.3 Min-Plus Algebra

In [31] Cruz presents the beginning of a network calculus. This work introduces the concept of a 'service curve'; a service curve is a mapping between an arrival process and a departure process. It can be used to represent the service over a single queue, or the service over a number of queues. Service curves can be convolved, allowing networks of queues to be represented as a single service curve. From a service curve it is possible to extract a bound (min or max) on the expected level of service.

In [29, 30, 2] Cruz et. al. presents a calculus using service curves. This calculus, know as the Min-Plus algebra, is able to calculate end-to-end network delays. By using service curves this calculus can yield detailed information about the probability distribution of delays. As a whole the calculus can capture a variety of network elements, including: schedulers, links and regulators (shapers). A specific scheduling algorithm, Service Curve-based Earliest Deadline First (SCED) [89], has also been proposed to be used in conjunction with this work.

The Min-Plus algebra is in some respects a refinement of the original work by Jackson - in that it can yield detailed probability distributions, without the restrictive requirement of Markovian servicing. It is more complex than the simple MVA approach that we also looked at. One aspect that the Min-Plus algebra does share in common with Jackson, BCMP and MVA is that it does not explicitly model loss. The author believes that such an admission limits both the accuracy of the results and the applicability of these methods.

2.7 A Quality-Centric Model

In this section we explore some new research into QoS, and the effects it has on current thinking. Our main focus is on the following paper [34]. This paper outlines a method for servicing different

flows of traffic with differing service requirements in a single queue. This approach has the following properties:

- Management of different classes of service in a single queue.
- Can operate in saturation, where the offered load is greater than the service capacity.
- It can provide strong statistical bounds on loss and delay of flows.
- Provides a method to allow end-to-end quality calculation.

In order to deliver these promises there are a number of assumptions. If these assumptions are met with relative confidence then it is possible to implement this approach. It is assumed that:

- Loss is mainly due to buffer overflow and not transmission corruption.
- Packets can be marked so that they can be associated with a class of service.
- Flows of packets are well behaved, i.e.. they are within a contract[6].
- The contracted maximum bandwidth of a flow is fixed for the duration of the connection.

To achieve this a Loss/Delay model of packet servicing is outlined. This is based on a fundamental property of finite queues, in that they have two degrees of freedom in three parameters. The three parameters are throughput, loss and delay. So if the throughput is fixed a relationship between loss and delay is created.

2.7.1 Two Degrees of Freedom

Current approaches to providing QoS support are based around bandwidth requirements. This includes all packet scheduling algorithms that are based on Generalised Processor Sharing (GPS) [75] such as WFQ [36]. These approaches are based on a bandwidth driven model of networks which assumes the existence of infinite buffering. It is possible to calculate waiting times (i.e. delay) for these models, but unfortunately in such models the loss rates are zero.

Some applications can tolerate loss, either by ignoring it or by retransmission, while others stall or suffer latency due to end-to-end retransmission. It is clear that sensitivity to loss is entirely application dependent, and as such it should be possible to provision for different loss rates. Approaches based on bandwidth driven models are ill suited to this. Loss is an unavoidable phenomena of packet switched networks and should therefore always be considered when modelling such networks.

[6] It is assumed that flows are policed at the edges of the network.

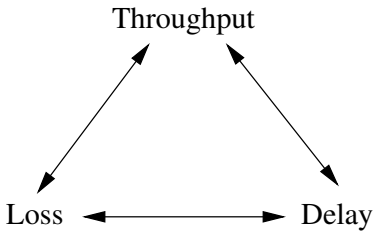


Figure 2.2: 2° of Freedom Triangle

Attempts have been made to remove loss from networks by using large buffers; this simply will not work. Under overload the buffers will always fill, as long as this situation is sustained long enough (most likely a few seconds). In this case the loss will not be avoided and extremely large delays will be introduced.

In any queueing system there is a relationship between throughput, loss and delay [34]. If there is infinite buffer capacity then loss is impossible; in such a situation the delay is determined entirely by the throughput. Where there is finite buffer capacity the following relationships exist:

1. For a fixed loss rate, reducing the throughput will cause the delay to reduce.
2. For a fixed throughput, reducing the mean delay (i.e.. number of buffers) will cause an increase in loss.
3. For a fixed delay, reducing the loss will reduce the available throughput.

Figure 2.2 shows this relationship. Any approach to providing QoS support must model these relationships; failure to do this will make it impossible to model the real behaviour of a system.

An important implication of this work is that any scheduling algorithm based on a bandwidth driven model is unlikely to be successful at managing differentiated classes of service[7], especially when the queue is pushed into overload. The major shortcoming is the inability to model loss, and hence provide predictions for the loss that would be experienced. This results in inaccurate predictions for delay, as delay is related to loss and throughput, not just throughput. In some situations, such as when the network is underutilised, this is not the case; but in such a situation loss is unlikely so the infinite buffer models are applicable.

2.7.2 The Loss-Delay Model

It is well known that applications have differing requirements for loss, delay and throughput. In [34] a model for classifying and servicing flows with differing loss and delay requirements is presented.

[7] We are assuming the absence of a feedback mechanism which would prevent overload.

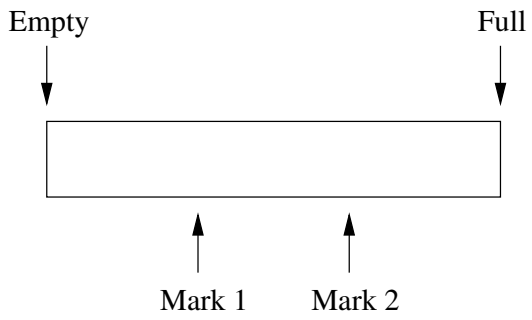


Figure 2.3: Partial Buffer Sharing

This approach allows trading within the two degrees of freedom found in finite queueing systems. As such it can address loss and delay requirements with confidence.

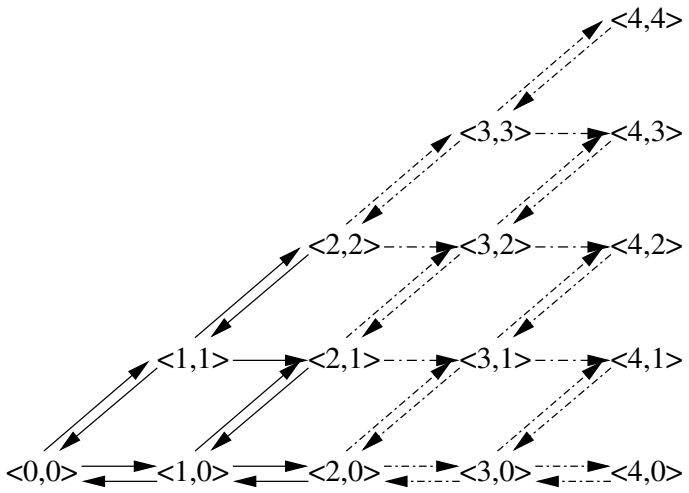
This method provides a way of controlling the two competitions that packets are subjected to in a finite queueing system. These are:

1. Competition to enter the queue. Failure to do so results in loss.
2. Competition to leave the queue. Failure to do so results in delay.

The loss-delay model classifies traffic in two dimensions: cherish and urgency. Cherish is the desire to experience less loss, so the higher the cherish level the less loss we would expect to suffer. Urgency is the desire to experience less delay, the higher the urgency the lower the delay should be. The motivation for this two dimensional classification is to capture the trading that occurs due to the two competitions in the queue.

The loss-delay model can be viewed as partial buffer sharing with strict priority. When packets arrive at the queue, they are admitted if there is free space in their portion of the buffer. Figure 2.3 shows a diagram of a buffer, which has been shared out between three classes of traffic. The first class has the lowest cherish and the third class the highest cherish. When the buffer is empty all three classes can be admitted. When the contents passes the first water mark (Mark 1) then only the second and third classes are admitted. Finally when the contents passes the second water mark (Mark 2) then only the third class will be admitted. Once a packet is in the buffer it is serviced in strict priority order, that is packets with a higher urgency will always be serviced before packets with a lower urgency.

To solve the loss-delay model a Markov chain is used. Figure 2.4 shows a Markov chain for a queue of two buffers with two levels of cherish and urgency. As we progress to the right the buffer fills, and after we have more than two items (the cherish limit) in the queue we no longer admit uncherished



Represents Poisson arrivals of not urgent traffic that includes both levels of cherised traffic

Represents Poisson arrivals of urgent traffic that includes both levels of cherished traffic

Represents Poisson arrivals of not urgent traffic that includes only cherished arrivals

Represents Poisson arrivals of urgent traffic that includes only cherished arrivals

<x,y> x represents the number of packets in the system, y is the number that are urgent packets

Figure 2.4: Loss-Delay Markov Chain

traffic. Through the methods outlined in the paper [34] it is possible to calculate the waiting time and loss rate for each class.

Additionally, we can also calculate the standard deviation in order to have a measure of the reliability of the results. It is possible, by following the derivation, to increase the number of classes to an arbitrary number if this is required.

Overall [34] presents a new type of queue which allows loss-delay trading to take place. Flows are marked with a cherish and urgency to indicate what service they would like to receive. The queue is modelled as a Markov chain, assuming Poisson inputs and service, allowing stochastic loss and delay predictions to be calculated. Given that the queue can trade in loss and delay it can provide better differentiated service than classical bandwidth based queues, which only trade in throughput leaving the relationship between loss and delay unmanaged.

There are two shortcomings of this model as presented. Firstly, it is assumed that the arriving traffic is Poisson distributed; this is unlikely in any real network. Secondly, packet lengths are not considered when modelling the servicing process or the queue admission process[8]. This means that the predictions presented may not accurately reflect that of real traffic. However, the first can be overcome by use of shaping and the second by a more complete model with bounds on the delivered service.

2.7.3 Markovian Shaping

The use of the loss-delay model, as has been presented, relies on traffic arriving with a Poisson distribution; to do this requires the traffic to be shaped. As presented in [15] shaping can be obtained using standard queues, but this itself creates problems.

A naive way of shaping is to use a queue with an extremely large buffer, serviced in a Markovian way. The service rate of the queue can then be adjusted to ensure that the queue rarely empties. As the buffers are large the possibility of loss is reduced. The problem with this method is that the delay introduced will most likely be unacceptably high.

Any shaper must attempt to keep its buffers non-empty for as much time as possible. When the queue is empty the output can be considered to be non-Markovian. An arrival after this point triggers the re-sampling of the server and the process continues. If the buffers fills then the output is still Markovian but loss will be introduced, this clearly being undesirable. Ultimately we wish the shaper to avoid either condition occurring. As you can see, any shaper has to be optimised in two dimensions. Firstly it must chose a buffer size that does not introduce huge delays or high losses. Secondly it must service the queue fast enough to prevent loss, yet slow enough not to exhaust the packets in the buffers.

[8] It is assumed that one packet occupies one buffer

One possible solution to this problem uses dummy packets to keep the server process running, and hence the output Markovian. When the server process finds the queue empty it creates a dummy packet to service, once the service for the dummy packet has completed the queue is examined again. As many dummy packets are created as is necessary to keep the server process running. While the authors of [15] do note that this is an area for future work, it does appear to be a promising solution to Markovian traffic shaping as it keeps the server process running; hence, keeping the shaped traffic as close to Markovian as possible.

2.8 Current "Best" Practise

In this section we will examine some of the current approaches to implementing QoS solutions. Here we are concerned with how people currently capture and implement QoS in their network. We show that these are far from ideal solutions to the problem.

There are, in general, four approaches to running a network today to provide QoS support. They can be used in combination for greater effect. However none of these adequately addresses all of the concerns that will be highlighted in the next chapter. In the general case it is also hard to predict what the performance of the configuration will be.

2.8.1 Requirements Capture

Requirements are generally expressed as Service Level Agreements (SLAs) between a provider and a consumer. These are used extensively on the Internet between backbone providers and Internet Service Providers (ISPs) as well as ISPs and end users. They agree how the traffic between the two entities will behave.

Unfortunately most SLAs fail to capture adequately the requirements that the user asked for. To illustrate this point we will use an example of an organisation that wishes to run VoIP between two of their sites. To do this they negotiate an SLA with an intermediate service provider that on the surface appears to satisfy their needs.

The ISP promises for each VoIP call to deliver 99.9% of packets in a month with a delay less than 100ms. The organisation also agrees to stay within a pre-determined bandwidth limit. The SLA is valid for a whole year, and the link is to be used 24 hours a day for the whole year. Both parties keep to their respective agreements. The loss and the delay are low enough for near perfect audio quality; but is this guarantee sufficient for VoIP?

Using this SLA it is possible to deliver no packets for nearly 45 minutes a month, or 2628 seconds to be precise. It would be possible for the ISP to unplug the link for 30 minutes and still maintain

their guarantee. This is clearly an extreme case and such an event would clearly be unlikely. Even if we were to evenly distribute these failure seconds, in one second bursts, throughout the working day it still may not provide acceptable quality for all applications. We investigate these issues further is section 4.6.

2.8.2 Under-utilisation

The first method of supporting applications that require QoS support is underutilisation. Here the network is not used at anywhere approaching its full capacity, in many cases less than 2% of the overall bandwidth available is utilised.

This method is generally successful, and this is not surprising. Firstly packets arriving at a switching element see an empty queue, and as such they see a small delay due to queueing. Secondly, because the queue is generally empty, the probability of loss is small. There are problems as a result of packet bursts, but this is minimised due to the general under-utilisation.

Unfortunately this is an extremely wasteful approach to providing QoS support. In some situations, such as Local Area Networks (LANs), it is acceptable, as the cost of providing excess bandwidth is not prohibitive. For the wide area Internet, where links between remote sites are expensive and it is important to utilise these links fully, it is most certainly not a cost effective solution. That is not to say that such networks do not exist. In the long-run, return on investment is important; more users will likely be added to recoup the costs of developing the network. As this happens the network will no longer be underutilised and QoS provision will become harder.

2.8.3 Measure and Improve

Another even more common solution [42] is to measure an existing network to see if it is capable of supporting a particular application. This may be done by literally injecting packets into the network and measuring the result, or by simply attempting to use a given application. Where a problem occurs, more capacity can be added; this is effectively underutilisation again.

The same source suggests building a parallel network solely for the purpose of testing the performance of the real network. This is unbelievably wasteful, and most likely extremely expensive. Again such approaches are adopted by a number of people despite this fact, but only where there is a sufficient amount of resource to justify the approach.

2.8.4 Delay Minimisation

For some applications, such as VoIP, low delay is critical to its correct functionality. These critical applications can be improved by giving them preferential treatment over other applications. Priority queueing is one such method of implementing this approach, there are many more, including [40, 99].

Unfortunately delay minimisation fails to properly manage the two degrees of freedom inherent in queues [34]. This means that the loss is hard to determine beforehand. Clearly the loss could be measured to see if it is acceptable, or the network could be designed so that it is grossly underutilised so that the problem is less apparent. While both of these approaches will work they are not ideal, especially if you wish to fully utilise your network.

A side issue with priority queueing is Denial of Service (DoS) attacks. If an attacker was to fill the highest class with bogus traffic then the lower classes would be denied service. This is especially harmful when the lower classes contain traffic such as routing information - the result being router flap. Bandwidth policing could be used to alleviate this problem; care would need to be taken not to over-police and deny legitimate traffic from the class.

2.8.5 Bandwidth Policing

The final approach to QoS provisioning is to use bandwidth policing [36, 10, 75, 93]. Here each class of traffic is assigned a rate at which its traffic is sent. The classes are then serviced in some order depending on the exact servicing discipline in use.

Managing only bandwidth fails to control loss and delay; again this does not manage the two degrees of freedom. Where the arriving traffic is bursty it becomes extremely hard to determine what the loss and delay of a class will be. One solution to this is to ensure that the arriving traffic is shaped and policed to the correct bandwidth.

2.9 Conclusions

In this chapter we have seen that traffic measured on the Internet has self-similar properties [61]. This would seem to suggest that using Poisson based models to reason about Internet performance are inadequate [77]. However, recent work [53] has shown that self-similar and Poisson-like models are in fact a continuum of behaviour. Poisson models can be used but they are restricted to limited regions (light or heavy loading).

Research into application behaviour [77, 33, 24] has shown that the majority of applications have heavy-tailed connection lengths and inter-connection times. The emergence of self-similar behaviour

can be explained [103] by multiplexing together a number of on-off sources with heavy-tailed 'on' or 'off' periods. This explanation relies on the 'on' periods producing work at a constant rate; TCP could be seen to produce such behaviour, although in practise it does not.

Self-similar traffic causes problems in providing QoS support due to its bursty nature. Bursts arrive at unpredictable intervals and last for unpredictable lengths. This affects the length of queues and as a result loss probabilities, and waiting times, become unpredictable. This lack of predictability makes it hard to make QoS guarantees with any accuracy, unless the network is underutilised.

Much of the work on QoS support has concentrated on bandwidth sharing [75, 36, 43, 44, 10, 55, 93, 107]. These scheduling disciplines attempt to emulate the behaviour of the ideal GPS discipline. Depending on the implementation gains in computational simplicity of fairness are made. GPS is based on the notion of a fluid-flow model, where packets are infinitely divisible. It is assumed that there is infinite buffer capacity, as such loss is not considered.

Effective bandwidth [56] provides a way of characterising a number of diverse sources. It can be used to calculate the amount of bandwidth that has to be delivered to a flow in order to meet its quality requirements. However, its accuracy is dependent on the correct choice of s and t parameters which is non-trivial.

Recent research [34] has shown that finite queues have two degrees of freedom. That is they can manage throughput, loss or delay by managing the other two parameters. By definition scheduling disciplines based solely on bandwidth are unable to manage the relationship between loss and delay, making them unsuitable for QoS support. To take advantage of this fact the loss-delay queueing model was developed to enable throughput to be managed by controlling loss and delay.

Chapter 3

Understanding QoS

3.1 A Perspective on QoS

We shall start our discussion by taking a high-level view of a network and examining what its stake-holders are concerned with and what they expect an ideal network to provide. Here we are mainly concerned with the networks ability to support QoS. However this is not the only consideration; networks exist to support some activity, and it is this activity that defines how the QoS is delivered. Delivering QoS is the process of assuring that the experienced quality degradation is within established bounds. Understanding how each stake-holder views the activity is key to producing a framework that addresses all of their needs.

3.1.1 The User Perspective

Networks exist to support the activities of users; ultimately it is their experience that we are trying to control. A successful network will manage the network to meet the users expectations.

In general, users are not interested in how a network functions, so long as it allows them to accomplish the task in hand. This means that they will express their requirements in terms of the activity that they are performing. We can understand a great deal about the aspirations of users by finding out what they do not like about the performance of current networks. For example:

- Web-pages take so long to load here; they do not at home.
- File downloads seem to progress at unpredictable rates.
- I have tried to use voice conferencing but the quality is dreadful.

- It takes a long time to connect to my mail server.
- Parts of the Internet appear to be disconnected at times.

All of these statements point us to some aspirations the user has about how the network should behave. It is easy to convert these into high-level requirements that the network should honour. What is important is that if we were able to meet these aspirations, at acceptable cost, the user would see the network as a success. It would be even better if the user was simply unaware that it could behave in any other way.

Making users unaware of the differing behaviour of the network is an extremely important point. Taking the file download as an example; if the user had never experienced fantastically fast downloads, do you think they would be unhappy with slower ones? The answer is probably no; [92] provides some interesting insights into this area. This is what it means to manage users' expectations. We could choose to slow down fast downloads so that users are unaware that fast downloads are possible. Or even better make downloads fast at defined times of the day that are well known to the user.

We can rewrite the user's complaints into requirements in language they can understand. By doing this the user can easily evaluate the performance of the network, and even participate in setting the requirements. Taking the above bullet points we can rewrite them as follows:

- Web-pages will load in under 10 seconds[19].
- File downloads will be between 100-150Kbps during the working day.
- Voice conferencing will always provide acceptable audio quality
- Connections to local mail servers will succeed in under a second.

While the statements above may look like a specification, they are not. What we have captured here is the user's qualitative aspirations. For example, research has shown that users lose interest in a website if they have to wait more than 10 seconds for the page to load. So it is the aspiration of the user, or at least the inferred aspiration, that the page should load in under 10 seconds. Clearly this depends on the size of the page; however, this is taken into account when mapping the aspirations into a configuration. What we have started to do here is to convert the users qualitative statements into a quantitative specification.

The last point "Parts of the Internet appear to be disconnected at times" is a harder problem, as it requires management of the Internet as a whole. The cause of such problems is usually router flap. One cause of this is when a number of routing messages get lost in a row, usually due to over utilised links. When these messages are lost, the routes that they advertise get removed from the routing

table, and it is this that causes the disconnection of parts of the network. The author believes that this could, in principle, be solved by using QoS in the Internet core (see section 8.2).

Packet loss is not the only cause of router flap. Today the number of routes that have to be stored by core routers is expanding; older routers may not have enough memory to store this information, causing some routes to be dropped, again leading to router flap. Another cause of router flap is convergence problems with BGP, the most common of inter-domain routing protocols used on the Internet; [45] provides a good overview of these issues.

Understanding the users' perspective and capturing their requirements informally, in their language, are relatively straight forward. Next these requirements can be formalised and turned into a specification. However, this process is extremely hard to do well. We shall cover this process and provide a method to deliver the requirements in later chapters.

3.1.2 The Management Perspective

Managers oversee both the users and the administrators. They are concerned with the general operation of their organisation. In a positive way, they wish to enable the administrators to support the users, where this is consistent with the organisational needs.

Managers can examine the high-level user aspirations to see if they are important to the business. For example, they may decide that having web-pages loading in under 10 seconds is too broad, they may restrict this to only those sites that they deem important to the business. They may also decide that VoIP is critically important, and hence should be treated in preference to web-pages.

This process of refinement is important to both the users and administrators. To the users it provides a clear statement of what will and will not be supported, and to the administrators a clear set of goals to reach. It is relatively trivial to capture these requirements, and it is essential that it is done in a language that in understandable to the management and the other stake holders with a non-technical background. For example:

- VoIP is critically important to expansion. It should work reliably with acceptable audio quality during the working day.
- Websites that are critical (such as suppliers and customers pages) should load in under 10s. During lunchtime this may apply to all websites.
- Connections to mail servers should be given preference at peak times, such as morning and the end of the working day.
- File transfers are unimportant, and should be treated as such.

As you can see these aims are a clear statement of what a user of a QoS enabled network should experience, it is this which we wish to deliver. It would be desirable to have confidence in the network. The choice of technology should ensure that the network is dependable and fault tolerant. This is increasingly important, given most organisation's dependence on networking technology.

Managers are concerned with a view of the system as a whole. This means that they are more interested in the overall efficiency of the network, than they are in an individual user's experience of the network. This view affects the choice of technology; ideally the technology should also focus on overall efficiency and not the treatment of an individual flow.

The final requirement of management, who in our model are responsible for purchasing decisions, is that the network should be cost effective. To this end, it is desirable to pick a technology that allows for the maximum utilisation possible while still meeting the goals that have been set.

3.1.3 The Administrative Perspective

Administrators are responsible for the smooth running of the network from a technical perspective. Their job is to configure and run the network to meet the goals that have been set by the users and management. To perform their function they have to have access to the aspirations of their masters in a form that allows them to produce a configuration.

Networks are a collection of switching devices connected together by physical links. It is likely that this infrastructure already exists. The administrators are able to control the configuration of these devices, and have a choice of a wide variety of queueing and buffering semantics. Their problem is how to best configure their infrastructure to meet the constraints placed upon them, and where to increase capacity in future expansions.

In order for them to produce the configuration they need to capture the requirements in their own language. This is different from the language of the users; it is more concerned with measurable properties of the network. The first tool that they require is a method of translating the high level requirement into a more solid mathematical representation. This is, in fact, more difficult than it may first appear, as we will show later.

Given a solid set of requirements the next task is to produce a configuration. This again is no simple task. To produce such a configuration it is essential to understand how the choice of buffering semantics and queueing disciplines will behave in the configuration. Without this, it becomes a continual process of reconfiguration and testing. While this may lead to success in a number of situations, it is not an ideal solution.

Gathering statistics on the behaviour of the network is another common task for administrators. It can help to highlight problems in the current configuration. It would be desirable to have a method

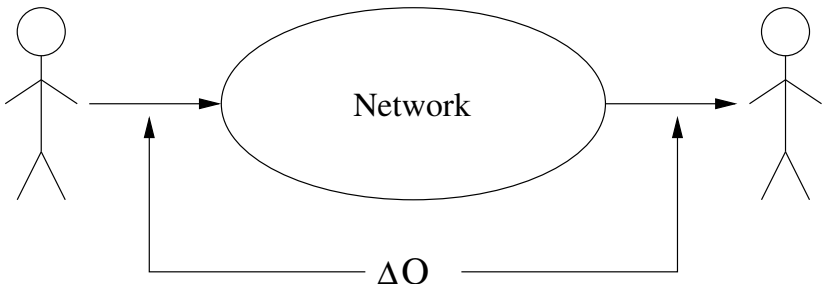


Figure 3.1: Change in Quality over a network

for the administrators to provision the usage of their network and to plan for future expansions. Again statistics gathering helps with this process; it highlights what the current usage patterns are. This can then be compared to the management aspirations to see if the configuration is successful.

However, without understanding how a change to the configuration will affect the problem, it is hard to find a solution quickly. In many instances the answer is simply to add more bandwidth, or buy improved equipment.

3.2 A Management Methodology for QoS

In this section we will examine what a QoS Management methodology should incorporate, making reference to the different perspectives of the network. To do this we will take a brief look at the low level QoS parameters of a network. Next we will show what the requirements of the methodology are. Finally we will identify what the criteria for success are of both the framework and the network.

3.2.1 What is Quality?

The idea of what quality is depends on your perspective. As such it is hard to quantify in general. However from both the users' and managers' perspectives quality is attained when the network meets the requirements that have been placed upon it.

Here we will concentrate on what quality is to the administrator. To do this we will use an abstract change in quality we call ΔQ. This quantity represents how quality degrades from its source to its destination. Figure 3.1 shows a digram of this.

It is possible to measure the performance of a network using a relatively small number of parameters. In general we will restrict ourselves to the following:

- Throughput
- Loss Rate
- Delay
- Jitter

We have chosen to normalise these quantities to bytes and seconds through the rest of this document for convenience. The time scale that these quantities are measured over, and hence averaged, depends on the guarantee that we are trying to meet. All of these parameters form part of our concept of ΔQ.

When packets traverse a network the flow that they belong to is degraded; this means that a number of the quality parameters of that flow will be modified. We call this change in quality ΔQ. It can be measured between any two points in a network in which a flow traverses.

3.2.2 Framework Application

A QoS framework needs to encompass all the perspectives of the network into a coordinated approach to providing a QoS solution. Applying the framework should involve executing the following steps, targeted at your particular choice of networking technology.

1. Capture the users' aspirations and check that they are aligned with the needs of the organisation. These should then be prioritised in terms of their criticality to the organisation. This provides us with a clear view of what the network should achieve.
2. Convert the aspirations into requirements. Requirements should be expressed in the administrator's language, this being numerical or constraint[1] based. Care should be take to make sure that the result accurately reflects the users original aspirations.
3. Check that the collated set of requirements can be satisfied. This requires knowing about the topology of the network and information on the usage patterns of its users.
4. Generate a configuration for the network from the requirements. This involves knowing how a configuration will behave under the given conditions. The configuration can then be incrementally redefined until it meets the requirements.
5. By monitoring the performance of the network, and gathering statistics, evaluate how good the process is up to this point. This will provide information on how accurate the original usage patterns were, and help to find new ones, as well as information about how well the requirements were.

[1] By constraint we mean a absolute maximum quality degradation, beyond which the delivered quality would be insufficient to meet the requirements that have been set.

6. Using the available statistics, refine the configuration. This could include introducing increased provisioning for some services, or perhaps placing restrictions to encourage usage of some services at other times of the day.

7. Using the available information identify hot-spots in the network. These could highlight where more investment is needed.

While the points presented above may not be new they do highlight the need for an understanding of the operational behaviour of a network. It is essential to understand what degradation is introduced to a given flow of traffic traversing a network that has been configured in a particular manner. The network topology and configuration are well known; however, the effect on arriving traffic is still elusive in a number of situations.

Ideally we would like to take the requirements we have captured from the users and generate a configuration that will satisfy these requirements, for a given topology. However it would be just as permissible to provide guarantees about the degradation that would occurs given a configuration and a topology. It is this approach that we shall now concentrate on.

What is required is the ability to guarantee the degradation, or ΔQ, a given flow will receive. This guarantee should be statistically sound, and hold under all normal[2] conditions.

3.2.3 Criteria for Success

Once we have calculated the degradation that a flow will receive as it crosses a network we need to check if it is sufficient to support the target application. The two pieces of information that we have available are the requirements set by the user and the guarantee provided by the framework.

Applications generally express their requirements as a worst case quality i.e. $\lceil \Delta Q \rceil$. We denote this by ξ, to distinguish it from a ΔQ that has been measured. This is of course application dependent. We get this requirement by converting the user's aspirations into a numerical form. We can express the criteria for a successful guarantee as follows.

$\Delta Q \leq \xi$

ΔQ is constructed from a number of statistically properties, these properties are the same measures that we use when measuring the performance of a network; namely throughput, loss, and delay. These properties should hold over all time scales; however, the accuracy of these parameters are expected to change depending on the time scale. One approach to evaluating this inequality is to compare each of the parameters in turn. This is not the only approach; some applications will be able to trade between the parameters and hence simple evaluation of each of the parameters may not be sufficient. However as a first cut approximation it is likely to be sufficient for most applications.

[2] This excludes disaster management and recovery processes; due, for example, to hardware failure or user error.

3.2.4 Approach Requirements

Implementing the QoS framework can be achieved through a number of different approaches. However to be ultimately successful it must provide some low level facilities to allow the framework to be fully implemented. Below is a list of the key requirements:

- Ability to capture user requirements in a mathematically meaningful way.
- Ability to predict the behaviour of a configuration.
- Ability to run the configuration at a high utilisation.
- Ability to understand the interaction between QoS and user experience.
- Management of the 2 degrees of freedom [34].

Chapter 4

Methodology

4.1 Introduction

In this chapter we will look at some concepts and insights that have motivated our investigation of QoS. The definitions given here are far from being mathematically rigorous, they are intended to highlight the way in which we think about the problems associated with QoS. We will refer back to these concepts throughout the thesis as we shed more light on the subject. In this chapter we cover the following:

- A description of our assumptions about the way the network behaves. In general, we consider any packet switched network, however, we are mostly concerned with networks based on Internet technology.
- The concept, called Intrinsic quality - there is only a finite amount of quality that the network can provide. This quality can be shared out to provide better treatment to some flows and lower quality to other flows.
- Introduce the concept of loss and delay creators and the fundamental rules that govern packet switched networks.
- Define a Quality Degrader Function (QDF) to capture the behaviour of loss and delay creators.
- The concept of Instantaneous Quality; a time independent measure of quality.

Overall we view QoS as the prediction and management of the behaviour of the network under all conditions. It is the ability to meet established targets that makes a QoS methodology successful.

4.2 Defining the network

Before introducing our framework we feel it is important to outline our view of the network we are studying. Here we are interested in the underlying nature of the network only. Later we will reintroduce some of the mechanisms that we factor out here.

Firstly we are interested in modelling packet switched networks specifically. Such networks carry a number of streams, where a stream is a connected set of packets from a protocol such as TCP. It is assumed that there are sufficient number of streams arriving at a network element to maintain statistical independence. Streams are grouped together into flows, which may be grouped into a larger flow. Flows can be grouped together and split apart as required. The grouping can be done on an arbitrary basis[1]; however in general the treatment of the flow is dependent on the applications' requirements. Our approach deals with flows, and not individual streams. Flows are assumed to be continuous. By this we mean that over a long period of time (hours or more) the average rate remains constant. For a high-speed links this is satisfiable, especially when looking at a particular type of traffic like WWW; although the rate is variable, depending on the time of day, the average for a given hour is relatively stable[2].

Packets are assumed to have a variable length, between some maxima and minima; the Maximum Transmission Unit (MTU) and Minimum Segment Size (MSS) respectively. They also have a way of identifying their source, destination and quality requirements (colour). All of this information is contained in the packet itself. For the purposes of simulation we will also assume that there exists a time at which the packet was sent, and a unique serial number. These two pieces of information allow us to determine the delay and loss rate of the stream. There is also routing information, it tells us the next multiplexing point that the packet will traverse.

Our network is also protocol agnostic. We are not interested in any specific connection control, encoding, flow control or error recovery techniques. We shall deal with these later. We do assume that the packets can be examined to colour them[3], despite the fact that this requires understanding the protocol headers.

Fragmented packets are not considered here, but they can be accommodated in this approach. Fragmented packets can create a problem in determining what class a fragment should be placed in; this is not always the case, as it is highly dependent on the protocols that are being used in the network. In general only the first fragment, unless the information is contained at a higher protocol level, contains the information required. Fragmented packets also cause problems for other areas other than QoS (such as routing), but these are outside the scope of our investigation.

[1] The grouping could be defined by Internet standards per application, or by the administrators of a given network.
[2] It is possible to observe this from the graphs in [95].
[3] Essentially this amounts to a way of assigning a packet to a quality class, without specifying the method used (like ToS fields for example).

Throughout this thesis we will talk about statistical properties of flows. We take an observational approach to explaining the meaning of these properties. The statistical properties that we are interested in are averages, although we will also look at variance and standard deviation of these properties as well. All averages are obtained by observing some property over a period of time, we ensure that the measurement period is sufficiently large that the resulting averages are statically sound. For well known distributions we use the central limit theorem to ensure that our measurement period for the simulations is sufficient; in many cases we measure such properties for substantially longer than required.

The first property that we are interested in is throughput. Packets flow between a source and a destination over the network. The throughput is the rate of packets, over a given time interval, observed at a point in the network. We may refer to the throughput using a number of different terms, depending on the context. Where packets are injected into the network at a source we call the throughput the offered load, as the network has not yet transported any packets. Where the throughput is measured at the destination we may call it the transported load, or alternatively good-put (as this is the work that has arrived successfully at the destination).

There are a number of way to measure the throughput. The most direct of which is to count the number of packets observed in a given time interval. This approach ultimately results in an average over a given time period; care must be taken to choose an appropriate measurement period[4]. The approach taken here for measuring throughput requires us to first measure the average inter-packet time. The throughput is defined as $\frac{1}{IPT}$ measured at the point of interest. The inter-packet time is the interval of time between the head of first packet and the head of the following packet; note that measuring the tail of the packets is equivalent. The benefit of measuring throughput by inter-packet time is that we can make measurements at a packet level, these can later be averaged, if required.

The variation in inter-packet time gives us our first, and most common, measure of jitter. There are two common measures of jitter. The first is the variation in time between successive packet arrivals, which is the variation in inter-packet times. This is essentially a measure of the change of rate over time. The second measure of jitter is the variation in end-to-end delay. We refer to these measurements as variation in inter-packet time and variation in end-to-end delay to avoid confusion over the term jitter.

The final property is end-to-end delay. The delay of an individual packet is the interval of time between the head of the packet entering the network and the head of the packet leaving the network. Again we could choose to measure the tail of the packet instead, which would lead to an equivalent measurement. As we have already mentioned variation in end-to-end delay is also a measure of jitter.

[4]Measuring throughput over periods of time such as minutes may give misleading results; especially when there are short lived periods of high load.

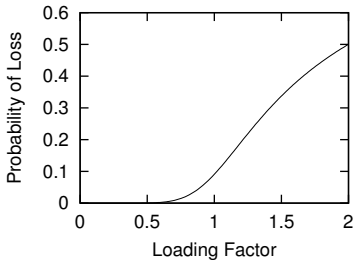


Figure 4.1: Probability of loss in an m/m/1/k queue

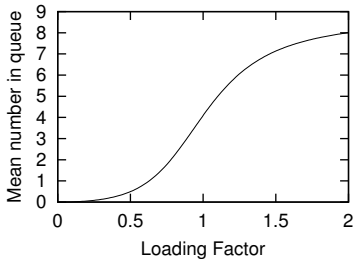


Figure 4.2: Length of an m/m/1/k queue

4.3 Intrinsic Quality

In this section we will start to investigate the problem of providing QoS. To do this we use an example of a simple M/M/1/K queue (See appendix A) to demonstrate our point. Intuitively there can only be a limited amount of quality in a queueing system, any attempt to provide more quality than this is simply infeasible.

The following graphs, figure 4.2, 4.3 and 4.1, show; the average length of the queue, the average time spent in the queue, and the probability of being refused entry to the queue. These graphs can be used to gain some understanding of the quality that a user will receive at loading factors of 0% to 200%. (Where k=10 on these graphs).

Let us examine the quality that we receive in relation to the loading placed on the queue. We know that the probability of a packet being lost is related to the length of the queue. Where the loading is 0% the length of the queue is 0. As the loading increases then so does the length of the queue, which then increases the probability of packet loss (figure 4.1). In effect the higher the loading the higher the loss.

As we have already stated; the higher the load, the longer the queue (figure 4.2). If we join the queue at this point we have to wait until we get to the front to be serviced. We are of course assuming

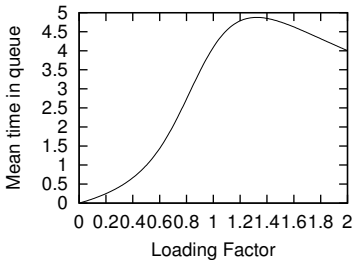


Figure 4.3: Time spent in an m/m/1/k queue

that the queue is being serviced in a First-in First-out (FIFO) order. So as the loading increases so does the average delay that packets will experience. If we look at the amount of time spent in the queue (figure 4.3) we see what initially looks like a surprise - the time spent in the queue starts to drop as the load increases. Packets that are not admitted to the queue spend zero time in the queue and hence contribute a delay of zero to the overall delay. This causes a drop in the average time spent in the queue. However, for packets that are admitted to the queue they will perceive a waiting time that is related to the queue length.

If the user of the system we are considering has dedicated bandwidth[5] then to get a low delay and loss requires using less than the allocated bandwidth. This is good news for the customer, because here is a guaranteed way to get a low loss and delay link. However for the provider this is bad news because it is likely that customers will demand more bandwidth for their allocations than is available or even practical[6].

There is another effect that is caused by the relationship of quality to loading. Users are likely to notice that they get better quality at certain times (when the network is lightly loaded) and not at others. This is undesirable; in the users' opinion they paid for a service and they would like it to deliver the highest quality all of the time. What is required is a method of setting and managing users' expectations. We can achieve this by simply degrading the quality to some bounded value at all times. As a result the users' will receive the same service all the time.

What we can see is that a user experience of quality is dependent on the load placed on the system; indeed this is no great shock. At a given loading factor there is only a given amount of quality available, we call this the Intrinsic Quality. When looking at QoS guarantees it is simple to see that we cannot make guarantees better than the quality that the intrinsic quality of the queue.

The system we have looked at here only has a single flow of traffic entering it. However it is not hard to imagine that this flow is in fact the composition of a number of other flows.

[5]We assume that the loading factor is one when the allocated bandwidth is fully used.

[6]We are assuming that the provider is using a packet-switched network and attempting to exploit the statistical multiplexing as much as possible.

Intuitively it must be true that the sum of quality degradation that the individual flows receives must be equal to the total quality degradation of the queue. This provides us with the notion of trading; that is in order to provide better quality (absence of degradation) to one flow we must provide worse quality (more degradation) to another flow. Moreover this trading can only occur within the bounds of the total intrinsic quality of the queue.

4.4 Network Quality Invariants

In this section we present some rules that govern the processing of packets as they traverse a network. Understanding these is important, as it encapsulates the basic behaviour of the network.

For a given flow, loss will always increase. As packets traverse a network there is always a possibility that they will be lost. Communication links could corrupt a packet causing it to be discarded; however, we generally assume that the likelihood of this is relatively small[7]. More likely the packet will be refused entry to a buffer due to contention and hence will be discarded. As we are uninterested in error recovery the packet is simply lost; if a higher level function were to retransmit the data this is viewed as another packet altogether. With this in mind there is no way a packet can be un-lost, and hence loss is always increasing.

For a given flow delay, as measured from the source, will always increases. Almost every element in a network introduces delay. When a packet is transmitted across a transmission line it is first serialised, then transmitted and then deserialised. Serialisation and deserialisation introduce a delay dependent on the length of the packet, and transmitting each byte of the packet adds a delay that is dependent on the length of the link and its characteristics. There are also queueing delays, where the packet has been successfully accepted into a buffer. This delay is caused when packets are being serviced from another flow (inter-flow delay) or from the same flow (intra-flow delay). Importantly once a packet has been delayed it can never be un-delayed; that time has simply elapsed. Thus a flow of packets will always be increasingly delayed.

For a given flow throughput will always decrease. Whenever flows interact for a resource, such as bandwidth on an outgoing link, sharing must occur. At this point the throughput of each individual flow will be decreased. The throughput cannot be increased once degradation has occurred. One could increase the service rate to increase the bandwidth, but this can only be sustained while there is a backlog of packets in queues in the network. Once this backlog has been removed its obvious that you cannot remove packet from the network faster then they enter it.

By definition these three parameters are essentially the ΔQ that was introduced in the last chapter. At each point in the network the quality of a particular flow will be decreased by a local ΔQ. Over

[7] Note that we view contention for a shared resource, such as a WLAN, as part of the service facility of the queue; failure to transmit is not considered to be corruption.

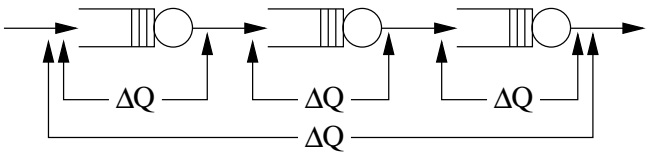


Figure 4.4: ΔQ Being Composed

the whole network these amount to an overall ΔQ. It is this quantity that we are interested in understanding and eventually calculating. Figure 4.4 shows three queues, each of which contribute a ΔQ to the total ΔQ.

As we can see networks essentially lose, delay, and restrict packets. Logically this has to be the case, unless the laws of physics can be broken in packet switched networks. So what is quality? Our definition is "Quality is the minimisation of degradation to a specific flow". By minimising the degradation of one flow implies that other flows will be degraded more. Essentially providing quality of service on a packet switched network involves trading different kinds of degradation.

4.5 Quality Degradation Functions

Quality Degradation Functions (QDF) are an abstract representation of elements within a packet switched network. Their purpose is to capture the quality degradation, ΔQ, that a flow would receive if it were to traverse a given network element. The quality degradation observed follows rules imposed by the Network Quality Invariants. As such the function may only increase delay, loss or anti-put of the flow that traverses it. It may increase any number of these parameters at a given time. Note that, depending on the distribution of packets, other measures such as jitter are not subject to these rules.

Degradation parameters are represented numerically as probabilities. Loss, for example, is the probability that a packet will arrive at a full queue. When measuring quality parameters it is important to obtain a statistically accurate result, as such measurements have to be conducted when the system under consideration has reached a steady state, and then for sufficiently long to get an accurate result. The amount of time taken to reach steady state can be calculated for a single M/M/1/K queue [100], and then the central limit theorem [96] can be used to calculate the measurement time.

A QDF can be viewed as a black box (see figure 4.5) that accepts a flow as an input, and returns a degraded flow as an output. Additionally we have a lost traffic flow to capture any lost traffic from the QDF; clearly in a real network packets are just dropped and do not exit the system by another

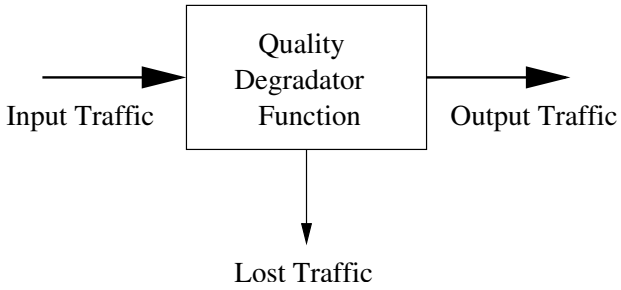


Figure 4.5: Quality Degradation Function

route, we include the loss flow for completeness. What is important is that a QDF may not create traffic internally nor may it lose traffic other than through the loss traffic flow.

4.6 Instantaneous Quality

In section 2.8.1 we introduced the concept of requirements capture. As was demonstrated specifying a requirement on the total percentage loss is not sufficient to adequately capture the requirements. In this section we demonstrate, through an example of burst losses, that capturing requirements accurately is more complex than simple averages. Or aim is to find simple ways of capturing behaviour which can be used for requirements capture, measurement and admission decisions.

We take an example guarantee for VoIP calls; here we promise to deliver 99.9% of packets within 150ms. During this example we will concentrate on the dynamics of loss in this system. We define a successful VoIP call to be one that never experiences 4 losses in a row; this has been chosen because such a situation would cause a serious glitch in the audio quality [11]. If a user experiences such a glitch then they are likely to judge the call as having bad quality.

Taking our 99.9% packet delivery we distribute the failure seconds evenly over an 8 hour working day, in one second bursts[8]. Given that there are 2628 failure seconds in a month, there is a $3x10^{-3}$ probability of any second being a failure second. That is during such a second all packets are lost, this will without a doubt cause 4 losses in a row. The probability of an individual second being successful is 0.997. For a three minute VoIP call there is only a 60% (i.e. 0.997^{180}) chance of success, because each second of the call must not be a failure second. While this is far from an accurate model it does demonstrate that our original specification was far from accurate.

We can model burst losses in this example using a discrete time Markov chain (See figure 4.6). The first state represents no loss, as we move to the right we experience a number of consecutive losses.

[8]The choice of failure seconds is arbitrary; it could be failure half-seconds or failure two-seconds.

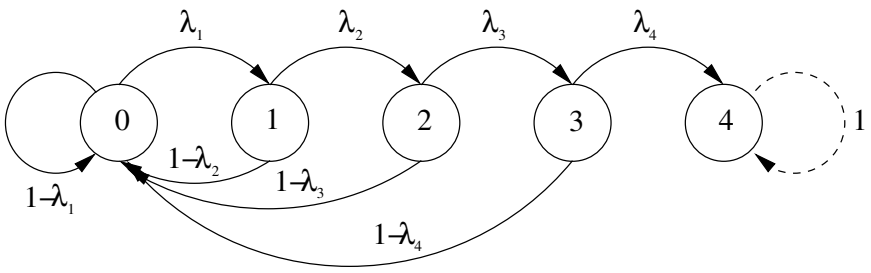


Figure 4.6: Instantaneous Quality Markov Chain

The final state, state 4, is an absorbing state; it is this state that we are most interested in as the probability of being in this state is the probability that the call had unacceptable audio quality. The values of lambda represent the probability of a single loss, and given a single loss the probability of another loss, and so on.

This chain can then be solved using the power law method described in [3], or any other text book on the subject. We start with an initial guess of what state we are in, which in this case at the start of the call is state 0. We then iterate until we reach a solution. The number of iterations is important, as each iteration represents an event. In this example VoIP sends 50 packets a second, so we need to iterate for 50 times the number of seconds the call lasts for.

Figure 4.7 show the probability of a call being considered a failure, assuming that all losses are independent, for a number of different loss rates. Unfortunately previous research [12] has shown that losses are not independent. If we include their finding in our model we quickly find that the probability of failure is almost always 100%. This clearly has some significant implications for VoIP support on the Internet. Moreover, a simple measure of the loss probability is insufficient to capture the behaviour.

What is required is a measure of quality that is independent of time and true at every instant; we call this "Instantaneous Quality". This property needs to capture the behaviour of loss and delay in some meaningful way such that we can be assured that we can meet guarantees that we require. At the moment we use average delay and unconditional loss probability as our measures of instantaneous quality; however, these need to be refined.

4.7 Trade-offs and Constraints

In this section we will look at a number of simple trade-offs that can be made in packet-switched networks. These are generally applicable to any queueing discipline or networking technology.

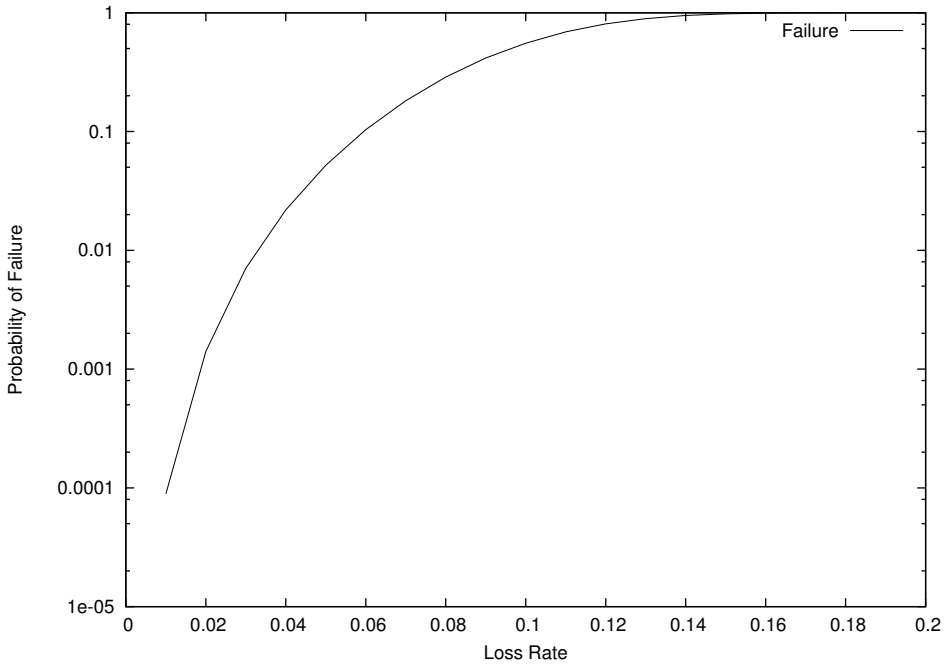


Figure 4.7: Failure of VoIP calls with independent loss

4.7.1 Packet Size Effects

Today it is common for a packet switched network to carry packets that have a variable length. Technologies such as IP do exactly this, where as technologies such as ATM do not[9]. However, it is common to place a restriction on the maximum packet size that will be transported, this is called the MTU. Packets that exceed the MTU may be fragmented, depending on the technology.

Clearly different sized packets take different amounts of time to service, assuming a unit is serviced at the same rate. For this reason it is convenient to normalise service times to packet service times. This allows a simple comparison between the waiting time for different sized packets.

It is important to understand that the size of the largest packets affects the performance that we can expect. Whenever a packet is enqueued the number and size of the packets ahead is unknown. While we may, as we will see in the next chapter, be able to predict the average number of packets ahead, predicting the size of those packets is harder. This has a direct effect on the performance that we can expect.

When predicting the service time for a packet we cannot expect to make predictions more accurate than one packet service time. This is because at any queueing point there may be one packet more,

[9] Although they handle variable sized loads through adaption layers.

or less, than we expected ahead. Attempting to make predictions at the granularity of one packet is unrealistic.

When there are packets with multiple sizes in the same network we are restricted to predictions of the order of the packet service time for the largest packet. The same argument follows, in that we do not know, to the accuracy of one packet, how many large packets are ahead. For the small packets this represents an accuracy of a number of service times. For example, on an Ethernet LAN we would expect an accuracy of one packet service time for maximal sized packets (1500 bytes) and around twenty three service times for small packet (64 bytes).

Whenever we make predictions about the performance of a network we are constrained by these facts. The problem occurs when we wish to make predictions for small packets that are more accurate than the packet service time of the largest packet.

One possible design choice that we have available is to lower the MTU size. By doing so we can make more accurate predictions for smaller packets at the cost of decreasing efficiency for applications using larger packets. Making such a trade-off is acceptable where the gain in accuracy, and hence predictability, out-weights the loss in performance for large transfers. This is a choice for the implementor of a quality network.

4.7.2 Buffer Size Effects

Packet-switched networks are based on the concept of store and forward. A packet arrives at a node and is queued until it receives service. The amount of time that it takes to be serviced is dependent on the number of buffers in the queue and the amount of time required to service the preceding packets. The amount of buffer capacity is finite, and dependent on the implementation of the node.

Some approaches to QoS assume that there is infinite buffer capacity, regarding packet loss as a rare event. While this may ease analytical analysis, the author believes that such models are on the whole inaccurate as they do not account for the effects of packet loss. In this thesis we model queues with a finite capacity, as is the reality. For the ease of analysis we assume that a packet, irrespective of size, occupies a single buffer. This is not uncommon in queueing theory, the mathematical basis that we use throughout.

The choice of buffer size affects the performance of the network when it comes to the provision of QoS. Therefore, we consider the choice of buffer size important when considering the performance that we would like to achieve. In real networking technologies the buffer size is fixed, and usually quite large, however, it is usually possible to configure the capacity if required. It may seem unusual to consider buffer sizes in a methodology for QoS, but as we will show, it is another trade-off the network administrator may use.

Large buffers sizes allow more packets to be buffered. When considering delay we find that large buffers imply more delay, this is because any packet arriving at a heavily loaded queue will have to wait for any preceding packets to be serviced. Conversely, large buffers reduce loss, as they can smooth out short term fluctuations in load. However, when the load is fluctuating so will the occupancy of the queue, this serves to increase the variance in delay.

Small buffer sizes allow only a few packets to be buffered. The delay introduced by a queue with a small number of buffers is also small, as is the variance. Unfortunately the side effect is to increase the loss, especially when there are short term fluctuations in load. Small buffer sizes, of the order of 10 packets, generally result in unacceptable loss rates. However, they are far more predictable in terms of delay.

Buffers can be seen as giving a queue a memory. The length of this memory is dependent on the number of buffers. So a short queue will only be effected by fluctuations in load for a short period of time, where as a larger buffer capacity will have a longer memory. For queueing theorists the time taken to reach steady state is the important measure of the memory. The larger the buffer the more time is taken to arrive at steady state. Where the buffer capacity is measured in terms of Megabytes it may take months to come to steady state, assuming that the input load remains constant.

It is advantageous to have a queueing system that has less memory of recent events. This means that short term fluctuations in load have a smaller effect. As a result the analytical models that queueing theorists use are more likely to accurately model the behaviour of a real system. We include the choice of buffer size in the methodology because it has a direct impact on the performance of the network.

We have a trade-off between large buffers, with a large memory and small loss rate, and small buffers, with a small memory and a larger loss rate. Huge buffer sizes, as found in the majority of todays network equipment, reduce loss but at the expense of larger delays and variance in delay. Choosing an appropriate buffer size requires making an acceptable trade-off between loss and memory. In this thesis we do not analytically investigate this topic, we do however make some educated guesses as to the acceptable size of buffers.

Chapter 5

The Model

5.1 Applying the Methodology

In this section we will show a method, based on our Quality Methodology, for providing QoS support to a network. To achieve this our choice of technology must satisfy the following criteria:

- A computational or algebraic model of all the network elements. This is required so that it is possible to calculate the ΔQ introduced by each component.
- A method of composition. This is required so that the degradation introduced by each component can be combined to yield the total degradation, or ΔQ, across the entire network.
- Instantaneous Quality. A time independent model of the degradation is needed so that the results from the composition are meaningful at all timescales.

To achieve these goals we are going to service packets in a Poisson manner; however, this is not the only model we could use, it is just the simplest distribution with the properties that we require. This means that the time taken to service any individual packet will be determined by a random process. It is hoped that by introducing such randomness into the system, it will become more stable, and predictable; allowing us to make accurate quality guarantees.

The first benefit of using Poisson traffic is that it is the most well understood mathematically. By ensuring[1] that the traffic is Poisson distributed at the edge of the network we can take advantage of well known mathematical techniques. Using queueing theory we are able to calculate the waiting times and loss probabilities of the traffic, and their distributions. This provides us with both a

[1] This could be achieved using traffic shaping.

model and an implementation of the QDFs that we introduced earlier. Notice that the waiting time increases the delay of the packets, the loss probability increases the number of packets lost, and as a result both reduce the throughput. These therefore follow the rules for the Quality Invariants.

Poisson streams have the PASTA (Poisson Arrivals See Time Averages) [104] principle. If we consider a number of flows arriving at a QDF, in steady state, they will observe the same time averaged properties, including delay. As a given flow traverses the network it will see the average delay of each QDF that it traverses. The end-to-end delay can be obtained by adding the mean delay at each point in the network. The same argument also applies to the loss, where again this can be added together. We shall use this method of computation from now on.

It is worth noting that this is not the only method available to us; using Laplace transforms it is also possible to symbolically (and numerically) compose these quantities. The formulae used to represent a QDF can be transformed into Laplace space. These density functions can be convolved, a potentially complex operation that becomes multiplication in Laplace space. Finally an inverse Laplace transform can be performed on the convolved formulae, allowing us to calculate the CDF (cumulative density function) of the final distribution. Such a method provides us with more accurate predictions for the performance of the network, but would cost substantially more in terms of calculation complexity. A similar approach for solving such systems can be found in [14]. We leave this method as an area for future investigation, where the methods that we present in this thesis are insufficiently accurate to produce useful answers.

Poisson traffic is by its very nature memoryless. This means that the time of the next event is not dependent on the previous event. As a result, it allows us to maintain statistical independence throughout the network, giving us the instantaneous measures we are looking for. Finally, assuming that the distribution is Poisson and the rate is known, it is possible to calculate other moments of the traffic for example Jitter.

5.2 Exponential Service

One of the key differences between our approach and others to providing QoS is that we use exponential servicing throughout the network. This is in stark contrast to the majority of other approaches that use deterministic servicing; where the service time for a packet is found by multiplying the size of the packet by the time taken to service one unit of the packet.

When referring to exponential servicing we are in fact talking about an inverse-exponential inter-packet time distribution. A packets service time is defined, in part, by a random process and not solely by the length of the packet. In fact the service time for a packet is found in the following way. An exponentially distributed random sample, taken from a random number generator, is used

to give us a random service rate. This service rate is then multiplied by the length of the packet to give the service time. As a result every packet will have a different random service time.

One benefit of such an approach is that it makes the analytical models simpler. A large proportion of queueing theory deals with Markovian servicing (denoted by M in Kendal's notation; see appendix A). The Poisson (or exponential) distribution is used for the service facilities in such queues. Using the approach outlined above we can closely approximate a Poisson distribution, and as a result the system becomes more mathematically tractable. This is a key benefit when attempting to make predictions about network performance.

Another benefit of servicing packets exponentially is that the merging properties at a multiplexing point are fairer. When packets arrive at a multiplexing point from a number of exponentially serviced queues the order of arrivals is indeterminate. If the queues are deterministically serviced there is a high likelihood that flows from different queues would have a phase relationship, causing short term unfairness. In other words at any given interval of time one of the flows would receive better treatment as it was always slightly ahead of the other flows. When we service packets in a exponential[2] manner this time-dependent phase relationship is destroyed, improving the fairness.

A disadvantage of exponentially servicing packets is inefficiency. Clearly we are introducing more delay and more variance than we would in the deterministic case. Again we have a trade-off, on one hand we have efficiency and on the other mathematical tractability. In this thesis we hope to show that the decrease in efficiency is warranted where the gain in predictability is more important.

5.3 Composing Queues in a Network

For our approach to QoS to work correctly we have to be able to compose queues together, this hinges on the traffic in the network being Poisson distributed. Unfortunately, it is well known that the departing traffic from a finite Markovian serviced queue is not strictly Poisson. The question is can we safely assume that it is a good approximation and retain the composition properties that we require.

This work can be seen to be a continuation of the work on Jackson Networks [51] and more recently BCMP Networks [9]. These models deal with queueing networks that have infinite buffer capacity and are loaded less that 100%. By allowing infinite buffers these models of networks do not encounter problems associated with non-Markovian traffic. They have to be loaded less that 100% or waiting times could tend to infinity quickly.

We would like to be able to model packet networks that can be loaded more than 100% and have finite buffering. For this reason these methods are not directly applicable, although the general idea

[2] This property is likely to hold for a number of other distributions as well.

remains the same. As our model now has finite buffers we must also consider loss, and not just delay. This however is not a great problem. However, we do return to the problem where the output from a finite queue is not Poisson.

A recent piece of research [49] has shown that a traffic's pattern is strongly determined by the service discipline. This demonstrates, at least graphically, that CPR traffic becomes exponentially distributed after passing through a number of exponentially (Markovian) serviced queues. Further to this they show that for single buffer queues that this is the case mathematically. It does also seem possible to prove this for the general case. This work suggests that traffic that is near Poisson on input will become increasingly Poisson as it traverses the network.

We do not attempt here to prove mathematically that we can safely ignore the problems associated with non-Markovian traffic. However, we do intend to demonstrate that in practise we can safely ignore such problems.

5.4 Modelling Variable Sized Packets

The approach presented so far has a limited applicability. One of the assumptions made was that packets did not have a length. This is clearly unrealistic and unhelpful for modelling real networks. For the approach to be useful it must possess the ability to correctly calculate guarantees for packets that have a length, as this affects both the service time and the distribution of packets.

Some networking technologies, such as ATM, handle packets (or cells) that have a fixed length. Here data is fragmented into these fixed sized cells for transmission over the network. However, technologies such as ATM are in a minority; most networking technologies, such as Ethernet and IP, possess the ability to handle packets of variable sized length. For our approach to be generally applicable it must handle flows arriving at a network element that contain packets of different lengths.

We shall assume, for the moment, that flows contain packets that have identical lengths. This is not an unreasonable assumption, for example: FTP transfers are likely to have maximal sized packets, and WWW requests minimal sized packets. This constrains the problem to flows of differing sized packets merging together. The following discussion will take a number of approaches to calculating the guarantees, and evaluate them through simulation. It is assumed that the worse case is two flows merging, where the two flows have maximal and minimal sized packets. Here we take these to be 64 and 1024 bytes respectively. It is important to note that in our queue model a packet occupies one buffer irrespective of its length.

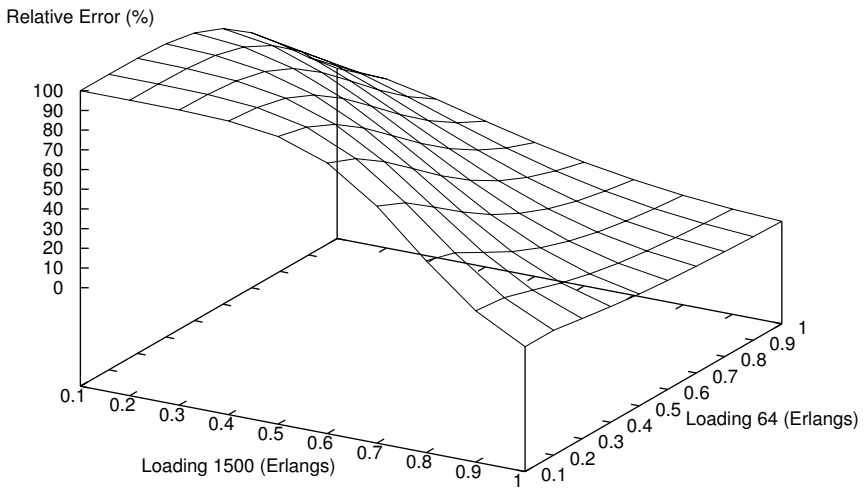


Figure 5.1: M/M/1/k Loss

5.4.1 Using the M/M/1/k Queue

The first approach we are going to examine is using the standard M/M/1/k queueing formula [3, 96] to perform the calculations. To do this we scale the quantities appropriately.

The following graphs show the relative error between the calculations and the simulations. In this scenario there is a single queue with two flows arriving at its input. The two flows have their packet arrival rate varied between 0.1 and 1.0 Erlangs (see appendix A), providing us with a surface. Each flow has packets of a different size, 64 and 1024 respectively, and arrives in a Poisson distributed manner.

The following formula is used to calculate the relative error. Using the queueing formula we can predict the outcome of some random variable, this expected value is denoted by $E[x]$. Using the a simulator we then measure this value, x, allowing us to compute the relative error.

$\epsilon = \frac{x - E[x]}{x}$

Looking at figure 5.1 we can see that the calculations provide a poor approximation to the observed behaviour. The probability of loss is determined by P_k, the probability of the queue being in the k'th state i.e. full. Both flows observe the same queue, and as such observe the same probability of finding the queue full. As both the flows experience the same loss we only show one surface. The

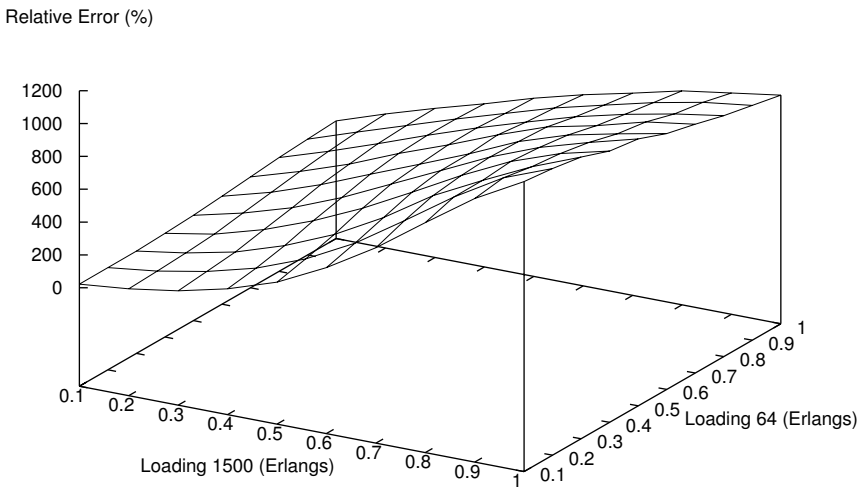


Figure 5.2: M/M/1/k Delay

error reduces when the loading factor on the queue is one or more; this is because the majority of the packets are lost at this loading factor. What this tells us is that the M/M/1/k queueing formula, when applied in this way, incorrectly approximates the state probabilities. The cause of this is the difference in service time between the two flows, where the larger packets take longer to service than the smaller packets.

Figure 5.2 shows the relative error in delay for the 64 byte flow. The relative error for the 1500 byte flow is similar in structure, but not as severe; we have shown only one flow for simplicity. As the combined loading factor increases, so does the error. This is because the service rate used in the mathematics does not take account of the packet sizes. As the loading factor increases so does the average number of packets in the queue. The problem is that as the service rate is inaccurate, and skewed by the packet sizes, the more packets in the queue, the worse the results are.

The conclusion is that the M/M/1/k queueing formula provides an inaccurate approximation when applied in this way. This is exactly what we expected. What is required is a set of mathematics that more accurately models the behaviour of this system.

5.4.2 The M/G/1 Priority Queue

There are many well known results from queueing theory that are widely available. What we required is a set of queueing formula that more accurately approximate our situation. Looking at [3] there are some results for the M/G/1 Priority Queue. In this type of queue there a n customer classes, where each customer is serviced with a generally distributed service time. More importantly there are classes of service but no priorities, in the simplest form (despite its somewhat confusing name). This means that jobs are serviced in a FIFO (First-in First-out) way. While this type of queue is not ideal, in as much as it assumes infinite buffering, it does allow for different service rates within the same queue. This, it is hoped, will more accurately approximate our two packet-size system.

The first step is to evaluate this queue to see if it can be applied in our scenario. To do this we again return to our example configuration with two flows having 64 and 1024 sized packets respectively. However, this time we have to ensure that the loading factor remains under one, as this would lead to an infinitely long queue. The following formulae are taken from page 700 of [3].

The first step is to covert the packet arrival rate R_i, packet size S_i and service rate μ and convert them into the form required for the calculations. The arrival and service rate per flow can be calculated as follows:

$$\lambda_i = \frac{R_i}{S_i}$$
$$\mu_i = \frac{\mu}{S_i}$$

From these quantities we can calculate the expected service time $E[s_i]$ and its second moment $E[s_i^2]$ as follows:

$$E[s_i] = \frac{1}{\mu_i}$$
$$E[s_i^2] = \frac{2}{\mu_i}$$

At this point we are going to restrict our discussion to the two flows in our example. The general forms for the following formula can be obtained from [3]. Next we can calculate the total arrival rate into the system, λ, as follows:

$$\lambda = \lambda_1 + \lambda_2$$

Next we can specialise the formula to calculate, W_s - the expected customer service time for the whole system, $E[s^2]$ - the second moment of the expected service time for the whole system, and W_q - the amount of time waiting for service as follows:

$$W_s = \frac{\lambda_1}{\lambda} E[s_1] + \frac{\lambda_2}{\lambda} E[s_2]$$
$$E[s^2] = \frac{\lambda_1}{\lambda} E[s_1^2] + \frac{\lambda_2}{\lambda} E[s_2^2]$$
$$W_q = \frac{\lambda E[s^2]}{2(1-\lambda W_s)}$$

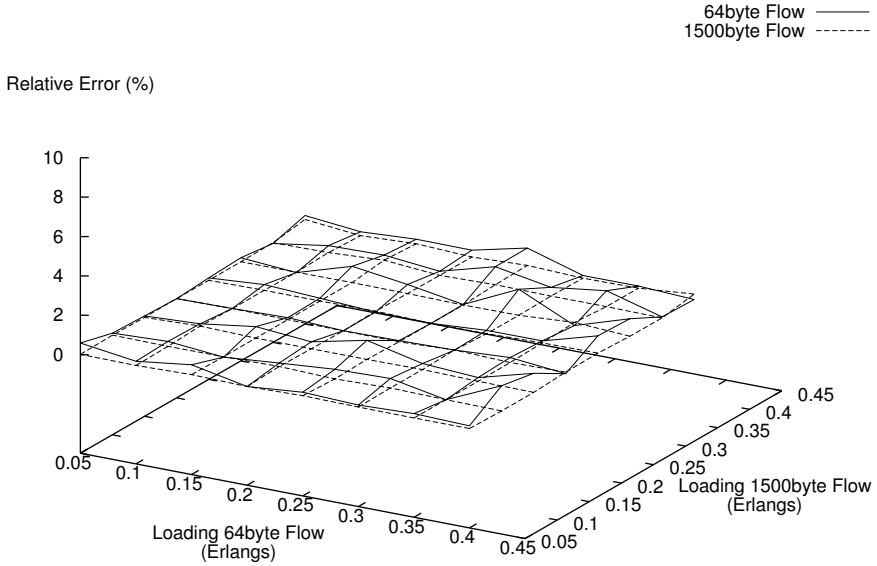


Figure 5.3: M/G/1 Delay

Finally we can calculate the time spent in the system per flow. This is equal to the time spent in the queue (W_q) added to the time taken to service that class ($E[s_i]$).

$$W_i = W_q + E[s_i]$$

In addition it is useful, as a means of checking the correctness of the simulation, to know the average waiting time for all classes in the system.

$$W = \frac{\lambda_1}{\lambda} W_1 + \frac{\lambda_2}{\lambda} W_2$$

Figure 5.3 shows the relative error between the simulation and calculation of the delay. Note that we cannot calculate the loss, as none occurs. The packet arrival rates are fixed to a maximum of 0.45, ensuring that the maximum load on the queue does not exceed 0.9, as we do not want to cause an infinite queueing delay. As you can see from the graph the maximal relative error is around 1.4%. This is around one packet service time for the largest packet, which is within the bounds that we would expect (see section 4.7.1). The flow (1) with the smaller sized packets suffers more in terms of accuracy that the flow with the large packets; this is to be expected as the effect of the large packets is much more significant than that of the small. In general this type of queue approximates our scenario sufficiently well.

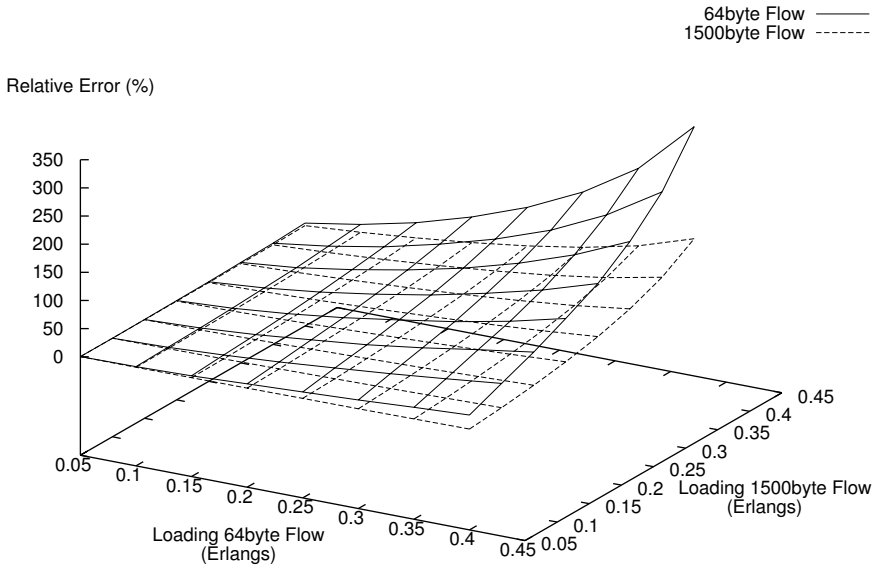


Figure 5.4: M/G/1 Delay, with finite buffers

5.4.3 Using the M/G/1 queue as an approximation

The M/G/1 queue, as show above, is not entirely suited to our situation. Specifically, it assumes that there are infinite buffers in the system. While the results above are promising they are of little value unless we can handle queues that have a finite number of buffers. In such a case we would also like to calculate the loss, per flow, that occurs in the queue.

The next approach we will take is to use the same M/G/1 queueing formula, but used to approximate a queue that has a finite amount of buffers. In other words, we are going to perform calculations using the M/G/1 queueing formula while simulating a queue that does not have a infinite number of buffers. The example used is the same as before, other than the queue has only 10 buffers that can be shared between the flows. We still restrict the arrival rates so that the loading factor on the queue is less than one to again ensure that the queueing time is not infinitely long.

Figure 5.4 show the relative error between our approximation and the results from the simulator. As you can clearly see, this is by no means a good approximation, and is completely unworkable. This is because the maths takes no account of loss, which is unsurprising as it assumes infinite buffering, and as such returns results that are far removed from those that are simulated. Another method is clearly required.

5.4.4 Using the M/G/1/k Priority Queue

Recent research [101] into M/G/1/k queues has yielded valuable results, which we can use to better model our results. It is possible using this research to calculate the loss ratio (LR) of the queue. The loss ratio is, in essence, the same as P_k in that it returns the probability of the queue being full. This loss ratio can then be applied to both flows equally. In addition, by calculating all of the state probabilities, it also allows us to calculate the average number of customers in the system L.

As we know the average number of customers (packets) in the system we can calculate, by Little's Law, average waiting time in the system. Note that we use the rate of traffic accepted into the system, λ_a, in the following:

$$\lambda_a = \lambda(1 - LR)$$
$$W = \frac{L}{\lambda_a}$$

By substituting this waiting time, W, into the waiting time of the M/G/1 queue and expanding we are left with the following for the time spent in the queue.

$$W_q = \frac{L}{\lambda_a} - \frac{\lambda_1}{\lambda}E[s_1] - \frac{\lambda_2}{\lambda}E[s_2]$$

The time spent in the queue for an individual flow is given by:

$$W_i = W_q + E[s_i]$$

Using these formula we can re-evaluate our previous simulation results. Figures 5.5 and 5.6 show the relative error in the delay and loss respectively. The large spikes on the loss graph are due to simulation errors, this is because the probability of a loss occurring is extremely low and any loss that does occurs will skew the results. Both graphs are predominately under 1% relative error - making this type of queue a good approximation of our scenario.

5.5 Performing the Calculations

In the previous sections we have show a number of methods for modelling flows with different sized packets. Using the M/G/1/k queue gives us the most true representation of the performance of a variable sized packet system. What is required to use this model is a solution to the M/G/1/K class based queue. Recent work [101] has produced closed form mathematical solutions to this queue for two and three classes of traffic. This has concentrated on solving the balance equations for each state, and then by re-arrangement returning the closed form solutions; clearly this technique requires human intervention.

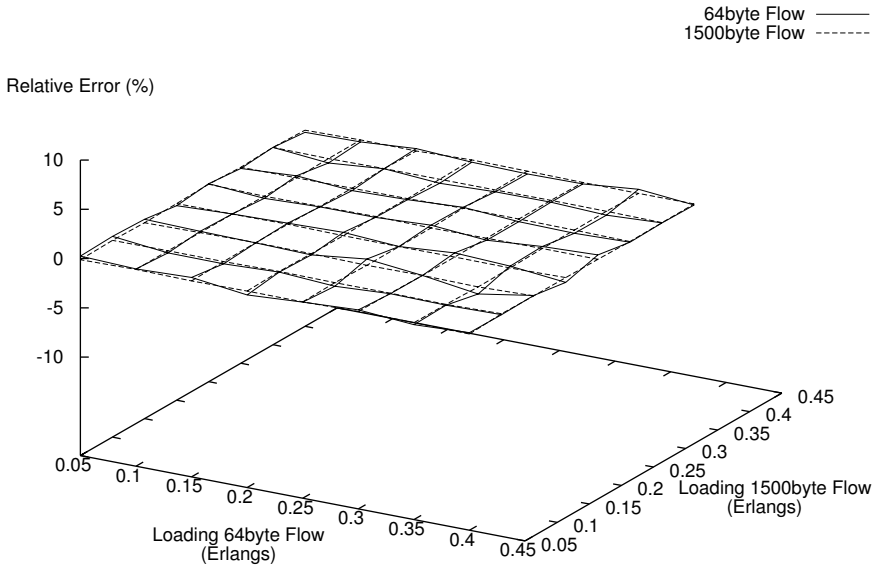


Figure 5.5: M/G/1/k Delay

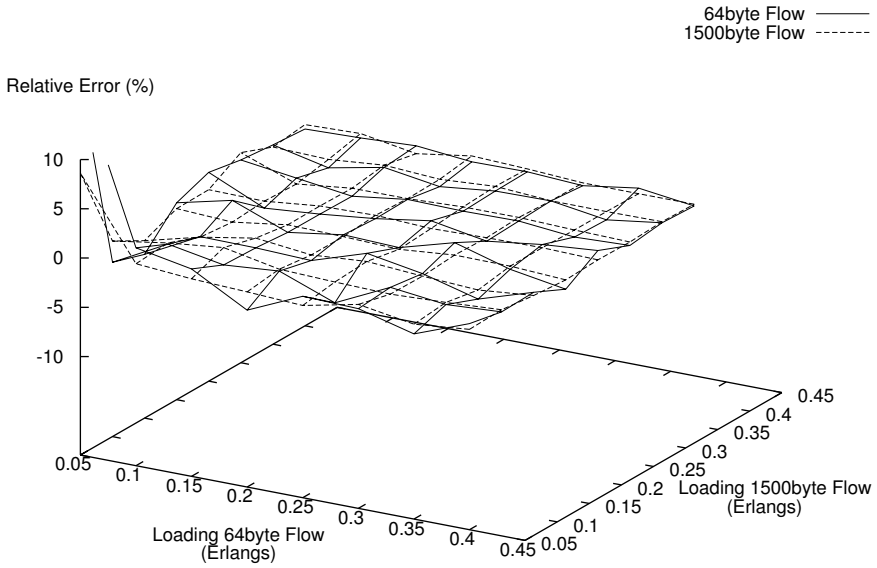


Figure 5.6: M/G/1/k Loss

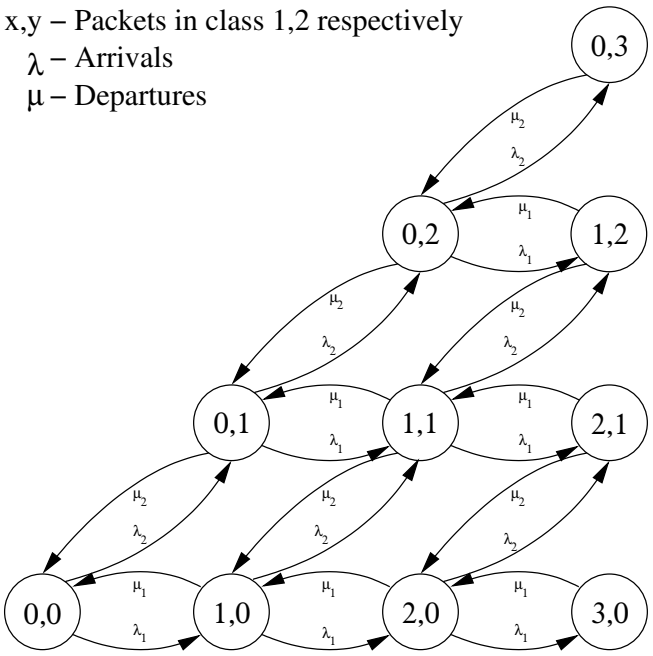


Figure 5.7: Two Class Markov Chain

To avoid having to produce closed form solutions for each of these problems we propose use of matrices to solve each case individually. In order to do this we need to construct the Markov chain representing the problem, and from there produce the transition matrix representing it.

What is required is a way of automatically generating the Markov chain for the problem that we are trying to solve. Figure 5.7 shows the Markov chain for a two class queue with three buffers. The state represents the number of packets in the queue from class one and two respectively. In each state there can be an arrival from either class, this will cause a transition into another state - other than when the queue is full and the packet will be discarded as usual. Departures are similar, but may only occur if there is a packet in that class requiring service. The queueing discipline of this queue is first-in first-out (FIFO).

Using the methods presented in Appendix A and [3, 96] it is possible to construct a matrix in order to solve this Markov chain. Using such an approach it would be possible to solve arbitrary combinations of flows with different sizes. Note that if two flows have the same packet size then we can simply consider them as an aggregate flow in this system. Generating and solving chains for this style of problem could be done automatically, avoiding the use of the closed form solutions.

The drawback of using such an approach is the time taken to compute the answer. It some situations the algorithms used to solve such chains may not converge, although this depends on the algorithm chosen. Clearly such a method could not be used in admission control decisions; however, it could be used in provisioning decisions where the time constraints are lower.

5.6 Modelling Burst Loss

Burst loss probabilities are important for a number of applications, such as VoIP. In the past loss had been modelled unconditionally, that is to say the probability of a loss occurring was independent of a previous loss occurring. Previous studies [12] have shown this not to be the case in practice and results from our own simulator agreed with these findings.

The model of networks that we are investigating is based on Poisson traffic, which is well known for having memory-less properties. This means that the time of an arrival is not determined by the time of the previous arrival. It would seem to follow that a loss would not depend on the loss of the previous packet, however, this is not the case, as queues are not memory-less. When considering a queue in steady state we have a measure of the average loss probability, this is the probability of finding the queue in the last state or full. To consider the arrangement of losses we need to take a more detailed look at the tail behaviour of a queue.

Looking at traces from our simulator we found that there is a probability of an initial packet loss, and another probability of a second or subsequent packet loss occurring. After examining the results further we found that the probability of losing a packet after the last packet was lost remains constant. That is, given that we have lost the first packet, the probability of losing subsequent packets remains constant.

This means that there are two probabilities required to capture the process of burst loss. The first is the probability of losing a packet initially. This happens when the queue is full and an arrival happens. The following formula calculates this probability:

$P_\alpha = P_k(\frac{\lambda}{\lambda+\mu})$

The second probability is the probability that an an arrival happens. As the system is already full this will cause a loss. As the arrival process is memory-less arrivals are independent; therefore, this probability remains the same until a departure occurs to reset the system. The following formula is the probability of a conditional loss:

$P_\beta = \frac{\lambda}{\lambda+\mu}$

Using these two probabilities we can construct a Markov chain to model this system. This chain is infinitely long as an infinite number of losses in a row could occur. Figure 5.8 shows this:

Notice that we can loop into state 0, no losses, to represent successful arrivals. After this we start traversing through the states in the chain, where state 1 represents one loss in a row and so on. At any point a successful arrival can occur, reseting the chain back to state 0.

The Markov chain presented in figure 5.8 has been simplified to allow it to be solved by iteration (see below). The Markov chain is figure 5.9 shows the addition of the loss states to a standard M/M/1/k

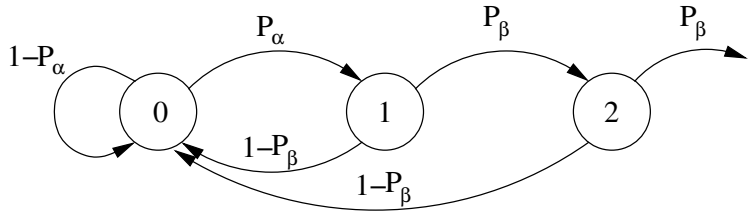


Figure 5.8: M/M/1/k Loss Chain

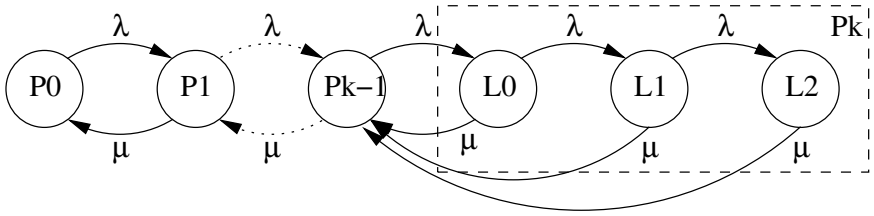


Figure 5.9: Full M/M/1/k Loss Chain

queue. The last state in the queue, P_k, has been expanded into three states, although it could be more if required. The state L_0 now represents the queue being full; however, no loss occurs in this state. Arrivals when the queue is full move the queue into the L_1 and higher states; each of these states corresponds to a number of losses occurring in a row. By constructing balance equations it is simple to show that the L states have the same probability mass as the original P_k state; as such this chain is equivalent to a standard M/M/1/k arrangement, but provides more information.

The chain presented in figure 5.8 is also theoretically equivalent to the chain presented in figure 5.9. The states P_0 to L_0 have been collapsed into a single state, where the transition out of that state is dependent on the queue being full. The benefit of this approach is that it allows the loss process to be examined in isolation from the rest of the queue, all that is required is the probability of the queue being full, the arrival rate and the service rate. We are confident that this is a correct representation for Markovian systems; however, more work is required to mathematically prove this both for the Markovian and general case.

Using this Markov chain (figure 5.8) we can calculate the probability of getting a given number of consecutive losses in a row. To do this we construct a probability matrix representing the Markov chain and iterate over it. Each iteration represents a packet arrival, so we must iterate for the number of packets we are interested in. Its important to note that over an infinite number of packets we would expect to see each length of burst losses in a row, as such, solving this chain for steady state does not make much sense.

Figure 5.10 shows the probability of a VoIP call of a given length succeeding at different loading factors, when transported by a M/M/1/k queue with ten buffers. We are assuming that the call is

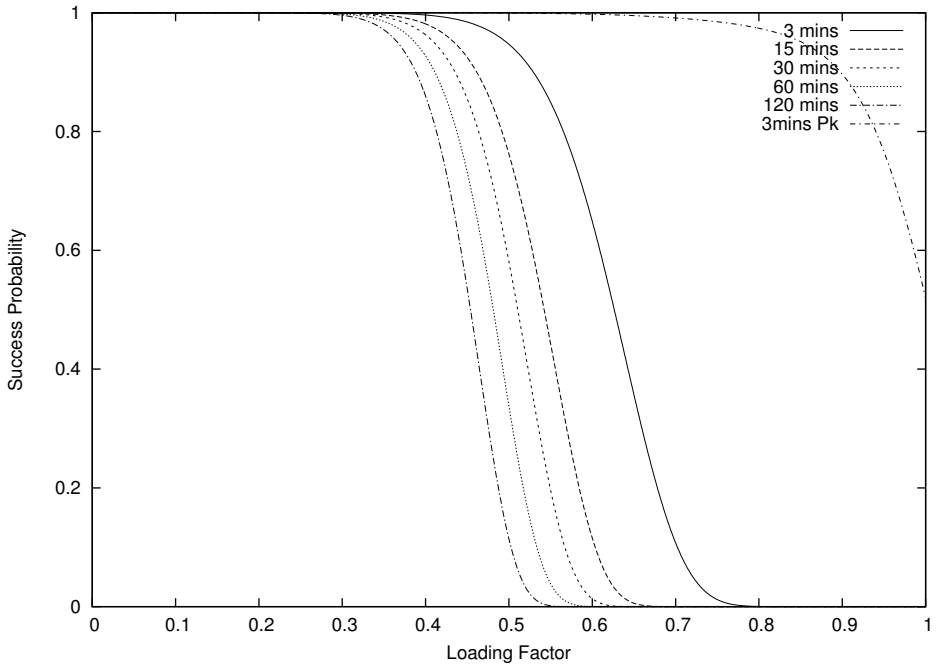


Figure 5.10: VoIP Failure Probabilities

the only flow using the queue. As you can see, as the queue approaches 50% loading the probability of a call succeeding drops dramatically. The line "3mins Pk" represents the success probability calculated using just the unconditional loss probability; which is the probability of the queue being full[3]. As you can see it seriously underestimates the loading factor at which VoIP calls will start to fail.

Calculating the failure probability in this way is useful to us as it allows us to determine what loading factor a VoIP call can tolerate. We can quickly see that to deliver VoIP successfully it can only pass through queues that are less that 50% loaded. This gives us a much better idea of what the Instantaneous Quality we must deliver to VoIP.

5.7 Summary

In this chapter we have developed the analytical tools that we require to solve the QoS problem using our methodology. We have covered the following areas:

- Convolution of queues numerically using the PASTA property.

[3]In reality there is a subtle difference between the probability of the queue being full and the probability of loss, since the former can occur without loss occurring. However, these are generally considered to be the same.

- Using exponential service to move the problem into a mathematically tractable domain.
- Queueing systems that can model queues which service packets of variable sizes.
- Methods for solving queueing systems, including: numerical convolution, automated Markov chain solving and Laplace transforms.
- Extending queueing systems to allow burst-losses to be modelled clearly and explicitly.

Chapter 6

The Testing System

6.1 A Haskell Packet Simulator

6.1.1 Introduction

This chapter introduces a discrete time packet simulator written in Haskell. Using Haskell allows the programmer to clearly express many forms of problem, including discrete event simulation. Our motivation for developing a packet simulator in Haskell was to produce a prototype that could be used as a basis for further work. Credit should be given to Neil Davies, and others as Degree2 Innovations, for their inspiration and code (see appendix B) for the Haskell simulator.

Ultimately the simulator proved too slow for long simulations, even after it was compiled using the Glasgow Haskell Compiler (GHC). However, many of the ideas presented here were used when developing the C variant of the simulator. What follows is interesting if only to demonstrate how Haskell can be applied to a number of different uses.

6.1.2 Time Dependent Evaluation

In any discrete time simulator it is important that events are processed in the correct order; in Haskell this is no different. Most programs that are written in Haskell use infinite lists, this simulator is again no exception. The problem comes when evaluating an infinite list; it is important to avoid any unnecessary references to the beginning of the list, which would cause the whole list to be stored. If the whole list is retained then the memory requirements of the program grow rapidly, which is clearly undesirable.

In this simulator we make time explicit in the design. Each value that is passed around has a time associated with it. Using this time we can correctly merge together a number of lists, while maintaining the correct order. To do this we define a type, called TimedThing, which associates a time with any type. Its definition is as follows:

```
type TimedThing a = (Time, a)
```

Using this type we can now merge two lists of timed objects together. As a bonus, because we use a polymorphic type, we can perform the merge without having to evaluate the data that is associated with the time. This is important because we do not wish to evaluate anything until it is necessary, this further helps the space efficiency of our program. The following function allow us to merge a number of lists of TimedThing into a single list of TimedThing.

```
timeMerge :: [[TimedThing a]] -> [TimedThing a]
timeMerge []
  = [] timeMerge [xs] = xs
timeMerge xss
  = timeMerge_ (timeMerge left) (timeMerge right)
    where
      splitPosition = length xss `div` 2
      (left, right) = splitAt splitPosition xss
timeMerge_ :: [TimedThing a] -> [TimedThing a] -> [TimedThing a]
timeMerge_ [] ys = ys
timeMerge_ xs [] = xs
timeMerge_ xs@(x:xs') ys@(y:ys')
  = if
      fst x <= fst y
    then
      x:(timeMerge2 xs' ys)
    else
      y:(timeMerge2 xs ys')
```

6.1.3 Simulating Networks

This section explains how we build on the above models of TimedThing to build a packet simulator. We start by defining a type to represent a packet. This type is then encapsulated by the polymorphic type TimedThing to create a TimedPacket. The type definitions for this are as follows:

```
type TimedPacket = TimedThing Packet
type Packet = (SourceId, PacketId, Time)
type PacketId = Int
type SourceId = Int
type Time = Double
```

Using the timeMerge function that we have defined earlier we can now correctly merge two flows of packets, called flow1 and flow2 of type [TimedPacket], into a new flow called flow3. The following snippet of code shows this:

```
flow3 = timeMerge [flow1, flow2]
```

The following subsections show how we: generate timed packets, queue them and finally how we measure the results.

6.1.4 Generating Packets

Packets are generated using the ExpStream module. To start the process off we create an infinite list of samples from an exponential distribution with parameter alpha. The following code performs this operation:

```
expStream :: Double -> Int -> [Time]
expStream alpha seed = [ id $! (f x) | x <- uniStream]
  where
    f x = - 1.0 / alpha * log (1-x)
    uniStream = randoms (mkStdGen seed)
```

Next we calculate a set of packet departure times by adding a new sample to the value of the last departure. The following code shows this:

```
poissonArrivalStream :: Double -> Int -> [Time]
poissonArrivalStream alpha seed = scanl1 (+) stream
  where
    stream = expStream alpha seed
```

Finally we generate an infinite list of TimedPacket from the poissonArrivalStream. These packets have the same flow Id, but have an incrementing packet Id starting from one. The following code shows this:

```
poissonTrafficSource :: SourceId -> Double -> [TimedPacket]
poissonTrafficSource srcid alpha
  = zip times (zip3 (repeat srcid) [1..] times)
    where
      times = poissonArrivalStream alpha srcid
```

6.1.5 Queueing Packets

The following subsection describes how we model queues in the simulator. It is based heavily on the SimulatorQueueModel module developed by Neil Davies (readers are directed to appendix B for more details).

First we define the initial state of the queue. This contains a state tuple of the items that are held on the queue and the list of service times. We also reference an arrival and departure function (which we will describe shortly). The following code does this:

```
init = Running {
  stateOf = ([], serviceStream),
  doArrival = arrive,
  doDeparture = depart
}
serviceStream = expStream mu seed
```

Next we describe the arrival function. There are two cases. The first is when the queue is empty. Here we simply add the arrival to the queue state. The second case is where the queue has at least one item in it. In this case we check that there is sufficient space for the arrival; if there is then it is added to the queue; failing that it is simply discarded. The following code implements this:

```
arrive ([], stream) arrival
  = ( init {stateOf = ([arrival], tail stream)},
      [], Just (id $! (head stream)))
arrive (q, stream) arrival
  = if (length q) < maxQ
    then ( init {stateOf = (q ++ [arrival], stream)}, [], Nothing)
    else ( init {stateOf = (q, stream)}, [], Nothing)
```

Finally we define the departure process. This is evaluated whenever there is an item on the queue. This function extracts the first item on the list of packets and the head of the service times list.

It then sets the state to be the same, but without these two values. Then it emits the packet at the head of the queue with a service time defined by the head of the service stream. The following shows this:

```
depart ((q:qs), stream)
  = if null qs
    then ( init {stateOf = (qs, tail stream)}, [q], Nothing)
    else ( init {stateOf = (qs, tail stream)}, [q], Just (head stream))
```

6.1.6 Calculating Statistics

To extract useful information from the simulator we have to build statistics. To do this we have the notion of an observer. This function takes a list of TimedPacket and generates an array with statistics for each flow, where the index to the array is the flow Id. We provide this function with the maximal flow Id that it will receive, so that it can construct the array; and the number of flows that are to be averaged, so we know when we have completed our task. The function has the following type signature:

```
aveFlows :: Int -> Int -> [TimedPacket] -> FlowArray
```

Finally we have a function that can print the FlowArray, its type signature is:

```
showAveFlows :: FlowArray -> String
```

6.1.7 Pulling it Together

The following function shows how to use the simulator to simulate a network with two queues in a chain. Two flows, with flow ids 1 and 2 respectively, enter the first queue. The first queue then feeds into a second queue. Finally the second queue feeds into the observer object.

```
main
  = let
    -- Configuration
    rate = 0.5
    buffers = 10
    -- Traffic Sources
```

```
    source1 = poissonTrafficSource 1 rate
    source2 = poissonTrafficSource 2 rate
    -- First Queue
    input1 = timeMerge [source1, source2]
    output1 = gm1k 1.0 buffers 3 input1
    -- Second Queue
    output2 = gm1k 1.0 buffers 3 output1
    -- Results
    result = showAveFlows $ aveFlows 2 2 output2
in
    putStr result
```

6.2 A QoS Test System

In this section we are concerned with the system that is used as a means of testing our end-to-end quality system. Throughout this thesis we present a number of sample network configurations; this test system provides a way of simulating, calculating and comparing any configuration we choose.

The motivation is simple: we wish to be able to predict the behaviour of a network accurately. To do this we need both the predictions, provided by the Calculator, and an example of the resultant behaviour, provided by the Simulator. These are then finally compared using the Comparator.

Ultimately we need a way of comparing simulations and calculations. The result of this system is a set of relative errors, which we will examine in later chapters (see chapter 7), between our predictions and a simulated network. The parameters that we are interested in are:

- Rate - the number of packets arriving in a given interval.
- Loss - the percentage loss of packets.
- Delay - the average end-to-end delay of packets.

6.2.1 Overview

Figure 6.1 shows a diagram of the overall test system. In later sections we will provide more detailed information on the design and implementation of the separate functional components; their function should be clear from their name. In this section we will look at the data files, specifically the Scenario and Results files.

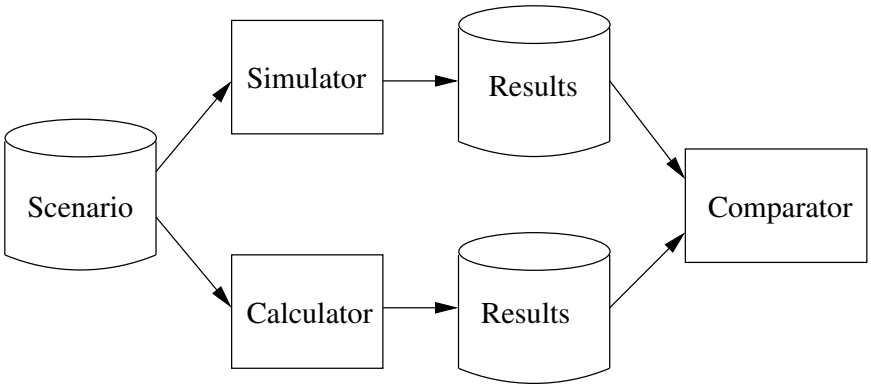


Figure 6.1: System Overview

The file format is based on XML. The decision to use this is because it allows us to use a number of standard tools to read and write the files. There is also a number of tools that allow transformations to be performed on the files, which allows for greater flexibility in the overall system.

6.2.2 Scenario Files

Scenario files provide us with a way of describing a sample network and the traffic that flows across it. A scenario can be seen as the result of a routing function applied to a topology, which describes the layout of the network, and an offering, which describes the traffic that will flow across that network. It is our intention to expand the system to allow this functionality in due time.

Below is an example scenario for a chain of queues topology. This particular topology has two queues, the first feeding into the second. A single traffic source, at a load of 0.5, crosses both queues.

```
<scenario>
  <title>Queue Chain, Rate(0.5), CLen(2), QDF(mm1k), K(10), Mu(1.0)</title>
  <date>Tue Aug 20 18:50:22 2002</date>
  <author>David C. Reeve</author>
  <source name="source1" sendsTo="router0">
    <flow id="1" rate="0.5" type="Poisson" lengthType="fixed" lengthMean="1"/>
  </source>
  <object name="router0">
    <port name="port1" qdf="mm1k" rate="1.0" buffers="10" sendsTo="router1">
      <select id="1" class="be"/>
    </port>
```

```
    </object>
    <object name="router1">
      <port name="port1" qdf="mm1k" rate="1.0" buffers="10" sendsTo="sink0">
        <select id="1" class="be"/>
      </port>
    </object>
    <sink name="sink0">
      <consume id="1" packets="10" samples="100"/>
    </sink>
  </scenario>
```

As you can see, the file is split into four main sections: preamble, traffic sources, objects and sinks. The preamble is included for informational purposes only, and is not used in any part of the system. We shall now describe the operation of the other three sections.

The source section provides a way of describing a number of flows that are present in the system. A flow is a label that is applied to a number of packets that are to receive the same treatment. In the source section there can be a number of distinct sources, defined as required by the user. Each source has a name, in order to distinguish it from other sources, and a name of an object that it sends its traffic to. We shall explain about objects in a moment.

Within each source there are a number of flows. These flows will be sent to the same location, as specified in the specification of the source. Flows are uniquely identified in the system by a number, which we call the FlowId. Along with this Id there are a number of parameters that describe the characteristics of the flow; specifically, the rate at which packets are injected into the network, the distribution of inter-packet times, the average length of the packets, and the distribution of the lengths (Poisson or Fixed). The flow can be one of the following types:

- Poisson - for Poisson inter-packet times.
- CPR - for Constant Packet Rate, ie. fixed packet intervals.
- ARS - for an adaptive rate source (see section 6.5.2.2).

The next section is the objects section. Objects are a simple abstraction that can be used to encapsulate a number of network elements. Each object has a unique name to distinguish it from other objects. An object has a number of ports, which describe how traffic leaves the object. We are assuming an output buffered model, but this does not exclude us from modelling other buffering semantics. Output queueing proved to be a simpler design than specifying both input and output ports, as it is closer to the way it is evaluated in Haskell.

A port is identified uniquely within each object, allowing the name to be reused. A port has associated with it a QDF, by which traffic is degraded. The service rate and the number of buffers per class are also described. The available QDFs in the system at the moment are as follows:

- gm1 - for a queue with infinite buffers and Markovian servicing.
- gm1k - for a queue with a finite number of buffers serviced Markovian.
- gd1k - for a queue with a finite number of buffers serviced Deterministically.
- WFQ - for a DRR queue.
- gos - for a loss-delay queue.
- delta - for the identity function[1].

The port also has a sendsTo, which contains the name of the next object or sink that the traffic will be sent to. As all traffic arrives to the object, a method of selecting what traffic leaves by a particular port is required. This is provided by the select statement. Select statements take two parameters: the flow Id of the flow that is destined to leave this port, and the class that it belongs to. The class describes how the QDF should treat the traffic. For example, in the case of the priority queue it will describe a strict priority queue that this flow should be admitted to. This allows us to support a number of different queueing disciplines.

The final section is the sinks section, which can contain a number of sinks. A sink describes where traffic leaves the network; it again has a unique name. Within each sink statement are a number of consume statements. These consume statements define the point in the network where statistics about a flow are collected. When packets are consumed with a continuous tracer (see section 6.5.2.5) the number of packets per sample, and the number of samples are also used.

We have shown the format of a scenario, now it is necessary to impose a number of rules to ensure that it is semantically correct. The same checks are performed by both the Calculator and the Simulator. These rules are as follows:

1. Source names must be unique
2. Sink and Object names must be unique
3. Port names, within an object, must be unique
4. Flow Id's must be unique
5. All sendsTo statements must be correctly routed

[1] The delta function has no effect on the traffic. That is it does not introduce delay, nor does it introduce any loss.

6. Each select statement must be able to select the given flow

7. All flows must eventually reach a sink

Unfortunately, the current implementation cannot handle routing loops; evaluating them requires iterating to a solution. As yet we have not investigated such methods, although an extension to this approach would be possible. It is worth noting that it is possible, with only a few minor alterations, to support one-to-many multi-cast. This is achieved by routing the same flow to more than one destination.

Multi-cast flows are well known for causing correlation problems in a network. This is because the flow of traffic is repeated throughout the network, other flows interacting with the multi-cast flow can become correlated with it. While we do not look at multi-cast issues in this thesis we do look at correlation issues (see section 7.4.3). Multi-cast flows require higher quality than other uni-cast flows, as there are a number of subscribers. Investigating both multi-cast correlation and quality issues are left as an area for future research.

6.2.3 Results Files

Once a scenario has been passed through the Calculator or Simulator it produces a results file. A results file describes the degradation that a flow has suffered by the time it reaches its destination sink. An example of a results file, from the Calculator is shown below.

```
<results>
  <meta>
    Created on: Tue Dec 17 21:49:52 GMT 2002
    Input File: test.scenario
    Modified On: Tue Dec 17 21:49:39 GMT 2002
    Directory: /home/dave/PhD/Software/Simulator
  </meta>
  <sink name="sink0">
    <flow id="1"
          arrival="0.49951267154762774"
          loss="9.746569047445197e-4"
          delay="3.979520790521888">
      <deg point="router1.port1"
           rate="0.49951267154762774"
           loss="4.8637472336850277e-4"
           delay="1.9892959615873818"/>
```

```
        <deg point="router0.port1"
             rate="0.49975574010747437"
             loss="4.885197850512946e-4"
             delay="1.9902248289345061"/>
      </flow>
    </sink>
  </results>
```

There is an important difference between the results produced from the Calculator and the Simulator. That is the results from the Simulator do not contain information about the degradation at intermediate points, only the degradation that has been suffered overall. It was decided to take this approach as we are far more interested in the final degradation after the flow has crossed the network than we are at intermediate points. This is however different in the case of the Calculator where we wish to know as much information about the network as is possible.

Results files contain two sections. The first is some meta information, which like the Scenario file, can be used to store any additional information about the file that is required by the user. The second is the set of sinks found in the corresponding scenario file.

Each sink is identified by name, which is the same name specified in the scenario file. Within each sink are a number of flows, again identified by their flow Id. As we mentioned earlier, this system can cope with multi-cast and as such we need to distinguish between the same flow arriving at a number of different sinks. For each flow we record the arrival rate, loss probability and average end-to-end delay of the flow in question.

In the case of the Calculator we also record the degradation along the way. This is achieved using the deg tag, of which there will be one per port that the flow has traversed. The degradation records the point at which the flow was inspected, by convention this is "object.port", and the same overall quality parameters.

When results are collected using a continuous tracer in the simulator the results are presented in a different format. For details on this see section 6.5.2.5.

6.3 The Scenario Library

The scenario library is responsible for loading scenario files into an intermediate data structure and performing the sanity checks listed above. It is common to both the Calculator and Simulator. Additionally it provides a way of storing the results into a results XML file. Finally some types are included that are used throughout for convenience.

The library is written in Haskell, a lazy evaluated function programming language with a strong type system. The decision to use Haskell was motivated by its lazy evaluation, which removes the need to worry about the order of evaluation. Writing the same functionality in C would require significantly more code, which would undoubtedly obscure the semantics.

6.3.1 The Types

Throughout the following type definitions are used, which simplifies the design. We start by defining some simple types for the quantities that are used when describing various names and measures. These are as follows:

```
type Name = String
type FlowId = Int
type FlowRate = Double
type FlowType = String
type Loss = Double
type Delay = Double
type MeanLength = Int
type LengthType = String
type ServiceRate = Double
type Buffers = Int
type NumSamples = Int
type NumPackets = Int
```

Next we define what a flow is. Confusingly a flow is a list of FlowToken, where a FlowToken specifies the lowest level of detail about a flow that we are interested in, that which is uniquely identified by a FlowId. At any point in the network the Flow will be arriving at a given rate (FlowRate) and with a given distribution (FlowType). It also has a mean length (MeanLength), and a length distribution (called a LengthType). Finally a Flow has a list of degradations, initially set to the empty list, to store degradations that occurs as it passes through the network. A Degradation simply records the location, defined by the type PortRef, and the rate, loss and delay of the flow at that point in the network.

```
type Flow = [FlowToken]
data FlowToken = FlowToken FlowId FlowRate FlowType
                  LengthType MeanLength [Degradation]
data Degradation = Degradation PortRef FlowRate Loss Delay
```

Flows of traffic are always degraded by a Quality Degradation function, as specified in the port tag in the Scenario XML file. The degradation a flow receives depends on the policy that the arriving traffic is subjected to. The precise value of the policy depends on the QDF that is in use. This is essentially where the information in the class attribute from the select tag resides. QDFs are defined as a function, which when given a PortRef and a list of (FlowToken, Policy) returns a new flow that has been degraded.

```
type Policy = String
type QDFName = String
type QDF = (PortRef -> [(FlowToken, Policy)] -> Flow)
```

Next the various parts of the Scenario are described in a number of types. These types capture Sources, Objects, Ports and Sinks. The overall Scenario is defined as a list of these types.

```
data Scenario = Scenario [Source] [Object] [Sink]
data Source = Source Name Flow DestinationRef
data Object = Object Name [Port]
data Port = Port Name QDFName ServiceRate Buffers
             [(FlowRef, Policy)] DestinationRef
data Sink = Sink Name [FlowRef]
```

To allow other parts of the Scenario to be referenced we define a number of reference types, these capture the name of a destination object. We use them to keep the type signatures of various functions clean, so as to make them clearer to the user.

```
data FlowRef = FlowRef FlowId
data SourceRef = SourceRef Name
data ObjectRef = ObjectRef Name
data PortRef = PortRef Name ObjectRef
data SinkRef = SinkRef Name
data DestinationRef = DestinationRef String deriving (Eq)
```

6.3.2 The Loader

To make the task of loading and saving files in XML easier the HaXml library is used. The HaXml library provides a way of automatically producing functions to load and save any XML file from its DTD file. The program used is called DtdToHaskell. As its output it produces a Haskell module that can load and save the XML file, as well as a set of types to encapsulate the information.

Unfortunately the compound type that is produced by DtdToHaskell is not exactly as we would like it to be; specifically it treats attributes of an XML element as a separate type, thus complicating the resulting data structure. For example an Object would be parsed into the following data types:

```
data Object
  = Object Object_Attrs [Port]
      deriving (Eq,Show)
data Object_Attrs
  = Object_Attrs { objectName :: String }
      deriving (Eq,Show)
```

For convenience this is re-parsed into a new data structure that flattens out the data structure. In the case of the above example this is stored in the Object data structure described above. Once the data has been moved into the new data structure the sanity check is performed. This is essentially the list of checks that are outlined in the previous section.

6.4 The Flow Calculator

The Flow Calculator, as the name suggests, provides a way of calculating the degradation that a set of flows will receive as they traverse a network of QDFs. Its functionality is broadly split into two sections, those responsible for calculating and applying the degradation and those responsible for evaluating the network such that the former can happen. Additionally it uses the Scenario support library to load the Scenario and to save the results of the calculations.

6.4.1 Evaluating the Network

The network is evaluated from sink to source. As Haskell is lazily evaluated this is a somewhat easier process, as most of the hard work is performed by the way Haskell is evaluated. Each sink is evaluated in turn, which causes each flow in each sink to be evaluated and so on.

When a sink is evaluated we attempt to find all of the objects that send a particular flow to the sink. The extractFlow function finds all of the FlowTokens that are sent to this particular sink. We construct a DestinationReference of the sink name we are interested in to create a filter. Once we have all of the flows that send to this sink we then filter only the flows that we are interested in. The function selectFlows then ensures that all of the required flows are available and that no more are present. The following snippet of code shows this.

```
evaluateSink :: Scenario -> Sink -> Flow
evaluateSink scenario (Sink sinkname flowrefs)
  = selectFlows sinkname flowrefs $ extractFlow scenario destref
    where
      destref = (DestinationRef sinkname)
```

When we extract flows matching a DestinationRef, using extractFlow, we have to evaluate all of the objects and sinks. However, as Haskell is a lazy evaluating language, only as much of the graph is evaluated as is necessary to return a result. Essentially what we are doing is to find all the Sources and Sinks that send to a specific destination, and then evaluate the Sources and Sinks to yield the Flow (which is a list of FlowToken) that would emerge.

Evaluating a Source is a simple process. All we need to do is to return the list of FlowTokens contained within that sink. As the following piece of code shows:

```
evaluateSource :: Scenario -> Source -> Flow
evaluateSource scenario (Source name flow dests) = flow
```

Evaluating an Object is a little more involved. In extractFlow, when extracting flows from an Object, we call evaluatePort for each of the Ports in that Object. Objects get their input flows from other Objects or Sources, so we must first extract and select the flows that arrive in the same way we do for a Sink. However, this time we extract all the flows that arrive at the Object, and only select those flows that leave from the given port. Once we have the list of flows that depart from a port we then apply the QDF to the Flow to yield the output Flow. While this description is somewhat brief the code below should expose some more of the finer detail.

```
evaluatePort :: Scenario -> Object -> Port -> Flow
evaluatePort scenario
             (Object oname ports)
             (Port pname qdfn rate buffs refplcys destref)
 = outputflow
     where
       estr       = (oname ++ "." ++ pname)
       destref    = (DestinationRef oname)
       inputflows = selectFlows estr flowrefs $ extractFlow scenario destref
       flowrefs   = map fst refplcys
       portref    = (PortRef pname (ObjectRef oname))
       qdf        = getQdfByName qdfn rate buffs
       outputflow = qdf portref (map f refplcys)
       f (flowref, plcy) = ((selectFlow estr inputflows flowref), plcy)
```

In essence the whole process involves evaluating the Object or Source that sends to this Sink or Object. Once this has happened, we can then perform the operation of this component. When Sources are evaluated they return a Flow as defined in the Scenario, and are therefore terminals in the evaluation. When Objects are evaluated they must first evaluate those Sources or Objects that send to them (which in turn causes the evaluation of their up-stream Objects and Sources) and then degrade the Flow. Finally, Sinks provide a handle to start the process.

6.4.2 QDFs and Degradation

Throughout the Calculator QDFs are distinguished by a string that determines what function to apply. Whenever we refer to a QDF we do so by this string. There is a simple mapping that returns a function, of type QDF, that will perform the required degradation. This allows us to pass some additional arguments, such as number of buffers and service rate to the real QDF function before returning it.

For each type of QDF, say an M/M/1/K queue, we have a function that allows us to calculate the degradation and apply it to the flows that traverse it. Below is an example of one such function.

```
mm1kQdf :: ServiceRate -> Buffers -> PortRef -> [(FlowToken, Policy)] -> Flow
mm1kQdf mu k portref tknplcys
  = outflow
    where
      inflow = map fst tknplcys
      lambda = totalArrivalRate inflow
      loss = mm1kLoss lambda mu k
      delay = mm1kDelay lambda mu k
      outflow = map (applyDeg portref loss delay) inflow
```

The operation of these functions follow a common theme. In this example we do not show how more than one class of service can be accommodated, although the steps are the same for each of the classes. The steps are as follows:

- Find all of the flows that we are interested in degrading.
- Calculate the total arrival rate, which is simply the sum of the arrival rates of all the separate FlowTokens.
- Calculate the loss and delay that this type of queue would introduce.

- Apply the degradation to each of the input FlowTokens to yield the output Flow.

The function applyDeg adds a new Degradation to a FlowToken. This function calculates the departure rate from this QDF and stores it in the ArrivalRate field of the FlowToken. Additionally it adds a new Degradation to the list of Degradations in the FlowToken to store the effects of this QDF. The following code performs this function.

```
applyDeg :: PortRef -> Loss -> Delay -> FlowToken -> FlowToken
applyDeg portref loss delay (FlowToken fid ratein ltype lmean degs)
  = (FlowToken fid rateout ltype lmean (deg:degs))
    where
      deg = (Degradation portref rateout loss delay)
      rateout = (1-loss) * ratein
```

6.5 The Flow Simulator

The Flow Simulator provides a way of doing packet based discrete event simulation. It is split into three major constituents. The first provides a set of support functions for timer manipulation and packet routing. The second is a set of components written using these functions, and finally a Haskell based loading interface.

6.5.1 The Interfaces

The Simulator provides two sets of support functions. The first is for manipulating timer events, and their associated call backs. The second is a way of routing packets between the components in a way that allows us to incrementally build a network from scratch.

Overall the simulator is run by calling simulation_loop. This function simply runs through the timer queue waiting for the next event to occurs. Time is moved forward to the next event located at the head of the timer queue. The simulation will keep running until the global variable g_simulation_loop is set to zero.

6.5.1.1 Time

In the simulator time is represented using a Double, which always increases. It is expected that the clock will wrap around in some situations. This is not a problem and a number of macros have been provided to protect against this. At any time it is possible to retrieve the simulator time by using the current_time global variable.

6.5.1.2 The Timer Queue

In order to simulate a large number of queues at speed it is necessary to process events quickly. This is especially important when the queues are loaded, as there will be more events waiting to time out. As the Simulator spends a fair proportion of its time processing the timer queue it is worthy of optimisation.

The timer queue uses a heap-ordered tree, which is sorted on insert. The insertion time is O(log n) where n is the size of the queue at the time of insertion, and repeated insertions tend to balance the tree. The tree does not guarantee to remain balanced as elements are removed from it, as such the removal time is O(n log n).

Events may be scheduled at any time in the future; events that have already passed cause an error to be raised. Details of the event time and the call-back function are stored in the event structure. In order to allow state information to be passed when the event is triggered, the event structure can be embedded in another structure. By use of the timer_event_object macro we can extract the pointer to the original structure.

We can define a structure for a imaginary queue using the event structure as follows. The example also contains examples of the pins interface that we will explain shortly.

```
struct queue_internals {
    struct pin *input;
    struct pin *output;
    int length;
    int occupancy;
    struct timer_event event;
    char *packets[];
}
```

Assuming that we have allocated ourselves a new structure, of type queue_internals, called queue we can do the following. First we must set the name of the function that is to be called at the event time, this need only be done once when the component is initialised. Next we need to set the time that the event is to trigger, here we assume that some_time is a time in the future. Finally we call timer_event_insert with a pointer to the event structure.

```
queue->event.function = queue_callback;
queue->evant.event_time = some_time;
timer_event_insert (&queue->event);
```

When the event is triggered the callback function is called with the timer event. If we wish to retrieve the original data structure then we can use the timer_event_object macro which is passed: the event structure that the function was called with, the type of the structure that we want to extract, and the name of the element that the event structure is stored in inside the data structure. Once this has been done we can proceed as normal.

```
static void queue_callback (struct timer_event * event) {
    struct queue_internals * queue
        = timer_event_object (event, struct queue_internals, event);
    /* more code here */
}
```

6.5.1.3 The Pins Interface

The Pins interface provides a generic way of connecting a packet producer and consumer. This interface provides the basic plumbing to join all of the components in a network together. It also allows one producer to send to many consumers by copying the packets as they pass across a pin. In essence it maintains a list of consumers and, when passed a packet by a producer, it passes the packet to those consumers.

A new pin can be created as follows:

```
struct pin * my_pin = pin_create ();
```

Once a pin has been created there are two operations that we can perform on it. The first, and most simple, is to send a packet to it using the pin_put_packet functions shown below.

```
void pin_put_packet (struct pin * pin,
                     struct packet * packet);
```

The second function is to attach a sink. This function, shown below, takes the following arguments: the pin that the consumer (or sink) is to be attached to, a function to call when a packet arrives, and some additional data which is defined by the user. Also shown below is the definition of pin_sink_function.

```
void pin_attach_sink (struct pin * pin,
                      pin_sink_function function,
                      void * data);
typedef void (*pin_sink_function)
             (void * data, struct packet * packet);
```

6.5.2 The Components

There are three main types of component: generators that are responsible for creating packets, queues that are responsible for implementing various types of queues, and tracers that are responsible for gathering statistics about various flows.

6.5.2.1 Generators

Generators are by far the simplest of the components. They use a structure to store their state, such as the flow Id and rate. Below is in an example of the generator for Poisson traffic:

```
struct poisson_generator {
   struct timer_event event;
   struct pin * pin;
   double mu;
   long flow;
   long seqno;
   int length_type;
   int mean_length;
};
```

When a new generator is constructed, by say add_poisson_generator, a new structure is allocated and filled. Next we calculate an inter-packet time, this is generator specific, and set an event for now plus that time.

```
generator->event.event_time = current_time + ipt_time;
timer_event_insert (&generator->event);
```

When the event fires the event function is called which, using pin_put_packet, emits a packet. The above lines are then repeated so that the next packet is emitted.

6.5.2.2 Adaptive Sources

Adaptive sources, as presented here, are intended to model the behaviour of a simple elastic-source. That is sources that attempt to utilise as much as the available bandwidth of the network as possible. One common example of an elastic-source is TCP; while some details of the adaptive

source we present here resembles TCP it is not our intention to model it in any detail. While the behaviour of our adaptive source is not exactly the same as TCP it does capture an important part of the behaviour, namely the source will attempt to use as much bandwidth as possible.

The adaptive sources work by modifying their sending rate dependent on the observed behaviour of the flow at the destination. To achieve this packets are intercepted by the source at the destination. Both the end-to-end delay and loss events are collated when making decisions about adapting the sending rate; more details on this shortly. When the sending rate is modified it is done so instantaneously, without incurring and end-to-end delay (as for example TCP would). This is for simplicity and also because we do not have a reverse path in the simulator.

When the source is started it does so with an initial rate. This initial rate is used to calculate the inter-packet time for the source. The source always sends packets spaced by the inter-packet time stored in its internal data structure. This structure is modified by a receiving process to update the rate, however, the rate is not changed until 100 packets[2] have been collected to ensure that the averages are correct. The initial rate is set as close to the expected steady running rate of the source as possible. Protocols such as TCP do not do this, instead they use slow-start algorithms to reach a steady rate. We have chosen to avoid slow-start for simplicity, and more importantly to allow the network to reach a steady state faster.

The source performs the following steps when packets arrive at their destination:

- Performs the end-to-end delay calculations (see below).
- Checks to see if enough packets have been collected to start adapting, if not no further processing is done.
- If the window size has not been set, set it to the number of packets currently in the network.
- Adjust the window size (see below).
- Set the inter-packet time of the sender.

The inter-packet time of the sender is set as the delay divided by the window size. This is a rearrangement of the bandwidth-delay product formula, shown below. The bandwidth delay product gives us a measure of the expected volume of data, measured here in packets, in the network. Initially we know both the delay and the number of segments in the network, this allows us to calculate the inter-packet time.

$$window = bandwidth * delay$$

[2]The choice of 100 packets was considered 'enough' to arrive at a steady round trip time after looking at results from the simulator.

$bandwidth = \frac{window}{delay}$

$IPT = \frac{delay}{window}$

The delay calculations are preformed by the following snippet of code. The calculations are taken from Van Jacobson's Congestion Avoidance and Control Algorithm [52] used in TCP.

```
delay = (double)time_to_seconds(current_time - packet->initial_timestamp);
/* Check that this is not the first measurement, if not update the delay */
if (generator->delay < 0) {
    generator->delay = delay;
} else {
    error = delay - generator->delay;
    generator->delay = generator->delay + (0.125 * error);
}
```

The size of the window is adjusted depending on the following two rules:

- If two packets in a row are lost then the window size is halved.
- If eight packets in a row are successfully received then the window is doubled.

6.5.2.3 Queues

Queues are the second most complex component. They maintain their state in a structure similar to queue_internals; shown above. When a new queue is added to the network this structure is allocated and its values are defined. This includes setting a timer event function and a sink function. The former is used for departures and the latter for arrivals, which call queue_timer and queue_arrival respectively. The following code shows this:

```
queue->event.function = queue_timer;
pin_attach_sink (input, queue_arrival, queue);
```

The queue_arrival function is called when a packet arrives at the input pin. It performs the following operations:

- Checks if the queue is full, if so discard the packet and return.

- Enqueue the packet on the rear of an internal list.
- If the queue was previously empty, schedule a departure event.

Note that we only schedule a departure event if the queue was previously empty, and the arriving packet causes it to be no longer; otherwise we rely on the departure function to schedule the departure events.

The queue_timer function is responsible for emitting packets when their service time has elapsed. It is called whenever the timer event expires. It performs the following operations:

- Emit the packet at the head of the queue.
- If there are no packets left in the queue then return.
- Calculate the service time of the packet at the head of the queue.
- Schedule a departure event for now plus the service time.

Note that in normal operation there is always a packet at the head of the queue when the timer function is called. This is because we only insert as many timer events are there are packets that arrive at the queue.

6.5.2.4 Tracers

Tracers are responsible for producing statistics about a flow that terminates at its input. One tracer will only collect statistics about a single flow Id, and will ignore packets with any other flow Id. To maintain their state they use the following structure:

```
struct packet_tracer_state {
   char *name;
   struct pin * pin;
   int flow;
   int state;
   unsigned long first_seq;
   Time last_arrival;
   unsigned long count;
   double delay_sum;
   double ipt_sum;
};
```

When a tracer is first started it will skip 10,000 packets, giving the queues enough time to settle. From this point onward it will start collecting statistics. When the next packet arrives its sequence number is recorded in first_seq, count is incremented and last_arrival is set to the current time.

As each packet arrives the following steps are performed by the tracer:

- Check that the packet has the correct flow Id.
- Increment count.
- Calculate the delay of the packet, and add it to delay_sum.
- Calculate the inter-packet time, and add it to ipt_sum.
- Set last_arrival to be the current time.

When count reaches the desired value, in this case 100,000, we stop the tracer; in this state no further statistics are collected. Additionally if it was the last tracer active then it stops. Then it causes the main simulation loop to be broken such that the results can be stored. From this state we calculate the average delay, rate and loss probability. The following code performs this operation:

```
rate = 1.0 / time_to_seconds(state->ipt_sum / state->count);
delay = (double)(state->delay_sum / state->count);
loss = 1.0 - (
              ((double)state->count) /
              ((double)(state->last_seq - state->first_seq))
             );
```

6.5.2.5 Continuous Tracers

Continuous tracers work much like normal traces other then they produce a continuous output and not an averaged result. They are used when more detailed information about the behaviour of a flow is required. A continuous tracer takes a number of packets and produces a line of output. The number of packets per line of output is defined in the consume statement of the scenario, found in a sink. A continuous tracer is marked as completed when it has collected enough samples, this is also defined in the consume statement. Although a continuous tracer is marked as completed it will continue to produce results until all of the other tracers have completed; this allows us to collect data for all flows over the same time period. Additionally a continuous tracer does not skip a predefined number of packets, this allows us to gather information about the startup conditions in the network.

The output from the continuous traces produces a space separated text file with one line per result, it has the following fields:

- The name of the sink that the result was collected from.
- The flow Id that the result belongs to.
- The time that the first packet in a result was collected.
- The time that the last packet in a result was collected.
- The sequence number of the first packet in a result.
- The sequence number of the last packet in a result.
- The sum of packet delays.
- The sum of packet delays squared.
- The sum of inter-packet times.
- The sum of inter-packet times squared.
- The number of packets that were lost in the result.

The result file is parsed after the tests has completed to compute the average and variance of: delay, inter-packet time, and loss. As we have intermediate results we can compute a instantaneous result as well as long term running averages.

6.5.3 The Processor

The Simulator processor is responsible for constructing a scenario in the Simulator. It is, in fact, written in Haskell allowing it to take advantage of the Scenario Library. In order to construct the scenario in the Simulator it is necessary to call C functions to add pins and components to the state. This is achieved using the green-card [86] library.

The green-card suit takes a specification and automatically produces marshaling code, allowing the user to call C functions from within Haskell. An example of one such specification is show below:

```
%fun addPoissonGenerator
%  :: Pin -> Double -> Int -> LengthTypeEnum -> Int -> IO ()
%call (pin output) (double service) (int flowid)
%       (lengthTypeEnum length_type) (int mean_length)
%code add_poisson_generator (output, service, flowid,
%       length_type, mean_length);
```

This produces a function in Haskell called addPoissonGenerator which calls the C function add_poisson_generator with the same arguments. The return type from the Haskell function is IO(); this means that the function is side-effecting but returns no result. Similar specifications exist for all of the functions that we wish to call from Haskell.

To process a scenario we first use the Scenario Library to load the scenario into a convenient data structure. From here we need to call the marshalled C functions in the correct order to build the network, run the simulation, and collect the results. First we create a list of named pins, which are a tuple of String and Pin, for each of the objects and sinks in the scenario. Then we process all of the sources, objects and sinks in order. The following code is the top level function:

```
processScenario :: Scenario -> IO ()
processScenario (Scenario sources objects sinks) =
  let {
    objectnames = map objectName objects;
    sinknames = map sinkName sinks;
  } in do {
    namedpins <- mapM createNamedPin (objectnames ++ sinknames);
    mapM (processSource namedpins) sources;
    mapM (processSinks namedpins) sinks;
    mapM (processObjects namedpins) objects;
    return ();
  }
```

The function processSource finds the NamedPin that it sends to, found by the DestinationRef in the Source data type, and creates a new generator to this pin. It is as follows:

```
processSource :: [NamedPin] -> Source -> IO ()
processSource namedpins (Source name flow (DestinationRef dest)) =
  let {
    destpin = getNamedPin namedpins dest;
  } in do {
    mapM (createGenerator destpin) flow;
    return ();
  }
```

In essence the process is the same for processSinks and processObjects. That is we find the required pin and attach, using the marshaled C functions, the desired component to it. Once this has been completed for the whole network a special marshaled C function, runSimulation, is called which calls the simulation_loop() function in the core of the Simulator.

In much the same way the results are returned from the C part of the Simulator and saved into a results file using the Scenario Library.

6.5.4 Random Numbers

When we simulate a particular scenario a number of random numbers are used. This is to make sure that the results are meaningful, and not just an artifact of the random number seed that was used for a single run. In the next section we will elaborate more on the effects of the random number generator.

Each scenario is simulated one hundred times using different random seeds. The first fifty of these seeds have been pre-generated and are stored in a file; these seeds remain constant throughout all of the tests. The second set of fifty numbers are generated at the time of execution, and are stored along with the results as the test runs. The first set of numbers gives us a way of repeating the tests, and the second set of numbers insures that there is no bias in the first set of numbers.

In the simulator the MT19937 generator of Makoto Matsumoto and Takuji Nishimura is used. It is a variant of the twisted generalised feedback shift-register algorithm, and is known as the "Mersenne Twister" generator. This is provided as part of the GNU Scientific Library (gsl).

6.6 The Comparator

The comparator is responsible for comparing the results between the Simulator and the Calculator. It compares a calculation result file, from the Calculator, with a number of results files from the Simulator. From these it produces the following results for delay, rate and loss:

- Mean result from the Simulator.
- Absolute error between the mean Simulator result and the Calculator.
- Percentage relative error between the mean Simulator result and the Calculator.
- Variation in Percentage error of each Simulator result compared to the Calculator.
- Standard deviation of the above.

The results are printed in a plain text suitable for processing with awk, sed, and finally gnuplot.

The comparator works by joining all of the Sinks and then Flows found in a number of scenarios. From there it can then work out the mean value for the loss, delay and rate. These are then compared to the result that the Calculator produced.

6.7 Summary

In this chapter we have presented a QoS test system which we can use to investigate the accuracy of our QoS methodology. The test system comprises of two main components, the Simulator and the Calculator. These components rely on a custom library, the Scenario Library, which provides basic functions for loading and saving data in a common XML format. The Simulator is able to perform a packet based simulation of a network and produce results at varying levels of detail. The Calculator uses mathematics from queueing theory to predict the behaviour of a network. The results from the Simulator and Calculator can be compared to check the accuracy of the methodology. In the following chapters we will use this system to investigate our ability to predict the behaviour of networks.

Chapter 7

Predicting Networks

7.1 Introduction

In this chapter we will present a comparison between our results from the Simulator and Calculator. The aim is to show that we can predict, to some level of accuracy, how a given network scenario will behave; that is the calculated parameters match closely those that are gathered from simulations.

All of the results presented in this chapter are for networks that do not provide differentiated service. This means that all packets will be treated the same by the queues in the network. The motivation is to show that it is possible to construct a network that behaves as closely as possible to a sound mathematical model.

Throughout this chapter we will use a number of simple test cases. These are designed to exercise well known problems in networks. In total three test cases are presented. These are explained in section 7.4.

In addition to the three test cases we have three different levels of network simplification. At each step we introduce more factors that could affect the accuracy of the results. Sections 7.5, 7.6 and 7.7 deal with each type of network simplification in turn. The different simplifications are:

- Network with packets modelled as point processes.
- Networks with sequences of equally sized packets.
- Networks with sequences of mixed sized packets.

Finally in section 7.8 we draw some conclusions from these tests.

7.2 Expected Outcome and Constraints

Before we present the results in this chapter it is important to understand what our expectation of the results are, and what these results are constrained by. The following points highlight some important facts about our results.

1. Mathematical reality and simulation reality are not the same. In creating the simulator we have attempted to stay as close to the mathematics as possible, however, differences are likely to be introduced.
2. Where differences between mathematical prediction and simulation exist those differences are: bounded and predictable.
3. The sources of the inconsistencies can be analysed using the the mathematical formulae themselves.
4. At one level the inconsistencies can be explained by the nature of random number generators and their property of short term bias.
5. There are other properties that we could see having an effect; we can see them and outline ways of understanding their bounds and predictability.
6. Errors do not make the system unpredictable. They imply that you either have to accept that the answers have an associated error or that you need more information/work to characterise and reduce the error to an acceptable level.
7. All this means that predictability is possible; however, it is a prediction that comes with an associated level of confidence.

7.3 Testing Methodology

7.3.1 Subject of the Tests

The purpose of this section is to describe our approach to testing the framework. We have implemented the framework using a simple model of queues based on Queueing theory using the Poisson distribution. This provides us with the following:

- A mathematical basis, derived from queueing theory, from which we can begin our investigation.
- A theory that allows us to compose these queues to model a given network topology.

- A method for calculating the predicted performance of a network configuration in terms of average delay, throughput and loss ratio.

In essence we can decompose a given, perhaps real world, network topology into a connected set of queues. From this we are able to predict, through calculations, what the performance of this network will be. This can then later be compared to a set of simulation results to evaluate both the usefulness of our approach as well as its accuracy.

There are a number of assumptions that have been made while constructing this framework. It is our aim to show that while these assumptions do affect the accuracy of the results, they do not invalidate the usefulness of the approach in general. There are a number of deviations in implementing the approach; these are as follows:

- The mathematics is based on the Poisson distribution; however, in real world networks packets in a network are not permitted to overlap[1]. We do not therefore strictly use a Poisson distribution in the simulator.
- The packet lengths, when using Markovian servicing, should also be Poisson distributed. This is again unlikely to happen in any real network; we therefore restrict each flow to a fixed size of packets[2].
- We are assuming that the traffic arrives at the edges of the network with Poisson distributed inter-packet times. This can, however, be achieved using a shaper.

The use of a shaper is an important requirement in the way we have implemented the framework. Using our framework we can calculate the effect on a flow that a shaper introduces. However, as it is not always possible to know what the input pattern is, it is not always possible to accurately predict the outcome. Conversely, where we do know the input pattern, say in the core of the network, it is possible to accurately predict the effect on the flow.

The framework is useful for modelling networks that accept a number of merged flows, with differing service requirements, at its input. While this does not apply to a large number of networks it is sufficient to demonstrate the usefulness of the approach in general.

One restriction is that flows must contain packets that are equally sized. This clearly does not reflect the mix of packet sizes found in a real network. It is the hope of the author that, in time, it will be possible to model trains of packets, where the tailing packets have a smaller size; this is left for future investigation.

[1] Queueing theory generally deals with point processes, which exist in a single instance of time. Packets can be modelled as a point process when you consider the arrival (or departure) of the head (or tail). However, in real systems, packets cannot overlap but mathematically we have not taken into consideration the length.

[2] We do however allow flows of different sized packets to mix together in the network.

Given the deviations above we have a number of assumptions that we hope will hold true when tested. These are as follows:

- Exponentially distributed inter-packet times should closely resemble a Poisson distribution, despite the fact that the queues are finite.
- By modelling queue service distributions as generally distributed (to a given mean), when dealing with multiple sizes of packets, the length of the packets should not affect the predictions from the calculations[3].
- As traffic traverses the network, and is subjected to more Poisson service times, it will approximately become more Poisson distributed.

This means that we use the mathematics to model Poisson inter-arrival times, whereas in reality we are simulating something close to Poisson inter-packets times. Similarly we model departures as generally distributed, but we service packets using a exponentially distributed service rate (the service time is therefore this rate multiplied by the packet length).

We conjecture that our assumptions will hold, allowing us to use the mathematics to make predictions, so long as we introduce enough randomness in the system. By randomness we mean to service packets with an exponentially distributed rate, approximating the maths as closely as possible.

7.3.2 Aims and Motivation

Before we can explain how we are going to test this approach it is necessary to define what the high level outcome of this approach is. In essence we are aiming to design and build predictable multi-service networks. In this context we take predictable to mean:

- We have the ability to predict the emergent behaviour of the network through calculations.
- The network must behave in a predetermined way when subjected to sudden changes in loading.

Predicting the behaviour of a network under a given set of conditions is almost a necessity when it comes to providing guarantees. However, predictions are next to useless unless they accurately reflect the performance that would actually be observed. With accurate guarantees it is possible to provision network resources correctly, and, as a result, provide and meet guarantees to the end users.

By attempting to stay as close as possible to a Poisson distribution we gain two important effects:

[3]Other moments, such as jitter, may be more inaccurate.

- The mathematics for predicting the behaviour of the network is simple, and is easily computable.
- We avoid any serious synchronisation problems inherent in deterministic systems (see section 2.2.1).

The second point is extremely important. In deterministic systems there are two places where predictions of behaviour fail. The first is where there is a time dependent synchronisation (or phase relation) between two streams, and the second is where the load of the network approaches 100%. In the former it is impossible to predict at which moment one of the two streams will gain a better treatment. In the latter, the introduction of loss into the calculations make them extremely complex and in some cases intractable. Our approach avoids both of these two problems.

Here we are attempting to give guarantees that are sufficiently flexible that we can meet them, while at the same time still being useful to the application that wishes to use them. At the same time we wish to protect flows for which we have provided guarantees from those for which we haven't, or that are out of contract.

7.3.3 Criteria for Success

Defining the criteria for success is extremely difficult, and highly dependent on the application that requires the guarantee. As we have already explained, applications can tolerate degraded flows until they cross some threshold; this threshold is application dependant. If a flow is received with a lower degradation than this threshold, then we can consider it to be successful. Here we are more interested in evaluating how accurate our assumptions are. However, we should always evaluate these assumptions with their final purpose in mind; namely the provision of acceptable levels of quality to applications.

Ultimately the results from the simulator and the calculator provide us with an average delay, throughput and loss ratio. From here we can then compare the two and find the absolute and relative (percentage) difference between them. The question is how close is close enough? As we saw in section 4.7 the following bounds on performance are considered to be acceptable.

- Delay should be accurate to less than one packet service time.
- Loss should be accurate to the same order of magnitude.

What is perhaps more important is that the relative classifications of the flows should be maintained. That is to say, that in all cases a flow that has a lower predicted loss should always get a lower actual loss. This is also the same for delay.

7.3.4 Expectations and Directions

In this thesis there is an underlying theme, namely, to make the networks behave in a way that closely follows a sound mathematical model. It is for this reason that we have chosen to use the Poisson distribution, as it is the most well modelled in queueing theory. However, this does not preclude us from using other distributions in the future, especially where other distributions are likely to result in better packet by packet performance. For the time being we are more interested in the behaviour of the system as a whole.

During this chapter we will present a number of results for a number of simple networks. These results are, on the whole, presented as absolute errors. The mathematical model is taken as the base case. A positive error indicates that the simulation produces more loss or delay than we expected, where our expectations are set by the mathematical model.

The power of this approach in the provision of QoS is simple. If we can build a system that behaves in the same way as a mathematical model, then we can predict the behaviour of the network. These predictions can then be used as a basis for provisioning and control of the networks that we build.

7.4 Test Cases

The following subsections will deal with the three test cases used in this chapter in turn, and explain what they are designed to test, and how.

For all of the tests we consider networks with two different sizes of buffers in the queues; namely, 10 and 100 buffers. For each of the flows in the test cases we vary their loading factor (which is measured assuming a service rate of 1.0) between 0.1 and 1.0 in steps of 0.1. Note, this loading factor is calculated per byte and not per packet; this means that the packet departure rate is this loading factor divided by the packet length (when considering point-processes a length of 1 is assumed).

7.4.1 Chains of Queues

The first and most simple test case is a chain of queues. Here a single flow enters at the start of the chain and traverses all the queues in the chain before being measured. The test is repeated with a chain length of 1 to 10. Figure 7.1 shows a queue chain with two queues.

The purpose of this test is to check that our model is composable. We already know that the output from a queue with exponential service is not truly Poisson. However, it may be close enough that the performance predictions are still accurate to an acceptable tolerance.

Figure 7.2 shows the loss and delay predictions for a chain of ten queues. Packets are modelled as point processes and ten and one hundred buffer sizes are shown.

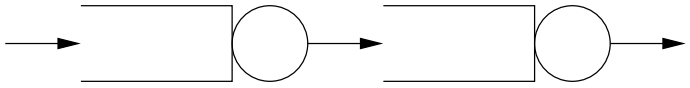


Figure 7.1: Queue Chain Test

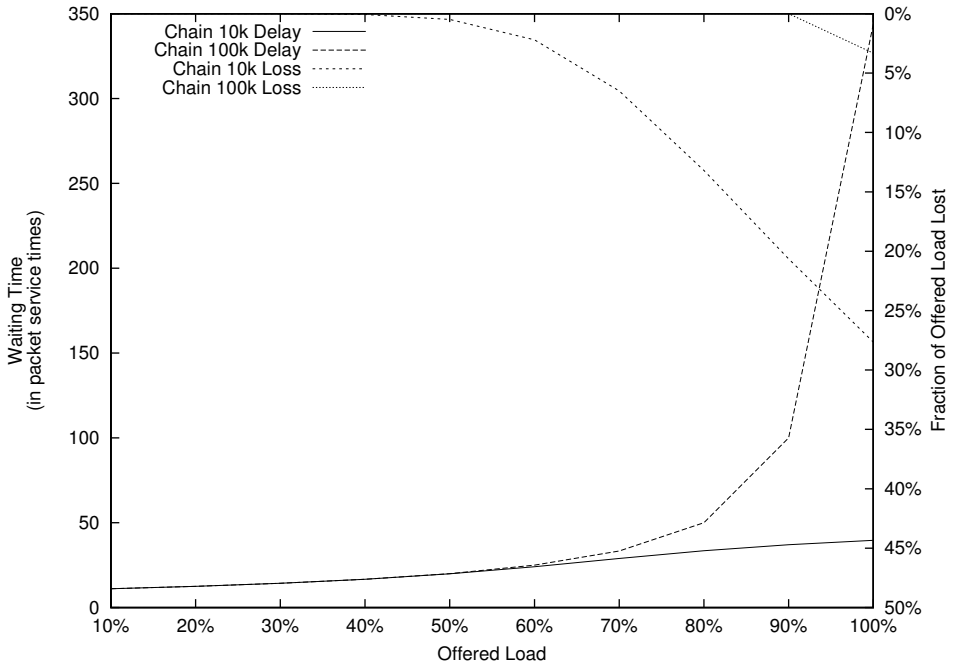


Figure 7.2: Queue Chain Test Predictions

7.4.2 Crossing Flows

The second test case again consists of a chain of queues; however, this time in addition to the first flow, which traverses all the queues, there are additional flows crossing each of the queues in turn. We measure the flow that traverses all the queues. This test is repeated with a chain length of 1 to 10. Figure 7.3 shows this test with a chain length of 3, notice that flow 0 traverses the whole network while 1-3 traverse only one queue.

The purpose of this test is to see how much interference the flow (flow 0) that traverses all the queues suffers as a result of the crossing flows. If the model is composable then we would expect any effect due to the crossing flows to be accounted for. This would result in the predictions for flow 0 being accurate.

Figure 7.4 shows the loss and delay predictions for a cross flow test with ten queues. The crossing flows place a 50% loading on each of the queues, the main flow is varied between 0-100%. Both ten

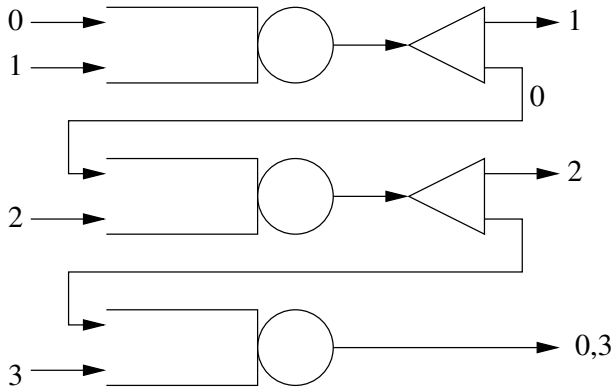


Figure 7.3: Cross Flows Test

and one hundred buffers have been calculated.

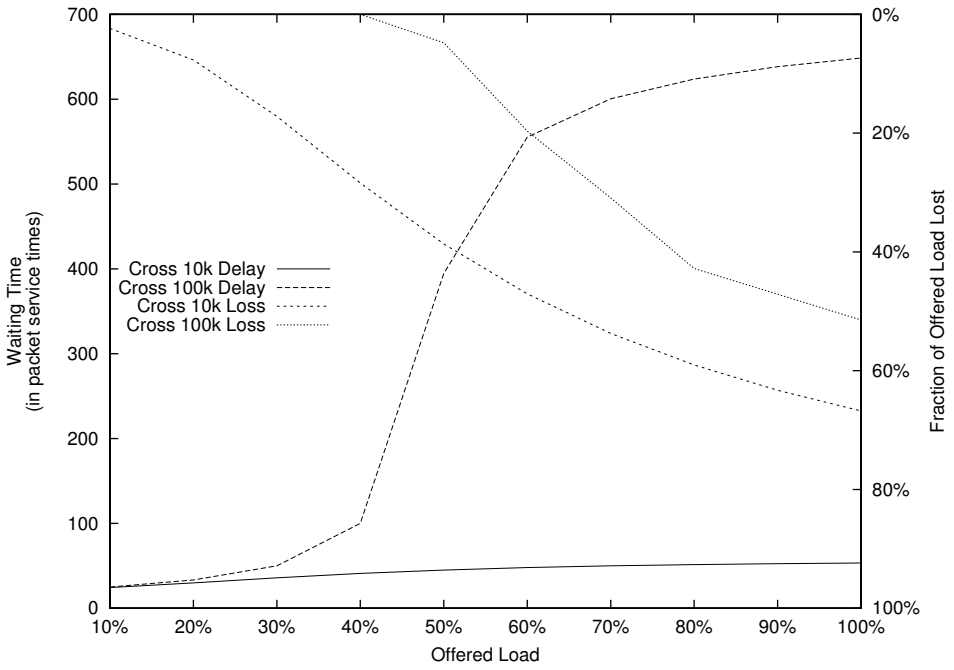


Figure 7.4: Cross Flows Test Predictions

7.4.3 Correlation

The final test is the correlation test. This consists of three queues connected together as show in figure 7.5. Flows 1 and 2 share their first queue with flow 0, then both share the final queue. Flow

0 traverses two queues, that are the first queue that both 1 and 2 see respectively.

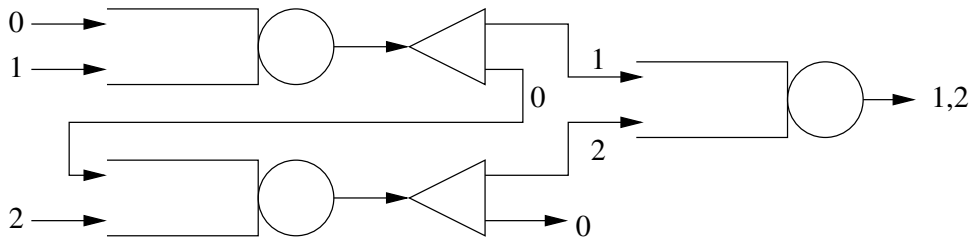


Figure 7.5: Correlation Test

This test is designed to correlate flows 1 and 2 such that they receive different treatment when they are queued together. To achieve this an interfering flow, flow 0, interacts in a queue with flow 1 then flow 2. If the flows correlate then we would expect the treatment of flows 1 and 2 to be radically different. However, as we service packets exponentially, this correlation should be destroyed making our predictions accurate.

7.5 Packet Point Processes

The first set of tests models packets as point processes. By a point process we mean that the packet it transmitted in a single instant of time. This means that the length of the packet does not affect its service time. Clearly this does not model real word networks with any accuracy; however, it is as close as possible to the mathematical models that we are using and is therefore a good base case test. We shall now investigate each of the test cases in turn.

7.5.1 Queue Chains, 10 Buffers

Figure 7.6 shows the absolute error in delay measured in packet service times against the loading factor of the sample flow. This is repeated for a chain length of 1 to 10. We can see from the graph that, at its worse, the delay is one packet service time greater than it was predicted when ρ is 1.0. Also the delay is 0.2 service times less than expected, where ρ is 0.6. From the applications point of view this is not a problem, as it will benefit from the increased performance. However, when considering the system as a whole we would like to minimise this kind of error as it will result in lost opportunities to carry more traffic.

Figure 7.7 shows the absolute error in the packet loss rate against loading factor for a queue chain of 1 to 10 in length. The loss error is at its worst at around 0.025 less than expected. This is about a 25% improvement in performance. Clearly any application that has been predicted a higher loss rate would benefit in this situation.

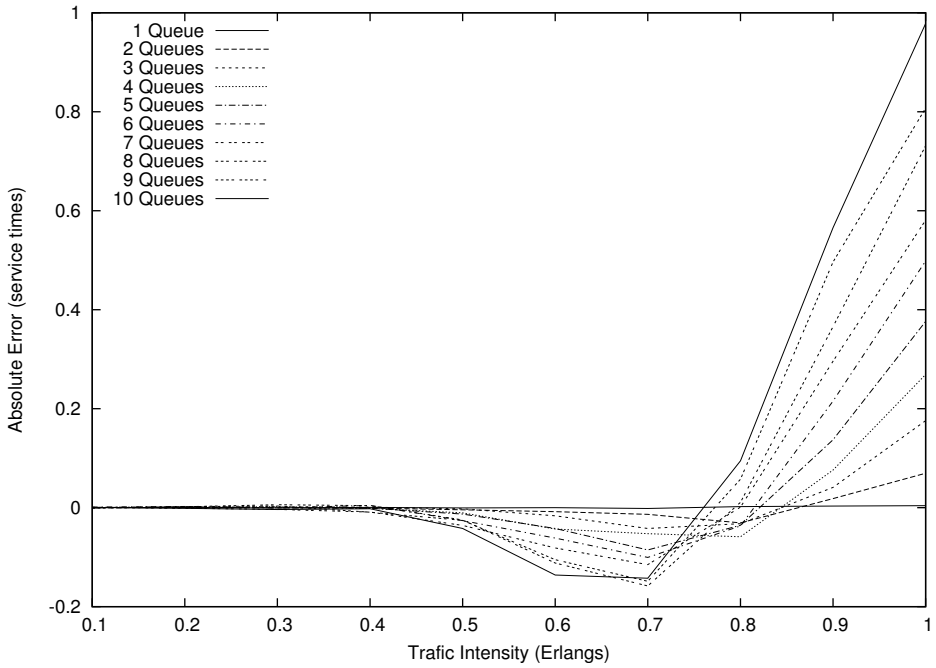


Figure 7.6: Queue Chain Delay, 10 buffers

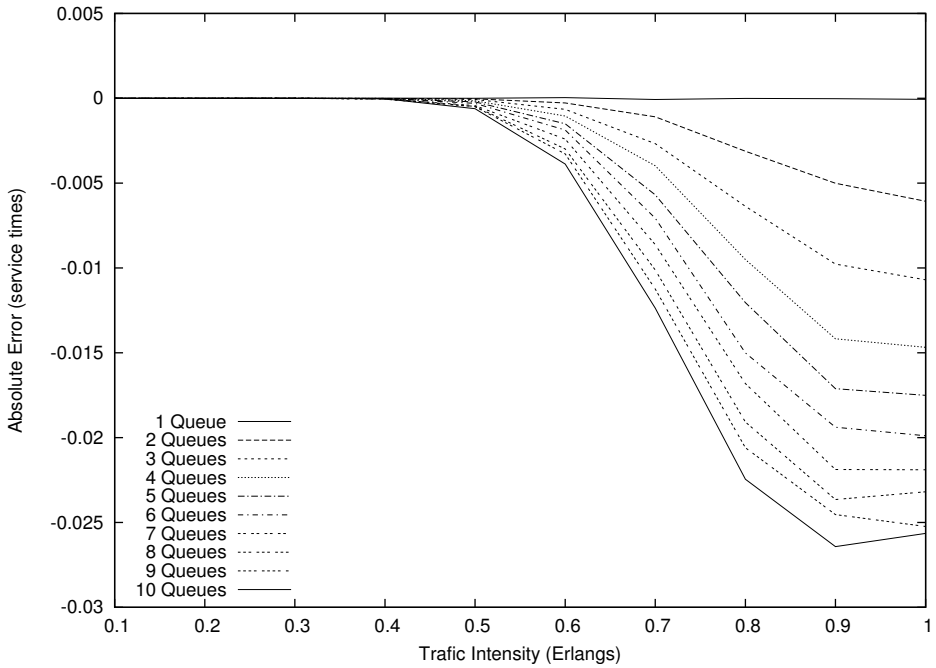


Figure 7.7: Queue Chain Loss, 10 buffers

7.5.2 Queue Chains, 100 Buffers

Figure 7.8 shows the absolute error in delay measured in packet service time against the loading factor of the sample flow. This is repeated for a chain length of 1 to 10 with 100 buffers per queue. As you can see when the loading factor is 1.0 for 10 queues the delay is nearly 2.5 packet service times higher than expected. How this would effect a real application is dependent upon that application. However, notice that this is not a problem until we are loaded in excess of 90%, a fact that we will use in later chapters.

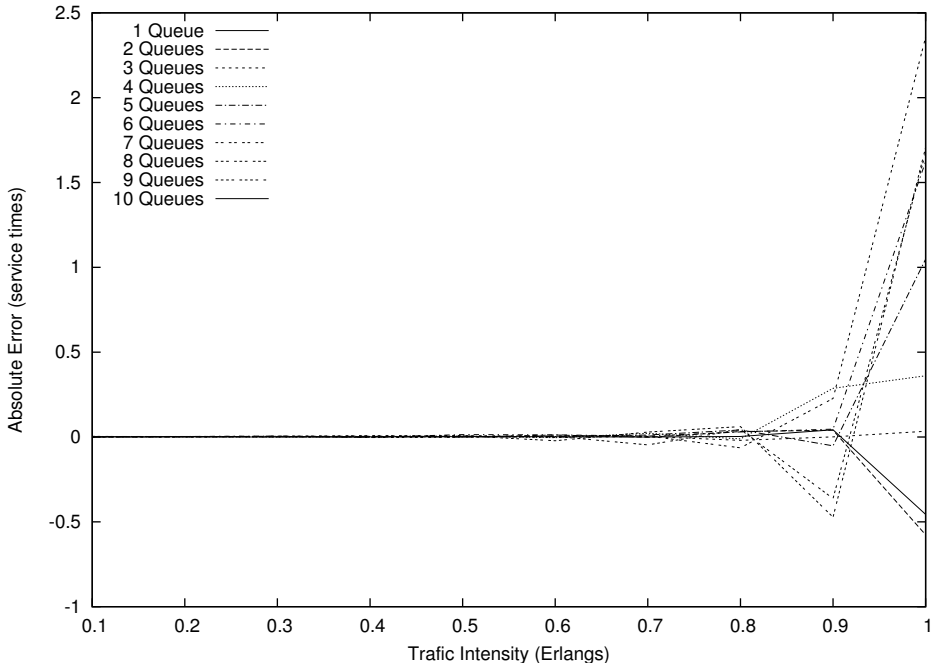


Figure 7.8: Queue Chain Delay, 100 buffers

Figure 7.9 shows the absolute error in the packet loss rate against loading factor for a queue chain of 1 to 10 in length. As you can see at a loading factor of 1.0 the loss is 0.0035 less than expected, this is again in the application's favour. The error in the packet loss rate increases significantly after a loading factor of 90%. If you refer back to figure 7.2 you will see that significant (ie. $> 10^{-10}$) loss does not occur for a 100 buffer queue until this point. What we see in this graph is that loss is less likely in the simulator that the mathematics predict. However, it should be noted that loss is relatively rare (<1%) at 100% loading, its quite likely that a longer simulation run would be required to gain enough loss results to be meaningful.

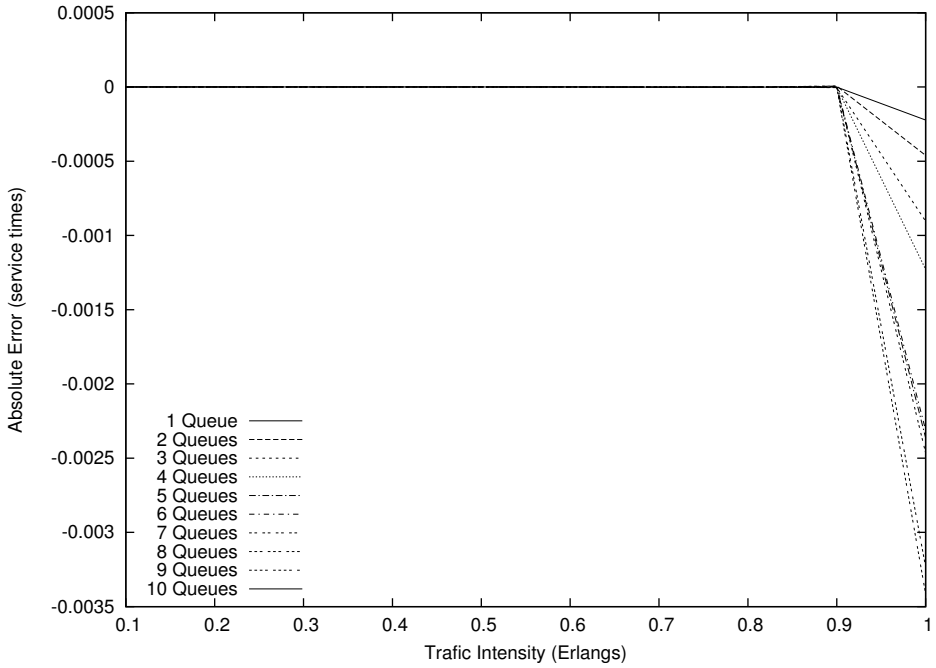


Figure 7.9: Queue Chain Loss, 100 buffers

7.5.3 Cross Flows, 10 buffers

Figure 7.10 shows a surface plot of the absolute error in the delay against the loading factor of both the sample flow and the crossing flows. As you can see, when both the crossing flows and the main flow are at a loading factor of one, the error in the delay in packet service times is 2. This means the delay is longer by two packet service times. Over 10 queues this equates to about a 3.5% positive deviation.

As we saw from figure 7.6 the delay was one packet service time larger than expected. Here it is 2 and we are dealing with two flows. It would appear that the error experienced in the delay is additive. However, given the small relative error in this situation it is quite acceptable.

Figure 7.11 shows the absolute error in the predicted loss for a cross flows test with 2 to 10 queues. Similarly to figure 7.7 the delay is 0.025 less than expected at its worse. This shows that there is not a significant increase in inaccuracy in this situation over others.

7.5.4 Cross Flows, 100 buffers

Figure 7.12 shows the absolute error in delay for a cross flows test with 2 to 10 queues. Whenever the combined loading factor exceeds 80% there is a positive error introduced of around 5 packet

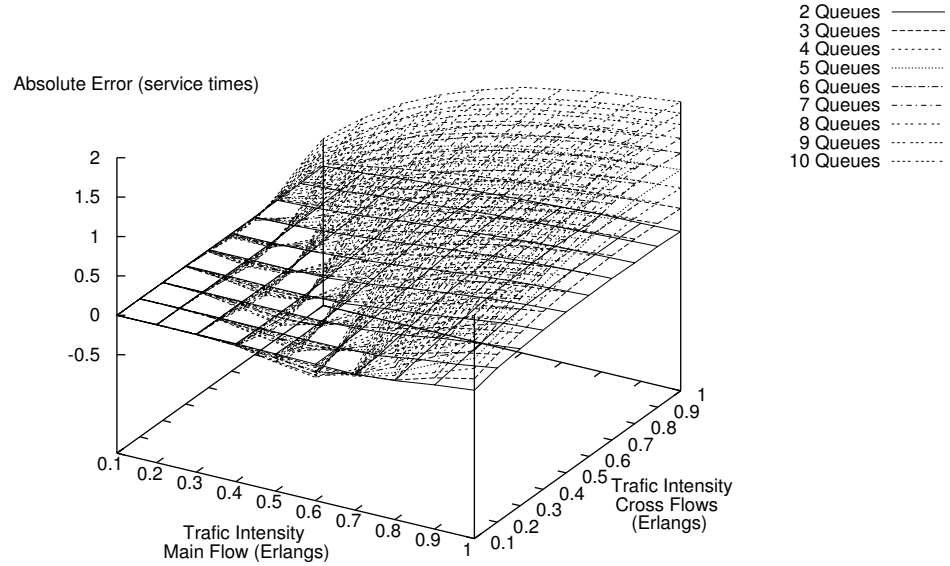


Figure 7.10: Cross Flow Delay, 10 buffers

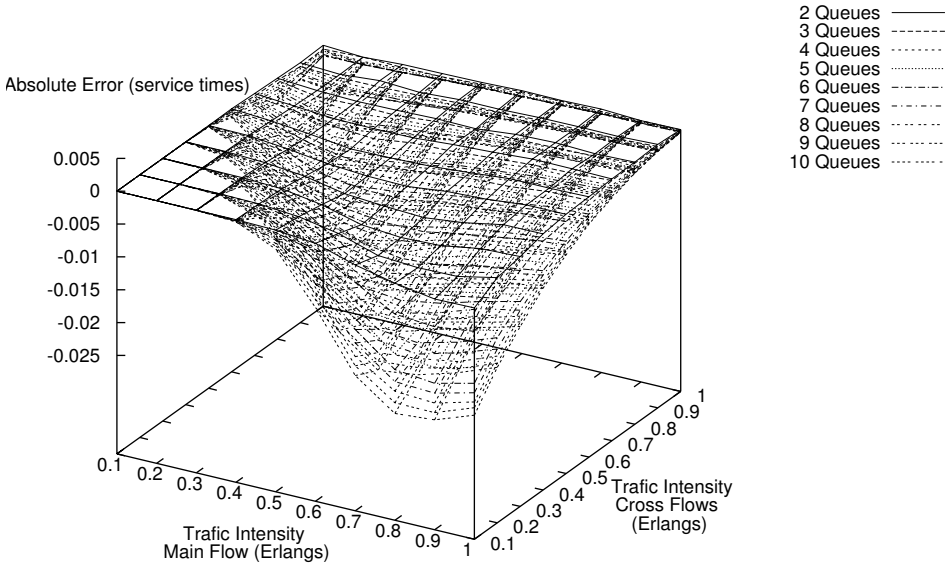


Figure 7.11: Cross Flow Loss, 10 buffers

service times. This accounts for less that 1% relative error in the delay over 10 queues. Again we see that this error is approximately twice that found in the corresponding queue chains example (see figure 7.8).

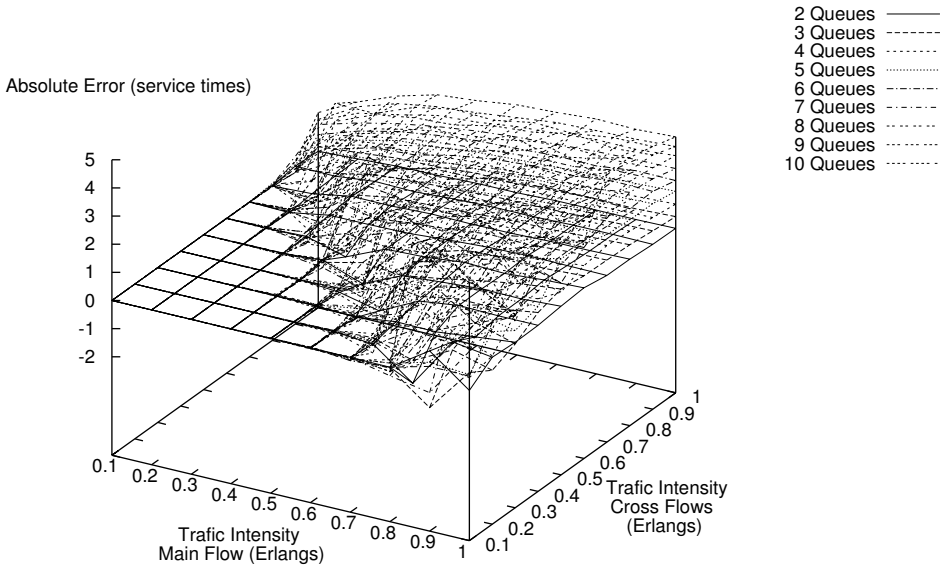


Figure 7.12: Cross Flow Delay, 100 buffers

Figure 7.13 shows the absolute error in the packet loss rate against loading factor for a queue chain of 2 to 10 in length. At its worst it is 0.003 less than expected, which is about 30% more loss. This again is almost exactly the same error in loss that can be found for a queue chain test for 100 buffers (see figure 7.9).

The reason for the change at 80% is due to loss occurring, which is rare before this point. As can be see from the graphs for both loss and delay the change happens at the same point. You will also notice that the point of change happens at a lower loading factor when there are 10 buffers, this is because loss occurs at a lower loading factor when there are a smaller number of buffers.

7.5.5 Correlation Test

Figure 7.14 shows the absolute error in the delay for a correlation test with 10 buffers. Firstly, notice that flows 1 and 2 receive almost the same treatment. Flow 2's error is slightly higher than flow 1's; however, given that the difference is so small it is likely to have been caused by an error in the rate calculations for flow 0 as it leaves the first queue. As the difference is small, a fraction

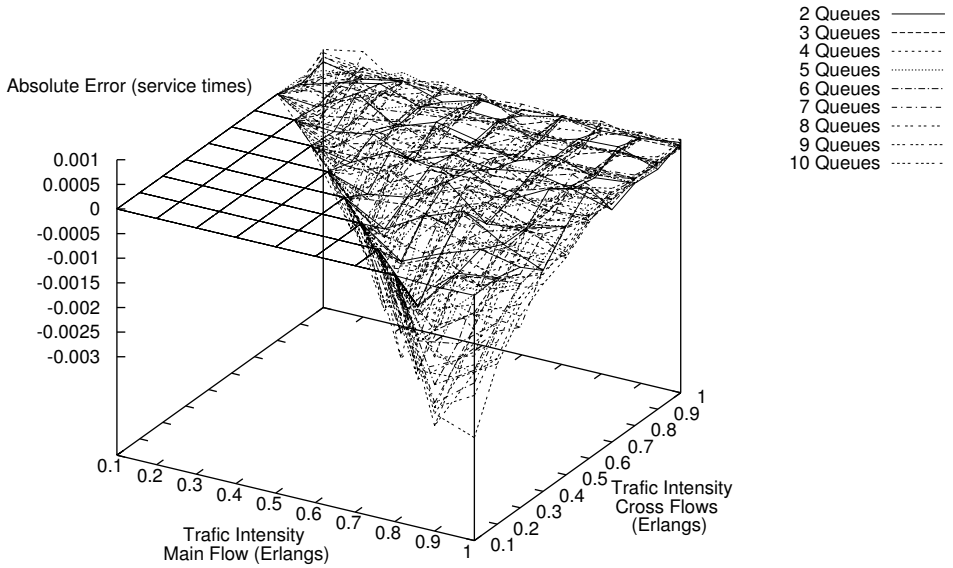


Figure 7.13: Cross Flow Loss, 100 buffer

of a packet service time, we do not consider the traffic to be correlated. For both flows 1 and 2 and flow 0 the delay is 0.4 packet service times worse than expected.

The error in flow 0 increases as its rate increases, which is to be expected. This error is increased more as the crossing flows, 1 and 2, increase their rate. This gives the shape of the graph for flow 0 as it peaks at the back right hand corner. The error for flows 1 and 2 is similar in that as they increase their rate so their error increases.

Figure 7.15 shows the absolute error between the predicted and measured loss for a correlation test. The treatment of all flows is generally uniform. At its worst the loss is 0.006 less than expected.

7.6 Fixed Packet Sizes

In our second set of tests we model a network where packets have a length. However, we restrict the network to carrying packets of a single fixed length. This is similar to networks such as ATM which carry 53 byte cells only. The packet length affects the service time, which in a deterministic system with fixed sized packets would also be fixed. In this approach we find this by multiplying the packet length by an exponentially distributed service time (see chapter 5 for more details).

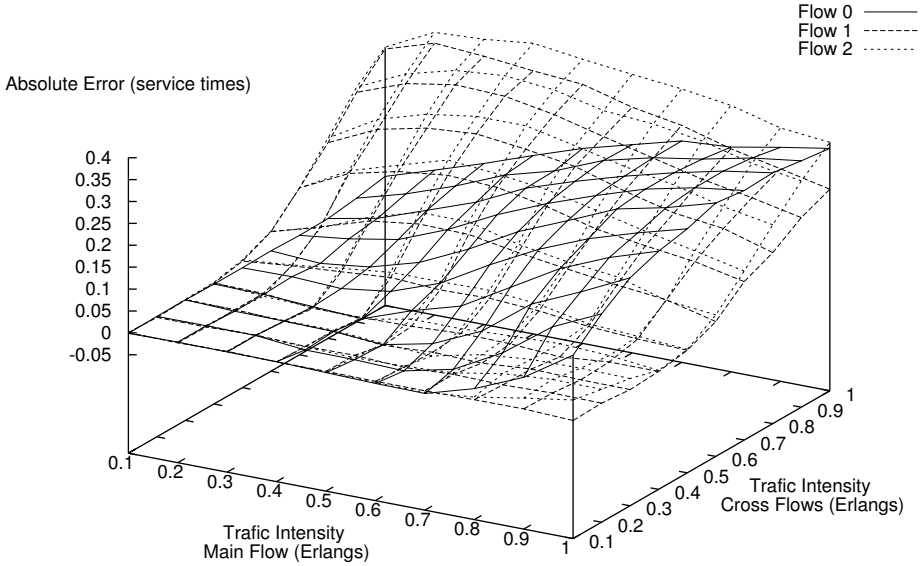


Figure 7.14: Correlation Delay, 10 buffers

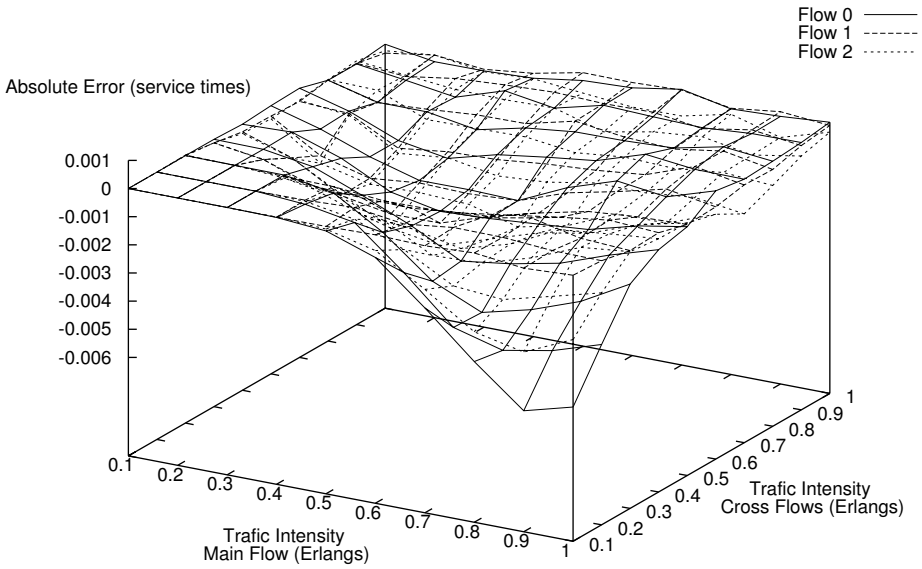


Figure 7.15: Correlation Loss, 10 buffers

We have chosen to test two sizes of packets in this set: 64 and 1500. These numbers were chosen because they are the minimum and maximum packet sizes that are found on a standard Ethernet network[4].

For both sets of packet size the resulting graphs closely resemble the graphs from our first set of tests using point processes; for this reason we have not included the graphs. Table 7.1 shows the maximum errors for loss and delay for the two packet sizes in this test and those of the point process set. We have included the first set so it is possible to examine the graphs if required.

Test Name	Max. Abs	Point Processes	64 Byte Packets	1500 Byte Packets
Chain 10	Loss	-0.025	-0.025	-0.025
	Delay	+1	+1	+1
Chain 100	Loss	-0.0035	-0.003	-0.003
	Delay	+2.5	+3.5 -0.5	+3 -0.5
Cross 10	Loss	-0.025	-0.025	-0.025
	Delay	+2	+2	+2
Cross 100	Loss	-0.003	-0.0035	-0.003
	Delay	+5	+4	+5
Correlation 10	Loss	-0.006	-0.005	-0.005
	Delay	+0.4	+0.4	+0.4
Correlation 100	Loss	0.006	0.008	0.008
	Delay	+1.2 -0.4	+1 -0.6	0.8

Table 7.1: Absolute Errors for Fixed Packet and Point Process tests

You will notice from the table that the results for packet sizes 64 and 1500 are almost identical. What is more encouraging is that the results from both sets of packet sizes are also almost identical to the results that we got in the first set of tests, namely using point processes. It is worth noting that the errors are at their largest when the loading factor is in excess of 80%, for the majority the errors are at the worse over 100% loading. Other approaches yield interesting accurate answers when the loading is lower, this approach maintains accuracy in excess of this point. This is strong evidence for the applicability of this approach. We can now predict the behaviour of a semi-realistic network with no less accuracy than a hypothetical network based on point processes.

7.7 Mixed Packet Sizes

In the third and final set of tests in this section we look at packets of mixed sizes in the same network. Remember that each flow has packets of a fixed size, and there are a number of flows with different sized packets. This provides us with a more realistic model of real networks.

[4]It is possible to have packet sizes greater that 1500 on Gigabit Ethernet. This is a propriety extension, called Jumbo Frames, to the Ethernet standard used by some Gigabit Ethernet Switch manufactures.

Each of the tests have 64 and 1500 byte packets in the network. In the case of the queue chain test we send two flows through the network for each packet size. For the cross flows and correlation tests we have two complementary tests. The first is where small (64 byte) packets cross the whole network and large (1500 byte) packets interfere (we label this 'large'). The second is the opposite, where large packets cross the whole network and are interfered with by small packets (we label this 'small').

Table 7.2 summarises the results from this set of tests. For each test we show the worse result obtained. Overall we can see that the trend is for the smaller packets to receive much worse service than we were expecting when they are interfered with by the larger packets.

Test Name	Max. Abs.	64 Byte Packets	1500 Byte Packets
Chain 10	Loss	+0.25	-0.14
	Delay	+90	+-0.4
Chain 100	Loss	+0.06	-0.05
	Delay	+80	-1.8
Cross 10 large	Loss	0.18	-
	Delay	300	-
Cross 10 small	Loss	-	-0.08
	Delay	-	-0.6 +0.4
Cross 100 large	Loss	0.18	-
	Delay	1000	-
Cross 100 small	Loss	-	-0.04
	Delay	-	-3.5
Correlation 10 large	Loss	+0.06	+-0.01
	Delay	+6	+1
Correlation 10 small	Loss	+0.07	0
	Delay	+3	0
Correlation 100 large	Loss	+0.06	+-0.01
	Delay	25	5
Correlation 100 small	Loss	+0.035	0
	Delay	20	0

Table 7.2: Absolute Errors for Mixed Packets Tests

Of the tests in this set the Cross Flows test is the most interesting. Specifically, where a flow of small packet size is interfered with by a number of large packet sized flows. As the large packets take longer to service the smaller packets see a larger average waiting time. This causes our predictions to be inaccurate (see section 7.8.4 for an explanation).

Figure 7.16 shows a flow, containing 64 byte packets, as it traverses a number of queues. The queues are shared with a flow, containing 1500 byte packets, that traverses each queue only once. As you can see, the more queues the flow of 64 byte packets crosses the worse the predictions become. What is interesting is that the inaccuracy is dependent entirely on the load offered by the 1500 byte packets; as the load that this flow offers increases so does the error, irrespective of the load that the 64 byte packets offer.

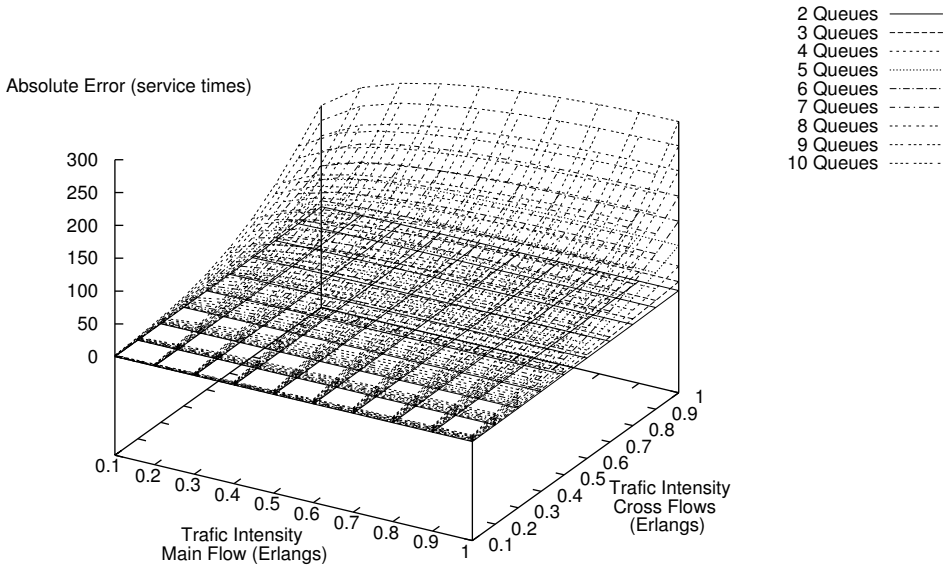


Figure 7.16: Cross Flows 10 buffers large Delay

Figure 7.17 shows the same test but using 100 buffers and not 10. As you can see the errors in the predictions do not become severe until the 1500 byte packets reach a loading factor of 0.9. Beyond this point the errors are significantly worse than the results for 10 buffers. Its clear that more buffers increases the accuracy of the predictions, at least for a larger region of the problem space. This is not surprising given that more buffers gives us a larger memory, which in turn makes the predictions about the number of packets of each size more accurate.

This may seem to suggest that this approach is only useful for particular buffer sizes; this is not the case. As we discussed in section 4.7.2 the choice of buffer size effects the performance of the network. This also effects the extent to which we can make accurate predictions. Where the buffers are small, and so is the memory of the queue, the loss is higher. Firstly the increase in loss makes the predictions more inaccurate, as we have seen before. Secondly the small memory provides a poor average of the number of packets of each size in the queue.

7.8 Interpreting the Results

In this section we look at some probable causes for differences between our calculations and simulations. This is important, as any real world implementation, also based on random services, would likely suffer from similar problems. First we show that the random number generator we have drifts

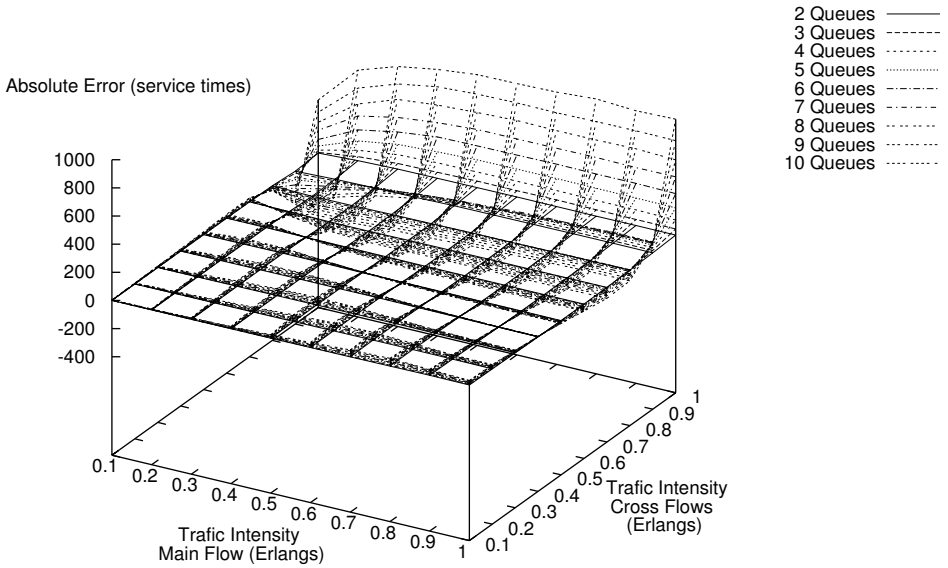


Figure 7.17: Cross Flows 100 buffers large Delay

around its mean value over a number of samples. We then show how this error in the mean produces a positive bias in the delay calculations. Finally we show how this bias is likely to cause higher delays in the simulations.

7.8.1 Errors in the mean

It is well known that random number generators drift around a mean value. It is this mean value that we set when we are asking for a particular rate of traffic to be generated. It is assumed that when averaged over a large number of samples that the observed mean value will be the same as the value that has been requested. This clearly depends on the number of samples that have been taken.

To test what drift in the average value of a number of samples from the random number generator we did the following. We took 100,000 rolling averages and for each of these rolling averages calculated how much higher or lower the observed average was than the predicted value. During the test the number of samples per rolling average was varied to see what the drift was likely to be. Throughout the test the expected value was set to 1.0. The test was also repeated a number of times with different seeds to ensure that the results were representative of normal operation. The table below shows the results.

Samples per Average	Approx Absolute Error
100	0.045
1,000	0.015
10,000	0.005

These results are not surprising, they are well within the bounds of possibility, as expressed by the Central Limit Theorem. If we sample a random variable, in this case an exponential distribution, we find that as we increase the number of samples we approach the mean of the original distribution. The more samples that we take the more accurate the mean becomes, and the lower the variance in the mean. Variation from the mean over short runs from the random number generator creates bias in the resulting simulations. The cause of this bias on the theoretical results from the queueing theory is important to understand.

7.8.2 Resulting Bias in Queueing Formula

In this section we will explain how the results are affected by accuracy of the sample taken from the random number generator. To do this we will look at how this affects the calculated delay for an m/m/1/k queue with 10 buffers. The formulae for loss and delay in any queueing system are non-linear. A period of bias from the random number generator will have effects which do not cancel out. A temporary increase in the load will cause more degradation (ie more loss or increased delay) than the corresponding temporary decrease in load. In this subsection we will quantify this effect.

Figure 7.18 shows the percentage error in the delay calculations, for the absolute errors given in the table above. To calculate this we use the following formula:

$$\frac{delay(\lambda+\epsilon)-delay(\lambda-\epsilon)}{delay(\lambda)} * 100$$

The reasoning behind this formula is as follows: The maths predicts that the delay at a given loading factor is $delay(\lambda)$, when λ is measured over an infinite period of time. However, we know that if λ is measured over a smaller number of samples and will therefore have an error of $\pm\epsilon$, in accordance with the central limit theorem. We wish to know if the delay added by a positive error is cancelled by a corresponding negative error. To answer this we subtract the delay when λ is smaller from the delays when λ is larger, and then divide by the ideal λ to find the percentage error (this can be seen as a crude approximation to the first derivative of the delay formula).

As you can see, there is a positive bias, that is the area under the curve above the x-axis is larger than that below. This means that when drift is introduced into the simulator the results are likely to come out higher than those that had been predicted by the mathematics. This is because a small increase in λ results in more delay than the same decrease in λ would decrease the delay. When a number of samples of λ are taken, being evenly distributed around the mean, more delay results due to the overshoot.

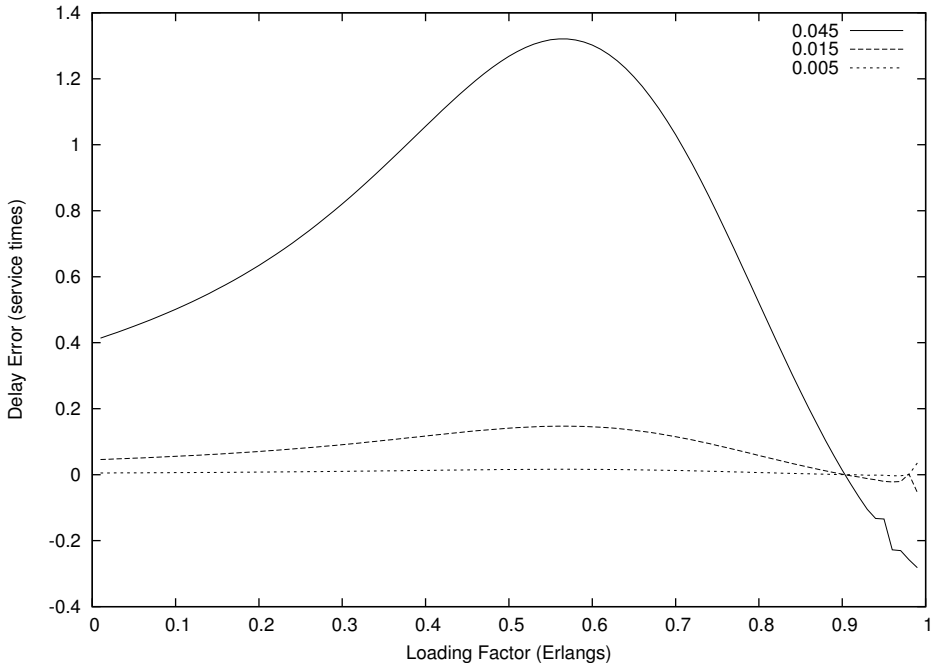


Figure 7.18: Bias in Delay Calculations

Its worth noting that the same bias does not exist for loss. There are large variations in the level of loss at low loading factors, which small changes in load could effect; however, as the loss rates are so small they are essentially irrelevant. It would appear that loss is less susceptible, at least theoretically, than delay. However, it is worth remembering that the loss effects the delay and vice-versa as the system has two degrees of freedom. So while there may be no significant effects on the loss, for small changes in load, they may cause larger variations for delay.

7.8.3 Explaining the Deviation

There are a number of artifacts in the results that need to be explained. The most significant of these is when a number of queues are composed in a chain the delay results from the simulator are always higher than that which the maths would predict. This applies to any flow that crosses a number of queues in our system whatever the topology. We have taken care to ensure that this effect is not caused by the choice of random numbers, and have repeated the tests in a number of different ways to confirm this.

To explain this result we start by considering a simulation, lasting for T seconds, of a single queue. We split the simulation time into a number of discrete time intervals $t_0..t_n$. Each of these time intervals starts and ends when the queue becomes empty. During each time interval the queue

observes an average arrival rate of packets, given the time interval is finite then so is the number of packets that can arrive at the queue during the interval. It follows that the observed average during each time period will be close to the long term mean arrival rate with some error, ϵ.

We assume that the errors in the observed rate are normally distributed, that is there are, on average, the same number of positive errors as there are negative errors. However, we know that a positive error causes more overshoot in delay than a negative error undershoots, when compared to the delay expected by the long term average rate. The result is that more delay will be experienced than predicted.

The increase in delay for a single queue, as a result of sampling and the bias in the queueing formula, is likely in reality to be quite small. Delays are additive, so over a number of hops the extra delays introduced will add up, causing the higher delay observed in the simulations.

As an area for further investigation would be to validate this hypothesis. A good starting point is the occurrence of empty events. We know that the probability of the queue being empty is given by P_0; the rate of empty events is therefore given by $\frac{1}{P_0}$. Using this it would be possible to calculate the average period of time between empty events. Applying the central limit theorem would give us a measure of the expected error in the observed rate. Finally it would be possible to get a more true measure of the expected error.

7.8.4 The effect of packet size

As we have seen from the cross flows test networks with mixed packet sizes are harder to predict. This is because the mathematical models that we are using do not accurately reflect the network. We are using a M/G/1/k class queue to model this network. Firstly the arriving traffic is not strictly Poisson distributed, as it has been produced by a finite queue. This may not indeed be a problem as the traffic is progressively shaped by the network. Secondly the service discipline is best modelled using a Gaussian distribution, the service facility calculates the departure time as a random Poisson service rate multiplied by the packet length, while this may have the same mean as a Gaussian model its moments are different.

To make better predictions a better model of the service discipline is required. Some candidates for this could be Hyper-exponential or Erlang distributions. Using such a model would require us to do some analytical work in order to arrive at some suitable equations. While we have not done this it would be possible to use such models in our methodology.

What is perhaps more interesting is the effect of the buffer size on our predictions. The formula for the waiting time in the queue that we are using is as follows (See Appendix A):

$W_s = \frac{\lambda_1}{\lambda} E[s_1] + \frac{\lambda_2}{\lambda} E[s_2]$

The waiting time for the entire queue is therefore the sum of the average waiting time for each class. We can obtain the waiting time for each class as the ratio of packets of that size, given by $\frac{\lambda_x}{\lambda}$, multiplied by the expected service for that class.

To explain the effect of the buffer size on the accuracy of the predictions, consider a single queue. At any instant of time we can take a snapshot of the packets that are in the buffer. At this instant of time there is a average number of each size of packet. When there are a small number of buffers the average number of packets for each size are less likely to be close to the expected value. As we increase the number of buffers the average number of each size of packet becomes closer to the expected value. We can relate this to an experiment where a number of coins are tossed in the air simultaneously; the more coins we add, the more likely we are to find 50% heads or tails, when compared to say two coins where there is a low chance of a head and a tail.

In reality the problems with the average number of packets in the queue occurs because of loss. In a system with infinite buffering the relative number of packets of each size will always be the same as the offered number. In a system with finite buffers the number of packets of each size is determined the unpredictable ordering of losses, as a result the ratio of packets of different sizes in the queue is not always the same as the offered ratio. The waiting time for the queue is determined by this ratio, and if that is not accurate nor are the results. The more buffers there are the closer the ratio of packet sizes in the queue is to the offered ratio.

The more buffers we add the closer the ratio of packets of each size is to the averages the maths predicts. Choosing a buffer size that is large enough to make this average reliable is clearly important. However if it is too large then the time taken for the queue to reach a steady state will also increase. This is undesirable, as it makes the network less resilient to changing loads.

An area for future research is to investigate a better model for mixed packet sized networks based upon a different service distribution. In addition, the effect of buffer size needs to be studied in relation to this to ensure that it is large enough to gain good averaging properties without being too large, so as to cause instability.

7.8.5 Link service rates

Thus far we have looked at delay in relation to normalised packet service time. While this is useful for comparing the results between simulation and calculation it gives us very little idea of performance of a real network. What can be tolerated in terms of delay is application dependent, and is more usually specified in seconds not packet service times. Table 7.3 shows the service time for a 64 and 1500 byte packet at different link speeds.

As you can see from the table the service time unsurprisingly reduces as the link speed increases. This is important when considering errors of a number of packet service times. For example an

Technology	Data Rate	Service Time for 64 bytes	Service time for 1500 bytes
Mobile Phone (GPRS)	16kbps	32ms	750ms
Modem	56kbps	9ms	214ms
Dual ISDN	128kbps	4ms	93ms
xDSL	512kbps	1ms	23ms
E1/T1	1.544Mbps	0.3ms	8ms
Gigabit Ethernet	1Gbps	0.5us	12us

Table 7.3: Link Service Rates

error of 10 service times at a Gigabit would give a delay for 64 bytes of 0.05ms; when real time constraints for traffic are expressed in 100's of milliseconds this is not significant. However at 16kbps this is extremely significant. Note that we are more interested in 64 byte packets as we know that predicting their delay is harder.

The errors in our predictions should always be considered in terms of the rate at which the packets were serviced. Where the link rates are high the errors in the predictions become relatively insignificant when compared to application requirements. However, when the link rates are low the errors in the predictions are more damaging. Getting a better understanding of the magnitude of these errors is required, as it is likely that some applications performance will be severely damaged by problems at low link speeds (probably when entering and leaving the core network).

7.9 Conclusion

In this chapter we have compared simulation results to predictions based on calculations for a number of test cases. We conducted three sets of tests modelling packets as: point processes, fixed length, and mixed length.

For point processes and fixed packet sizes we find that our analytical models closely match the results from our simulations. Importantly, this shows that assumptions about the Poisson nature of the traffic leaving a finite queue still hold even when the traffic is not, at least theoretically, Poisson. One possible reason for this assumption holding is due to reshaping of the traffic as it passes through the network [49]. As a result only the first few queues contribute to errors in the calculations.

For networks that contain mixed packet sizes we find that our predictions are acceptable for large packets, but inaccurate for small packets. While this can be eased by increasing the buffer size to get a better averages it is not as accurate as hoped. This is due to the models, based on Gaussian distributions, that we are using. However, as an area for future research it looks possible to obtain the same accuracy using a better model. This would allow us to make strong predictions on the behaviour of the network.

Chapter 8

Quality Requirements

8.1 Cherish and Urgency

In this chapter we will look at QoS requirements for a number of common applications. We shall also look at some problems that these applications suffer on the Internet today. The focus will be on classifying the applications within our framework to allow them to be supported on a network that has differential QoS support.

In [34] a model for classifying and servicing flows with differing loss and delay requirements is presented. Traffic is classified along two axes:

- Cherish; the desire to experience less loss.
- Urgency; the desire to have a lower delay.

This can be represented graphically (see figure 8.1) as a grid. This allows for four classes of service: cherished and urgent, not cherished and urgent, cherished and not urgent and not cherished and not urgent; although it would be entirely possible to extend the model to provide more levels of cherish and urgency. In the rest of this chapter we will use this two dimensional classification.

This chapter is organised as follows: In section 8.2 we look at the problems and requirements of routing protocols. In sections 8.3, 8.4, and 8.5 we look at the requirements for NTP, VoIP and HTTP respectively. In section 8.6 we look at how the IP ToS bits map onto our cherish-urgency model, and look at the implications of such a mapping. Finally we will finish with a summary of the findings in this chapter.

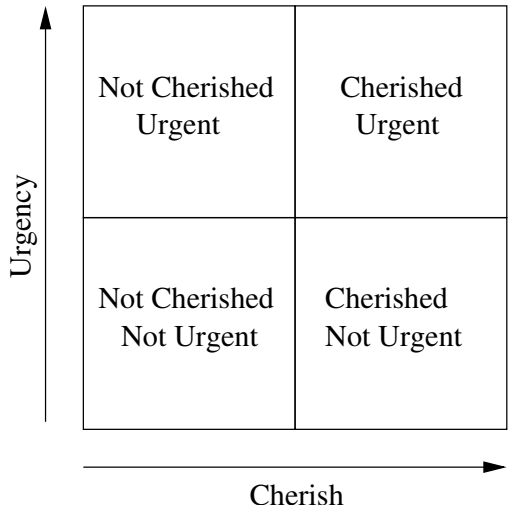


Figure 8.1: Cherish Urgency Grid

8.2 QoS and Router Flap

In this section we will investigate how Quality of Service can be used to alleviate the problems causing router flap. IP networks are interconnected using routers. These devices operate at the Network layer of the ISO seven layer model [35]. Each router maintains a routing table that describes how to reach portions of the IP address space. The routing table is used to send packets down an appropriate link, bringing them closer to their destination. A detailed description of this process is outside the scope of this document and readers are directed to [7] for a fuller discussion.

Routing tables are in general maintained in one of two ways, depending on the size and complexity of the network. In a small network they are usually maintained by hand by an administrator. Each route entered in this way is called a static route. In larger networks a routing protocol is used.

Routing protocols enable routers to pass information about routes in their routing table to adjacent routers. Each router, in addition to its static routes, holds a set of dynamic routes which are updated by a routing protocol. Both the static and dynamic routes are now used to send packets to their destination. This process allows a router to discover parts of the network that it is connected to automatically.

In addition to the route itself there is often a metric passed as well; the metric describes how suitable the path is. For example hop-count may be used to choose the shortest path to a destination; if there are two routes to a destination then the path with the shortest hop count will be used. This allows for redundancy in the routing infrastructure, where there is more than one route to a destination. Such metrics are usually protocol specific, and we will not deal with them in detail here.

Router flap occurs when dynamic routes disappear and appear in quick succession. In some situations a route may disappear for a longer period, in which case we would more likely call it a net-split. The effect of these dynamic routes disappearing causes parts of the network to be unreachable, or reachable via a longer path. This is clearly undesirable, and will affect any traffic that is attempting to reach such a destination.

In such an environment it is almost impossible to provide QoS guarantees, as a fixed path cannot be determined. Even without the desire to provide multi-service facilities, it is extremely irritating to the end user to be unable to reach parts of the network; especially when it was possible only a few moments ago.

8.2.1 Causes of Router Flap

One cause of router flap is when routing messages fail to reach their destination in time[1]. Each dynamic route has an expiry time; if the route is not updated during this time then it is removed. This is not the case for all routing protocols, but a number of well used protocols will behave in this way. Will shall examine this more in a moment.

When routes are lost in this way there is a secondary effect. That is, routes propagated from this point onwards do not contain routes to the destination that was deleted. This means that the loss of routes is not just confined to adjacent routers but is actively propagated around the whole network, causing yet more inconvenience. To add to this, routing protocols take time to converge into a stable state, which may not happen in the presence of router flap. Even worse, it is possible to converge into an incorrect state in some protocols such as RIP [48, 67]. These problems are well known.

What causes the router messages to be lost? The first and most obvious reason is that there has been a hardware failure, physically separating two parts of the network. If there is an alternate route then this can be used in the interim, otherwise there is nothing that can be done, other than replacing the hardware. The second reason is that the link, which is carrying the router messages, is heavily congested. The congestion causes messages to be discarded, as there is insufficient buffer capacity to hold them. This situation is made worse when there is contention between the routers[2], or if a link has failed causing a backup link to carry additional traffic.

Using RIP and OSPF [72] as an example, we will investigate how router flap could occur using these protocols. What follows is a brief outline of the operation of these two protocols, and the problems that they have when dealing with loss.

[1] There may be other causes of router flap, such as routing tables overflowing due to large numbers of routes.
[2] This could happen when there is a switched network between the two routers.

8.2.2 RIP

Routing Information Protocol (RIP) is a simple UDP based routing protocol designed to be used on a Local Area Network (LAN). It is, however, used in a wide variety of other places. There are two versions of this protocol, 1 and 2, the latter is an extension to the first to add security; for our purposes their behaviour is the same.

Each router that is running RIP sends its routes to its neighbour, usually every 30 seconds. When a router receives a RIP message it examines it and adds or deletes routes as appropriate. However, for the routes to remain in the routing table, updates must be sent, or the routes will be deleted. If no message is received after 180 seconds then the routes are marked as unusable.

Thus if 6 messages in a row fail to reach their destination then routes will be deleted; if the intervening network is heavily loaded this is quite possible. The situation is made worse if broadcast frames are used to transmit the routing messages, as they are more likely to be discarded by switching elements in the network than uni-cast frames.

8.2.3 OSPF

Open Shortest Path First (OSPF) is a link-state routing protocol designed to be used inside a single Autonomous System. It is layered directly on top of IP, and has its own mechanisms for providing reliable updates. It is a complex protocol and we will only look briefly at its operation here.

Each router running OSPF talks to its neighbours using uni-cast, multi-cast or broadcast frames depending on how it is configured. The figure 8.2 shows the basic exchanges that OSPF goes through when it initially transfers its routing information. Once this has happened HELLO frames continue to be sent to keep the routes active, and Link State Updates are sent when required. The exchanges can be summarised as:

- Send HELLO frames to discover, or keep-alive, neighbouring routers.
- Transfer the details of the links that are available in the database.
- Make updates to the available links when the topology changes.
- Send HELLO frames to inform neighbours that you are active.

Generally in OSPF changes to the topology are acknowledged either implicitly or explicitly. This makes it far more tolerant to loss than RIP. However, similarly to RIP, HELLO frames have to be sent to neighbours in order to keep the advertised routes active. In OSPF, HELLO frames are

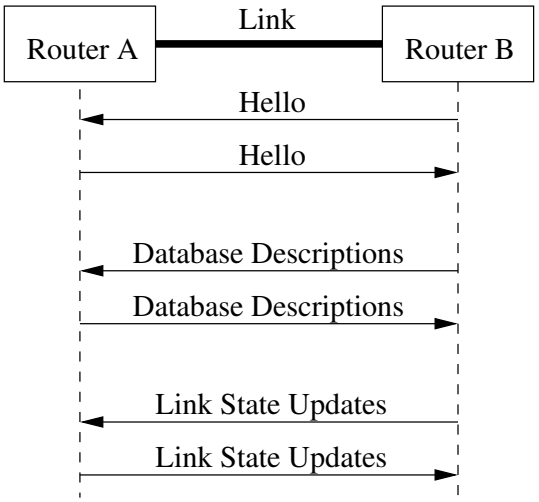


Figure 8.2: Basic OSPF Exchanges

smaller than they are in RIP, as they do not contain routing information, hence the bandwidth required when no topology changes occur is smaller.

If no HELLO frame is received from a neighbour within a certain amount of time, called the Dead Time, then the routes expire from the routing table. This in turn causes the shortest path to be recalculated and the changes to be propagated. We are assuming here than no Link State Update frames are sent, as there have been no alterations to the topology in other parts of the network. Both the HELLO advertisement rate and dead time are dependent on the underlying network type. The table below shows the HELLO period, and Dead Time for a number of network types. Included is the standard from which the numbers come and the number of losses that will cause a link failure.

Type	Hello Period	Dead Time	Standard	Losses
Point-to-Point Non-broadcast	30	120	Cisco	4
Point-to-Point	10	40	Cisco	4
Broadcast	10	40	Cisco	4
NBMA	30	120	RFC	4
Point-to-Multi-point	30	120	RFC	4

As you can see, in all cases, four losses in a row will cause the link to assumed to be down. This is actually lower than that of RIP, but you have to remember that the bandwidth required is lower. The effects and causes are the same as RIP.

8.2.4 QoS Solutions

As you can see from the examples above, the main cause of router flap is packet loss; using QoS we should be able to protect against this more efficiently than best effort can achieve.

In this framework we classify traffic by two metrics: cherish and urgency. Routing protocol traffic is highly cherished but not urgent. This is because the traffic is sensitive to loss, especially when there is a high contention ratio, but it has a number of seconds to reach its destination before a failure occurs. For this reason in our architecture we would place this traffic in the most cherished but least urgent class.

Using other QoS techniques we could do the following:

- Priority Queueing (separate buffers). Here we would place the routing traffic in the highest class, preventing loss. As the bandwidth required is low, problems to do with starvation of the lower queues are unlikely.
- Push-out buffering. Ensure that the routing protocol traffic is always able to take buffer space. We are not concerned with the delay that would be suffered. However, this is detrimental to other traffic.
- WFQ. Assign the routing traffic a class of its own with a low bandwidth. This is assuming that each of the classes has its own buffers.

8.3 NTP and QoS

Network Time Protocol (NTP) [69] allows a computer's local clock to be synchronised to the correct time. This is achieved by synchronising with one or more time servers across the Internet. What follows is a brief description of some of the terminology and working of the NTP protocol.

Each NTP server has a concept of how accurate its time source is. If it has direct access to an external time source, such as a caesium clock or GPS receiver, then it is considered a stratum zero server. The stratum is a measure of how far away from an accurate time source you are; so if you synchronise to a stratum zero server your stratum is one and so on.

An NTP server has its own virtual reference clock. In the case of a stratum zero server this is synchronised to the external time source. Other NTP servers synchronise their reference clock to one or more other NTP servers having a lower stratum. This reference clock is then used to set the system time, in small increments.

NTP is a UDP based request-response protocol. A client sends a request to a server, the server then processes this request and returns a response. When sending out a request, the client stores its own time (originate time-stamp) into the packet being sent. When a server receives such a packet, it will in turn store its own time (receive time-stamp) into the packet, and the packet will be returned after putting a transmit time-stamp into the packet. When receiving the reply, the receiver will once more log its own receipt time into the packet.

From this exchange the client can calculate the transit time of the packet. This is estimated to be half of the total delay minus remote processing time, assuming symmetrical delays. This is then used to calculate the time offset between the two machines. Clearly a number of these exchanges have to take place before the offset in the time can be averaged, and considered valid. NTP has some sophisticated methods for doing this, which are outside the scope of this discussion.

The client uses this averaged offset, and that from other servers, to synchronise its own reference clock. Finally it uses this reference clock to set its own local clock, as described above.

NTPs ability to accurately synchronise time is dependent on how accurately it estimates the offset in the clocks. This is in turn dependent on the round-trip time calculations. The shorter and more predictable the round-trip time is, the more accurate the estimate will be. Essentially we want the round trip time to have as little jitter as possible.

In our approach to QoS we would place NTP in the most urgent but least cherished class. The motivation for this is that we would rather the NTP packet did not arrive than be delayed. Additionally there are very few other types of traffic that have similar requirements; this means that there will be little contention for the outgoing link, hence, the jitter will be lower. Jitter will still be introduced as a result of the service of the preceding packet (which may be in another class) and any shaping.

8.4 VoIP and QoS

When referring to Voice-over-IP (VoIP) in this thesis we are generally talking about the H.323 protocol suit. However, this does not exclude the discussions from applying to other types of voice or video conferencing and streaming technologies. H.323 was chosen as it is an industry standard and one of the more interesting real time applications.

H.323 is an umbrella that covers a number of protocols. From a QoS perspective we are most interested in the data streams containing voice data. As such we do not consider protocols that are responsible for call signalling, although it is possible to do so. H.323 is not covered in any great detail here, due to its complexity.

H.323 carries audio data encapsulated in an RTP [90] data stream. This has an overhead of approximately 58 bytes on an Ethernet LAN [3]. The amount of data carried in each segment is dependent on the codec in use. Throughout we will be using the G.711 codec at 64Kbps. Samples are sent at a rate of 50Hz, so at 64Kbps we have a raw packet size of 160 bytes. Adding the protocol overhead to this gives us a packet size of 218 bytes sent at 50Hz intervals; this corresponds to a data rate of 87.2Kbps.

For simulation purposes VoIP traffic is much easier to generate than other traffic. This is because it sends a fixed sized packet at fixed intervals. The size of the packet may vary between connections depending on the codec, which is chosen during call setup, in use. During the simulations in this thesis we will assume that VoIP calls are always placed using the G.711 codec at 64Kbps.

VoIP traffic has a number of simple quality constraints [97]. These are as follows:

- The end-to-end delay, including all processing costs, must be less than 150ms.
- The inter-packet delay variation must be less than 60ms.
- The loss rate must be less than 1%, with no more than three losses in a row.

The end-to-end delay is constrained by our (human) ability to detect delay in a two way conversation; if the delay is less than 150ms we will not notice it. The restrictions on the jitter are to ensure that audio samples arrive in good time at their destination. The loss has to be low, so as not to lose too much data. The requirement for less than three losses in a row will ensure that there is not a serious glitch in the audio quality, however, this may depend on the codec in use.

In the loss-delay model we would chose to place VoIP traffic in the highest class, that is cherished and urgent. We chose cherished to minimise both overall loss and burst loss, so long as the class is not over contended there should be very little loss. We chose urgent both to keep the end-to-end delay low and to minimise jitter. In the most urgent class we will only have to wait for the same or any other classes to finish processing the current packet before we receive service.

In bandwidth-centric networks we can only allocate the VoIP call the correct amount of bandwidth. This may protect against loss where there are separate buffers for each class, otherwise it must compete with the other traffic. The delay is dependent on how many other classes have to be serviced, and this is extremely hard to predict in advance. Additionally where the VoIP class has to wait for other classes to service jitter may be introduced. The amount of jitter is dependent on the number of classes and properties of their instantaneous traffic characteristics, and cannot be determined beforehand.

[3] 20 bytes of IP, 8 bytes of UDP, 12 bytes of RTP and 18 bytes of LAN.

8.5 HTTP and QoS

HTTP is perhaps one of the hardest applications to model convincingly. Being based on TCP makes its packet by packet behaviour hard to simulate. We know that the length of a connection is likely to be Pareto distributed, as are the gaps between successive connections from the same source. During this thesis we will model HTTP connections as a long running transfer of data using a greedy source. The greedy source attempts to use as much of the bandwidth as possible at any given time; this is intended to model the behaviour of TCP.

TCP effectively turns loss into delay; that is, any losses are hidden from the user and extra delay is introduced. From the point of view of the user of TCP this decreases the throughput. The problem with applications like HTTP is we wish to maximise the throughput and minimise the delay; users simply do not like waiting for pages to load.

In this thesis we use HTTP as our best effort traffic, not cherished and not urgent. HTTP does not have as strict loss requirements as VoIP or routing protocols, due to TCP being a reliable transport protocol. It also does not have strict delay requirements in the same way VoIP does. It is for this reason that we consider it as best effort.

Another important reason for making HTTP best-effort is that it is perhaps the most common application in use on the Internet. As it stands HTTP does work reliably for most purposes, however, applications like VoIP are harder to support. We believe that there needs to be a portion of the total network traffic that is best-effort, as this allows the higher quality classes to borrow spare capacity when required.

8.6 IP ToS Mapping

In order to provide basic support for classes of service to the Internet Protocol [84] part of the IP protocol header contains what is known as the ToS (Type of Service) bits. These are defined in [4]. This RFC defines four classes of service described below:

1. Minimum Delay. This class is used when the time a packet takes to travel from its source to its destination is most important.
2. Maximum Throughput. This class is used when the volume of data transfered in any given period of time is most important.
3. Maximum Reliability. This class is used when some certainty is required that the data reaches its destination without retransmission.

Class of Service	Application
Minimum Delay	telnet/ssh ftp control SMTP control UDP DNS query
Maximum Throughput	ftp data SMTP data DNS zone transfer
Maximum Reliability	IGP SNMP
Minimum Cost	TCP DNS query ICMP EGP BOOTP

Table 8.1: RFC1060 ToS Mappings

4. Minimum Cost. This class is used when it is important to minimise the cost of data transmission.

To specify which class of service is required a bit is set in the ToS field of the IP header. If none of the bits are set the the packet will receive "normal service"; otherwise known as best-effort.

While this RFC specifies the value of the ToS field, by defining the classes above, it does not mandate that other interpretations of the field are not allowed; this is important as it allows the field to be re-used in the future. It does however fix the size of the field to four bits, updating several previous RFCs.

It is important to note that these bits are intended to provide a way of allowing an application to describe its desire to receive a particular class of service. The network does not have to honour this, indeed, it may not even support this functionality.

8.6.1 Setting the ToS bits

The assigned numbers RFC [88] specifies what class of service well known applications should use. Table 8.1 shows the class to which some well know applications belong.

8.6.2 ToS to Loss-Delay Mapping

Given the definitions of the classes of service for the ToS fields we can map these classes into the classes for the loss-delay model. Figure 8.3 shows the loss-delay grid with the names of the classes of service in place.

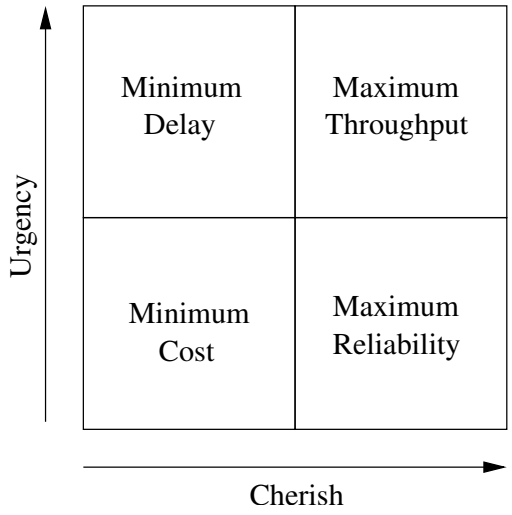


Figure 8.3: Loss-Delay ToS Grid

The reason for these mappings are as follows:

- Minimum delay means that we should service the packet as soon as possible. This class does not specify any constraints on loss. We have therefore classified this class as urgent but not cherished.
- Maximum reliability means that we should always admit the packet into a queue. This class does not specify any constraints on delay. We have therefore classified this class as cherished but not urgent.
- Minimum cost is taken to mean using less resources. In our model resources are buffer capacity and service capacity. Therefore we have classified this class as not urgent and not cherished.
- Maximum throughput is harder to classify. We know that throughput is related to both loss and delay. To achieve the highest throughput we require the lowest loss and lowest delay. We therefore place this class in the cherished and urgent class.

8.6.3 Important Observations

It is important to realise that the Maximum Throughput class will dominate all other classes if this mapping is implemented. This may not be what was originally intended when some well known applications were placed into classes.

For example, let us assume that there is a long running FTP transfer, which is mapped into the Max. Throughput class. In this instance a telnet session could appear to be very slow! This is

because both the FTP transfer and the telnet session are at the same level of urgency. It is likely that there are significantly more ftp packets than telnet packets; this will result in the telnet packets having to wait longer for service.

Taking the same long running FTP session, we may also expect to see problems with SNMP queries. This is because the FTP session and the SNMP query are at the same level of cherish. As a result, they will both be contending for the same buffer capacity, and as the FTP session will have more packets in flight, it is more likely to succeed; this is assuming that the SNMP query is a low bandwidth flow.

The problems do not stop there. Our FTP transfer completely dominates the Minimum Cost class, which is not cherished and not urgent. As a result, any traffic in this class is likely to be discarded or suffer a long delay.

Of course, this may not actually be a problem, as it depends on how important the FTP transfer is in comparison to the other traffic. While the mapping we have presented will give the application the service that it requires, it may do so at an unacceptable cost to other traffic.

There are two ways of tackling this problem. The first is to change the class of service applications request to better reflect the priority of the application. The second is to cap the available bandwidth of each class so as to prevent unacceptable starvation of other classes. It is likely that a combination of these two approaches will be the most successful.

8.7 Summary

In this chapter we have looked at the quality requirements for a number of different types of traffic. We have seen that these can easily be classified in terms of cherish and urgency requirements. This provides us with a simple to understand way of prioritising network traffic. We may also need to capture additional information like the number of consecutive losses that can be tolerated, or the inter-packet jitter that is acceptable.

We have provided a brief description of the following protocols, as well as assigning them to a cherish-urgency class:

- Routing Protocols - cherished but not urgent.
- NTP - urgent but not cherished.
- VoIP - cherished and urgent.
- HTTP - not cherished and not urgent (best-effort).

In networks that provide QoS support by bandwidth there is no way of trading loss and delay. We can only assign each application a bandwidth dependent on its expected requirements. This is in contrast to networks that allow loss-delay trading.

Chapter 9

Differential Quality

9.1 Introduction

In this chapter we are going to develop a simple case study to demonstrate the techniques we have presented so far. Using these techniques we will show how it is possible to construct a network that is capable of delivering QoS support to a number of applications with high confidence.

We deviate from previous approaches here in that we will not calculate the performance of the network beforehand. As we have shown it is not currently possible to accurately calculate the performance of networks that transport mixed packet sizes. However, the author believes that this will become possible in due course.

Our aim is to show in this chapter that, using the principles we have already shown, we can construct a network with QoS support that is better than other well known approaches. By using a fixed topology and mix of traffic we will be able to investigate the effect of using different queueing disciplines. A given network has a fixed amount of quality it can provide, we call this the Intrinsic Quality (see section 4.3), by choosing different queueing disciplines we change how this quality is shared out. By managing the two degrees of freedom we will show that it is possible to provide better QoS support than is possible with bandwidth alone.

9.2 Assumptions

9.2.1 Traffic Distributions

In previous chapters we have restricted ourselves to using exponentially distributed traffic. The motivation for doing this was to allow us to explore the mathematical properties of the queueing

systems that we have presented. In this chapter we are interested in the performance of more realistic traffic, and for this reason we do not dictate a distribution. We will however use traffic distributions that are aligned with the real world behaviour of the applications in question.

9.2.2 Traffic Loading

The network we will present carries a mixture of different types of traffic, each with their own constraints. It is important that we do not place an unrealistic load on the network. For example to load a network at 100%, when 90% of the traffic is VoIP, is unrealistic. The reason for this is that it is extremely unlikely that we would be able to meet that guarantees of that many VoIP calls. A good way of approximating how much traffic, of a particular type, a network can carry is to ask if it is possible to meet the guarantees only with that traffic type being carried. If it is not then it is almost certainly not going to perform correctly when other traffic is also present.

An important observation about running a multi-service network is that it is important to have a portion of the traffic as best effort. Without this best effort traffic we have no one who can receive a worse treatment. The provision of quality is the absence of degradation, for a given intrinsic quality we can chose who receives the worse treatment; best effort traffic provides such a possibility.

Finally we will arrange for the network to be heavily loaded at the edges to more accurately represent the loading on real networks. The centre of the network while moderately loaded should not be in heavy contention.

9.2.3 Fixed Traffic and Topology

Firstly we are assuming a fixed topology. That means that there are no sudden appearance or disappearance of routers. It also means that traffic will always take a well known path. Secondly we assume that the number of streams, and hence flows, is fixed for the duration of the case study.

Both of these things are relatively unlikely in a real network. However; they are both ultimately supportable. It would just involve re-calculation on a topology or traffic change. This would take us into issues such as signalling, which while interesting is not the thrust of our example.

In some senses we could allow changing traffic, so long as it does not exceed the maximum we have provisioned for (which requires policing). In this case we would give worse case guarantee. To allocate to peak in this way requires us to have an understanding of how large the peak is, when it occurs and how long it is likely to last for. This then leads us into over booking, traffic trends and alike. Again this is not the thrust of the example. Using the techniques that we present in this thesis would allow us to make inroads into these problems; we leave this as an area for future research.

9.2.4 Packetisation and Transmission Costs

For the purposes of this example we ignore the cost of data (de)packetisation at the sources and sinks. These costs are related to the size of the packet that is to be sent. While this means that the results are less realistic, it would not be hard to factor this in if necessary.

We also ignore the costs of packet transmission. This means that packets move across a link in zero time. Transmission times are fixed per byte, with some overhead per packet. We have chosen not to model this for simplicity, again if necessary it could be factored in. In both cases we are interested in the ability of the network to support differential service.

9.2.5 The traffic is well behaved

As we mentioned above the traffic arrival rate and distribution is fixed. Additionally we assume that we have managed to make them not break their guarantees. So we do not let more traffic arrive than we intended. This is easy in a simulator but harder in real life.

9.3 Trade-offs

This section provides a review of the trade-offs introduced in section 4.7. When constructing the example that we use in this chapter we have used a number of these trade-offs to improve the expected performance of the network.

9.3.1 MTU Size

The MTU of the network, that is the largest size packet that can be sent, affects our ability to make quality guarantees. Reducing the MTU size of the network also reduces the average service time of the packets. This reduction in service time leads to more accurate guarantees. Large packets have a large residual service time, any small packet behind a large packet is likely to experience a delay that is much larger than we would have predicted in the mathematics. This for the purposes of our example is non optimal.

For this reason we will chose to set the MTU of the network to 512 bytes. The choice to use this size was motivated by measurements of packet sizes on the Internet. Most packets in reality have less than 64 bytes of payload, so we would not effect the majority of packets. The same study [95] showed that peaks in packet length histograms occurred at 64, 128, 256, 512 and 1500 bytes. By choosing 512 we only cut off one common size of packet, which accounts for about 10% of the traffic.

It is most likely that 1500 byte packets are generated by file transfer applications such as FTP[1]. This means that we will decrease the efficiency of such applications. However; the benefit is better guarantees for traffic that has real time constraints. We feel that this is an acceptable trade off as ultimately we are more interested in getting applications, such as VoIP, to work predictably.

9.3.2 Buffer Sizes

The buffer sizes found in queues on a network affect the reliability of the quality guarantees, as well as our ability to provide some levels of service. This is an unavoidable fact, as there is a relationship between the number of buffers and the observed delay and loss; less buffering implies more loss, more buffering implies more delay. Choosing an appropriate buffer size is therefore important.

If the buffer sizes are too small then at higher loading factors we are likely to see an unacceptable high level of loss. Adding more buffers in the face of continued over utilisation will not help reduce the loss rates, it simply delays the problem for a few milliseconds. One benefit of small buffer sizes is that it gives the queue a smaller memory. This means that the effect of short bursts of over-utilisation do not effect us for long periods of time.

If the buffer sizes are too large then we are likely to get unacceptably large delays. What is worse is, because the queue has a longer memory, we are more likely to suffer from longer periods of recovery from short term overload; this has the effect of increasing the variance which is also undesirable.

What is required is a buffer size that is large enough to prevent unacceptable loss in short bursts, yet small enough not to increase the delay and variance unacceptably. A buffer size of 200 should be sufficient; at a loading factor of 100% it has a loss ratio of around $\frac{1}{2}\%$ (assuming an M/M/1/200 queue). This is sufficient to prevent short term losses. At the same loading factor there is an average service time of around fifty times the average service rate.

9.4 The Topology

The topology we will use as our example is show in figure 9.1. It depicts a number of sources sending data across a set of contended routers to a sink. Starting with a source on the left we traverse a fan-in router followed by a backbone router, then the backbone link; then the same in reverse until we reach a sink.

For simplicity we only model one direction of data flow. Each of the sources sends a mixture of traffic, to one or many sinks. We will explain more about the nature of this traffic later in the section.

[1]Where HTTP is used to download large files it will suffer similar reductions in performance.

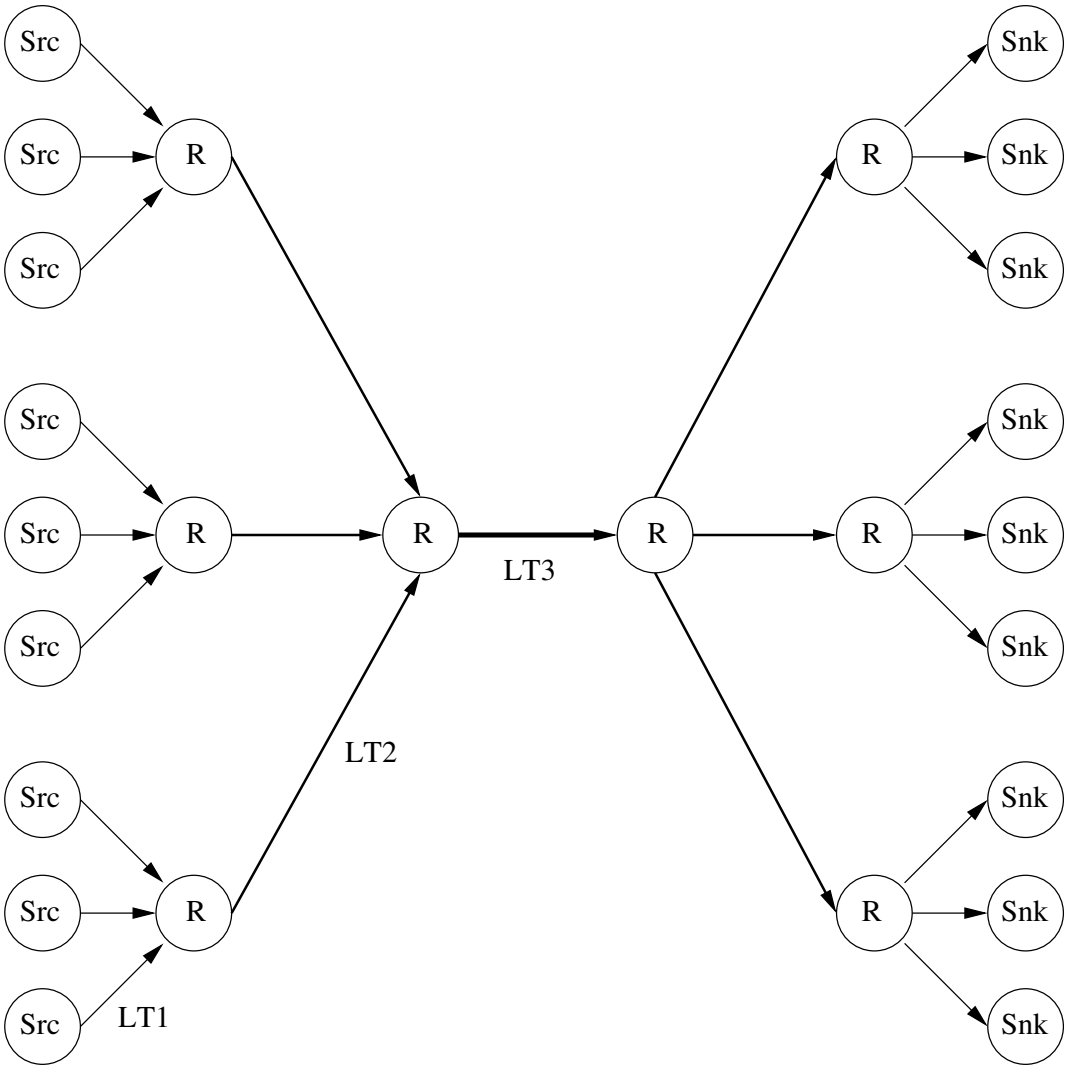


Figure 9.1: The Network Topology

This example has links of three different speeds. The link rates of these are 512Kbps, 1.544Mbps, and 6.312Mbps respectively, which we have labelled LT1-LT3 respectively. Queues feeling into these links are serviced at the rate of the link.

In this example the sources produce enough traffic to keep their LT1 links loaded to full capacity. This allows us to calculate the loading factor that each of the routers is under. The fan-in routers accept three LT1 connections and multiplexes them onto a single E1 line, this means that their loading factor will be 0.99 Erlang's. For the fan-in backbone router the loading factor is around 0.73 Erlang's. The fan-out routers are less loaded as they have less contention for bandwidth in the direction of flow.

We have chosen to place the multiplexing points at, or close to, saturation. In this state resources are highly contended. It is in this kind of situation that is a good test of the success of QoS schemes.

9.5 Simulation Traffic

For the purposes of the example we need to decide on a number of types of traffic that we are going to support. Seeing as we have some information on the constraints of the following sorts of traffic they would seem an obvious choice. The quality traffic types that we will support are:

- VoIP calls
- HTTP traffic
- NTP traffic
- RIP traffic

Remember that the ratios of these sorts of traffic are constrained by the intrinsic quality of the network. VoIP is likely to be the most problematic, seeing as it is high bandwidth with stringent loss, delay, and jitter requirements. NTP and RIP are much lower bandwidth, and have only one set of constraints, i.e. delay and loss respectively. Finally HTTP would ideally like low loss and plenty of buffer space to achieve maximum throughput; however, in this example we are going to consider it as best-effort because we are more interested in the performance of applications like VoIP.

9.5.1 VoIP

In the simulation two VoIP calls will be made from each source to the sink directly opposite. In total there will be 18 concurrent VoIP calls running.

For VoIP we need to evaluate two things: the delay variation and the burst losses. If either of these is too high then we may consider a call to be a failure. We are also interested in the probability of call failure for a number of calls, as in reality this is a more useful measure for network providers.

In this example we chose to model VoIP calls using a G.711 codec. The codec has a sample rate of 64Kbps. The samples are packetised at a rate of 50Hz, ie. 50 packets a second. This produces raw packets of 1280 bits, or 160 bytes, at a rate of 50 a second.

These raw packets have additional headers added to them. We are assuming that the VoIP call is placed using H323, and as such is carried by an RTP stream. This would impose overheads for LAN, IP, UDP, RTP of 18, 20, 8, 12 bytes respectively. This gives an extra 58 bytes of additional overhead; bringing the packet size to 218 bytes. Our VoIP call has a packet rate of 50pps and a packet size of 218 bytes. This results in a data rate of 87.2Kbps.

For VoIP to deliver the best voice quality it requires that the jitter of the incoming packets is less than 60ms and an end to end delay less than 150ms. Minor packet loss can be tolerated, but burst losses of greater than three cause a significant loss of quality. These quantities are measured over a number of three minute intervals, each representing a VoIP call. If the jitter exceeds 60ms or their is a burst loss of more than three, we say the call has failed.

9.5.2 NTP

In the simulation each source sends NTP traffic to each of the sinks, resulting in 81 concurrent NTP sessions. While this may seem a lot of traffic NTP has a low bandwidth, in total this accounts for less than 1% of the total network capacity.

For NTP we want the delay to be as low and as predictable as possible. It is hard to quantify the results of this protocol, but the mean and variance of the delay should be sufficient to give an indication of the performance.

We model NTP as sending one packet every 10 seconds to each peer. With a packet size of 64 bytes. This corresponds to a data rate of around 51bps per peer.

9.5.3 RIP

In the simulation RIP packets are sent from each source to all other sinks. Again this results in 81 concurrent RIP sessions where the total is less that 1% of network capacity.

RIP sends a packet every 30 seconds to each of it neighbours. If a neighbour does not receive a packet in 180 seconds then the routes are dropped and there has been a failure. This constraint would be measured over the whole timescale of a simulation.

We assume that these RIP packets are 128 bytes in size. In a real network their size would be determined by the number of routes that RIP advertises. At one packet every 30 seconds the data rate is 34 bits a second.

For RIP we are most interested in the burst losses. What we want is there to be at least a packet every 180 seconds, so that the routing tables are not changed as a result of a route not being updated.

9.5.4 HTTP

In the simulation we take HTTP traffic to be best-effort. This means that we will not evaluate the performance of HTTP in any detail, we will however provide some indication of the likely throughput of an HTTP session in the network. Each source sends HTTP data packets to all of the sinks, this is similar to RIP and NTP. We are choosing only to model the data part of HTTP, in effect we are assuming that acknowledgement packets do return successfully. We shall fix HTTP packets to be the MTU of 512 bytes. The bandwidth used by HTTP will be the remaining capacity after the other applications have taken their share.

As we are modelling HTTP, which is based on TCP, we are going to use our adaptive source. This source will attempt to increase its bandwidth when it receive 8 successful packets in a row, and two losses in a row cause it to decrease. The receiver is able to change the bandwidth of the sender in the simulator without transmitting additional packets, for this reason it only models on half of the interactions. While this is a extremely simple model it does capture one of the main features of TCP, namely its greediness. It is hoped that this will provide a reasonable model of TCP for our purposes.

9.6 FIFO Queueing

The first set of tests that we have run use FIFO queueing throughout the network. Although we are interested in differential queueing in this chapter we have run these tests to provide a base line comparison of the topology and traffic. All the queues have 200 buffers that can be used by any of the applications. We ran tests with Deterministic and Markovian service discipline to provide a basic comparison.

Each of the tests were run 10 times with different random numbers generator seeds. For each of the applications we produce an average result. We can do this because the topology is symmetrical; that is any path between a source and a sink is identical, a flow would cross the same type of queues and links. We will use this approach again in the rest of the chapter. Table 9.1 shows the results from this set of tests.

Service Discipline	Application	Delay	Delay Stdev	IPT	IPT Stdev	Loss Prob.
Markovian	VoIP	1.552s	0.391s	0.021s	0.018s	0.056
	NTP	1.536s	0.450s	10.809s	2.964s	0.070
	RIP	1.538s	0.451s	32.440s	8.499s	0.069
	HTTP	1.570s	0.450s	0.109s	0.103s	0.063
Deterministic	VoIP	1.622s	0.369s	0.021s	0.012s	0.053
	NTP	1.609s	0.402s	11.139s	3.625s	0.097
	RIP	1.610s	0.402s	33.387s	10.321s	0.095
	HTTP	1.641s	0.407s	0.108s	0.131s	0.070

Table 9.1: Results for FIFO Queueing

For each of the service disciplines you will notice that the delay for each application is broadly similar. This is not at all surprising as all the flows are treated the same by the queues, as a result they will see the same average queueing time. The effect of the packet size of an application can also be seen in the delay. HTTP for example has the largest packet size, and as a result has the largest delay; this is because the larger packets take longer to service, so the delay increases relative the the others. The other applications VoIP, RIP and NTP have 218, 128 and 64 byte packets respectively; they also have progressively smaller delays due to their packet size.

Another perhaps surprising result is that the average delay and delay standard deviation are of the same order of magnitude for both service disciplines. This is due to the heavy traffic approximation which states that waiting time in any queue will tend to be exponential when heavily loaded. The heavy traffic approximation is based on the G/G/1 queue, which is an approximation to the G/M/1/k and G/D/1/k queues that we are using here.

The loss probability for each application within a service discipline is of the same order of magnitude. One would perhaps expect to see the loss probability for each of the applications to be the same, as they cross the same queues. However this is not the case because the loss probability depends on the amount of arriving traffic of each application and its packet size. The loss probability for NTP is the highest as it has the smallest packet size. In decreasing probability we would expect to see RIP, VoIP and HTTP. However, as the HTTP flow is controlled by our greedy source it has a higher loss as it attempts to use as much of the available bandwidth as possible.

As you can see from the results, most of the applications are likely to fail. VoIP has an end-to-end delay of 1.5s. NTP has 450ms of end-to-end delay variation. RIP may work correctly depending on the distribution of losses. HTTP will of course work correctly as it is based on TCP, however, it may not be all that fast. We do not analyse the results further as they are only meant to be a base comparison.

9.7 Differential Queueing

The second set of tests have been run with queues that can support differential service. We compare two queueing disciplines here: FQ and loss-delay multiplexing. The implementation of Fair Queueing that we are using is Deficit Round Robin (DRR). This was chosen primarily for its ease of implementation, to avoid any mistakes in the implementation. The loss-delay queue is implemented as presented in chapter 2.7.2.

The loss-delay queue has 200 buffers; all of which are available to the cherished class, the non-cherished class may only use the first 100 buffers. The queue is also serviced in strict priority order, where the servicing priority is separate from the cherish priority. This means that the queue has four classes of service, we place the applications in the following classes:

- VoIP is Cherished and Urgent
- NTP is Not-cherished and Urgent
- RIP is Cherished and Not-urgent
- HTTP is Not-Cherished and Not-urgent

The DRR queue works by dividing the bandwidth of the outgoing link between the different classes. Each class has its own set of buffers, and therefore only has to contend for outgoing link capacity. A class is allocated 100 buffers, which only that class can use. This means that the queue has more than 200 buffers, which could be considered unfair. However, as VoIP and HTTP form the majority of the traffic it would make a poor comparison to only allocate them 50 buffers. Both the FIFO and loss-delay queueing have 200 buffers that can be shared, approximately half being used by HTTP and the other by VoIP; it is for this reason that we allocate 100 buffers per class. Its worth pointing out that this will make the loss for NTP and RIP extremely small, as they account for just a small fraction of the transported traffic. For each of the applications we allocate an percentage of the outgoing link, as we know what traffic is flowing we can allocate each application its share. The percentages for each application are as follows:

- VoIP 35%
- NTP 1%
- RIP 1%
- HTTP 63%

Table 9.2 shows the average results per application and service discipline for this set of tests. We shall now examine each of the results for each class in turn before presenting a comparison between them.

Service Discipline	Application	Delay	Delay Stdev	IPT	IPT Stdev	Loss Prob.
FQ	VoIP	0.538s	0.241s	0.020s	0.085s	0.017
	NTP	0.818s	0.413s	10.055s	0.565s	0.000
	RIP	0.820s	0.412s	30.196s	0.585s	0.000
	HTTP	0.977s	0.453s	0.112s	0.155s	0.071
Loss-Delay Markovian	VoIP	0.024s	0.012s	0.020s	0.014s	0.000
	NTP	0.025s	0.013s	10.703s	2.584s	0.060
	RIP	1.335s	0.536s	30.196s	0.756s	0.000
	HTTP	1.353s	0.521s	0.115s	0.130s	0.067
Loss-Delay Deterministic	VoIP	0.011s	0.003s	0.020s	0.005s	0.000
	NTP	0.012s	0.004s	11.008s	3.231s	0.086
	RIP	1.543s	0.482s	30.196s	0.678s	0.000
	HTTP	1.543s	0.480s	0.111s	0.134s	0.070

Table 9.2: Results for Differential Queueing

9.7.1 Fair Queueing

In a FQ the delay you receive is dependent on the percentage of the outgoing link you are allocated and the utilisation of the other classes. If you have been allocated a small percentage of the outgoing link then it may take some time to accrue enough tokens to send a packet. How often you are checked for service, that is, when it is your turn if you have enough tokens, is dependent to some extent on how busy the other classes are. If another class is using all its allocation then you will have to wait until it has finished to be serviced.

You will notice that the delays for NTP and RIP are quite high. This is because they have to wait for other classes to finish servicing. It could also be caused by having to wait for enough tokens to be serviced, but both NTP and RIP are over allocated by a factor of 10 in this case (as they account for around 0.1% of the traffic and are allocated 1%). The delay for HTTP is higher as it is likely to have a longer queue length, as can be seen from the loss probability, which means that there is more queueing delay than other classes. Indeed this is why the delay for VoIP traffic is lower. You will also notice that the delay variation is broadly the same as the FIFO queueing. This is not surprising as each class must still wait for other classes to finish servicing, in FIFO servicing packets have to wait for preceding packets to be serviced.

If we convert the IPTs into bandwidths, by inverting them and multiplying by the packet size in bits, we find that the percentage of bandwidth used by each application is the same as we allocated. This is not at all surprising as this is essentially what FQ does - share the link bandwidths between the classes. The standard deviation is also better than FIFO queueing, again FQ attempts to make the shares of the outgoing link fair.

The loss probabilities for NTP and RIP are, unsurprisingly, zero. As both these applications have been allocated 100 buffers each and their bandwidths are so low then the probability of queue

overflow is extremely low. The loss probabilities for VoIP and HTTP are higher than those of NTP and RIP, which is again unsurprising as they use a larger share of the bandwidth. As there is isolation between HTTP and VoIP in this type of queue the loss probabilities are lower than those of FIFO.

When it comes to the performance of applications under FQ they do not work any better than they do under FIFO queueing. The delays are better under FQ, than FIFO, as each application has an isolated queue; this prevents our greedy sources from filling all the buffers. VoIP has an IPT standard deviation of 85ms, which is larger than the 60ms target. NTP is unlikely to come up with accurate time estimates if the end-to-end delay standard deviations are nearly half a second. RIP would work correctly as there is no loss. HTTP does fair slightly worse under FQ, at least in terms of throughput, this is due to the isolation in FQ that prevents it from gaining a higher share of the capacity.

The important point to understand from this example is that, although the DRR queue did what it was supposed to, namely divide the outgoing link bandwidth, it was insufficient for the types of service that we required. Although the algorithm is functioning correctly it did not have the result we expected. We could increase the performance of the VoIP by increasing its portion of the outgoing link, this would lower the IPT standard deviation. As for NTP we could do the same, but it may still suffer due to the residual of other classes service time. Ultimately it is quite hard to get a feel for the behaviour of FQ algorithms in terms of delay and jitter.

9.7.2 Loss-Delay Queueing

We ran the loss-delay queueing with two different servicing disciplines: Markovian and Deterministic. The latter is to enable a more accurate comparison with FQ, which is also Deterministically serviced.

It is clear immediately from the results that the classes received the treatment that they should have. Looking at traffic in the urgent class, that's VoIP and NTP, we can see that the delay is dramatically lower than the non-urgent class containing RIP and HTTP. Looking at traffic in the cherished class, VoIP and RIP, we see that the loss is zero; while the loss for the non-cherished class, containing NTP and HTTP, is non-zero. This is exactly the behaviour we would have expected.

Comparing the Markovian and Deterministic servicing disciplines, we can see that the delay for the urgent class when serviced Deterministically is roughly halved. However, they are increased for the non-urgent class. The delay standard deviation for Deterministic servicing in the urgent class is decreased, but in the non-urgent it is also decreased but not as much. By servicing the queue deterministically we decrease the delay and variance, as there is no exponential element introduced that would increase them. However there will always be a variance in this type of system due to the packet sizes. This is why the non-urgent class does not improve that much, as it is still effected by the variation in service of the urgent class.

You will also notice that the loss probabilities change when we service the queue Deterministically rather than Markovian. This only effects the non-cherished traffic here as the other classes have no loss. The loss is higher in the deterministic case as the end-to-end delay calculations performed by the adaptive source are not as good. The Markovian servicing distributes the delays in such a way as to make the simple end-to-end average more accurate. We intend to do some more investigation to see if the same holds for real TCP.

When it comes to supporting differentiated service the loss-delay queue clearly wins. VoIP requires an end-to-end delay of less than 150ms, an IPT variation of less than 60ms and low loss. All of these are met with ease. However, it is worth noting that adding more VoIP flows to the cherished and urgent class would degrade the service for all other classes. Using this class in a real network would require policing without a doubt.

NTP requires an accurate estimate of the end-to-end delay, for this reason we want to minimise the end-to-end variance. In the loss-delay queue the end-to-end standard deviation is only 13ms, compared to around 400ms in the FIFO and DRR queues. While it is hard to predict how accurate the time estimate would be with a deviation of 13ms, it will likely be better than 400ms. As NTP is not-cherished it does suffer loss, but this is of the same order of magnitude at FIFO and DRR queueing.

RIP requires predominately low loss, it is important that a packet arrives at the destination in the order of 100s; clearly in most cases the end-to-end delay is unlikely to be this high. As RIP is cherished it suffers no loss, and as it is non-urgent it suffers 1.5s of delay. In these conditions RIP is likely to work as there is no loss.

HTTP is mainly concerned with throughput, we do not attempt to relate that to the actual amount of data that is transfered at the session level. The throughput for HTTP stayed about 36.5Kbps in all of the tests in this chapter. From the point of loss-delay queueing this means the application performed no worse than it did in other disciplines.

9.8 Conclusion

In this chapter we presented a simple network that carries a well known set of traffic. We allowed ourselves to change the queueing discipline in order to evaluate the performance, in terms of QoS, of the different disciplines. Using the requirements set out previously we then evaluated the results of each test.

In the first set of tests we used FIFO queueing with Deterministic or Markovian servicing. We found that the results for the two different servicing disciplines were broadly the same. This is due to the heavy-traffic approximation. We used this set of tests as a base line comparison.

Next we changed the queueing discipline to a variant of Fair Queueing (FQ) called Deficit Round Robin (DRR). We found that this did behave as it was designed, namely it correctly shared the outgoing link to a number of classes. However, it failed to meet the majority of the constraints on the traffic, although it did better than FIFO queueing.

Finally we changed the queueing discipline to loss-delay queueing. We found that all of the quality constraints for the traffic were met, and usually with room to spare. This was possible because a loss-delay queue correctly manages the relationship between throughput, loss and delay. In addition it provides a simple to understand way of classifying traffic, and more importantly its emergent properties are easier to understand.

Thought this chapter we have kept the topology and the offered traffic the same. The differences in results are due to the performance of the different queueing disciplines only.

Chapter 10

Modelling Real Networks

10.1 Introduction

The purpose of this chapter is to demonstrate how it is possible to construct scenarios, such as those used by the Simulator and Calculator, to model a real network infrastructure.

We already know that it is possible to model network components using QDFs. We will show that to model most real world network devices requires more than one QDF. However, this is not a problem, as it is possible using this framework to compose multiple QDFs together to form a composite component.

Figure 10.1 shows a very simplistic network consisting of two interconnected switches and a number of terminals. Each of the terminals is the source and sink for a number of flows. The switches are connected together with a single link that carries traffic between them. Similarly, each of the terminals is connected to a switch using a link. In order to model this network we need composite models for each of the following:

- Terminals or other endpoints.
- Network Links, such as Ethernet.
- Switches and Routers.

In the beginning of this chapter we deal with each of these components in turn. In the final part of this chapter we will look at the issues involved in producing QDFs that can model different types of switches, and some of the QoS problems that each switch design introduces.

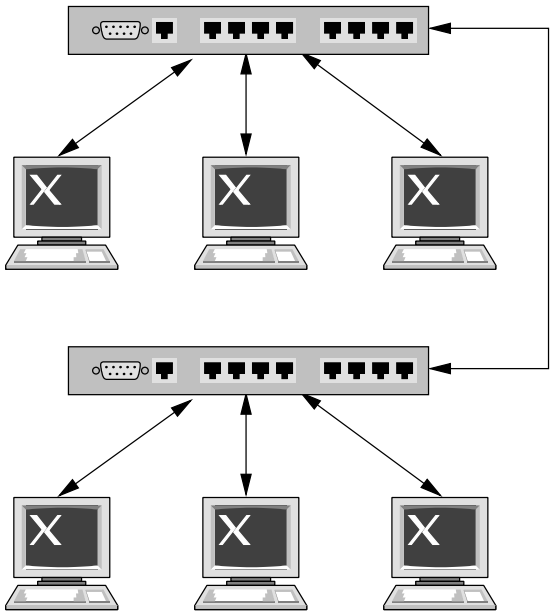


Figure 10.1: A Simple Network

10.2 Terminals and Endpoints

In this context we define an endpoint, or terminal, to be the source and sink of a number of flows. To model this in a scenario we would create a source and sink for each endpoint. We have already covered this in chapter 6.2.2, readers are directed there for further information. We would as an example add the following, for endpoint1, to a scenario file:

```
<source name="endpoint1_source" sendsTo="switch1">
  <flow id="1" rate="0.5" type="Poisson" lengthType="fixed" lengthMean="64"/>
  <flow id="2" rate="0.5" type="Poisson" lengthType="fixed" lengthMean="1500"/>
</source>
<sink name="endpoint1_sink">
  <consume id="3"/>
  <consume id="4"/>
</sink>
```

Note that two flows are generated by this endpoint, labelled 1 and 2, and a different two flows, 3 and 4, are sinked. This is because we treat a flow as a uni-directional transfer of packets. In this example we could be modelling two connections where 1 and 3, and 2 and 4 are paired together to form a bidirectional flow.

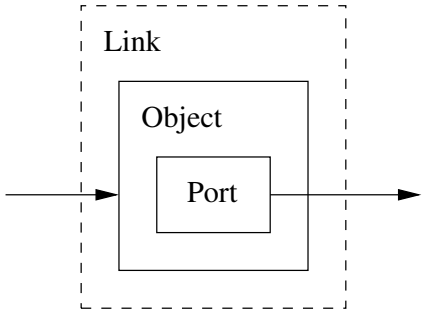


Figure 10.2: A Network Link

10.3 Network Links

Network links are used to connect together components on a network. We introduce a technology independent model here, which could for example model a point-to-point Ethernet link[1]. Network links primarily introduce a delay, due to transmission latency, which is dependent on the length of the link. They also introduce a small probability of loss, due to transmission corruption, but in general we ignore this as its effect is extremely small.

Figure 10.2 shows a diagram of how we would construct a link in a scenario. Note that this is again a uni-directional link, and another link would have to be used to model the opposite direction. The link consists of a single object which contains a single port. Traffic crossing the link arrives at the object and is degraded by the QDF specified in the port, finally the sendsTo attribute defines where the traffic should be sent. The following snippet of scenario shows a link transporting a single flow of traffic (labelled 1):

```
<object name="link0">
  <port name="port" qdf="ethernet" rate="1.0" sendsTo="switch">
    <select id="1" class="be"/>
  </port>
</object>
```

In this thesis we do not use this model of links. Instead we ignore the effects of transmission delay in favour of concentrating on the effects on flows caused by queueing. For this reason no QDFs to model links have been written, although doing so would be a simple extension.

[1] Modelling shared media access requires modelling the contention resolution processes; one approach to achieving this is to model queues with vacations.

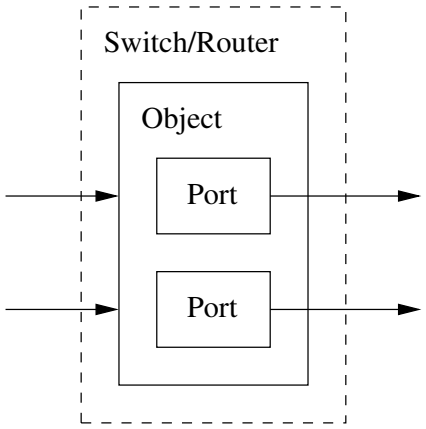


Figure 10.3: Modelling Output Queueing

10.4 Routers and Switches

In our model we consider routers and switches to be the same. We can make this generalisation because the path a flow takes across the switching fabric in either device is fixed in the scenario. This does limit us to static configurations, however, it is sufficient for our purposes.

Routers and switches can be constructed in a number of different ways. They can have buffers on the input, output, or both. They can also have a diverse set of switching fabrics. A full discussion of these matters is beyond the scope of this chapter. However, we will show how it is possible to model the following types of switches:

- Output queueing with a perfect[2] switching fabric.
- Input queueing with a perfect switching fabric.
- Input and output queueing with a perfect switching fabric.
- Input and output queueing with an imperfect switching fabric.

10.4.1 Output queueing

Output queueing is the simplest model considered here. We assume that there is a perfect switching fabric, ie. it introduces no loss or delay, which transports packets from the inputs to queues located at the outputs.

[2] A perfect switching fabric does not have any contention issues; that is, it does not introduce loss or delay.

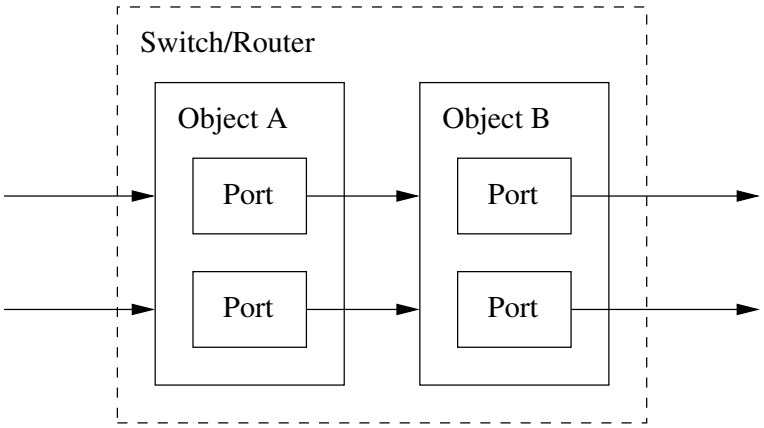


Figure 10.4: Input/Output Queueing

Figure 10.3 shows this graphically. Flows that arrive at this kind of switch do so at a single input object. This object then has a port, with associated QDF, for each of the output ports that we are attempting to model. Note that we send traffic directly to the object irrespective of which port it would have arrived on in a real switch.

We use this model of switches extensively in this thesis. This is because they lend themselves to easy implementation of the Calculator and Simulator. However, it would not be difficult to change the way that scenarios are modelled to be based on another method of switching.

10.4.2 Input Queueing

Input queueing is the opposite to output queueing. Here packets arrive at an input interface and are queued, then, once they have been serviced, they are transmitted over the switching fabric to the outputs. This is again assuming that we have a perfect switching fabric.

Figure 10.4 show how we construct the scenario graphically. Traffic arriving at this sort of switch does so at the first object (Object A), here it is queued per input port and subjected to the QDF defined by that port. All the ports in the first object send their traffic to the second object (Object B), this simulates our perfect switching fabric. The second object (Object B) is used to make a decision as to which output port to send a flow to; the QDF for all the ports is set to the identity (id) QDF which has no effect on the traffic.

Alternately we can change the QDF function of the second object (Object B) to a standard queue based QDF. In this case we can model a switch that has both input and output queueing with a perfect switching fabric.

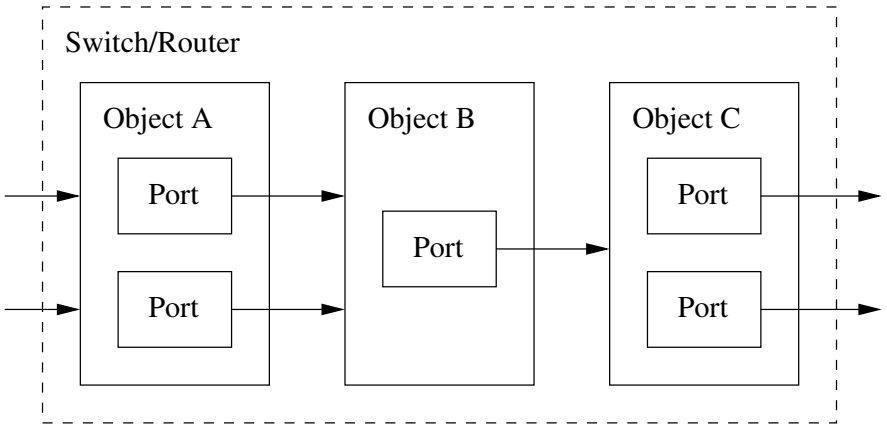


Figure 10.5: Input and Output Queueing

10.4.3 Combined Input and Output Queueing

Our final model of a switch has input and output queueing as well as an imperfect switching fabric. By an imperfect switching fabric we mean that it is possible to lose and delay packets as they traverse across the switching fabric.

Figure 10.5 shows graphically how we would construct such a switch in a scenario. Flows arrive at Object A and are passed through a port, and associated QDF, representing the input queue. Next all the flows from Object A are sent to Object B; this models the switching fabric and has a single QDF that encapsulates the degradation that would be suffered. Finally all the traffic from Object B is sent to Object C; this object models output queueing, and has a port per output port that we wish to model. The following snippet of scenario shows this:

```
<object name="inputs">
  <port name="port1" qdf="mm1k" rate="1.0" buffers="100" sendsTo="fabric">
    <select id="1" class="be"/>
  </port>
  <port name="port2" qdf="mm1k" rate="1.0" buffers="100" sendsTo="fabric">
    <select id="2" class="be"/>
  </port>
</object>
<object name="fabric">
  <port name="port" qdf="crossbar" rate="2.0" sendsTo="outputs">
    <select id="1" class="be"/>
    <select id="2" class="be"/>
  </port>
```

```
</object>
<object name="outputs">
  <port name="port1" qdf="mm1k" rate="1.0" buffers="100" sendsTo="next">
    <select id="2" class="be"/>
  </port>
  <port name="port2" qdf="mm1k" rate="1.0" buffers="100" sendsTo="next">
    <select id="1" class="be"/>
  </port>
</object>
```

We do not have a QDF to represent a crossbar as it stands. In the next section we will investigate how this could be accomplished.

10.5 Performance and Modelling Issues

In this section we will take a look at some of the performance issues inherent in the different types of switch design. This section is intended to be a brief overview of the current literature in this area. We also look the how the choice of switch design effects the accuracy of our models. Finally we will look at QoS problems associated with switching technologies.

10.5.1 Output Queueing

Output queueing is the ideal switch architecture when it comes to performance. There are generally two ways to construct an output queued switch. The first method (crosspoint queueing) connects each input to a buffer on each output, where each output has a separate buffer for each input. The output port then schedules packets for departure from each of the queues it manages (one per input). The second method is to use a shared memory between all of the inputs and outputs. Both of these methods have implementation drawbacks, especially when dealing with high valency switches. When using crosspoint queueing the number of internal links grows by the square of the number of ports, and this can quickly become impractical. There is a variant of crosspoint queueing, called block crosspoint queueing, whose link requirements grow at a slower rate. When using a shared buffer the bandwidth requirements of the buffer memory can quickly become impossible to satisfy. However, these issues are less of a problem for low valency switches.

Output queueing does have a number of benefits. These are:

- Very high throughput.

- No head-of-line (HOL) blocking [54].
- Low delay in crossing the switch.
- Does not suffer from fairness problems.

Modelling Output queued switches is simple, we do not have to model contention or correlation issues within the switching fabric. Each of the output ports can be modelled as a number of queues serviced by a single service facility. In this thesis the model of QDFs that we have used is derived from queueing theory, as such we could also derive a model for this type of queue with relative ease. One of the key benefits of output queueing, from a QoS perspective, is that there is only a single point of contention within the switch - the output port. This allows us to place a QoS scheduler in one place, and it is easier to model. However, output queueing is only practical for low valency[3] switches and as such is less common. The drawback of output queueing is that the throughput has to grow linearly with the number of ports; adding more ports requires a growth in fabric capacity or memory speed. This is why output queueing is generally only used on low valency switches - as larger valencies become impractical.

10.5.2 Input Queueing

Input queueing generally uses a buffer-less switch fabric, although it is possible to construct a multi-stage switch fabric with limited buffering. Packets arrive at the switch and are queued in an input buffer. Each input must attempt to send the packet at the head of its queue to the destination port. As the switching fabric is buffer-less the input port must first request the use of the fabric. Once this has been granted the packet can be transfered across the switching fabric.

As the switching fabric is based on the concept of request-grant it arbitrates, or schedules, access to the fabric. The scheduler inside the switching fabric must attempt to make a stable marriage between its inputs and outputs, in other words it must attempt to match inputs to outputs without contention. This presents a problem in terms of fairness and of efficiency. Any matching algorithm must attempt to give each of the inputs a fair share, while not taking to long to make a decision (which become more difficult as the valency increases).

A well known problem that naive input queueing can cause is called head of line (HOL) blocking, this occurs when each input has a single FIFO queue. The packet at the front of the queue may not be granted access to the switching fabric, because of contention for the output port. However, the packet behind, if it were at the front, would have been granted access to its output port. The effect is that the packet at the front of the queue blocks other eligible packets from being serviced. To overcome this problem it is common to find switches based on input queueing to use virtual output

[3] We take low valency to mean switches with anything up to 32 ports.

queueing; here each input has a separate queue for each of the output ports, and hence overcomes the problem of HOL blocking.

Despite the use of virtual input queueing we are still left with the problem of matching inputs to outputs; to this end, a number of algorithms have been proposed [60, 68, 6, 91]. From a QoS perspective there is still only a single source of contention - gaining access to the switching fabric. The problem with input queueing switches, from a modelling perspective, is that they have back-pressure and are loss-less in the operation. It is quite likely that there are a number of inputs attempting to reach the same output, in this situation the scheduler has to choose which input port will be granted access. To model this sort of switch in our methodology is non-trivial, as we have chosen a generally Markovian approach. Clearly switches with this kind of back pressure are not Markovian, as there is a work-conserving deterministic choice being made about the input port that is chosen. In some situations, with a synthetic load, it may be possible to bound these decisions; however, this is unlikely to result in an accurate answer.

10.5.3 Combined Input and Output Queueing

Combined input and output queueing switches, as the name suggests, use queues on both the input and the output. Packets arrive at the switch and are queued, next they are transported across a switching fabric and enqueued on the output queue, where they are finally serviced. This type of switch typically has an internal speed-up (ie. the switching fabric is faster than the incoming links); the motivation for this is to ensure that packets are serviced from the input queues as quickly as possible such that a backlog does not build up. This essentially pushes the contention towards the output queues. It is possible to emulate an output queued switch where the internal speed-up is sufficiently high, see [22, 58] for more details.

Combined input output queueing is more complex to model. There are a number of sources of contention within this type of switch. Firstly they suffer from similar issues to input queued switches, as there is contention to get across the switching fabric. However, this can be overcome by internal speed-up in some situations. Secondly there is the contention for resources in the output queue, which is again similar to output queueing. From a QoS perspective all of these sources of contention need to be managed in order to provide quality guarantees.

10.6 Summary

In this chapter we looked at how to model networks using scenario XML descriptions. This included modelling sources, sinks and switches (routers). The main area of interest has been switch modelling. There are a number of different architectures we have looked at, each constructed in a scenario in

a different way. In order to simulate or calculate the network we need a model of each of the QDFs that we are using in the scenario. This presents more of a problem, depending on the switch design. For pure output queued switches we are able to use standard queueing theory models, as there is no contention for the switching fabric. Input queued switches cause more of a problem because of internal back-pressure, this is an area for more research. Finally combined input and output switches have similar modelling problems to input queueing switches. We have emphasised that in order to predict and control quality we must manage all sources of contention, both in the network and its internals. This will require us to have detailed models of the queues, and service disciplines used throughout.

Chapter 11

Conclusion

11.1 Aims and Motivation

In this thesis it was our aim to take a fresh look at the area of QoS. There are many solutions to specific problems in this area in the current literature; however, at the present time there is no coordinated solution that encompasses all of the problems and solutions into a single framework. What is required is a methodology for QoS that can yield results in the areas of network planning, provisioning, quality delivery and finally expectation management.

Ultimately it is our aim to allow the construction and management of predictable multi-service networks; able to deliver differentiated quality to a number of subscribers with confidence. The problem with current networking technologies is the lack of predictability under many conditions - not least of which is overload. A solution in this area must have a well defined emergent behaviour at all operating points. Today we see that major Internet back-bone providers dramatically underutilised their networks, thus ensuring minimum contention, and as a result predictable quality. While this is certainly an effective solution in terms of its outcome, few would argue that it is cost effective or sustainable.

Currently there is a gulf between what a network can deliver, the operations, and how we reason and express our desires, the aspirations. We may, for example, have a network that is capable of delivering a predefined amount of bandwidth between two points; such a network can be instrumented and tested to make sure that it does indeed achieve this. The problem occurs when we realise that we specify our requirements in terms of acceptable loss and delay at a given bandwidth. While we know the network can deliver the bandwidth we do not know, or cannot assert with confidence, what the observed loss and delay will be. Until this gulf is closed it looks unlikely that multi-service networks will ever live up to their promise.

Recent research has highlighted the fact that any finite queue has two degrees of freedom; this relationship is a fundamental property of any finite queue, irrespective of the queueing discipline. When you fix any one of throughput, loss or delay you create a relationship between the other two. So, for example, fixing the throughput (bandwidth) creates a relationship between loss and delay. The problem with doing this, as the majority of queues do, is that the relationship between loss and delay is not managed. This, in turn, makes it impossible to predict the effect of a queue on the traffic it services in terms of loss and delay. This is clearly a major problem; especially when we specify our aspirations in terms of loss and delay at a given bandwidth. It provides us with our first insight into solving the QoS problem, namely, managing bandwidth alone is not a complete solution.

The same research has also proposed a new type of queue called a loss-delay multiplexer. This queue differs from others in the literature by simultaneously managing loss and delay, leaving the throughput to vary. Traffic is classified in terms of cherish, the desire to experience less loss, and urgency, the desire to experience less delay; giving us a more natural way to reason about performance. In addition this queue has a sound mathematical basis, allowing us to investigate it analytically. Taking this approach gives us the potential, but not all the answers, to close the gap between aspirations and operations.

Our aim is to demonstrate that it is possible to build a network whose behaviour we can predict, even when the network is in saturation or overbooked. The predictions should also be comparable to our aspirations, thus allowing us to assert whether or not a given configuration will satisfy our needs. This will require a sound mathematical basis from which to approach the problem. Queueing theory provides us with such a platform, using statistics to express our expectations of performance. This work should be carried out in a broader framework which allows us to reason about QoS and to arrive at useful solutions.

11.2 Summary of Work

In chapter one we introduce the area of multi-service networks. One of the key requirements for building such networks is support for differentiated servicing or QoS. Current large scale networks, such as the Internet, do not on the whole support such services. The reason for this is the gap between what users desire, their aspirations, and what the network can deliver, the operations. In this chapter we also introduce the concept of treating a network like a utility and relate this to people's expectations of a network utility.

In chapter two we examine the previous literature in the area of QoS. We start by looking at measurements of real Internet traffic, and highlight how their self-similar, or fractal nature, makes the provision of QoS difficult. Next we look into the behaviour of applications, in an attempt to explain the emergence of self-similar traffic. We find that on-off sources with Pareto on-off periods

can lead to this behaviour through aggregation. We then look at bandwidth management techniques, while these do indeed work correctly it is not clear how to make predictions in terms of loss or delay. These bandwidth management techniques are then contrasted to some quality-centric approaches that deal with loss and delay, as opposed to bandwidth alone. Finally we look at the current "best practise" for managing QoS on todays networks.

In chapter three we take a high-level view of the area of QoS. We start by looking at the aspirations and responsibilities of the stake-holders in a multi-service network, and highlight what requirements these place on our choice of implementation strategy. The concept that quality is the absence of degradation is introduced. The degradation can be captured intuitively by the concept of ΔQ, which represents the change in quality as traffic crosses a network. Finally we outline the requirements for a methodology for tackling the end-to-end QoS problem.

In chapter four we introduce the mental model and associated insights that we use throughout the thesis. We start with the concept of intrinsic quality, which states that there is a finite amount of quality available in any network element, and hence the whole network; the choice that we have is how to share this quality out. Next we show that quality is lost through the actions of quality degraders, which represent where loss and delay are introduced into the network. This concept is then extended to allow us a definition of a Quality Degradation Function, which is a mathematical representation of the quality degradations. Finally we introduce the concept of instantaneous quality, that is the quality that you are likely to receive irrespective of the period of observation.

In chapter five we look at how we can start to attack the end-to-end QoS problem. Our starting point is to use simple Markovian queueing theory to allow us to compose queues. While the output from a finite queue is never truly Poisson we conjecture that we will be able to break such assumptions, allowing us to compose queues. Next we look at queueing theory in more detail to find ways of calculating network performance. This include strategies for handling networks with mixed packet sizes, and models of burst loss.

In chapter six we explain the design of the test system that we will use in the rest of the thesis; this consists of a calculator and a simulator. The calculator is based on the mathematics and methods presented in chapters four and five. It allows us to predict the behaviour of a network of queues under a given set of stationary sources. The simulator allows us to simulate the network we are investigating. By comparing the results between the simulator and calculator we can validate our assumptions and methods for solving end-to-end QoS.

In chapter seven we make some comparisons between our predictions and simulations; using a few simple test cases, each designed to affect the accuracy of the results. The test cases do not provide any differentiated QoS, they concentrate on the base line predictability. Packets are modelled initially as point processes, to be as close to the mathematical models as possible. We then continue this to look and fixed and mixed packet size networks.

In chapter eight we take a closer look at requirements for individual applications. For each application we look at its sensitively to loss and delay, and the bandwidth at which it operates. Using the concept of cherish and urgency we classify the applications relative to each other. We also outline what the conditions for success are for each of the sample applications.

In chapter nine we present results for a comparison between bandwidth-centric and loss-delay style queue disciplines. Using the sample applications in the previous chapter we construct a scenario of usage for a given network. Using the same topology we change the queueing discipline and perform some simulations. The results show that concentrating on bandwidth alone is not enough to guarantee the performance of an application, even when that application has indeed received the correct amount of bandwidth. Bandwidth does not adequately capture the required performance metrics such as loss and delay; however, the methodology and model that we have developed does uniquely address this issue.

Finally in chapter ten we look at how closely we could model real networks. The main focus is capturing the design of the networking hardware, so that we can model it within our calculator. We find that in some situations we can indeed produce acceptable models, but in others more work is needed. While we may have a theoretical solution to the end-to-end QoS solution there is some way to go before this becomes a reality in the hardware.

11.3 Major Contributions

In chapter four we introduced our QoS methodology; which, as will be discussed shortly, includes the following concepts: two degrees of freedom, quality degradation, intrinsic quality, ΔQ and instantaneous quality. We believe that this model provides a simple way of reasoning about quality. Such a model is important because to gain wide adoption QoS solutions must be easy to understand by the people who have to use them. Our model has allowed us to explain our work to a number of people not related to Computer Science, and all with great success. While we agree that bandwidth is also simple to reason about, we do not believe that it is sufficient to explain the problem area completely.

The underpinning of this methodology is the relationship between loss, delay and throughput. While it is well known that this is an inherent property of finite queues, the implications are less well understood. By specifying requirements in terms of cherish and urgency we can provide a more appropriate language in which to reason about quality. This is especially powerful when used with the loss-delay queueing discipline, which naturally handles specifications of this form.

We have introduced degradation as a key concept for reasoning about QoS. Quality is the absence of degradation to a particular flow, and not something special that happens to improve the quality. It is clear that in order to degrade one flow less, other flows have to be degraded more. This brings

us to another important concept - intrinsic quality. Under a given loading every network element has a finite amount of quality, or expressed another way, a given amount of degradation which has to be shared out. Degradation is therefore conserved once created, and can only be differentially apportioned. Providing QoS support is deciding how the degradation inherent in every network element should be distributed to meet the requirements that have been set.

Another important concept that we have developed is that of ΔQ. As flows of packets traverse a network they become degraded; their throughput drops, and their loss and delay increase. The concept of ΔQ captures this change in quality between two points in the network. For each network element, from the point of view of a flow, there is a change in quality as you traverse it. As a flow traverses an entire network it is subject to a number of changes in its quality (which is an increase in degradation), these changes are cumulative and allow us to capture the end-to-end degradation.

To reason about quality end-to-end we must be able to reason about the change in quality over a single network element. We define the concept of a Quality Degradation Function (QDF) to capture the change in quality over a particular network element. By modelling each of the network elements as a QDF we can essentially add up the degradation that a flow would suffer as it crosses the network, through our theory of ΔQ. Currently evaluating a QDF results in a loss probability, average delay and decrease in throughput; although as we will see later there are other ways of attacking this problem.

The final concept that we introduce is that of instantaneous quality; which is the quality that you receive irrespective of the length of your measurement period. Assuring quality over a long period does not necessarily imply that applications will work over that period (see section 4.6). It is this concept that allows us pull together our methodology. The model of QDFs that we use allows us to calculate a degradation, and this degradation can be considered instantaneous as it is true over all intervals of time. By using the concept of ΔQ we can convolve the degradation of individual QDFs to arrive at the overall end-to-end degradation.

In chapter five we developed a simple example of how it is possible to implement our methodology. This model essentially uses queueing theory to build a mathematical model of each of the QDFs that we wish to use; these are later tested in chapter seven. While we have chosen simple Markovian servicing disciplines to demonstrate our point, it is likely that the same process can be applied to other more complex servicing disciplines if required. One of the benefits of using a Markovian-based service discipline is that we can reason about the performance of the queue under saturation; this is important as greedy protocols, such as TCP, will continually push queues into overload. During periods of overload the most quality degradation occurs, it is clearly essential to be able to model such conditions accurately.

In chapter seven we have shown that it is possible to predict the behaviour of certain types of networks, specifically, those based on Poisson distributions and Markovian servicing. Queueing theory allowed us to explore this area, and by challenging some of the assumptions we were able

to extend the work further. For example, where the output from a finite Markovian serviced queue is not strictly Poisson, we can indeed assume that it is. The predictions we performed were only on networks where there was one class of service; however, the same techniques can be applied to multi-service networks. A key underpinning of this technique is to use "randomness" to make the solution statistically tractable. While this may seem "wasteful" by the standards of today's typical engineering practise, we believe it is justified, as the consequential ability to accurately predict network behaviour more than outweighs the apparent waste.

In chapter eight we tie network quality to application behaviour. For each of the applications that we study we are aware of the failure conditions that exist. By engineering quality constraints for each of the applications we can ensure that failure conditions do not occur, or only when another application is more important. Using the cherish-urgency model proved to be excellent at capturing these requirements. In applying our methodology we had to extend some techniques in queueing theory. The main contribution in this area is the modelling of burst-losses using a Markov chain. This allows us to predict the probability of application failure due to burst losses. We also show how it is possible to use some well known queues to model more complex networks, by breaking the assumptions we outlined above.

In chapter nine we present a comparison between bandwidth management and a loss-delay model of network management. We found that although the bandwidth management did work correctly, in that each application did indeed receive its contracted bandwidth, it was not sufficient to deliver the quality required. By using a loss-delay style of multiplexing, with the same transmission speeds and buffer sizes, we show that it is possible to meet the requirements of the applications with confidence.

Today much of the networking industry is primarily concerned with bandwidth management; engaging with these results implies the need for this to change. Such a change is vitally important, if we are to see a shift to multi-service networks.

11.4 Future Work

During the course of this thesis we have highlighted a number of areas that warrant more investigation. In this section we will take another look at these areas, and where possible we will propose some possible courses of investigation that may lead to a solution.

11.4.1 Modelling Mixed Packet Sizes

In section 5.4 we looked at how to model queues that service packets of variable sizes. Such a model is necessary because the majority of networking technologies, such as IP, handle packets with

variable sizes. Modelling technologies, such as ATM, which carry fixed sized packets can be achieved using standard queueing formulae, which model packets as point processes.

The model that we proposed, in section 5.4.4, uses an M/G/1/k priority queue with classes but no priorities. The key feature of this type of queue is that it supports a number of classes, where each class can have a different service rate. Different service rates are required because packets of different lengths take a different amount of time to service; by placing packets of a given size in their own class we can model a queue that handles packets of differing sizes. Ultimately we can calculate the loss and delay that each class will suffer.

One drawback of this approach is that it requires one class per packet size. Given that most technologies support a large range of packet sizes this could result in a rather large number of classes. However, research [95] has shown that packet lengths tend to cluster around well known MTU sizes; this would make it possible to have classes representing common packet sizes, and not classes for all sizes of packet. This is likely to introduce some calculation errors, which should be investigated before this approach can be used.

The M/G/1/k queueing formula can accurately calculate the performance of a singe queue handling mixed sized packets. However, when we attempted to extend this to a number of queues, see section 7.7, we found that the calculations became inaccurate. Even for the simplest of examples of composed queues we found discrepancies of hundreds of packet service times between calculations and simulations. This clearly makes such a model unusable in the real world.

We know by looking at the results that increasing the buffer size makes the predictions accurate at high loading factors. This is because adding more buffers reduces the loss probability for lower loading factors. This suggests that the inaccuracies in the calculations are caused by loss, where no loss occurs the predictions are accurate. The inaccuracies in the loss calculations could be caused by the fact that the packet arrival distribution is not strictly Poisson. Another cause is that the steady-state loss probabilities are not sufficient to model systems with mixed sized packets.

There are a number of avenues for further investigation open to us. We could consider modelling packet arrivals using different distributions, such as hyper-exponential or an Erlang-k distribution. Such an investigation could be started by looking at the distribution of inter-packet times observed in the simulator. If we find that steady-state loss probabilities are insufficient we could investigate using a different method of convolution, such as Laplace (see below).

11.4.2 Burst-loss Probabilities

In section 5.6 we looked at a burst-loss model for an M/M/1/k queue. We found that there is a probability of an initial loss, and another probability of additional losses. This gives us a power law

distribution of losses, which is very different from the evenly distributed losses that λP_k suggests. This has important implications for applications like VoIP, where burst-losses are important. It may also affect the way in which we solve queueing systems at the moment.

The method that we presented for calculating the burst loss probabilities relies on compressing all the queue states into a single state, and then extending the state space to produce a chain of losses; each state greater than zero represents a burst loss of that length, and a departure resets the system to the zero state. The benefit of such an approach is that it allows us to ignore the semantics of a particular queue, so long as we know what the probability is of the system being full, P_k. Some further investigation is required to ensure that compressing states in this way is mathematically sound. It would also be nice to extend this work to queues other than the M/M/1/k queue.

Another use of the burst-loss work is in producing better models of queues. By attaching the loss states to the end of a standard Markov chain we can model the loss process, and hence the average waiting time, much better than before. This may be an extremely useful tool for investigating how to model mixed sized packets, given that loss appears to be one of the dominating factors. The model presented for capturing burst-loss only considers a single arrival stream. It would also be nice to extend this work to cover multiple arrival streams.

11.4.3 Laplace Convolution

The method of convolution that we present in this thesis involves addition of mean values for delay and loss. On the whole this approach appears to work well where the network carries packets of a fixed size. However, when the network carries mixed packet sizes the results are less accurate. We have suggested that steady state predictions may be insufficient for modelling mixed packet sized networks, and that evenly distributed loss probabilities are unrealistic. What is required is another approach that can overcome these problems.

Laplace transforms could provide another, more accurate, method of convolution. By transforming the formulae for QDFs (Quality Degradation Functions) into Laplace space we can convolve them using multiplication. The benefit of such an approach is that we can extract distributions of interesting parameters, such as delay, by performing an inverse Laplace transform at the end of the convolution. Having an accurate approximation of the delay, loss and inter-packet distributions would provide us with much more useful information than averages alone. We would be able to compute confidence intervals and percentiles of the expected averages giving the approach more strength. It would also be possible to investigate the transient behaviour of networks caused by periodic changes in load or topology, and ultimately the change in quality that this would cause.

To apply such an approach would require us to represent the system in semi-Markov processes (SMPs). Given that we have chosen to represent our queueing systems as Markov-chains this would

not be too difficult to achieve. Recent research [14] on computing passage time using a distributed cluster of computers would provide a good starting point.

11.4.4 Distributions

Throughout this thesis we have restricted ourselves to reasoning about Poison distributed inter-packet times. We chose this approach due to its easy analytical properties, and well known queueing formulae. However, the Poisson distribution introduces a high variance into inter-packet times which may be undesirable in a number of situations. On the other hand it is this high variance, or 'randomness', that provides us with desirable properties such as independent arrivals and fair merging properties.

There are a whole series of well known distributions that have similar properties to Poisson, Erlang-k distributions for example, while not having such a large variance. It is worth investigating these distributions with respect to their effect on performance and predictability. It may be the case, and indeed quite likely, that other distributions with less variance than Poisson would still allow us to use the techniques presented in this thesis while increasing the overall packet-by-packet performance of the network.

11.4.5 Buffer Size Effects

In section 4.7.2 we highlighted the fact that choosing an appropriate buffer size, for queues within a network, has a dramatic effect on the performance of the network. Buffers can be considered to introduce 'memory' into the network, making it less tolerant of changes in offered load. Buffers also effect both the loss and delay that packets experience while crossing the network. Adding more buffers increases the delay but reduces the loss, similarly reducing the buffer size decreases the delay but increases the loss. Therefore, the buffer size is yet another parameter that we can use to tune the behaviour of the network.

The buffer size also affects the performance of queues in transient conditions, such as start-up and sudden changes in load. Theoretically it is possible to calculate the time it takes, after a change in offered load, for a queue to reach steady state. Understanding such effects is important, as sudden changes in load can adversely effect the delivered quality. Today it is quite common to find network devices that contain buffers measured in Megabytes; such large buffering could potentially take days to reach steady state, assuming the load remains constant during this period – which is unlikely. Investigating the effects of such large buffers could also yield interesting information on achieving consistent quality.

In this thesis we have concerned ourselves with investigating QoS using loss-delay models, and not bandwidth based approaches. Given that buffer capacity affects both loss and delay it is worth

investigating the effects that buffer size has on quality. In addition it is also worth investigating what effects buffer size has on quality under transient conditions. One possible avenue for research in this area is provided by Laplace transforms, which can reveal detailed information about the long term behaviour of queues and the use of instantaneous generator functions can provide information about their transient behaviour.

11.4.6 Adaptive Source Models

The adaptive rate source that we have presented in this thesis is intended to model simple greedy sources in steady state. A common example of a greedy source would be TCP, which attempts to utilise as much of the available bandwidth as possible. Such sources are responsible for pushing queues into overload, as they continually increase their bandwidth until they experience loss. They are also believed to be one of the causes of the observed self-similar, or fractal, behaviour of networks.

The first area for more investigation is that of the model itself. We have not made any attempts in this thesis to accurately model TCP. As a result we do not have any detailed understanding of the behaviour of TCP under our proposed network architecture. It is our hope that we can destroy the self-similar behaviour of networks by introducing exponential, or other, types of servicing. By building a better model of TCP we could investigate this area more throughly.

The second area of interest is modelling transaction-based TCP applications. This style of application usually transfers small amounts of data for which a quick response is required. By their very nature they do not use the parts of TCP that are designed to maximise throughput. For this type of application we would like to know the distribution of transaction times, so we can give realistic bounds on the performance of the application itself. To achieve this would require building an accurate model of TCPs state transitions.

The third area of interest is understanding the resulting behaviour of TCP. Like other reliable transport protocols, TCP essentially hides loss by introducing delay. The delay that is introduced is dependent on where losses occur within the packet exchange and the end-to-end delay. By possessing an accurate model of TCP we could investigate the effect of our chosen architecture on the behaviour of TCP. Ultimately it would be nice to provide predictions on the behaviour of TCP under the chosen configuration of the network.

Finally, it would be nice to incorporate some better traffic models for adaptive sources. Applications such as HTTP generate variable amounts work. A model based on Poisson thinking time, Pareto distributed file and request sizes may provide a better model for such applications. These models could be based on empirical measurement and previous literature [74, 5, 105, 18].

11.4.7 Changing Topology and Load

One of the assumptions we have used throughout this thesis, see section 9.2.3, is that both the topology and offered traffic remains constant throughout our experiments. While this reduces the problem space, allowing us to make a number of discoveries, it is not consistent with a real world view of networking. The load placed on a network changes depending upon the time of day, and is influenced by real world events like the 11th of September. Routing nodes within a network can, and do, fail causing changes in the path that packets take through the network. All of these factors affect the quality that is observed by the users of the network, and as such should be investigated.

To overbook resources successfully requires us to have a detailed understanding of the effects of the changing traffic patterns. To a large degree much of this information can be gathered by monitoring the edges of the network to collect statistical data. The question remains what to do with this data in order to make the network run smoothly. By adapting the configuration of the network at different time intervals we can change the behaviour of the network to reach our goals. We could even envisage penalising certain activities, such as large file transfers, at given times of the day to increase the overall performance of the network. To achieve this requires us to understand, in a sufficiently accurate way, how a given network configuration will affect the quality of the traffic that flows across it. Understanding the transient behaviour of changes in configuration is key in achieving this goal.

Adapting to failures, of for example routing elements, within the network is a harder task. It is unlikely that we will know the point of failure before it happens. When a failure does occur traffic should be, although it is not always, rerouted via another path. This could potentially cause quality problems for the traffic currently flowing across the backup path, as well as problems for the traffic that is being rerouted. There are two areas of interest here. The first is to look at quality routing protocols that will redirect traffic in such a way as to minimise the overall effect on quality, this requires us to have an understanding of the current and expected level of utilisation across the whole network. The second is to look at mechanisms for controlling delivered quality; it would be possible, for example, to modify the configuration of a multiplexing point to deliver the best quality it can, depending on policy.

11.4.8 Correlation and Multicast Modelling

In section 6.2.2 we noted that, although it is possible to support multicast, with a few minor modifications to the system, we have chosen not to investigate it in this thesis. This choice was again taken to reduce the problem space. Multicast flows introduce a much higher occurrence of correlation in the network, this is because the same flow can be found in more than one part of the network. We believe that such correlation issues can be destroyed by stochastic service, which

ultimately destroys time dependent correlation at a low level. However, we have not investigated correlation issues or multi-cast in this thesis.

Correlation occurs when the same temporal pattern is experienced by a number of flows which later interact. The temporal pattern is caused by the multicast flow interacting with other flows in the network. These flows interact with each other after being correlated by the multicast flow. In section 7.4.3 we present a simple test case that highlights the problem. However, we found that no serious correlation was introduced in this experiment, perhaps providing justification for stochastic service destroying correlation. Unfortunately this simple example does not provide enough evidence in itself and more results are needed. Multicast would provide an excellent vehicle for investigating this problem.

Multicast flows are also an interesting area of quality delivery. Given that a multicast flow is consumed by a number of end-points it requires a higher quality than the same unicast flow would require. It would be interesting to investigate what level of quality should be delivered to a multicast flow to match the quality delivered to a similar unicast flow. Indeed, it may be beneficial to allocate higher quality to multicast flows to encourage their usage over unicast flows, so long as any correlation issues do not affect the overall performance of the network.

11.4.9 Shared Media and Switching Fabrics

In this thesis we have considered queues that have unrestricted use of the outgoing link, this is useful for technologies such as point-to-point Ethernet; however, it does not currently model links such as shared Ethernet segments, wireless Ethernet (Wi-Fi) and cable modems. Where there is a shared link there is the possibility that an attempt to service a packet will fail, because the link is currently being used by another station. Commonly this causes the transmission of the packet from both sending stations to cease, before being resumed after a random time out.

It is possible by using queues with vacations to model contended links of this type. This approach has been used recently to model wireless Ethernet and similar technologies. It would be nice to include such models into our methodology, to broaden the scope of its applicability. Assuming that exponential service is still used, which may be plausible due to back-off timers, similar benefits in predictability may be gained.

We have also investigated models of switching fabrics in this thesis; however, these models are extremely limited. Switching fabrics have contention issues that are not dissimilar to those found on shared network segments. By again modelling queues with vacations it may be possible to make more accurate predictions of the performance of flows crossing a switching fabric.

11.4.10 Provisioning using Loss-Delay

The final area of research that deserves consideration is that of provisioning. In this thesis we have concentrated on using the loss-delay model to provide QoS support. The loss-delay model is unique in that it explicitly manages the two degrees of freedom found in any finite queue. By specifying an ordering in terms of loss and delay we can accurately control the throughput. Another way of viewing the loss-delay queue is its ability to provide a quality, measured in loss and delay, at a given bandwidth. This allows the possibility of doing accurate provisioning.

For a given offered load, that is a bandwidth and associated cherish-urgency classification, we can predict the performance of the network. As it stands we have been doing just that with a fixed set of offered loads and network configurations. While a fixed network configuration, which in terms of the loss-delay queue is the link service rate, buffers and watermarks, is quite likely - a fixed load is not. However, by policing the edges of the network it is possible to restrict the offered load to some well known maxima.

When the offered load is fixed to some maximum value it is possible to calculate the worse case performance of the network. That is the condition when all classes of service are at their maximum value. From this point it is possible to evaluate how successful a given application, such as VoIP, would be. Where there are performance problems the configuration can be changed so that they do not occur. Ultimately this process should be automated in some way.

11.4.11 Loss-delay implementation for NS2

In this thesis we have performed simulations using a custom built simulator written in C. This approach was taken as it allowed us a much greater understanding of how the simulator was written, and what design decisions were taken. Indeed it has served its purpose well, allowing us to investigate the methodology.

In the future we would like to implement some of our ideas in the NS2 [98] simulator. This would allow us to use a large volume of research that has already been implemented in NS2. Comparisons between our methodology and others could be performed, but with the benefit of using a more standard platform.

11.5 Final Words

In this thesis we have attempted to find solutions to the problem of providing end-to-end QoS support. We started by abstracting the network into a set of connected queues with flows of traffic

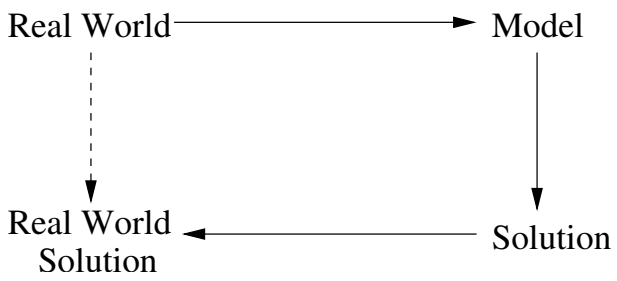


Figure 11.1: The philosophy

crossing them. Next we restricted the problem space to queues with Markovian service facilities carrying traffic with Poisson inter-packet times; by doing so we showed that it is possible to predict the performance experienced by the flows crossing the network. We then extended our model of queues to include the loss-delay model, which explicitly manages the two degrees of freedom (two of: loss, delay, throughput) inherent in finite queues. We believe that the work presented here offers a new way of approaching the problem of end-to-end QoS, although there is still more work to be done.

In this section we would like to explain one of the fundamental philosophies behind the approach that we have taken; namely that, we can arrive at useful real world solutions to the problems that we trying to solve. Figure 11.1 shows the mental steps we progress through to arrive at a solution.

Ultimately we want to create a real world solution to a real world problem, but we cannot do this directly. First we must abstract away from the real world to arrive at a model, in doing so we discard some information and potentially introduce some errors. We will only know if this is a good abstraction when we return to solve the real world problem later. In this thesis, as we have already mentioned, we have chosen to restrict ourselves to Markovian models which are mathematically tractable. Having done this we can set about searching for a solution, in the model that we have chosen. The final and most important challenge is turning our solution into a real world solution than can be implemented and will work as intended.

In this thesis we have not had the opportunity of implementing our solution in the real world; however, we have been able to investigate our finding through simulation. By choosing our abstraction carefully we have been able to provide realistic predictions about the performance of the network under all conditions. We have also shown that other abstractions, such as bandwidth-centric approaches, do not provide similar benefits in prediction and performance; although such approaches are used in the real world. Clearly the approach that we have presented here could work in the real world, but cannot be investigated in detail until hardware exists to test the theories[1].

As computer scientists we have a luxury that other sciences do not; namely, we can decide how

[1] U4EA Technologies are starting to produce hardware that does indeed implement the loss-delay model.

the systems that we build will operate. Other fields, such as Physics, have to build models that represent the real word. Currently much effort in the QoS field is expended trying to accurately model networking equipment. The author believes that we should concentrate our efforts into creating the correct model first, and then set about creating the hardware and software to support it.

Appendix A

Queueing Theory in Brief

A.1 Distributions

Throughout this thesis we will primarily be concerned with the exponential distribution. For us this has a number of interesting properties that make it suitable for designing a network.

The first property of the exponential distribution is that it closely approximates the Poisson distribution (which is a discrete-time distribution). This approximation makes it easier to model mathematically. In general we will use the term Poisson in this thesis, although we generally model this as exponential.

Exponential distributions are also memoryless (otherwise known as the Markov property). This means that no matter how long the system has been running the probability of a new event remains constant. In effect it does not age. This is useful as we do not need to account for time in our models.

The exponential distribution is also easy to generate. The function to do this is shown below:

$$F(x) = \begin{cases} 1 - e^{-\lambda x}, & \text{if } 0 \leq x < \infty \\ 0, & otherwise \end{cases}$$

A.2 About Queueing Theory

Queueing theory is a broad area of mathematics that attempts to model the behaviour of queues (as the name would suggest). It does not just apply to computer science but many other fields from

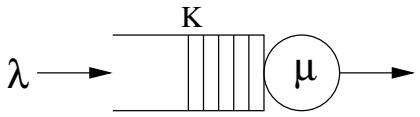


Figure A.1: A Queue

airport parking to production control. The following has a computer science slant, but the formulae are generally applicable.

Queues are usually drawn as in Figure A.1. Jobs arrive at a rate of λ with a given arrival pattern. They are then buffered; if there are a finite number of buffers then this is usually referred to as K. Finally the jobs are serviced by one or more service facilities (represented by the circle) with rate μ, again with a given distribution. In this context, when we talk about jobs we are referring to packets.

There are a number of conflicting names for describing where a packet is in the system. Throughout we shall use the term buffered to describe when the packet is awaiting service; note this is sometimes called "in-line" or "in the queue". When the packet is either buffered or in the service facility we shall call this "in the queue"; note, this is sometimes called "in-system".

In order to identify different types of queue we use Kendall's notation. The form of this notation is A/B/C/D/E where:

- A Distribution of inter-arrival times of customers
- B Distribution of service times
- C Number of servers
- D Maximum total number of customers which can be accommodated in system
- E Calling population size

Both D and E are sometimes omitted, in which case they are assumed to be infinite. A and B can represent a number of distributions, however, in this thesis we shall generally use the following:

- M Exponential Distribution (Markovian)
- D Degenerate (or Deterministic) Distribution
- Ek Erlang Distribution (k = shape parameter)
- G General Distribution (arbitrary distribution)

The following symbols are used to represent properties of a queue:

- ρ The traffic intensity
- λ_a The amount of traffic accepted into a finite queue
- P_n The probability of there being n jobs in the queue
- P_k The probability of the queue being full
- L_qThe number of jobs in the buffers
- L The number of jobs in the queue
- W_q The average waiting time in the buffers
- W The average waiting time of the queue

The ratio of arrivals to departures is called the loading factor, measured in Erlangs. An Erlang is a dimensionless unit of the average traffic intensity (occupancy) of a facility during a period of time. This is used to give an indication of how heavily the queue is loaded. We can calculate the loading factor of any queue as follows.

$\rho = \frac{\lambda}{\mu}$

Additionally for queues that have finite buffers we can calculate the amount of traffic that is successfully accepted into the buffers. This is as follows.

$\lambda_a = (1 - P_k)\lambda$

One final piece of useful information is Little's Law, which states: The average number of customers in a queueing system L is equal to the average arrival rate of customers to that system λ, times the average time spent in that system W.

$L = \lambda W$

A.3 Well Known Queues

There are a number of well known queues covered in the literature, in this section we will show the formulae for these. Interested readers are directed to one of the many books on queueing theory [3, 96] for an explanation of how the various formulae are derived. Throughout the thesis we will rely on these formulae for our calculations.

A.3.1 The M/M/1 Queue

The M/M/1 queue has infinite buffers and one service facility. Both the arrivals and service facility are Markovian, or exponentially, distributed.

$\mathrm{P_n} = (1-\rho)\rho^2$

$L = \frac{\rho}{1-\rho}$

$L_q = \frac{\rho^2}{1-\rho}$

$W_s = \frac{\rho}{\lambda}$

$W = \frac{W_s}{1-\rho}$

$W_q = \frac{\rho W_s}{1-\rho}$

A.3.2 The M/M/1/K Queue

The M/M/1/K queue is essentially the same as a M/M/1 queue, other than it has a finite number of buffers denoted by K.

$$P_n = \begin{cases} \frac{(1-a)a^n}{1-a^{K+1}}, & \text{if } \lambda \neq \mu \\ \frac{1}{K+1}, & \text{if } \lambda = \mu \end{cases}$$

$$L = \begin{cases} \frac{a[1-(K+1)a^K+Ka^{K+1}]}{(1-a)(1=a^{K+1})}, & \text{if } \lambda \neq \mu \\ \frac{K}{2}, & \text{if } \lambda = \mu \end{cases}$$

$L_q = L - (1 - P_0)$

$W = \frac{L}{\lambda_a}$

$W_q = \frac{L_q}{\lambda_a}$

A.3.3 The M/G/1 Class Queue

The M/G/1 Class queue represents a queue with infinite buffers and one service facility where the arrivals are Markovian. However, it allows for more than one class of traffic to arrive, where the classes have differing service requirements. These classes are serviced to a mean rate with no other stipulation. We also introduce $E[x]$, which is the expected value of x (or the service, s, as used here).

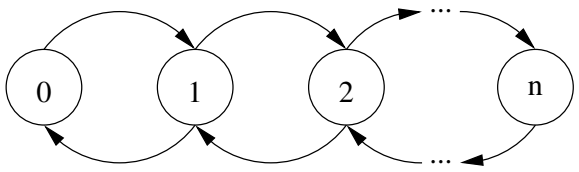


Figure A.2: A Simple Markov Chain

$\lambda = \lambda_1 + \lambda_2 + ... + \lambda_n$

$E[s_i] = \frac{1}{\mu_i}$

$W_s = \frac{\lambda_1}{\lambda} E[s_1] + \frac{\lambda_2}{\lambda} E[s_2] + ... + \frac{\lambda_n}{\lambda} E[s_n]$

$W_q = \frac{\lambda E[s^2]}{2(1-\rho)}$

$W_i = W_q + E[s_i]$

$W = \frac{\lambda_1}{\lambda} W_1 + \frac{\lambda_2}{\lambda} W_2 + ... + \frac{\lambda_n}{\lambda} W_n$

A.4 Markov Chains

This sections outlines some of the underlying mathematics required in order to solve queueing systems. It is our intention to cover this only briefly as any textbook on the subject will give a much better introduction to the subject than we have space for here.

Figure A.2 shows a simple birth-death Markov chain for a queue with n buffers. Each of the states represents the number of jobs that are in the system, starting at 0 and finishing at the maximum number of buffers (which may be infinite). As a job arrives we change states (to the right) with an average rate of λ. As jobs are serviced we change down states (to the left) with an average rate of μ. This is not the only sort of chain that we can construct, others will be covered as we progress through the thesis.

There are two methods for solving this chain. The first is empirically, for continuous-time chains. Here we are assuming that the chain is in a steady state, and hence no build up occurs in any state. For each state we construct balance equations, these are simply the sum of the rate of entering transitions minus the rate of leaving transitions. From this we can rearrange and solve for a general solution. It is these general solutions that we have shown above.

Another method to get the steady state probabilities is to use matrices. The following method is known as the 'power method'. First we construct the transition probability matrix (sometimes

called a stochastic matrix), Q, as follows: For each of the transitions in the Markov-chain we enter the probability into a matrix, i.e. q_{01} is the rate of transitions from state 0 to 1.

$$Q_{ij} = \begin{bmatrix} q_{oo} & q_{01} & . \\ q_{10} & q_{11} & . \\ . & . & . \end{bmatrix}$$

In the case of discrete-time Markov-chains each row must sum to 1; however, we are more interested in continuous-time Markov-chains, here each row must sum to zero. To achieve this we use self transition probability, i.e. $q_{00}, q_{11}...$, to make the sum of the row equal 0. For example:

$$q_{00} = -(q_{01} + q_{02} + ... + q_{0n})$$

$$q_{11} = -(q_{10} + q_{12} + ... + q_{1n})$$

The next step is to find q_{max}; the highest absolute value in the matrix Q. In this case we already know that the most likely candidate for such a value will be the top-left to bottom-right diagonal of the matrix Q. The formula is as follows:

$$q_{max} = max|Q_{ii}|$$

Using the following iterative formula we can now use are values of Q and q to find the steady state probabilities. This formula is obtained by rearranging and simplifying the steady state formula; this can be found in [96].

$$\pi^{(i)} = \pi^{(i-1)}Q^*$$

$$Q^* = I + \frac{Q}{q_{max}}$$

The value of the initial vector, $\pi^{(0)}$, is defined by placing a 1 in the column of the starting state. I is the identity matrix. We now iterate until the changes in the resulting value of π settle; however, this may never happen or take inordinately long. This is the drawback of this method, and its variations. Other methods, described in [96, 3], can overcome some of these difficulties but not all.

Appendix B

The Simulator Queue Model

This is an attempt to create the right simulation object for queues. The approach that is taken here is that a queue is an object that has state, an arrival operation and a departure operation.

```
module SimulatorQueueModel
  (
    SimulatedQueue(..),
    SimulatedQueueEmptyPredicate,
    queueSimulator
  )
where

import Maybe
```

The following data type holds the basic continuation information, at any step something either arrives (like a packet) or some departure occurs (at some time in the future - which could of course be immediately).

```
data SimulatedQueue qstate action interval
  = Running
    {
      stateOf     :: qstate,
      doArrival   :: !(qstate -> action ->
                       SimQYeild qstate action interval),
      doDeparture :: (qstate -> SimQYeild qstate action interval)
    }
  | Stopped
```

In this approach the queue object does not have a local concept of time and returns only relative time interval (therefore non-negative) at which a self-generated action will occur. The simulation methodology makes the assumption that all actions of a time step occur during that time step - an assumption that can be made true within a simulation environment.

There will need to be a later refinement of this model to explicitly manage time.

To make the above definition more readable here is a type definition of the return from each continuation step.

```
type SimQYeild qstate action time
  = (SimulatedQueue qstate action time, [action], Maybe time)
```

This gives us the individual steps in the finite state automata representing the basic operation of the queue - but does not give us a process that is easy to use elsewhere.

The first externally viable variant that we need to build is one in which a timed stream of actions is input into a queue and the output from this queue is also a timed action stream.

We need to handle two edge conditions:

- Empty input stream: We will make the assumption that when the input stream becomes empty that the output stream will empty itself and when it is empty the output stream will also terminate.
- The SimulatedQueue enters the Stopped state. In this case the output action queue will immediately terminate.

To cope with the first termination condition we will need an additional piece of information supplied by the queue implementation.

```
type SimulatedQueueEmptyPredicate qstate
  = qstate -> Bool
```

This now allows us to define what this process evolution looks like:

```
simQTimedActionStream
  :: (Num time, Ord time)
  => SimulatedQueueEmptyPredicate qstate
  -> Maybe time
  -> SimulatedQueue qstate action time
  -> [(time, action)]
  -> [(time, action)]
```

Termination cases:

First the case when the queue itself decides to stop.

```
simQTimedActionStream _ _ Stopped _
   = []
```

Next the case when the input queue has dried up. We will terminate on the first occurrence of one of two conditions: The first is that the queue becomes empty, the second is that the queue yields Nothing as its action time. This means that the queue process's evolution is dependent on an arrival - which will never come. There is a particular edge condition here that will result in an error being reported, when the queue is non-empty and the input list terminates and the time of the next action is Nothing. This would be a strange condition for a queue to be operating in - an example would be queues that keep a packet in the buffer until the next packet arrives. The reporting of an error in that case would be the same as reporting that a packet was not handled by the system. If this sort of behaviour is needed at some later date then the termination conditions will need to be amended accordingly.

```
simQTimedActionStream isemptyp suppliedtime simq []

  | isemptyp $ stateOf simq
    = []

  | otherwise
    = let
        (simq', actions, nextactiontime)
          = (doDeparture simq) (stateOf simq)

        currqtime
          = fromJust suppliedtime

        timedactions
          = [(currqtime, a) | a <- actions]

        continuation
          = case nextactiontime of
              Nothing  -> []
              (Just t) -> simQTimedActionStream isemptyp
                            (Just (currqtime + t)) simq' []
      in
        timedactions ++ continuation
```

Now the more general case, where the input queue is not empty. We need to decide whether the next evolution step is driven by a departure or an arrival. If the evolution time is Nothing then the queue can not evolve until some input action occurs.

Note that the queue has no internal concept of time, so whenever it yields a action it is calculated relative to the time of the action. In the arrival case, it is relative to the time of input action and, if not specified, it is the (original) time of the next departure action. For departure it is relative to current departure time.

```
simQTimedActionStream
  isEmptyP suppliedTime simQ
  allActions@((actionTime, action):futureActions)

  | isNothing suppliedTime || actionTime < currQtime
    = arrivalEvolution

  | otherwise
    = departureEvolution

      where
        currQtime
          = fromJust suppliedTime

        arrivalEvolution
          = let
            (simQ', actions, nextActionTime)
              = (doArrival simQ) (stateOf simQ) action
            timedActions
              = [(actionTime, a) | a <- actions]
            internalActionTime
              = case nextActionTime of
                  Nothing -> suppliedTime
                  Just i  -> Just (actionTime + i)
            continuation
              = simQTimedActionStream isEmptyP
          internalActionTime simQ' futureActions
          in
            timedActions ++ continuation

        departureEvolution
```

```
          = let
            (simQ', actions, nextActionTime)
              = (doDeparture simQ) (stateOf simQ)
            timedActions
              = [(currQtime, a) | a <- actions]
            internalActionTime
              = case nextActionTime of
                  Nothing -> Nothing
                  Just i  -> Just (currQtime + i)
            continuation
              = simQTimedActionStream isEmptyP
          internalActionTime simQ' allActions
          in
            timedActions ++ continuation
```

This supplies us with the evolution of the process given an input timed action stream. All that is now missing is the way in which the creation of the process occurs.

```
queueSimulator
  :: (Num timeType, Ord timeType)
  => SimulatedQueueEmptyPredicate localState
  -> SimulatedQueue localState actionType timeType
  -> ([(timeType, actionType)]
  -> [(timeType, actionType)])


queueSimulator pred simQ
  = simQTimedActionStream pred Nothing simQ
```

Bibliography

[1] Henrik Abrahamsson and Bengt Ahlgren. Using empirical distributions to characterize web client traffic and to generate synthetic traffic. In *IEEE/Globecomm'00*, San Francisco, November 2000.

[2] R. Agrawal, R. L. Cruz, C. Okino, and R. Rajan. Performance bounds for flow control protocols. *IEEE Transactions on Networking*, 7(3):310–323, June 1999.

[3] Arnold O. Allen. *Probability, Statistics and Queueing Theory with Computer Science Applications*. ISBN 0120510510, Academic Press, 1990.

[4] P. Almquist. RFC 1349, Type of Service in the internet protocol suite, July 1992.

[5] Eitan Altman, Konstantin Avrachenkov, and Chadi Barakat. A stochastic model of TCP/IP with stationary random losses. In *SIGCOMM*, pages 231–242, 2000.

[6] T. Anderson, S. Owicki, J. Saxe, and C. Thacker. High-speed switch scheduling for local-area networks. *ACM Transactions on Computer Systems*, 11(4):319–352, November 1993.

[7] F. Baker. RFC 1812, Requirements for IP version 4 routers, June 1995.

[8] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. In *Measurement and Modeling of Computer Systems*, pages 151–160, 1998.

[9] F. Baskett, K. Mani Chandy, R. Muntz, and F. G. Palacios. Open, closed and mixed networks of queues with different classes of customers. *Journal of the ACM*, 22(2):248–260, April 1975.

[10] J. C. R. Bennet and H. Zhang. WF2Q: Worst-case fair weighted queueing. In *IEEE Infocom '96*, pages 120–128, 1996.

[11] Razvan Beuran, Mihail L. Ivanovici, Bob Dobinson, and Peter Thompson. Network quality of service measurement system for application requirements evaluation. In *2003 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS'03)*, Montreal, Canada, July 2003.

[12] M. Borella, D. Swider, S. Uludag, and G. Brewster. Internet packet loss: Measurement and implications for end-to-end QoS. In *International Conference on Parallel Processing*, 1998.

[13] R. Braden, D. Clark, and S. Shenker. RFC 1663, Integrated services in the internet architecture: an overview, June 1994.

[14] J. T. Bradley, N. J. Dingle, P. G. Harrison, and W. J. Knottenbelt. Distributed computation of passage time quantiles and transient state distributions in large semi-markov models. In *PMEO*. IEEE Society, April 2003.

[15] J. T. Bradley and N. Thomas. Putting quality of service into a network by making the traffic markovian. In *Proceedings of the Fifteenth European Simulation Multiconference*, Prague, June 2001.

[16] Jin Cao, William S. Cleveland, Dong Lin, and Don X. Sun. Internet traffic tends to poisson and independent as the load increases. Technical report, Bell Labs, 2001.

[17] K. V. Cardoso and J. F. de Rezende. Design and use of an aggregated HTTP traffic model. Technical report, Universidade Federal do Rio de Janeiro.

[18] Neal Cardwell, Stefan Savage, and Thomas Anderson. Modeling TCP latency. In *INFOCOM (3)*, pages 1742–1751, 2000.

[19] Junghoo Cho and Hector Garcia-Molina. Parallel crawlers. In *The 11th International World–Wide Web Conference*, pages 124–135, 2002.

[20] Hyoung-Kee Choi and John O. Limb. A behavioral model of web traffic. In *Seventh Annual International Conference on Network Protocols*, pages 327–334, Toronto, Canada, November 1999.

[21] Gagan H. Choudhury, Kin K. Leung, and Ward Whitt. Calculating normalization constants of closed queuing networks by numerically inverting their generating functions. *Journal of the ACM*, 42(5):935–970, 1995.

[22] S. Chuang, A. Goel, N. McKeown, and B. Prabhakar. Matching output queueing with a combined input output queued switch. *IEEE Journal on Selected Areas in Communications*, 17(6):1030–1039, June 1999.

[23] K. Claffy, G. Miller, and K. Thompson. The nature of the beast: recent traffic measurements from an internet backbone. In *INET98*, 1998.

[24] William S. Cleveland and Don X. Sun. Internet traffic data. *Journal of the American Statistical Association*, 95:979–985, 2000.

[25] Professor Peter Cochrane. Highlights of the 1998 3M innovation lecture, 1998. www.cochrane.org.uk/opinion/papers/backto.htm.

[26] A. Conway, de Souza e Silva, and S. Lavenberg. Mean value analysis by chain of product form queueing networks. *IEEE Transactions on Computers*, 38:432–442, 1989.

[27] A. Conway and N. Georganas. Decomposition and aggregation by class in closed queueing networks. *IEEE Transactions on Software Engenieering*, 33:768–791, 1968.

[28] M. Crispin. RFC 2060, Internet Message Access Protocol - version 4 revision 1, December 1996.

[29] R. Cruz. A calculus for network delay: Part I: Network elements in isolation. *IEEE Transactions on Information Theory*, 37(1), January 1991.

[30] R. Cruz. A calculus for network delay: Part II: Network analysis. *IEEE Transactions on Information Theory*, 37(1):132–141, January 1991.

[31] R. Cruz. Quality of service guarantees in virtual circuit switched networks. *IEEE Journal on Selected Areas in Communications*, 13(6):1048–1056, 1995.

[32] Peter B. Danzig, Ramon Caceres, Danny Mitzel, and Deborah Estrin. An empirical workload model for driving wide-area TCP/IP network simulations. *Journal of Internetworking*, 3(1):1–26, March 1992.

[33] Peter B. Danzig and Sugih Jamin. TCPLib: A library of TCP/IP traffic characteristics. Technical Report TR CS-SYS-91-01, USC Networking and Distributed Systems Laboratory, October 1991.

[34] Neil Davies, Judy Holyer, and Peter Thompson. An operational model to control loss and delay of traffic at a network switch. In *Third IFIP workshop on the Management and Design of ATM Networks*, pages 218–231, Queen Mary and Westfield College, University of London, March 1999.

[35] J. D. Day and H. Zimmermann. The OSI reference model. In *Proceedings of the IEEE*, pages 1334–1340. IEEE Press, December 1983.

[36] A. Demers, S. Keshav, and S Shenker. Analysis and simulation of a fair queueing algorithm. *Computer Communication Review*, 19(4):1–12, September 1989.

[37] Agner Krarup Erlang. The theory of probabilities and telephone conversations. *Nyt Tidsskrift for Matematik B*, 20, 1909.

[38] Agner Krarup Erlang. Solution of some problems in the theory of probabilities of significance in automatic telephone exchanges. *Elektrotkeknikeren*, 13, 1917.

[39] Anja Feldmann, Anna C. Gilbert, Polly Huang, and Walter Willinger. Dynamics of IP traffic: A study of the role of variability and the impact of control. In *SIGCOMM*, pages 301–313, 1999.

[40] D. Ferrari and D. C. V. Verma. A scheme for real-time channel establishment in wide-area networks. *IEEE Journal on Selected Areas in Communications*, 8(3):368–379, 1990.

[41] R. Fielding, U. C. Irvine, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616, Hypertext transfer protocol – HTTP/1.1, June 1999.

[42] Mike Flannagan. *Administering Cisco QoS for IP Networks*. Ingress Media Inc, 2001.

[43] S. Jamaloddin Golestani. A self-clocked fair queueing scheme for broadband applications. In *IEEE INFOCOM '94*, pages 636–46. IEEE Computer Soc. Press., 1994.

[44] Pawan Goyal, Harrick M. Vin, and Haichen Cheng. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Transactions on Networking*, 5(5):690–704, October 1997.

[45] Timothy G. Griffin and Gordon T. Wilfong. An analysis of BGP convergence properties. In *Proceedings of SIGCOMM*, pages 277–288, Cambridge, MA, August 1999.

[46] IETF Network Working Group. RFC 2205, Resource ReSerVation protocol, September 1997.

[47] IETF Network Working Group. RFC 2475, An architecture for differentiated services, December 1998.

[48] C. Hedrick. RFC 1058, Routing Information Protocol, June 1988.

[49] G. Heide, C. J. Vowden, G. Willmott, and N. Davies. Evolution of traffic patterns in a multi-hop network. In *18th UK Performance Engineering Workshop*, University of Glasgow, July 2002.

[50] P. Hurley, Mourad Kara, J. Y. Le Boudec, and P. Thiran. ABE: Providing a low-delay service within best effort. *IEEE Network Magazine*, 15(3), May 2001.

[51] James R. Jackson. Networks of waiting lines. *Operations Research*, 5(4), August 1957.

[52] V. Jacobson. Congestion avoidance and control. *ACM Computer Communication Review; Proceedings of the SIGCOMM '88 Symposium in Stanford, CA*, 18(4):314–329, August 1998.

[53] Xusheng Tian Jie. A unified framework for understanding network traffic using independent wavelet models. In *IEEE INFOCOM*, 2000.

[54] M. Karol, M. Hluchyj, and S. Morgan. Input versus output queueing in a space division switch. *IEEE Transactions Communication*, 35:1347–1356, 1987.

[55] M. Katevenis, S. Sidiropoulos, and C. Courcoubetis. Weighted round-robin cell multiplexing in a general-purpose ATM switch chip. *IEEE Journal on Selected Areas in Communications*, 9(8):1265–1279, October 1991.

[56] F. Kelly. Notes on effective bandwidths. In F. P. Kelly, S. Zachary, and I. Zeidins, editors, *Stochastic Networks: Theory and Applications*, pages 141–168. Oxford University Press, 1996.

[57] J. F. C. Kingman. On queues in heavy traffic. *Journal of the Royal Statistical Society*, Series B(24):383–392, 1962.

[58] P. Krishna, N. Patel, A. Charny, and R. Simcoe. On the speedup required for work-conserving crossbar switches. *IEEE Journal on Selected Areas in Communications*, 17(6):1057–1066, June 1999.

[59] S. Lam and Y. Lien. A tree convolution algorithm for the solution of queueing networks. *Communications of the ACM*, 26:203–215, 1983.

[60] R. LaMaire and D. Serpanos. Two-dimensional round-robin schedulers for packet switches with multiple input queues. *IEEE/ACM Transactions on Networking*, 2(5):471–482, October 1994.

[61] Will E. Leland, Murad S. Taqq, Walter Willinger, and Daniel V. Wilson. On the self-similar nature of Ethernet traffic. In Deepinder P. Sidhu, editor, *ACM SIGCOMM*, pages 183–193, San Francisco, California, 1993.

[62] Dong Lin and Robert Morris. Dynamics of random early detection. In *SIGCOMM '97*, pages 127–137, Cannes, France, September 1997.

[63] J. Luthi and G. Haring. Bottleneck analysis for computer and communication systems with workload variabilities and uncertainties. In I. Troch and F. Breitenecker, editors, *2nd Int. Symposium on Mathematical Modelling*, pages 525–534. Technical University Vienna, February 1997.

[64] J. Luthi, S. Majumdar, and G. Haring. Mean value analysis for computer systems with variabilities in workload. In *IEEE International Computer Performance and Dependability Symposium, IPDS*, 1996.

[65] Reiser M. and So S. Lavenberg. Mean value analysis of closed multi-chain queueing networks. *JACM*, 27(2):313–322, April 1980.

[66] Bruce A. Mah. An empirical model of HTTP network traffic. In *INFOCOM (2)*, pages 592–600, 1997.

[67] G. Malkin. RFC 2453, RIP version 2, November 1998.

[68] Nick McKeown. The iSLIP scheduling algorithm for input-queued switches. *IEEE/ACM Transactions on Networking*, 7(2):188–201, April 1999.

[69] David L. Mills. RFC 1303, Network Time Protocol (version 3) specification, implementation and analysis, March 1992.

[70] P. Mockapetris. RFC 1035, Domain names - implementation and specification, November 1987.

[71] Robert Morris and Dong Lin. Variance of aggregated web traffic. In *INFOCOM*, pages 360–366, 2000.

[72] J. Moy. RFC 1583, OSPF version 2, March 1994.

[73] J. Myers and M. Rose. RFC 1939, Post Office Protocol - version 3, May 1996.

[74] J. Padhye, V. Firoiu, D. Towsley, and J. Krusoe. Modeling TCP throughput: A simple model and its empirical validation. *Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 303–314, 1998.

[75] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in intergrated services networks. *IEEE/ACM Transactions on Networking*, 2(2):137–150, April 1994.

[76] Vern Paxson. End-to-end Internet packet dynamics. In *Proceedings of the ACM SIGCOMM '97 conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, September 1997.

[77] Vern Paxson and Sally Floyd. Wide area traffic: the failure of Poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, 1995.

[78] J. Postel. RFC 768, User Datagram Protocol, August 1980.

[79] J. Postel. RFC 792, Internet Control Message Protocol, September 1981.

[80] J. Postel. RFC 821, Simple Mail Transfer Protocol, August 1982.

[81] J. Postel and J. Reynolds. RFC 854, Telnet protocol specification, May 1983.

[82] J. Postel and J. Reynolds. RFC 959, File Transfer Protocol (FTP), October 1985.

[83] DARPA Internet Program. RFC 793, Transmision Control Protocol, September 1981.

[84] DARPA Internet Program. RFC 971, Internet Protocol, September 1981.

[85] Pradeep Ramakrishnam. Self-similar trafic models. Technical Report CSHCN TR 99-5, Center for Satellite and Hybrid Communication Networks, 1999.

[86] Alastair Reid and Sigbjorn Finne. GreenCard: A Haskell FFI preprocessor. http://haskell.org/greencard/.

[87] Y. Rekhter and T. Li. RFC 1771, A border gateway protocol 4 (BGP-4), March 1995.

[88] J. Reynolds and J. Postal. RFC 1060, Assigned numbers, March 1990.

[89] H. Sariowan, R. Cruz, and G. Polyzos. Scheduling for quality of service guarantees via service curves. In *International Conference on Computer Communications and Networks (ICCCN)*, pages 512–520, September 1995.

[90] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RFC 1889, RTP: A transport protocol for real-time applications, January 1996.

[91] D. Serpanos and P. Antoniadis. FIRM: a class of distributed scheduling algorithms for high-speed ATM switches with multiple input queues. In *IEEE Infocom 2000 Conference*, Tel Aviv, Israel, March 2000.

[92] S. Shenker, D. Clark, D. Estrin, and S. Herzog. Pricing in computer networks: Reshaping the research agenda. *ACM Computer Communication Review*, 26:19–43, April 1996.

[93] M. Shreedhar and G. Varghese. Efficient fair queueing using deficit round-robin. *IEEE/ACM Transactions on Networking*, 4(3):375–385, June 1996.

[94] W. Richard Stevens. *TCP/IP Illustrated Volume 1.* ISBN 0201633469, Addison-Wesley, 1994.

[95] K. Thompson, G. J. Miller, and R. Wilder. Wide-area internet traffic patterns and characteristics. *IEEE Network*, 11(6):10–23, November/December 1997.

[96] Kishor S. Trivedi. *Probability and Statistics with Reliability, Queueing and Computer Science Applications.* Wiley, 2002.

[97] International Telecommunication Union. "one-way transmission time", transmission systems and media: General characteristics of international telephone connections and international telephone circuits, February 1996.

[98] Berkley. University of California. NS2 simulator, the LBLN network simulator.

[99] D. C. Verma, H. Zhang, and D Ferrari. Delay jitter control for real-time communication in a packet switched network. In *Tricomm 91*, pages 35–46, April 1991.

[100] Chris Vowden. Confidential technical U4EA report, 1998.

[101] Chris Vowden and Gerhard Heide. Confidential technical U4EA report, Feb 2002.

[102] S. Wang, D. Xuan, R. Bettati, and W. Zhao. A study of providing statistical QoS in a differentiated services network. In *The 2nd IEEE International Symposium on Network Computing and Applications*, 2003.

[103] W. Willinger, M. S. Taqqu, and R. Sherman. Proof of a fundamental result in self-similar traffic modeling. *ACM SIGCOMM Computer Communication Review*, 27(2):5–23, April 1997.

[104] R. Wolff. Poisson arrivals see time averages. *Operations Research*, 30(2):223–231, 1982.

[105] Ikjun Yeom and A. L. Narasimha Reddy. Modeling TCP behavior in a differentiated services network. *IEEE/ACM Transactions on Networking*, 9(1):31–46, 2001.

[106] T. Ylonen. The SSH (secure shell) remote login protocol, November 1995.

[107] L. Zhang. Virtual clock: A new traffic control algorithm for packet switching networks. In *ACM SIGCOMMN'90*, Philadelphia, September 1990.