Monitoring, Analysis and Simulation of Packet Switched Network Traffic

Ross Alexander

A thesis submitted in fulfilment of the requirements for the degree of
Masters of Science in Mathematics,
The University of Auckland, 1984.

Abstract

The original intention of this thesis was to examine network traffic entering and leaving the university campus. The aim was to try to obtain a measure of how saturated the link was.

While it is simple to give a figure for the average utilisation over time this is not adequate. Clearly if the link is heavily utilised, say over 60%, then users will experience delays when sending or receiving data. The problem is that even on lightly utilised links delays through congestion can also occur. This occurs because data communications traffic levels are not constant but fluctuate over time.

These fluctuations can occur over very short periods of time giving rise to the concept of a *burst* of traffic. These bursts of traffic can be of intensity more than five times that the average utilisation so that if a user is trying to send data and it coincides with a burst the user will experience delays. Traffic which exhibits these wild fluctuations is known as *bursty* traffic.

To this end it is important to gain an insight into the behaviour of this bursty traffic and try to measure its effect on overall network performance.

A common assumption in modelling computer networks is that arrivals occur as a *Poisson process*. In the thesis we challenge this assumption and examine the results of doing so.

To this end we decided to take experimental measurements of real computer networks to try and fit a model to them. The aim was to produce a theoretical model which was consistent with real traffic behaviour.

A collection of mathematical models were investigated. This included simulating their behaviour on computers and examining their output in comparison to that of observed network. The observed traffic show *self-similar* (or *fractal*) behaviour. The thesis examined which of the investigated models produced similar behaviour. The final results are summarised in the conclusion with suggestions for possible areas of further research.

Contents

List of Figures

List of Tables

Chapter 0

Overview of the thesis

The thesis is divided into five chapters, with an appendix and bibliography at the end. Below is a summary of each chapter.

Chapter 1 This chapter introduces computer networks and the software which controls them. It discusses in general terms basic computer network technology and then in more detail those technologies which were used for the thesis. The computer networks around the University of Auckland are also commented on.

Chapter 2 This chapter involves a detailed discussion on how the raw samples were collected and the problems associated with that exercise.

Chapter 3 This chapter is a detailed discussion of the statistical models investigated in the thesis. It contains all the underlying mathematics used later during the simulations. It also contains an introduction to the mathematics behind the observed *fractal* behaviour.

Chapter 4 This chapter summaries the results from the observed data primarily in the form of graphs, with commentary.

Chapter 5 Here the results of the simulations are discussed, along with how they were produced and how they compare with the observed data.

Chapter 6 The conclusion summarises the work done in the thesis and mentions possible extensions. It also discusses some of the problems which were found during the writing of the thesis.

Appendix A This appendix contains mathematical definitions for basic statistical concepts. This is background reference for those with limited statistical knowledge.

Appendix B Source code listing of software programs written and used for the thesis.

Chapter 1

Computer networks and protocols

1.1 Switched networks

Packet switched networks, and computer networks in general, have become an intrinsic part of the computer industry. With the fall in their cost and the increase in user-friendly software it is now possible for people to connect several computers together to form a network with ease.

Computer networks come in all shapes and sizes and there are a multitude of methods for connecting two or more computers together. Each individual network is different, depending on its topology, the transmission technology and the software running on it. To fully understand a network it is necessary to know about each element.

1.1.1 Packet switching versus circuit switching

1.1.1.1 Circuit switching

Circuit switched networks are one of two types of communication networks, the other being packet switched networks. Circuit switched networks are the older of the two as they are the basis of the telephone system.

Originally telephone networks were based around a central office where wires from each telephone terminated. When a call was placed the operator would physically connect the two wires together. This would form an electrical *circuit* which would remain intact throughout the duration of the call. When the call ended the operator would disconnect the two wires. It is not clear where the term switching came from but it has applied to the operation of connecting two parties together at a central office from the earliest times of telephony.

Today the technology is very different but the concepts of circuit switching remain the same, that is a fixed path between the end parties is set up at the start of a connection, remains intact throughout the duration of the call, and is torn down only after the call has ended.

One important concept with circuit switched systems is that of having a fixed resource. Whether you talk or not when using the telephone does not change the amount of resource being used. Exactly one circuit is being used regardless of how much information there is

at any given time, that is the resource usage does not shrink or expand once the circuit has been set up.

1.1.1.2 Packet switching

Packet switching differs from circuit switching in that whereas a circuit switching system treats a connection as a continuous stream, packet switching breaks everything up into discrete, limited size blocks. Each block is known as a *packet*. A common analogy is the idea of a postcard in a postal network. To send a long message several postcards have to be written and sent. Each postcard is treated independently and they may not arrive in the same order as they are sent or follow the same route in getting to their destination.

1.1.1.3 Hybrid systems

A common hybrid system between circuit switched and packet switched networks is based around the concept of a *virtual circuit*. A virtual circuit behaves like a standard circuit in that for a connection between two parties a circuit is formed, remains fixed throughout the duration of the connection, and is torn down once the connection has ended.

Virtual circuits differ in that rather than having a fixed resource throughout the duration of the connection to support a continuous stream of data the input is broken up into packets. Each packet follows a set route decided upon at the creation of the circuit, but if there is no input to be transmitted then no packets are sent. This way resources only need to be used when something is to be transmitted, the resources growing or shrinking as needed.

The idea is to keep the simplicity of a fixed circuit while trying to maximise the utilisation of limited resources. While it looks like a circuit it suffers from the major problem of a packet switched system, that is congestion. With proper circuit switching if inadequate resources are available, a connection will fail when the system attempts to open it (as in a busy signal or trunk full signal experienced with the telephone system) but in packet switching resources are only used when a packet is sent so having inadequate resources is only discovered when a packet is actually sent. A supplier may allocate more virtual circuits then the number of actual physical ones, assuming that not every connected party will want to send something at any given point in time. If too many parties do try to send something at the same time then the network's resources will be exceeded causing some packets to be dropped (the packet is ignored by the system and ceases to exist) or delayed.

Virtual circuit based networks are called *connection oriented packet switched networks*. In contrast packet switched networks where each packet is completely self contained are called *connectionless packet switched networks* or just *connectionless networks*.

1.1.1.4 Datagram networks

Connectionless packet switched networks are often called *datagram* networks. Because the packets are self contained they are often thought of as a individual blocks of data travelling through the network and are hence called datagrams. Most datagram networks follow the *hop-by-hop* paradigm, that is each intermediate system redirects datagrams without respect to the datagram's previous travels or its further travels. This implies datagrams do not keep

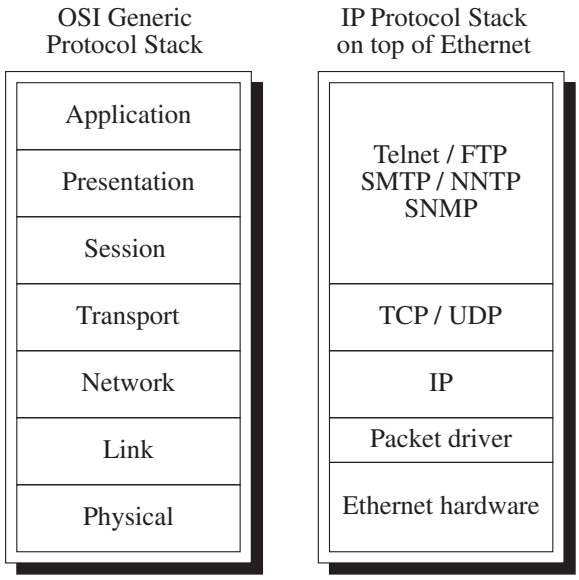

Figure 1.1: OSI and TCP/IP Protocols Stacks

a history of their travels but rely on the co-operation and co-ordination of intermediate systems to relay them to their destination.

1.2 Network protocols

A *protocol* is a set of rules on how to behave in given a set of circumstances. Protocols generally come grouped together as a *protocol family* and are normally arranged in a semi hierarchical fashion, with many protocols relying on others to do work for them while being relied on by other protocols. This pseudo hierarchy for a given protocol family is called a *protocol stack*.

Throughout the history of computing there have been many network protocols, most of them proprietary to specific systems. As market forces drive networks towards greater connectivity many of these protocols have fallen by the wayside, leaving a handful of system specific protocols and two major 'open' systems. Common network protocols include SNA (Systems Network Architecture) from IBM, DECnet from Digital Equipment Corporation, IPX (Netware) from Novell and AppleTalk from Apple Computer. The two major open systems protocols are TCP/IP (Transmission Control Protocol / Internet Protocol) and OSI (Open System Interconnection) (See Figure 1.1).

1.2.1 Internet Protocol

1.2.1.1 History and comments

The Internet Protocol, commonly known as IP, developed from ARPANET (Advanced Research Project Agency NETwork), one of the earliest packet switching networks developed [17]. ARPANET was funded by the United States Department of Defence, who required a

computer network able to survive a limited nuclear attack. For this reason emphasis was placed on distributing decision making (as any centralised system would be an obvious target) and reliability in extreme circumstances.

ARPANET no longer exists but has been replaced by the *Internet*, a world wide network of networks, all using IP as their base technology. IP is a datagram protocol and is usable over a very wide range of transmission technologies.

1.2.1.2 Deployment within the University

IP addresses are 32 bits wide and are split up into two parts, the first being the network address and the other the host address. IP routes packets to the network level where it expects to be able to communicate directly to the host. Originally IP addresses were split up into classes, for example class B addresses had 16 bits for the network address and 16 bits for the host address, with class C having 24 bits for the network address and 8 bits for the host address.

The class of the address was encoded into the high bits of the address so a router could tell how many bits it needed to make a routeing decision on. This fixing of the address part sizes was extremely limiting so it was replaced by a 32 bit contiguous mask. The mask indicates which part of the address is the network part.

TCP/IP is deployed throughout the university and is available to all departments. Currently the university has a class B address which is split up into 256 class C addresses using address masks. This is important for routeing packets within the university network. From the outside the university has a single 16 bit network address `130.216.0.0` (the full address is always used even though the only the top 16 bits are significant, non significant bits are set to zero) but inside the university all networks addresses are 24 bits wide, for example mathematics is `130.216.15.0`.

1.2.2 AppleTalk

1.2.2.1 History and comments

AppleTalk is the network protocol suite developed by Apple Computer Inc. when it released the Macintosh [7]. It was developed specifically for small local area networks. Originally it only ran over *LocalTalk*, a bus topology physical network. LocalTalk has a very low bandwidth by today's standards but it didn't require any extra hardware since all Macintoshes come with it built in. In later versions *EtherTalk* was added. This enabled the AppleTalk protocols to run over standard Ethernet.

It is mainly used for connecting Macintoshes to file servers and printers. Over small networks (up to the size of a campus network) the protocol runs without difficulty. It has a distributed directory service which makes using it simple, but limitations in that protocol now limit how large the network can grow.

AppleTalk cannot easily be used over wide area networks. This is mainly to do with limitations in its ability to address physical entities on the network. Also, the directory service breaks down over slower links.

AppleTalk splits the network up into logical *zones*, where one or more physical segment belongs to a zone. Zones are not used in packet routeing but are for human use only, dividing the network up logically to simplify directory lookups, that is AppleTalk has a two level directory consisting of zones and logical entities, which belong to exactly one zone. Because zones cannot belong to other zones, connecting two separate AppleTalk networks, say two universities, is infeasible without extreme care.

1.2.2.2 Deployment within the University

AppleTalk is also widely deployed throughout the campus. The majority of the network is over Ethernet with some outlying sections using LocalTalk.

1.2.3 IPX

1.2.3.1 History

IPX, or *Internet Packet eXchange* is the network layer protocol used by *Netware* [3]. Netware is a product from Novell Inc which provides file server and remote printing services. A Netware network is a collection of file and printer servers connected together using IPX. Client machines, running MS-DOS or Windows, connect to a server, and software on the client machine makes the remote file systems behave as if they were connected locally.

IPX itself is a very simple protocol with a two level addressing system. Physical segments are each assigned an address, and machines on the physical segments are addressed using their hardware address. This simplicity reduces the amount of computing required to send each packet, but limits the flexibility of the protocol. IPX runs almost exclusively over Ethernet.

1.2.3.2 Deployment within the University

IPX can go over any Ethernet segment provided it is able to be routed onto it. Most departments have enabled IPX routeing and often their Novell servers also do IP and AppleTalk routeing.

Netware also places a non trivial load on the network because every two minutes (or however long the administrator sets it) the router broadcasts information about every Netware service available. When the number of file servers exceeds twenty this information becomes large in size. Currently the information is about 500 kilobytes in size. On busy networks such as student labs, this is unwelcome extra traffic.

1.3 Physical transmission media

While the number of common network protocols is less than ten the number of ways of transmitting digital information is at least five times this. Each method differs in behaviour and hence alters the characteristics of traffic using it. Fortunately most methods fall into broadly defined groups, with each member of the group exhibiting similar behaviour.

1.3.1 Analog and digital transmission technologies

All data transmitted is electro-magnetic energy at some stage (whether it be light, radio or signals along copper wire) and as such is analog in nature. All modern data communication is digital in nature so transmission requires signals to be encoded. The term *bandwidth* originally came from broadcasting, where it determined how large a slice of the radio spectrum a transmitter could use. This in turn governed the frequency range of the transmission and how much information could be sent in a given period of time. The modern usage still measures the rate at which data can be sent but unless specified should not be seen as relating to the underlying analog technology.

1.3.2 Point to point connections

Point to point connections are the easiest to visualise. *Transceivers* (devices which both transmit and receive data) are placed at either end of a cable. If both transceivers can send simultaneously then the medium is said to support *full duplex* transmission otherwise the medium is said to be *half duplex*.

Point to point connections have two major characteristics. The first is bandwidth, measured in bits per second (bps) and in magnitudes of multiples of one thousand (this is different from computers where orders increase in multiples of 2^{10}). The other is *latency*, which is a measure of delay between when data is sent and subsequently received. This is normally measured in the orders of seconds (for example, milliseconds or microseconds).

1.3.2.1 Framing

Data is not sent in a raw stream but is encapsulated into discrete, limited size packets, known as *frames*. Framing the data avoids excessive error propagation as well as providing information for clock synchronisation. Frames may also contain information about addressing data to a particular transceiver and error correction.

Frame sizes vary from as little as 53 *octets* (an octet is eight bits, the term byte is normally used with respect to computers) to over 8192 octets (8 Kbytes). It is important for protocols using a particular transmission medium to match their packet sizes to that of the transmission frames.

1.3.2.2 Serial and parallel connections

Modern computers normally store data in blocks of 32 or 64 bits. When transmitting these blocks inside the computer they are sent in *parallel*, that is to say 32 or 64 separate connections are used, with each bit being sent simultaneously. This means that large amounts of information can be sent rapidly.

When sending data between two computers physical wires are required making very wide parallel cables extremely expensive. While eight bit wide parallel cables are common they only ever extend a distance of metres. Wider cables are now being used to connect computers to high speed peripherals but only up to a distance of one or two metres.

Over longer distances these blocks of data are sent bit by bit in sequence. This is known as *serial* transmission and only requires two copper wires or one optical fibre per direction.

Over any significant distance it is cheaper to make serial transmission faster than to make parallel cables wider.

1.3.3 Contention and bandwidth allocation

When two or more parties attempt to simultaneously use a limited resource then one or more must miss out. This fighting is called *contention* and the process for deciding who finally gets the resource is called *contention resolution*. Contention occurs in many places in computing and specifically in data transmission where it is associated with transmitters wanting to simultaneously send data with limited bandwidth.

Contention can be resolved in a number of ways, each having advantages and disadvantages.

- Stations are given set priorities. If a station with a higher priority wants to send then any sender which is lower must stop. This is called a *priority based* system.
- A centralised station gives the right to send to subservient or slave stations. This is called *polling*.
- The right to send is passed from station to station in an orderly fashion. This method is called *token passing*.
- Two or more stations attempt to send and collide. They then fight among themselves until there is a winner who gets the right to send. These are called *contention* networks.
- Data is transmitted in buckets and stations wanting to send must let a certain number of empty buckets pass before they can use one. This broadly falls into the category of *distributed queue* networks.

1.3.4 Broadcast media

Radio, television and satellite are all broadcast media. All receiving stations get an identical signal from the sender. This can in principle be extended to include all receivers connected to a single piece of wire.

This idea can be taken further and we may define a broadcast network as a network such that any station can send a single packet to every other connected station, which will receive an identical copy of that packet. This definition includes all physical topologies where signals are propagated to every connected station. Hence we can include such networks as token ring and dual bus networks.

Broadcasting is very useful for sending information to every station and is very efficient provided the transmission technology is based on network wide signal propagation, where broadcast essentially comes for free. When the physical technology does not allow for broadcasting it must be simulated. This means every station wanting to receive broadcasts must have the message individually sent to them. For any network of non trivial size this can become hugely expensive in bandwidth.

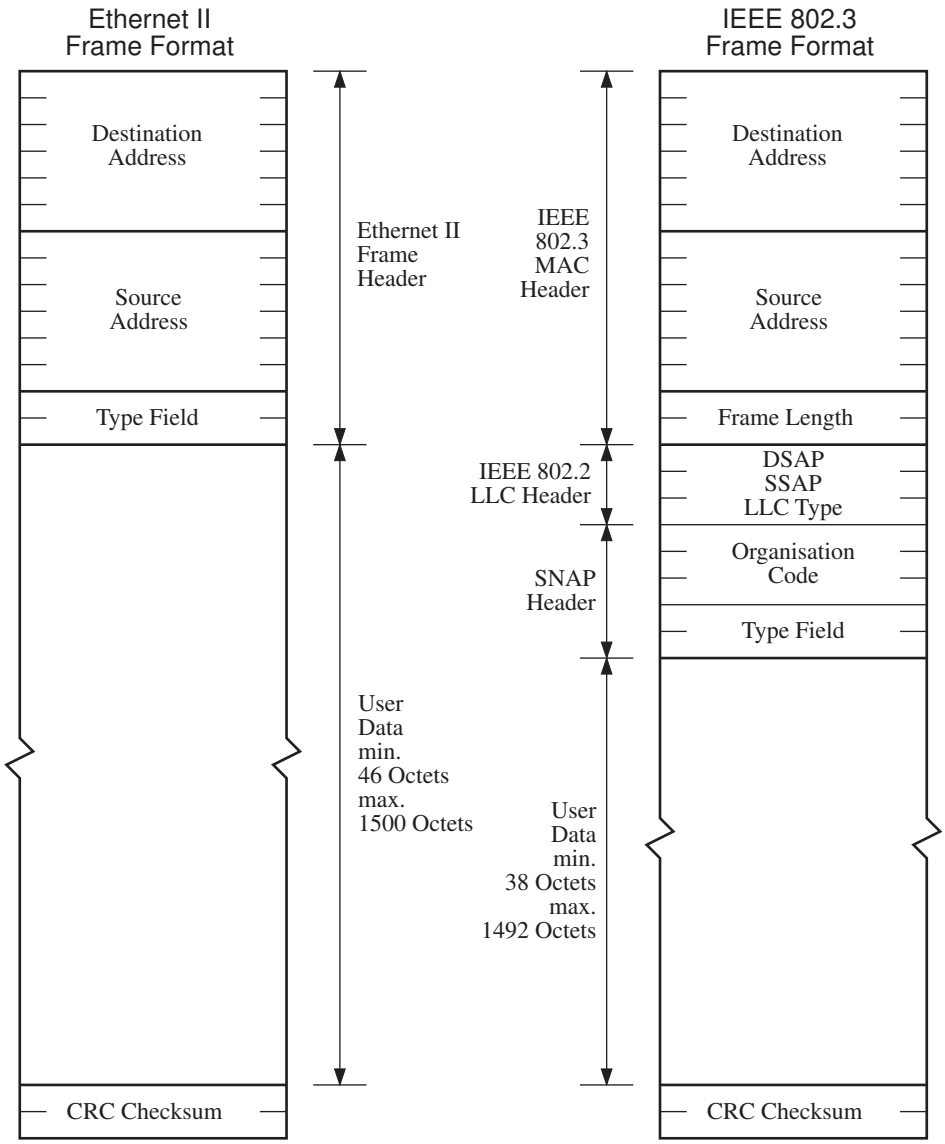

Figure 1.2: Ethernet II and IEEE 802.3 Frames

1.3.5 Common transmission technologies

Below is a quick overview of common transmission technologies. Full details of some of them will be given in later chapters.

1.3.5.1 Ethernet

Ethernet is a contention based broadcast network designed to work over short distances (tens to hundreds of metres). It is the most widely deployed transmission technology in computer networking with equipment cheaply and widely available. Its relative ease of installation and maintenance has lead to it becoming the baseline technology within the micro computer industry. It is now known as IEEE 802.3, the standard which currently defines it

[4] [15]. The raw transmission speed is normally 10 Mbps but work is proceeding on defining a 100 Mbps standard.

The original Ethernet is now known as Ethernet II or Blue Blue Ethernet, after the colour of the book which defined the standard. It is physically compatible with IEEE 802.3 but has a different frame structure (See Figure 1.2). Ethernet II is still commonly used, especially with respect to TCP/IP.

1.3.5.2 Token ring

Token ring, specifically the standard IEEE 802.5, is a token passing broadcast network [16]. It is not as common as Ethernet and is normally associated with IBM equipment. Token ring normally has a raw data rate of either 1, 4 or 16 Mbps. Note that it is unwise to make a direct comparison between these rates and that of Ethernet as raw transmission rate is a poor indicator of true throughput.

1.3.5.3 FDDI

FDDI, or Fibre Distributed Data Interface, is another token passing network. It is very similar to IEEE 802.5 except it runs on optical fibre rather than copper wire. Raw transmission rate (as seen by a connected station) is 100 Mbps.

1.3.5.4 PPP and SLIP

While not physical technologies these define how data is to be sent along serial connections. SLIP (Serial Line Internet Protocol) is a de-facto standard used because of its simplicity and wide availability [18]. PLIP (Parallel Line Internet Protocol) is an analogous standard for parallel connections but is fairly uncommon. PPP (Point to Point Protocol) is a full and complete standard for transmitting packets over point to point connections [22]. It provides Data Link layer functions such as error checking for the link.

1.4 The University campus network

The campus network is a collection of physical networks connected together via routers. A majority of the physical networks are Ethernet, either using copper wire or point to point connections using optical fibre.

1.4.1 Computer Centre network

The computer centre network did have five sub-networks when the experiments were being performed. Since then a major re-arrangement has taken place; this is not reflected here.

The network consisted of subnet 1, which connected the staff offices and the VAX cluster, subnet 2, which held the computer centre Netware servers and subnet 3 which had the Unix hosts. There was also a subnet 9 which served the terminal room containing PCs with Netware and Macintoshes.

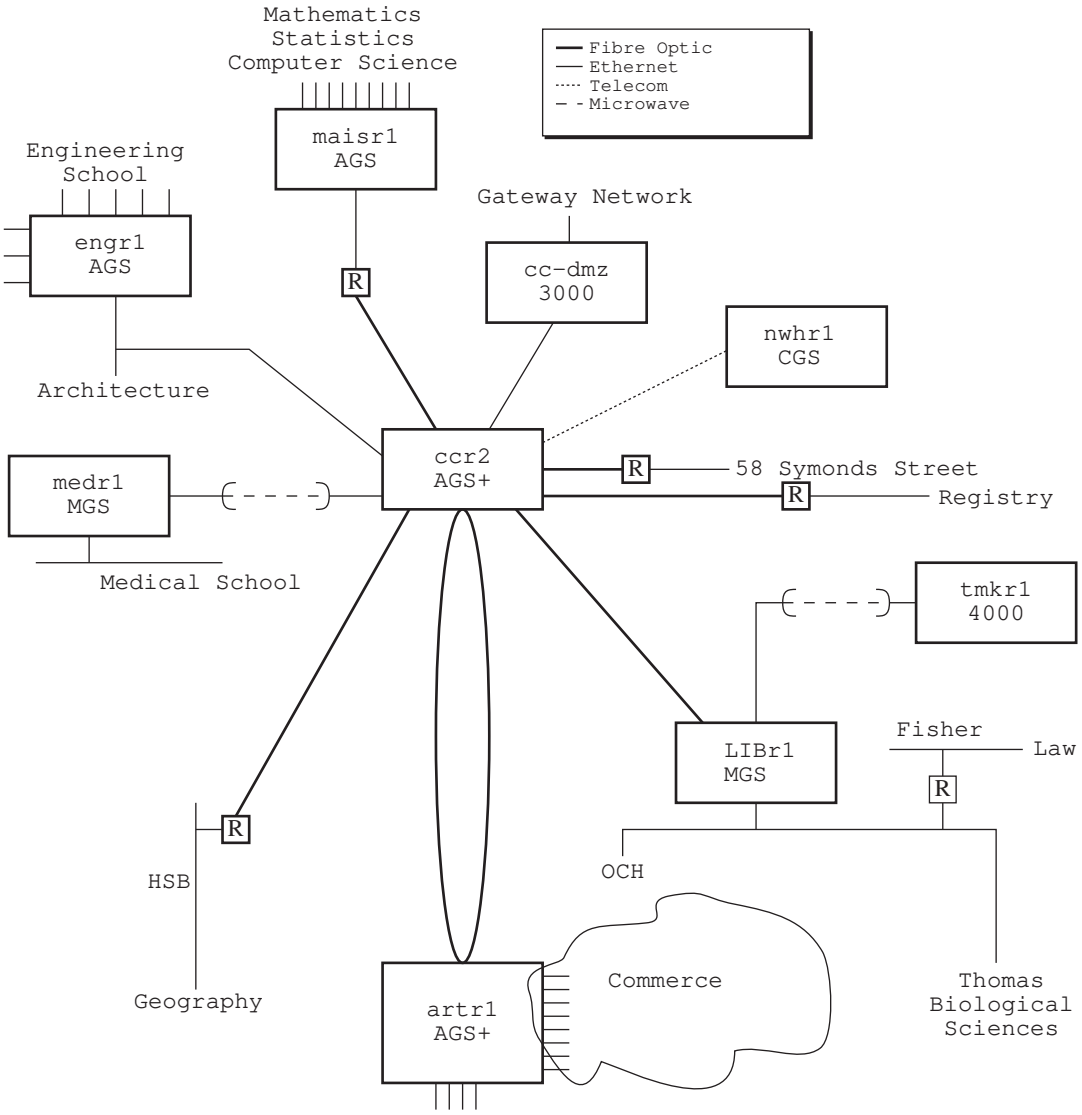

Figure 1.3: Topological map of the University of Auckland Network

1.4.2 Gateway network

The gateway network, commonly known as the DMZ (Demilitarized Zone) network, is used to separate the University network, the Auckland local networks connected though common carrier dialup and leased lines, and the frame relay connection to the rest of New Zealand and the world.

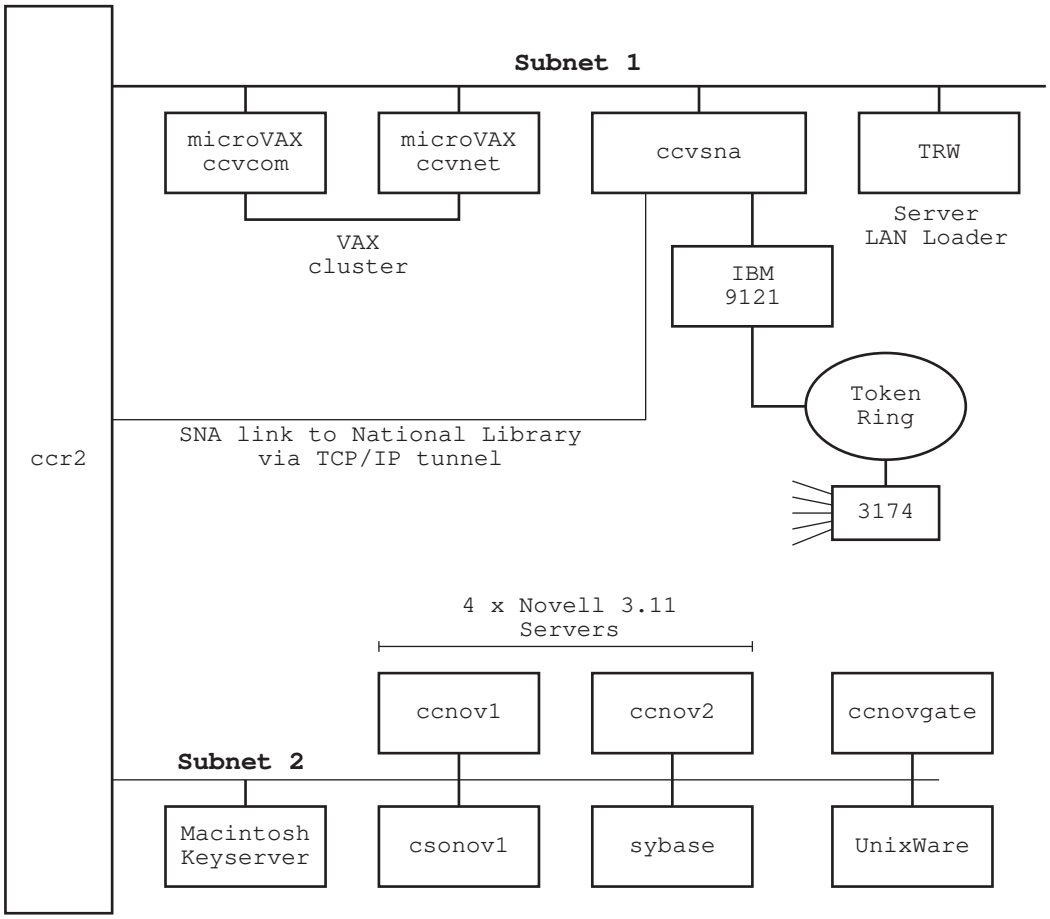

Figure 1.4: Topological map of part of the Computer Centre Network

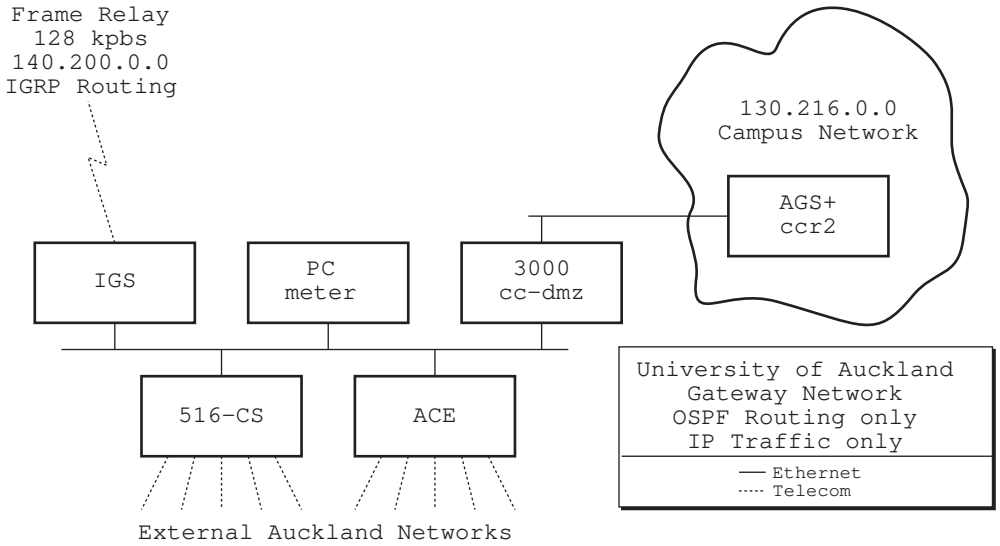

Figure 1.5: Topological map of the University Gateway Network

Chapter 2

Network tracing

A network (or packet) *trace* is a record of activity of a physical network over a period of time. A network trace can contain information about the length of each packet, the time it was seen and part or all of the actual packet contents.

The emphasis of this thesis is traffic intensity with respect to time. For this packet inter-arrival times are needed. This is done by recording a time-stamp in the trace. The more accurate the time-stamp the finer the analysis that can be performed.

For packet size analysis the length of the packet has to be recorded. Because packet sizes are a function of higher level protocols, for any in-depth analysis the origin of the packet has to be obtained. This can be done by examining the contents of each packet and extracting the relevant information. This is a time consuming task so it cannot be done during the execution of the tracing program. Instead the first forty to fifty bytes of each packet is recorded for later processing.

Packet traces are normally stored as large binary files. Programs can then examine these files and perform statistical analysis on the data. Recording a trace over any reasonable length of time produces large files, in the order of megabytes or tens of megabytes.

2.1 Producing a packet trace

The packet traces of Ethernet segment were collected using an IBM PC directly connected to the Ethernet. Because of the broadcast nature of Ethernet it is easy to record all packets sent on the network. The program records a time-stamp, the packet size and the first n bytes of the packet, where n is defined at compile time.

The number of packets the program records is set at compile time, and normally ranges from ten thousand to half a million. In a later version of the program it should be possible to specify the number of minutes the program is to run.

2.1.1 Format of the trace file

The trace file has an eight byte header containing four records, each a two byte big endian word. The contain the length of the time stamp (in octets), the length of the packet size, the number of octets of packet header and an unused word respectively (see Figure 2.1).

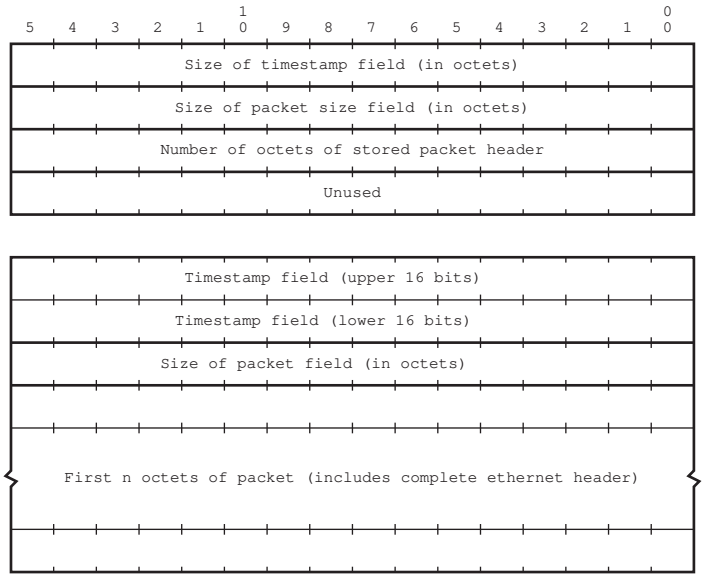

Figure 2.1: Format of the trace file

Following the header are the actual trace records, which contain the time stamp, the size of the packet (this includes the Ethernet header but not the frame checksum) and the first n octets of the packet.

2.1.2 Hardware used to produce the traces

The hardware used was a standard IBM PC compatible with an Ethernet controller. This was used because it was easy to write the software needed to produce the trace and because the hardware is widely, and cheaply, available. The IBM PC architecture does have several drawbacks.

- The clock used to generate the time stamps is limited in accuracy to approximately 1 ms.
- The input/output bus is limited to transfers of about 700 kilobytes per second. Very long bursts could be a problem.
- Interrupt events block other interrupts, causing problems with the timer.

The Ethernet controller has an eight kilobyte buffer so as to reduce the actual packet loses. This may affect the analysis as queued packets will be seen by the program as having a fixed inter arrival time.

2.1.3 Software used to generate the trace file

The software is written in C and then compiled. It is based on *packet driver* software. Packet drivers are pieces of code which isolate the programmer from the specific details of each

hardware controller, presenting a standard interface regardless of the actual controller. This enables programmers to write much simpler programs at the expense of a small extra overhead.

When a packet arrives at the Ethernet controller it generates a signal to interrupt the processor and process the packet. The packet driver, while in interrupt mode (that is, the processor has suspended what it was doing and is executing a special routine called an interrupt handler) calls a user program with the size of the packet. This returns a pointer into memory for the contents of the packet to be copied (this includes the packet header). The driver copies the packet and calls a second user program routine to tell it that the copy has been completed. Once the second routine has exited, the packet driver then returns from interrupt mode, and the processor restarts what it was originally doing.

The problem with interrupt mode is that code executing while in interrupt mode cannot itself be interrupted (this is not strictly correct but is true for packet drivers). Interrupts which occur while this happens are placed in a priority queue. For this reason interrupt handlers must do as little work as possible. In general programs just set flags and update simple data structures during interrupt mode.

The program itself loops repeatedly checking these flags for change. If a flag is set it then processes the packet from the head of the data structure. This looping is called *polling*.

2.1.3.1 Data structures used

The data structure consists of two fixed buffers. While one is being used the other is either waiting to be written out to a file or is empty. Each buffer has a fixed number of records into which packet headers can be written.

The *Output File* is a standard C file pointer and is defined as part of the C *stdio* (standard input/output) library. This enables the program to be portable to different operating systems.

There are also variables such as *Initial Time* and *Total Packet Count* for determining when the program should terminate.

2.1.3.2 Flow of polling loop

The program has three stages. The first initialises the required variables and data structures, including the timer and packet driver. The second is a repeated loop which continually checks to see if a buffer needs to be written out to a file and the last is the final clean up before exiting (Figure 2.2).

There are various reasons for the program to terminate. In the normal course of events either the number of packets recorded reaches a set amount or a running of the program exceeds a certain time limit. Both these values can be specified when the program is loaded into memory. If the writing to the file should fail for some reason then the program will also terminate. Generally this will be because the disk is full but other errors can occur.

2.1.3.3 Flow during interrupt

When an interrupt is received by the packet driver it in turn calls the user with the size of the packet and expects a pointer into memory as a return result. It also stores the current timer value and the size of the packet into the current record (Figure 2.3).

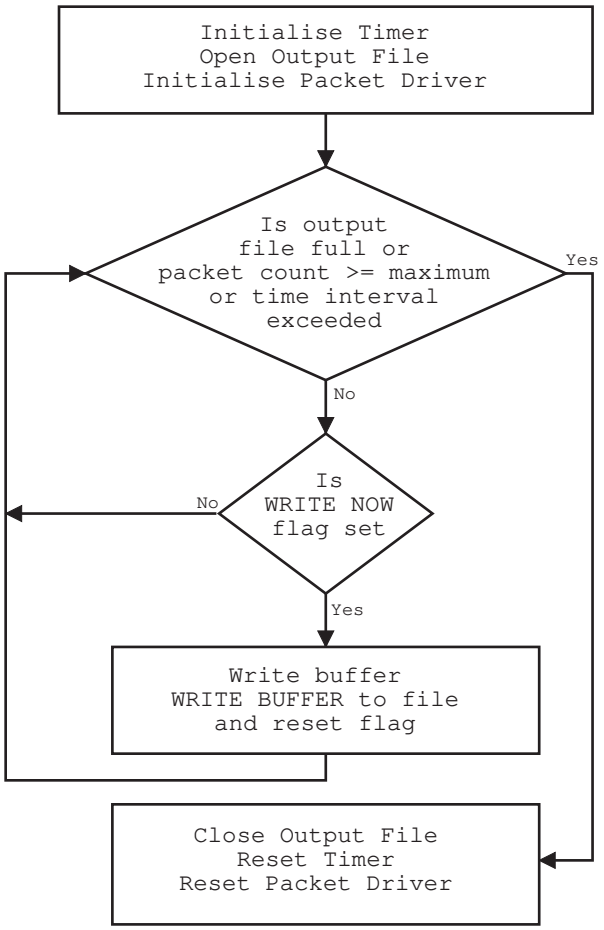

Figure 2.2: Flow of polling loop

The packet driver then copies the contents of the packet into the specified memory and calls a second user routine. This simply updates various counters and exits without any return value (Figure 2.4).

2.1.4 Known problems

2.1.4.1 Errors with the timer

The timer works by having an internal 16 bit hardware counter, say c_0. This starts at 0xFFFF (65535), counts down to 0x0000, then resets itself automatically and starts counting down again. When it reaches 0x0002 it causes an interrupt and a counter t kept by the timer code is increased. The counter is split into upper and lower bytes, c_u and c_l respectively. The value returned from the timer is

$$2^8 t + (256 - c_u)$$

which should theoretically always give the correct result.

The 1.19318 MHz clock generates the clock signals so that the standard PC BIOS time is accurate to $\sim$ 18.2 Hz or 54.9 ms. We use c_u to give us a rate of 4.66 kHz or 0.2 ms.

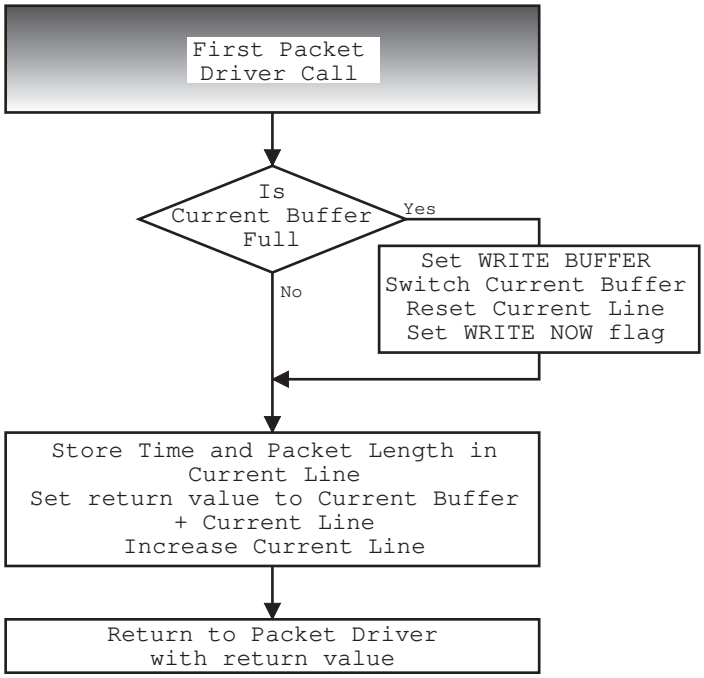

Figure 2.3: Flow for first packet driver call

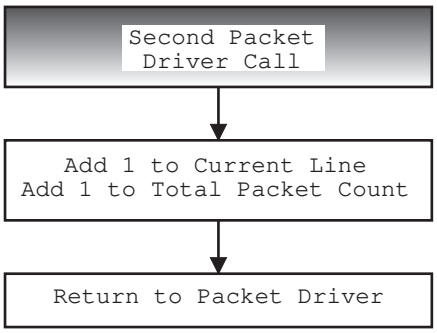

Figure 2.4: Flow for second packet driver call

Because the timer is asynchronous to the main processor, if t should fail to be increased at the correct time a problem may occur. With the Intel 80x86 processor a flag allows or blocks interrupts from occurring. If this flag is set then interrupts get queued, waiting until the flag is cleared so they can occur.

If a packet arrives when c is close to zero then an interrupt occurs and the packet driver is called. The packet driver sets the interrupt flag (since another packet may be received while the driver is still processing the first and if it was allowed to interrupt again the code would probably crash). While the driver is processing the interrupt the timer reaches 0x0002, causes an interrupt, which is queued, counts a little more and resets itself.

The tracing code then reads the value from the timer code, which is now incorrect, since t has not yet been incremented, and stores it within the current buffer. Afterwards the packet driver clears the interrupt flags and exits, the timer interrupt occurs, t is incremented to the correct value, and the timer value is what it should be.

The problem occurs because of a limitation of the Intel 80x86 architecture and cannot be avoided without considerable modification of the packet driver code to stop it from having the interrupt flag set while calling the user program code. It is unclear whether this is even feasible.

Because this problem is known it can be corrected in later processing without much difficulty, so rather than try to fix it at its origin it is simpler to make the corrections.

2.1.4.2 Machine crashes after program exits

After running the tracing program it is necessary to reboot the machine it ran on. This is more an inconvenience rather than a problem. The reason for the problem is unknown but the timer code is the most suspect. The amount of time needed to fix this problem is not worthwhile and since it does not affect the results it can safely be ignored.

2.2 Testing the packet tracer

2.2.1 Introduction

Having written this program a series of controlled experiments were performed to check the validity and usefulness of the program. Below are some of the requirements of interest.

- At what rate of packet arrival does packet loss occur?
- Does the size of the arriving packets affect packet loss?
- In what form does packet loss happen?
- How is packet loss affected by writing to permanent storage?

2.2.2 Design

This experiment was performed using two machines connected via an isolated segment of Ethernet. On one machine a packet generator executes while the other records all packets seen on the segment and writes them out to permanent storage.

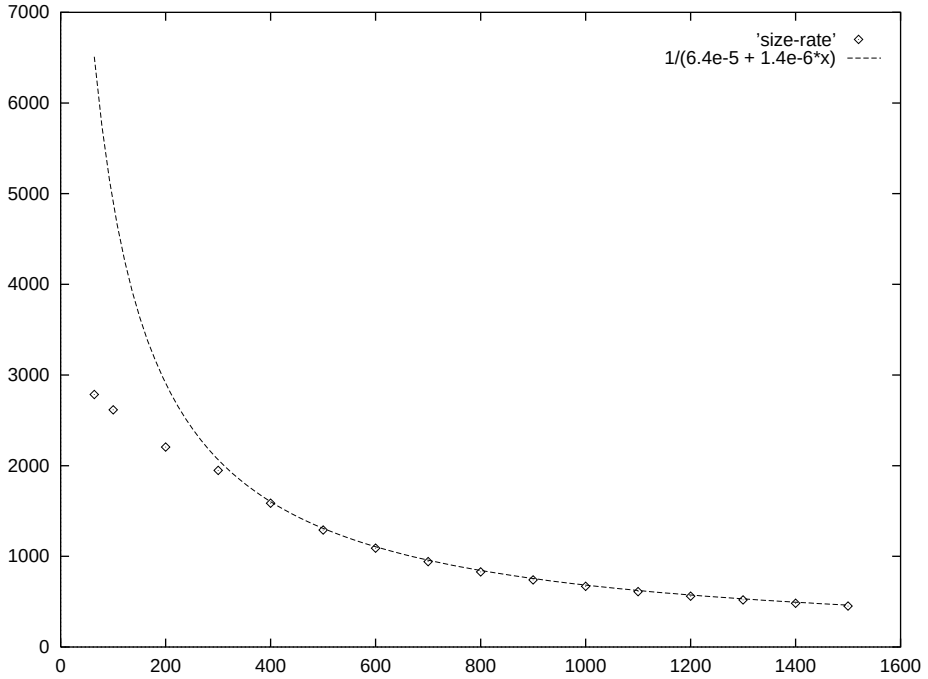

Figure 2.5: Output Rate versus Packet Size

One important note is the relative processing power of each machine. While both machines have compatible architectures one is at least twice as fast as the other. This asymmetry enables the recording program to be "tested to destruction" while the packet generating machine remains stable.

The experiment was carried out by subjecting the recording machine to various traffic loads (as specified by the generator's input parameters) and examining the resulting output file.

2.2.3 The packet generator

The packet generator generates c packets of fixed size s octets. Between the start of each packet is a delay d ticks where a tick is $\sim$ 0.2 ms. If the time required to send a packet is greater than the specified delay then the packets are sent end to end without any artificial delay. These three parameters can be specified on the command line.

```
stress [-s packet size] [-c number of packets sent] [-c delay]
```

$stress$ uses packet drivers for it communication with the Ethernet hardware. The code consists of a single loop which continually sends out packets using the Ethernet broadcast address. The current sequence number and time follow the address within the packet.

2.2.4 Experiment - determining the maximum output rate

To examine the behaviour of the packet generator a series of packet generations was carried out from the slower to the faster machine. The experiment looked at the maximum rate (in

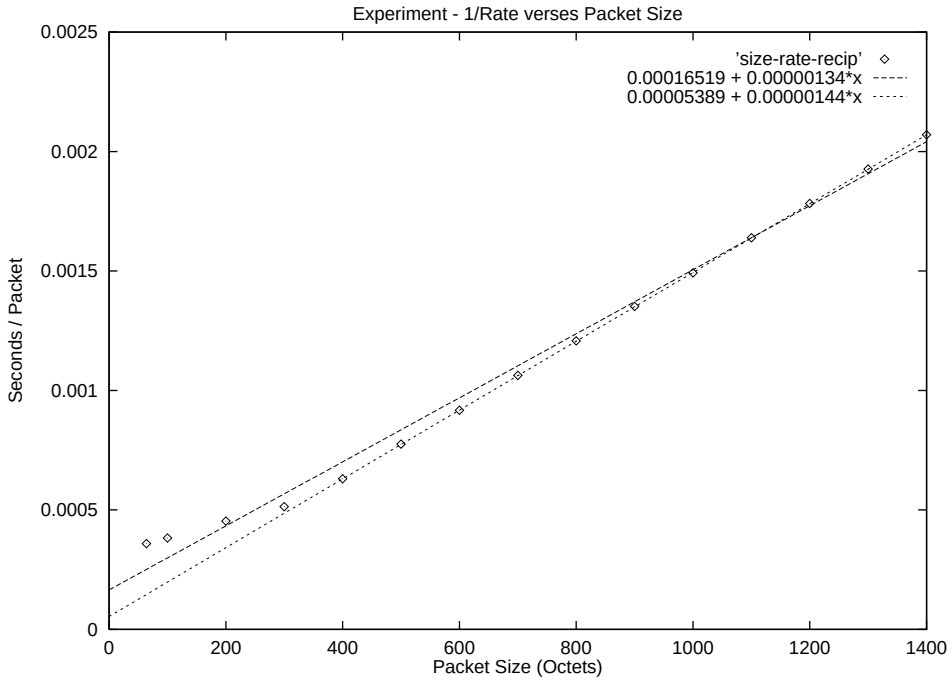

Figure 2.6: 1 / Rate versus Packet Size

Packet Size (octets)	Packets per second	Octets per second
0064	2785	178240
0100	2616	261600
0200	2206	441200
0300	1948	584400
0400	1586	634400
0500	1290	645000
0600	1090	654000
0700	0941	658700
0800	0828	662400
0900	0740	666000
1000	0670	670000
1100	0610	671000
1200	0561	673200
1300	0519	674700
1400	0483	676200
1500	0451	676500

Table 2.1: Results for maximum rate experiment

packets per second) the generator could produce given a specific packet size. The result can be seen in figures 2.5 and 2.6, and table 2.1.

These were produced by sending 10,000 packets back to back without delay between the two machines. The trace file is then analysed and the number of packets per second is extracted using `arrival`.

The curve on figure 2.5 was produced by taking the resulting values (table 2.1, ignoring the first four values, taking the reciprocal and fitting a linear line to it. In figure 2.6 you can see both line fits, one including all the points and the other excluding the first four points. This linear fit was then inverted and plotted against the original points.

The results suggest that beyond packet sizes of 400 octets the generator's output is bounded by the I/O (Input / Output) of the machine, at about 710,000 octets/second or 693 K per second. Packet sizes smaller than this are bounded by the processing time needed to send each packet so that the curve represents the maximum theoretical output in packets per second.

Note that 710,000 octets/second is less than Ethernet's theoretical throughput of around 1,200,000 octets/second so that on a faster machine with greater I/O bandwidth more than 700 K per second is easily achievable.

2.2.5 Summary of packet trace test

- No losses were observed in the experiment, and th packet rates used there are higher than those observed on the Campus network.
- In the experiment there was on evidence of packet loss during writes of the trace data to disk storage.

Overall the trace program performed very well, providing a simple and effective tool for collecting network traffic data.

Chapter 3

Stochastic Models of Packet Arrivals

To simulate packet arrivals we must first define models of packet arrivals. Most of the early work in this area used the Poisson process [6] as the basis of its models. Many more complex models have been used but many rely on the Poisson process somewhere for the packet arrival distribution. The Poisson process has the advantage of being well understood and mathematically tractable.

The Poisson process is related to the exponential distribution (§A.3.1). This distribution is one of the most widely used in statistics and the basis for several others. One feature of the distribution is it has a finite mean and variance (§A.1.2). The exponential distribution has the property that the time spent waiting for an event (that will occur within a time that is exponentially distributed) will not give you any information about when it will occur. This is known as the *memoryless* property of the exponential (and its discrete counterpart the geometric) distribution.

One important area of interest are processes which have inter-renewal times with infinite moments. We are mainly interested in those distributions with infinite variance such as Cauchy, t_2-distribution and Pareto.

The research into infinite variance distribution renewal processes has led to the idea of *fractal*, or *self similar* renewal processes. The basis of these names has to do with similar behaviour at different time scales. Poisson processes become smoother as you observe their average behaviour over longer time intervals. Pure fractal processes do not smoothe so that they are invariant with respect to the time period over which you observe them.

Such behaviour has serious consequences when applied to traffic models because it indicates that you will always get bursts of traffic at every time scale and you can never allocate enough queueing resources to cope with every situation.

Much of the original work into fractals was done by Mandelbrot [13] but it is only recently that his work has been applied to communications traffic by researchers at Bellcore [1],[9], [8], [10].

The use of finite variance distributions was challenged in [1]. In this thesis we examine the behaviour of Poisson process based models and demonstrate that they are inadequate for modelling network traffic observed on the University of Auckland campus network.

The researchers at Bellcore produced traces for Ethernet LAN traffic, ISDN packet traffic over public switched telephone systems and the output of variable bit rate video encoders.

The earlier chapters of this thesis attempted to duplicate their recording of Ethernet traffic and produce comparable results as confirmation of their work.

The Bellcore work then went on to examine many of the self-similar properties of the traffic and its relationship to theoretical work in fractal behaviour. In this thesis we simulate various statistical models to see whether the observed behaviour can be reproduced.

3.1 Poisson process

3.1.1 Definition

A Poisson process is a counting process

$$N = \{N(t) : t \geq 0\}$$

taking values from

$$S = \{0, 1, 2, 3, \ldots\}$$

conditioned on

$$N(0) = 0$$

and

$$\text{if } s < t \text{ then } N(s) \leq N(t)\ \forall s, t \in \mathbb{R}$$

defined by

$$P(N(t+h) = n + m | N(t) = n) =$$
$$P(N(t+h) - N(t) = m) =$$
$$\begin{cases} 1 - \lambda h + o(h) & m = 0 \\ \lambda h + o(h) & m = 1 \\ o(h) & m > 1 \end{cases}$$

3.1.2 Remarks

The Poisson Process is one of the most common and simple stochastic processes in statistics and operations research. Poisson processes occur in both time and space and are characterised by a single parameter λ. This parameter λ is a measure of the rate or intensity of the process.

Poisson processes in time occur along a real valued time line, normally $\mathbb{R}^+$. They create single events along the line as they advance towards $+\infty$.

The above definition has the following consequences.

- No event may occur at exactly time zero.
- The numbers of events in two non-overlapping time periods are independent. For example knowing the number of events in the period $(0, t]$ will not give you any extra information about the number of events in the period $(t, t+s]$, that is $N(t+s) - N(t) \sim N(s)\ \forall s, t \in \mathbb{R}^+$.
- The number of events in the period $(0, t]$ is distributed as a Poisson(λt) random variable, that is $P(N(t) = x) = \frac{e^{-\lambda t}\lambda t^x}{x!}$, independently of all other inter-event times.

It can easily be shown that these give us the following.

- The time between any two successive events is distributed Exponential(λ), that is $P(T_i - T_{i-1} \geq t) = e^{-\lambda t}$.

3.2 Renewal processes

A renewal process can be seen as a generalisation of a Poisson process. The distribution of the times between renewal events are independent and identically distributed random variables. For example the Poisson process is a renewal process because the inter-arrival times are independent and identically distributed (as exponentially distributed random variables). Renewal processes periodically restart themselves at the renewal points, forgetting their previous history.

The behaviour between renewal events may be complex. Often the renewal event is a marked event within a more complex process. An example is the simple random walk where the return to the origin or return to zero can be used as the renewal event. This is to maintain consistency with the concept that when a renewal occurs it is as if the process were starting from time zero.

3.2.1 Simple renewal processes

3.2.2 Definition

Let

$$T = \{T(k) : k \geq 0 \; k \in \mathbb{N}\}$$

where $T(k)$ is the time of the k^{th} event, conditioned on

$$T(0) = 0$$

Let

$$R_k = T(k) - T(k-1)$$
$$R_k > 0 \;\; k \geq 1$$

Then this is a renewal process if there exists a distribution function $F_R(x)$ such that

$$P(R_k \leq x) = P(R \leq x) = F_R(x) \; \forall x > 0 \text{ and } \forall k \geq 1$$

Furthermore, the R_k are mutually independent of one another. That is, inter-event times are independent and identically distributed.

3.2.3 Remarks

In simple renewal processes the process restarts every time an event occurs. This is the simplest type of renewal process and is very similar to the Poisson process. The main difference is that the Poisson process relies on events having an exponentially distributed inter-arrival time. In a renewal process any probability distribution may be used for the time until the next renewal.

Because the time until the next renewal has to be non negative many common distributions such as the normal or Cauchy have to be modified. This can be done by folding the negative half of the density onto the positive real line. The simplest method is sampling from the distribution and then taking the absolute value.

3.2.4 Processes with internal structure

Many processes are more complex then single events. Between renewals the process may produce other events. A M/M/1 queue is a simple queueing process where the time between arrivals is exponentially distributed and the time taken for the customer at the head of the queue to be served and leave is also exponentially distributed. We can make the renewal event the event of the system becoming empty. The times when the system becomes empty form a renewal process with the queueing process becoming hidden.

3.3 Modulated processes

Modulated processes are multi-state stochastic processes. They consist of individual states and a set of stochastic transitions which govern the process' change from one state to another.

The process is in exactly one of the states at any given time. While it is in a state it acts in accordance with the behaviour that state specifies. The process may change state at any time and must enter another state straight away. The rules governing how long the process stays in a state and which states can follow it may be state dependent or be controlled by an underlying global process.

The difference between modulated processes and renewal process is that each state in the modulated process has an associated renewal process which produces events. When a new state is entered a new renewal process corresponding to that state is created rather than each renewal process continuing to run but being hidden, producing events which are then discarded.

For Markov modulated Poisson processes there is no difference because of the memoryless property of the exponential distribution but for more general processes this distinction is important. This is due to the *edge effects* created by the renewal process creation at state changes. If the lifetimes for the states are short causing rapid state changing then these edge effects may seriously affect the overall behaviour of the process.

3.3.1 Markov modulated Poisson processes

3.3.1.1 Definition

Let

$$\mathcal{S} = \{s : s = 0, 1, 2, 3, \ldots, N\}$$

be a set of states with associated arrival rates λ_s, where λ_s is the arrival rate of the underlying Poisson process while in state s. Let

$$N = \{N(t) : t \geq 0\}$$

be the underlying Poisson process of rate $\lambda_{S(t)}$. Let

$$S = \{S(t) : S(t) = s \in \mathcal{S} \ \ \forall t \geq 0 \ \ t \in \mathbb{R}\}$$

be the state of the process at time t. Let

$$P(S(t+\delta) = k, \ 0 < \delta \leq h | S(t) = k) = e^{-\mu_k h}$$

where $\mu_k \in \mathbb{R}^+$ are state lifetime parameters, with

$$P(S(t+h) = k | S(t) = j, h = \inf\{S(t+\delta) \neq j \ , \delta > 0\}) = M[j,k] \ \forall j,k \in \mathcal{S},$$
$$M = M[m,n]$$

is a stochastic $|\mathcal{S}| \times |\mathcal{S}|$ matrix such that

$$\sum_n M[m,n] = 1 \ \ \forall m \in \mathcal{S}$$

3.3.1.2 Remarks

Markov modulated Poisson processes are the simplest of the modulated processes. Each state defines the arrival rate of an underlying Poisson process.

The amount of time the process stays in a state is exponentially distributed with a stochastic transition matrix governing the next state entered.

Markov modulated Poisson processes are often used to generate the arrivals for standard queueing models. This is so that more variability is introduced into the model and to add extra realism rather than having a constant arrival rate.

3.3.2 Generalised modulated processes

3.3.2.1 Definition

Let

$$\mathcal{S} = \{s : s = 0, 1, 2, 3, \ldots, N\}$$

be a set of state with associated distribution functions

$$L_s : (0, +\infty) \to [0, 1] \ s \in \mathcal{S}$$

Let

$$S = \{S(t) : S(t) = s \in \mathcal{S} \ \ \forall t \geq 0 \ \ t \in \mathbb{R}\}$$

be the state of the process at time t, $c(t)$ be the number of state changes by time t, $\sigma \in \mathcal{S}$ be a starting state $S(0) = \sigma$ and τ_n be the time of the n^{th} change of state.

$$\begin{aligned} \tau_0 &= 0 \\ \tau_{n+1} &= \inf\{t : t > \tau_n, S(t) \neq S(\tau_n)\} \end{aligned}$$

$$c(t) = n \quad \text{iff} \quad \tau_n \leq t \text{ and } \tau_{n+1} > t$$

Let

$$N_i = \{N_i(t)|N_i(0) = 0\}$$

be a collection of simple renewal processes where $N_i(t) = k$ if k events have occurred by time $(0, t]$ and $i \in \mathcal{S}$. These are template renewal processes.

Let

$$P(\tau_{\alpha+1} - \tau_\alpha > h) = 1 - L_{S(\tau_\alpha)}(h)$$

with

$$P(S(\tau_{\alpha+1}) = k \mid S(\tau_\alpha) = j) = M[j, k] \; \forall j, k \in \mathcal{S}$$

where

$$M = M[m, n] \;\; 0 \leq m, n \leq |\mathcal{S}|$$

is a stochastic matrix such that

$$\sum_n M[m, n] = 1 \;\; m \in \mathcal{S}.$$

Let the current renewal process at time t be

$$Q_\alpha = \{Q_\alpha(t) \mid Q_\alpha(\tau_\alpha) = 0\} \text{ where } \tau_\alpha \leq t < \tau_{\alpha+1}$$

where

$$P(Q_\alpha(t) = k) = P(N_{S(t)}(t - \tau_\alpha) = k)$$

so that the total number of events by time t is given by

$$R(t) = \sum_{i=0}^{c(t)-1} Q_i(\tau_{i+1}) + Q_{c(t)}(t)$$

3.3.2.2 Remarks

A generalised modulated process consists of a set of states and a stochastic transition matrix but unlike the Markov modulated Poisson process it is not limited to only using the exponential distribution as a lifetime distribution.

Each state has two distributions associated with it. One is a renewal process which generates events and the other is a lifetime distribution. When the process enters the state a lifetime is generated from the lifetime distribution. The process will remain in that state for the period of the lifetime.

While in that state a renewal process will run generating events. If an event is generated which would have occured beyond the lifetime of the state then it is ignored. Any generalised renewal process is able to run provided it is independent of all other states.

Once the lifetime of a state has expired then the process chooses a new state to enter based on the probabilities in the transition matrix. The transition matrix is constant and independent of any activity within any of the states.

3.4 Merged Processes

3.4.1 Definition

Let

$$\mathcal{C} = \{c : c = 0, 1, 2, 3, \ldots, N\}$$

be a set with associated general renewal processes

$$R_c(t),\ c \in \mathcal{C}$$

Let

$$T_c = \langle T_{c_1}, T_{c_2}, \ldots \rangle$$

be the sequence of renewal times of the process R_c.

Let

$$T = \langle T_1, T_2, T_3, \ldots \rangle = \bigcup^{\mathcal{C}} T_c$$

be an ordered sequence such that

$$T_i \leq T_j\ \forall i, k \in \mathbb{N}$$

then T is a merged process.

3.4.2 Remarks

A merged process consists of multiple renewal or modulated processes running concurrently with their resulting event traces merged together. In most models each process is identical and all must be mutually independent.

The idea behind merged (or superimposed) processes is to capture complex behaviour using simply renewal processes with few parameters. This would give us a much simpler model without losing the complex nature of the real behaviour.

3.5 Slowly decaying variances

3.5.1 Processes with infinite moments

There are several well known distributions with infinite moments. The t-distribution has infinite variance for $\nu = 2$ and infinite mean and variance for $\nu = 1$ (this is also known as the Cauchy distribution). Another is the Pareto distribution. This is a less common distribution and is not often used in the area of stochastic process modelling.

3.5.1.1 Definition of the Pareto distribution

$$F_X(x) = P(X \leq x) = 1 - \frac{\beta^\alpha}{(\beta + x)^\alpha}$$

and

$$f_X(x) = \frac{\alpha\beta}{(\beta + x)^{\alpha+1}}$$

so that $\alpha = 2.5$ with give a distribution with finite mean and variance, $\alpha = 1.5$ will give infinite variance and $\alpha = 0.5$ will have both infinite mean and variance.

3.5.2 Aggregating stochastic processes

One of the methods for testing whether a sample has been generated from a Poisson process is to see how the variance behaves under averaged aggregation [5].

Let

$$X = (X_1, X_2, X_3, \ldots)$$

be a stationary stochastic process (for us X_k will denote the number of events per time unit). For each $m = 1, 2, 3, \ldots$ let

$$X^{(m)} = (X_k^{(m)} : k = 1, 2, 3, \ldots)$$

denote a new (aggregated) time series obtained by averaging the original series X over non-overlapping blocks of size m. That is for $m = 1, 2, 3, \ldots$, $X^{(m)}$ is given by

$$X_k^{(m)} = \frac{1}{m}(X_{km-m+1} + \cdots + X_{km}) \ (k = 1, 2, 3, \ldots) \tag{3.1}$$

Note that $X^{(m)}$ defines a new stationary process for each m.

Consider any collection of random variables which have the following properties

- All variables are independent and identically distributed.
- Their probability distribution have finite mean μ and variance $\sigma^2 \in \mathbb{R}^+$.

Let

$$X = (X_1, X_2, \ldots, X_n)$$

and

$$\bar{X} = \sum_{i=1}^{n} \frac{X_i}{n} = \frac{1}{n} \sum_{i=1}^{n} X_i$$

then

$$\mu_{\bar{X}} = \mathrm{E}(\bar{X}) = \mathrm{E}(\frac{1}{n} \sum_{i=1}^{n} X_i) = \frac{1}{n} \sum_{i=1}^{n} \mathrm{E}(X_i) = \mu$$

and

$$\sigma^2_{\bar{X}} = \mathrm{Var}(\bar{X}) = \mathrm{Var}(\frac{1}{n} \sum_{i=1}^{n} X_i) = \frac{1}{n^2} \sum_{i=1}^{n} \mathrm{Var}(X_i) = \frac{\sigma^2}{n}$$

From above we can see that for aggregated renewal processes with finite mean and variance the variance is of order n^{-1} under averaging. The decreasing of variance is called *decaying variance*. So from equation 3.1 we obtain

$$\mathrm{Var}(X^{(m)}) \sim a_1 m^{-1}, \text{ as } m \to \infty \tag{3.2}$$

With *slowly decaying variance* we do not get this, instead we observe

$$\mathrm{Var}(X^{(m)}) \sim a_2 m^{-\beta}, 0 < \beta < 1 \text{ as } m \to \infty \tag{3.3}$$

For summation $n\bar{X} = \sum_{i=1}^{n} X_i$ we get $u_{n\bar{X}} = n\lambda t$ and $\sigma^2_{n\bar{X}} = n\lambda t$. This result states for the aggregated random variables both mean and variance increase linearly under summation. For processes that exhibit slowly decaying variance the result is that their variance grows faster than linearly. For a full example see § A.4.

Therefore to see the behaviour in equation 3.3 one or both of the above conditions must fail to hold. For any renewal process the random variables will always be identically distributed leaving either the distribution to have infinite variance and/or there to be some dependence between the variables.

3.6 Fixed interval counting

Let

$$T = \{T(k) : k \geq 0 \; k \in \mathbb{N}\}$$

where $T(k)$ is the time of the k^{th} event, conditioned on

$$T(0) = 0$$

Let

$$R_k = T(k) - T(k-1)$$
$$R_k > 0 \;\; k \geq 1$$

Then this is a renewal process if and only if there exists a distribution function $F_R(x)$ such that

$$P(R_k \leq x) = P(R \leq x) = F_R(x) \; \forall x > 0 \text{ and } \forall k \geq 1$$

and the R_k are mutually independent. That is, inter-event times are independent and identically distributed.

Let

$$\mathcal{I} = \{i_k : i_k = [ks, (k+1)s), k \in \mathbb{N} \cup \{0\}\}$$

be a set of non overlapping intervals of size s such that $\cup\mathcal{I} = \mathbb{R}^+$.

Let

$$\mathcal{C} = \{C_k : C_k = \{j : T(j) \in i_k\}\}$$

so that

$$c_k = |C_k|, k \in \{0, 1, 2, \ldots\}$$

be the number of events in the interval i_k, that is $(ks, (k+1)s]$.

Then

$$P(c_i = k, c_{i-1} = j) = P(c_i = k) \cdot P(c_{i-1} = j), i > 0$$

is true $\forall s$ if and only if T is a Poisson process. That is for general renewal processes c_i, c_{i-1} are not independent.

For example, if we have a renewal process with inter-renewal events distributed uniformly with mean 0.75, and a counting interval of length 1.0 then for any interval c_k where the previous interval c_{k-1} had no events must have at least 1 event in it.

This is because a uniform inter-renewal distribution with mean 0.75 has a maximum inter-renewal time of 1.5. Since the previous interval did not have any events ($c_{k-1} = 0$) in it then 1.0 of the maximum of 1.5 has already occurred. This means that event must occur within the current interval ($c_k > 0$). Hence the number of events that can occur in the current interval is dependent on the number that occurred in the previous interval, so the c_{k-1} and c_k are no independent.

Chapter 4

Experimental Results

4.1 Introduction

In this chapter the we examine four samples from around the University (See Table 4.1), taken over a period of five months during 1994. The results are displayed as graphs along with commentary about any notable features. There are also notes on how the graphs were obtained and any interesting results.

The samples have been taken from quite different environments to try to give a wide selection of network protocols and traffic conditions. Along with the results of the four samples, one of the samples has been split up by network protocol. The question of how each different protocol affects the traffic behaviour is one this thesis would like to try to answer, or at least gauge the basic similarities and differences between the protocols.

4.2 Processing trace files

Once the trace file has been recorded it is transfered to a Unix host where it is processed. The first processing done corrects the timer problem (Section 2.1.4.1). The file may then be divided up, based on the various protocols embedded in the packet, and used to generate packet frequency (by time interval) data.

Date	Location	Traffic Mix	Load	Time	Packet Count
14 May	Computer Centre Staff Network	AppleTalk, Telnet Novell client	Heavy	14 minutes	400,021
13 June	Gateway to Outside World	Various IP	Light	102 minutes	329,379
1 August	Statistics	IP, mainly NFS	Light	118 minutes	400,048
1 September	Commerce	NetBIOS from Windows NT	Medium	94 minutes	800,002

Table 4.1: Traffic Samples from around The University of Auckland

```
4649   65   11524
4650   66   12652
4651   61   12261
4652   56   10319
4653   54   12231
4654   57   10219
4655   58   10704
4656   66   12109
4657   49   10574
4658   60   11231
4659   49   10424
```

Figure 4.1: Sample output of *arrival*

4.2.1 Correcting the timer problem

The correcting of the timer problem is a simple process. Because the timer error causes a time to be out by 256, by keeping some notion of the minimum correct time we can determine whether a time-stamp is earlier than it should be. Adding 256 to the value corrects the time-stamp to the right value. This is done by the program

```
correct filename[.icr]
```

which produces a new file called *filename*`.tra`. `correct` also changes the time from ticks (~ 4648 ticks per second) to milliseconds. This greatly simplifies all calculations and removes any dependence on the PC timers.

4.2.2 Producing frequency data

The process of producing discrete time interval packet and octet counts is very simple. The program `arrival` is given the slot size in milliseconds and told whether the output should be a binary file or an ASCII text file.

```
arrival [-b binary] [-t interval time] filename[.tra]
```

The text output of `arrival` can be seen in Figure 4.1. The first column is the interval number, the second the packet count and the third the total number of octets. The binary output has the same format except that the entries are stored as 32 bit integer values. The binary file also has the total number of rows and columns stored at the end of the file as 32 bit integer values. This is to help programs which read the file.

4.2.3 Dividing trace into protocols

To provide information on individual protocols packet traces need to be separated. This is done by a program `demux` (for demultiplex), which takes each packet header in turn, analyses it and decides via a set of rules which output file to store it in.

Currently the rules provide for IP, AppleTalk, IPX and most of DECnet. As IP, IPX and AppleTalk make up the bulk of all the University's network traffic this is adequate.

4.2.4 Producing the histograms

The histograms (figures 4.6, 4.11, 4.16, ...) were produced using the following program:

```
hist [-w width] [-s smooth] filename[.tra]
```

`hist` reads in the files created by `arrival` and produces a frequency histogram table. As the input is packets per time unit (normally one second) this gives us an indication of how the traffic levels are spread. This table is stored as ASCII so it can be directly plotted. To get around the problem of too much detail an aggregation (smoothing) feature was added. This takes the histogram results and at successive intervals of size s it sums the frequency values, and outputs a single value for the interval at the average of the corresponding packets per second.

4.2.5 Producing slowly decaying variance results

To produce the slowly decay variance results the output from `arrival` has to be processed. This is done using the program

```
stat [-l logarithmic output] [-v verbose] [-m maximum]
filename[.abf] ...
```

which automatically detects whether the file is text or binary and reads it all into memory. From the input file it builds up a vector $\vec{u}$ in memory containing all the packet count values, say $size$ long. Let the command line maximum be stored as $cmdmax$.

It then loops from 1 to $\min(size/100, cmdmax)$ using the variable k. Let

$$\vec{v}_k \text{ be vectors of length } k$$

such that

$$\vec{v}_{k_l} = \sum_{i=k\cdot l}^{k\cdot(l+1)-1} \vec{u}_i/k \text{ where } l \in \{0, \ldots, size \text{ div } k - 1\} \subset \mathbb{N}$$

Let var_k be the sample variance of the vector $\vec{v}_k$. The output file *`filename`*`.sta` is the textual representation of the ordered pairs (k, var_k). If the `-l` command flag is present then another file called *`filename`*`.dat` is also produced with the ordered pairs $(\log_{10} k, \log_{10} var_k)$.

4.3 Mean and standard deviation of samples

Table 4.2 lists the means and standard deviations of the samples with different aggregation times. The aggregation can be seen as a simple non-overlapping summation (§3.5.2). The means increase linearly as expected due to the linear nature of expectation.

Taking the 10 millisecond results as a starting point, if the underlying model is a Poisson process then we would expect the standard deviations to increase by a factor of order $\sqrt{10}$ each step. Instead we see they increase much faster, giving strength to the argument that Ethernet traffic is non-Poisson in nature.

Sample	Interval	Mean	Std. Dev
Gateway Network	10s	537.50	115.33
	1s	53.70	17.48
	100ms	5.37	3.39
	10ms	0.53	0.86
Computer Centre	10s	3,008.00	1405.31
	1s	300.70	188.77
	100ms	30.08	23.37
	10ms	3.01	2.99
Statistics	10s	561.00	426.23
	1s	56.10	64.89
	100ms	5.61	9.13
	10ms	0.56	1.43
Commerce	10s	1,408.50	559.17
	1s	140.85	69.06
	100ms	14.08	9.05
	10ms	1.41	1.70

Table 4.2: Mean and Variance of Samples

4.4 Graphs of results

Below are a collection of graphs showing packet count against time (figures 4.2, 4.3, 4.7, 4.8, 4.12, 4.13, 4.17, 4.18). Although little statistical information can be gained from these graphs the wide range of different behaviours exhibited by Ethernet traffic is readily apparent.

An attempt was made to try and get a wide sample of different traffic behaviours from around the university campus. We start by examining graphs of packet counts against time.

4.4.1 Packet Count versus octet count

The packet traces produce two sets of data, namely packet counts and octet counts. Because Ethernet packets can have from 64 to 1518 bytes of data, octet counts and packets counts are not equatable quantities.

Packet sizes are not evenly distributed. In general they have a multi-modal distribution which is based on the underlying protocol stacks. Each protocol has a maximum packet size and packets will be that size unless they are the last packet in a sequence. For example, AppleTalk has a maximum of 600 bytes of data and TCP normally uses only 1024 bytes, whereas NFS and Netware will use all 1500 bytes if possible. Many of the packets are also acknowledgement packets which are only about 20 bytes long and fixed in size. IPX for example requires an acknowledgement packet for every data packet.

It was decided that packet counts were to be used rather than octet counts for simplicity. In some newer transmission technologies the packet size is fixed so the difference disappears, packets are padded with zeros if there isn't enough data to fill them. The use of packet counts rather octet counts also makes comparisons with simulations simpler.

When bursts of traffic occur it is normally because of large transfers. In this case a majority of the packets will be of maximum size so we can assume fixed maximum sized

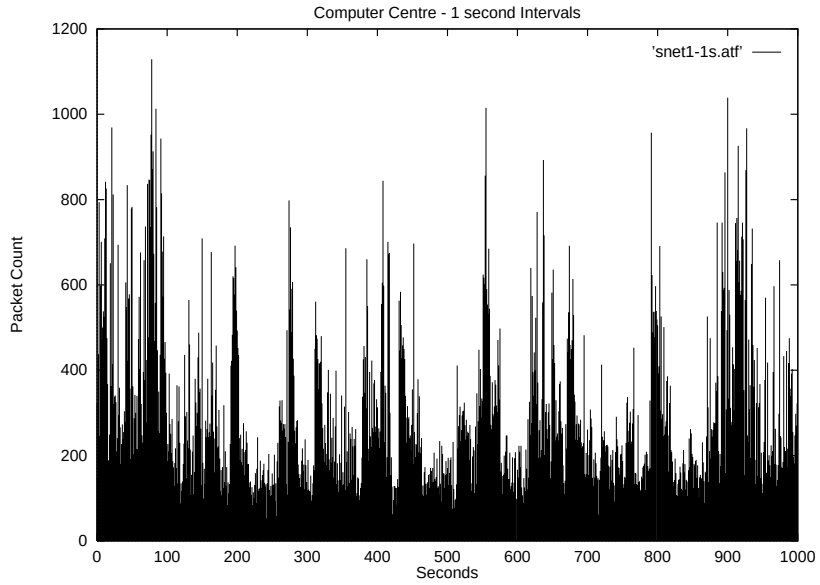

Figure 4.2: Computer Centre network with time interval 1 second

packets (at least when looking at bursts). In these cases packet counts are equivalent to octet counts.

From examing time series graphs and slowly decaying variance graphs produced for both packet counts and octets counts it was decided that their behaviour was similar enough that not using the octet count results would not be a loss.

4.4.2 Graphs by segment

4.4.2.1 University Computer Centre network

Figure 4.2 shows the 1,000 seconds of sample taken from subnet 1 of the Computer Centre staff network. This traffic has a steady background level of traffic with frequent bursts. In figure 4.4 a definite 120 second cycle can be observed. This is caused by the router sending out NetWare Service Broadcasts. The router collects information about every NetWare file server and print spooler and redistributes this information every two minutes. This amounts to about 500 Kbytes of information which is sent using back to back maximum sized packets. Figure 4.5 shows the autocorrelation of the 10 millisecond interval results. In this autocorrelation there exists an underlying positive dependency.

The histogram (figure 4.6) shows a wide range of traffic flow and that along with steady (and not insignificant) background traffic, the bursts are both large in number and size.

4.4.2.2 Department of Statistics network

The Department of Statistics staff subnetwork (also known as subnet 93) consists of about twenty Sun workstations using NIS and NFS and some Macintoshes. Apart from file and print server traffic it also carries some X-windows traffic.

Figure 4.7 shows a much more lightly loaded segment with minimal background traffic but still exhibiting very bursty traffic. The 10 millisecond autocorrelation (figure 4.10) shows

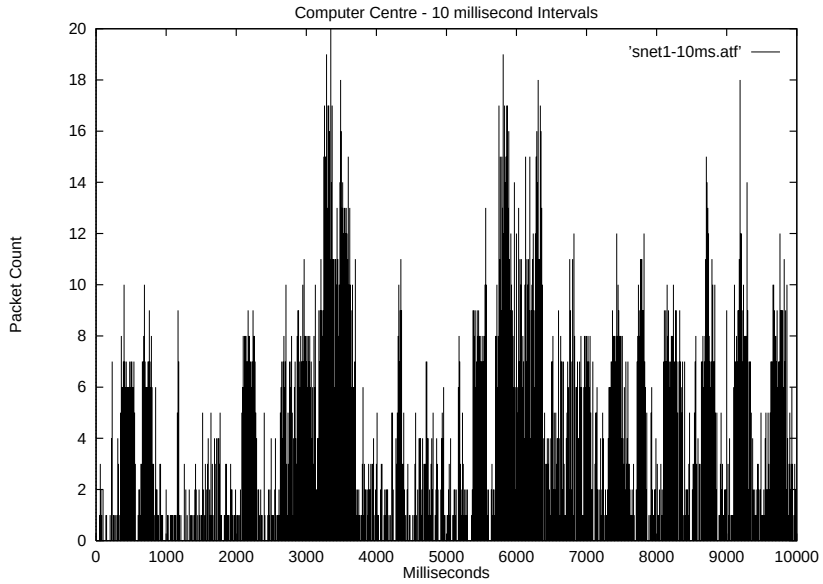

Figure 4.3: Computer Centre network with time interval 10 milliseconds

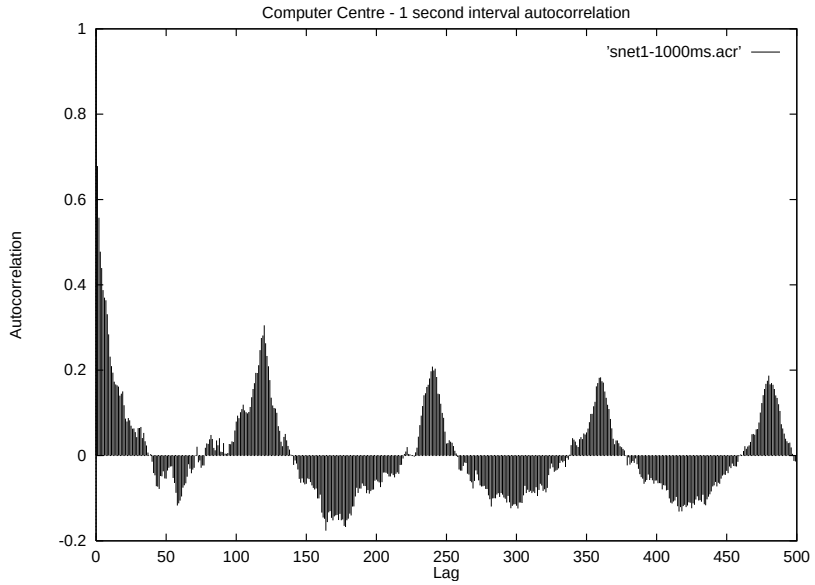

Figure 4.4: Computer Centre network autocorrelation with time interval 1 second

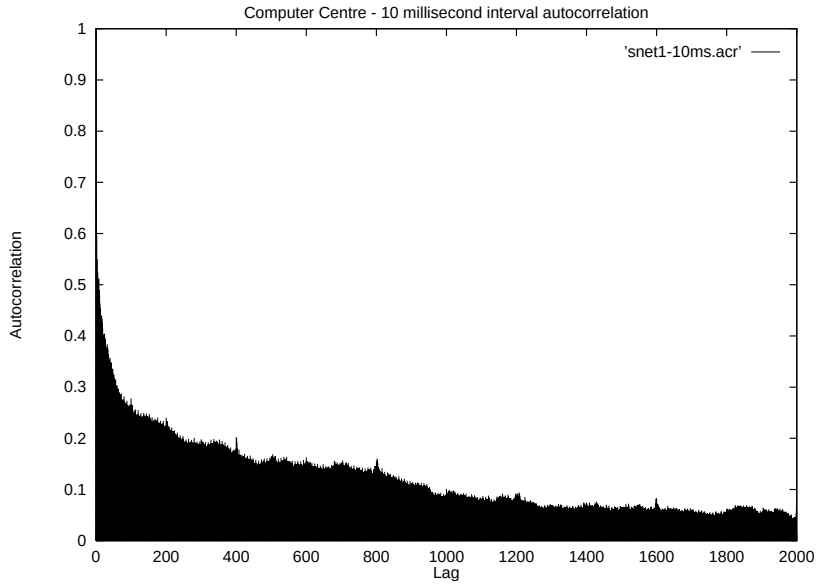

Figure 4.5: Computer Centre network autocorrelation with time interval 10 milliseconds

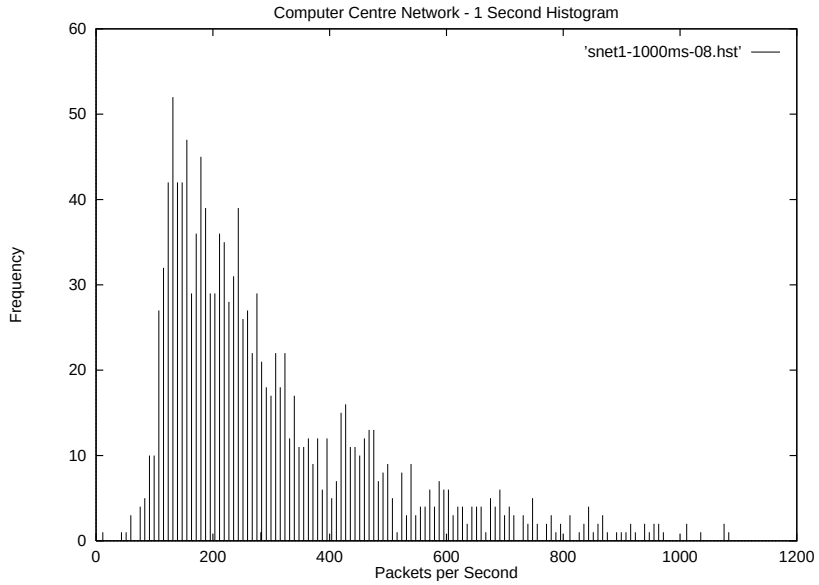

Figure 4.6: Computer Centre network histogram with time interval 1 second

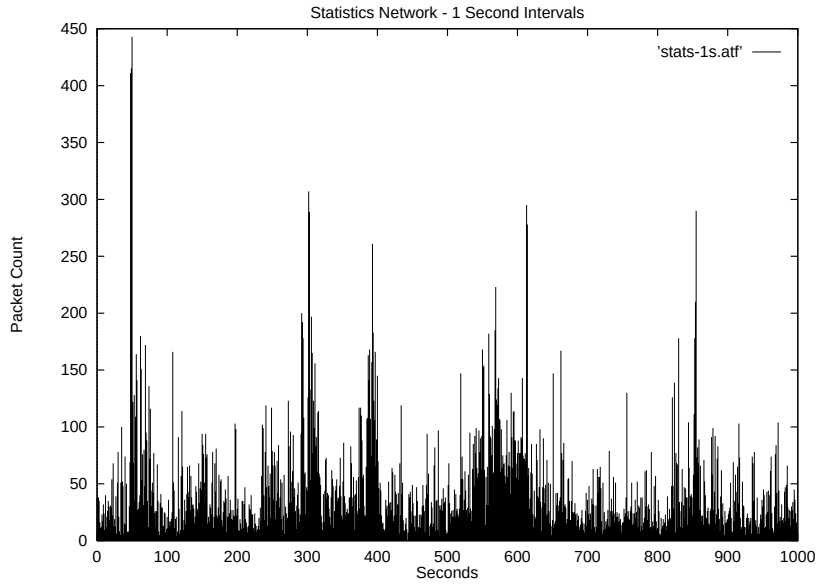

Figure 4.7: Statistics network with time interval 1 second

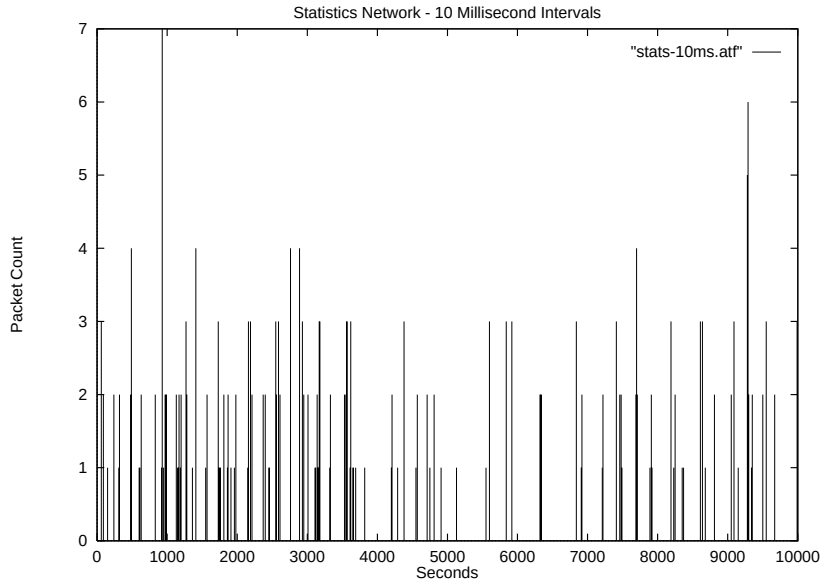

Figure 4.8: Statistics network with time interval 10 milliseconds

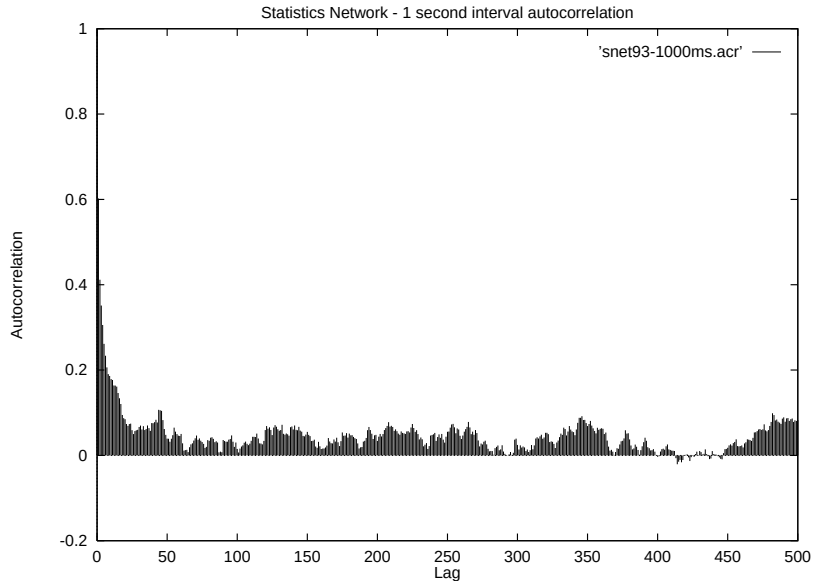

Figure 4.9: Statistics network autocorrelation with time interval 1 second

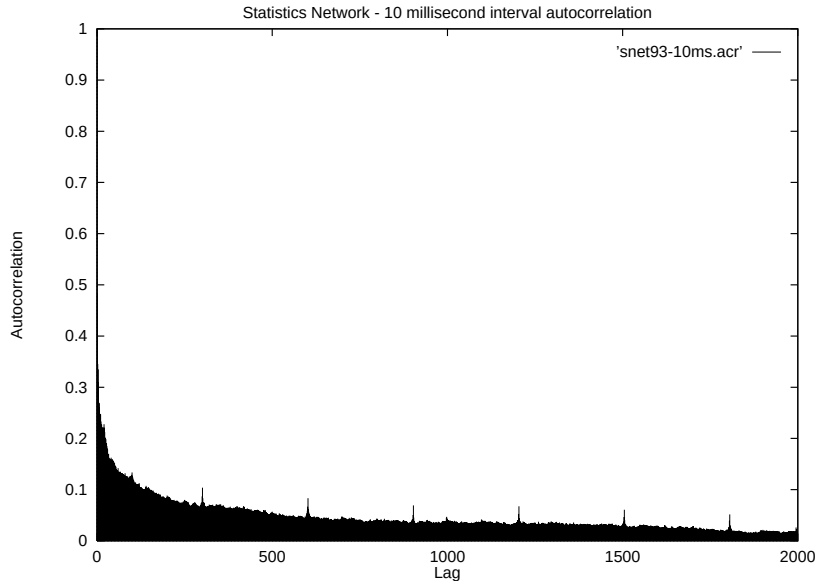

Figure 4.10: Statistics network autocorrelation with time interval 10 milliseconds

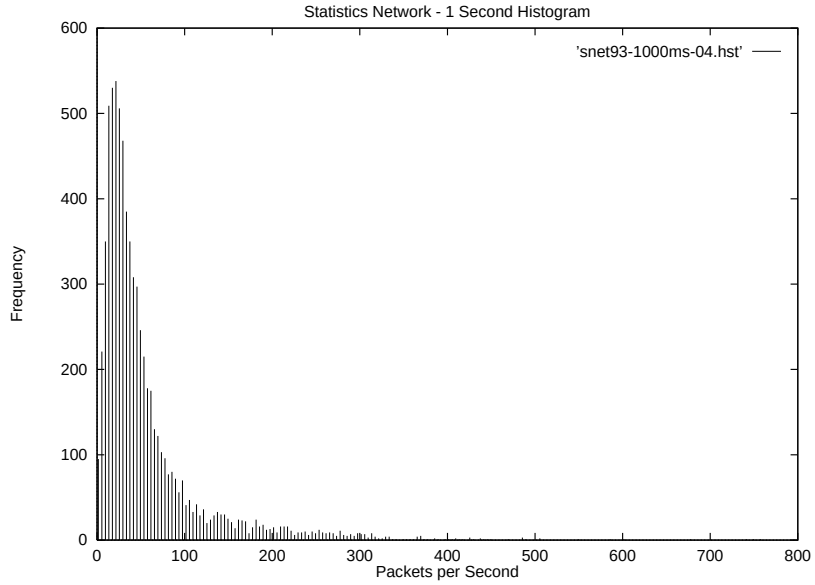

Figure 4.11: Statistics network histogram with time interval 1 second

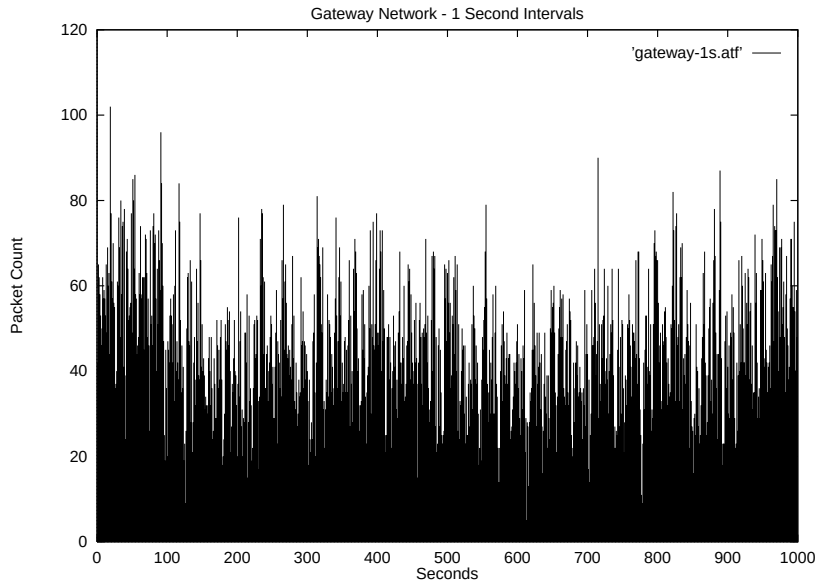

Figure 4.12: Gateway network with time interval 1 second

definite peaks every 300 milliseconds. This is caused by the router broadcasting IP routeing information. It is not a very large amount of information (about seven small sized packets sent back to back) indicating that the level of background traffic is small (since it does not mask the route broadcasts). The histogram (figure 4.11) shows most of the traffic flows at a steady level and that bursts are low in number and size.

4.4.2.3 External gateway network of the University

The Gateway Network (figure 4.12) has an extremely light load with a considerable amount (with respect to the total) of the traffic as a steady background flow. The bursts are much

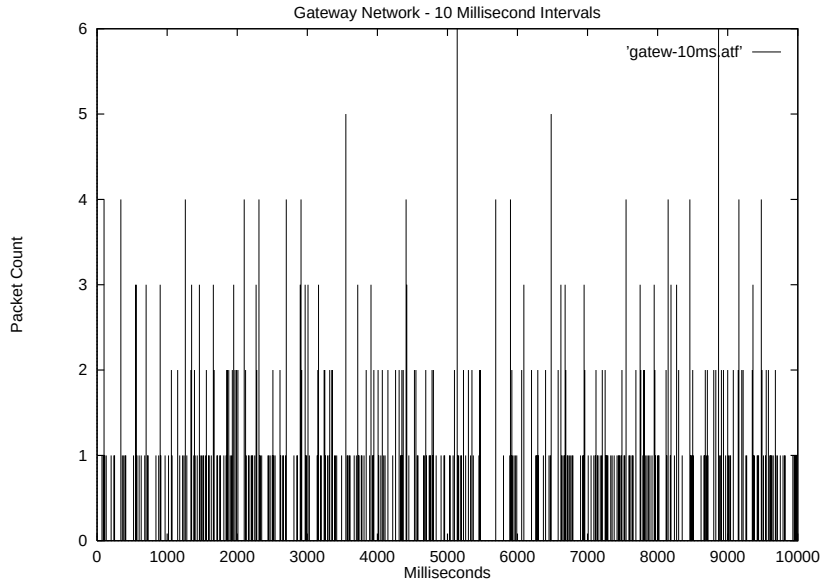

Figure 4.13: Gateway network with time interval 10 millisecond

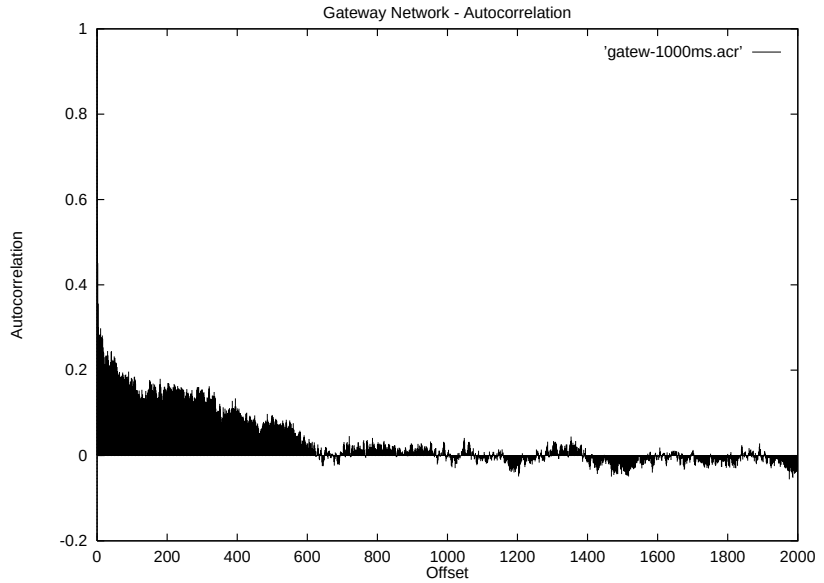

Figure 4.14: Gateway network autocorrelation with time interval 1 second

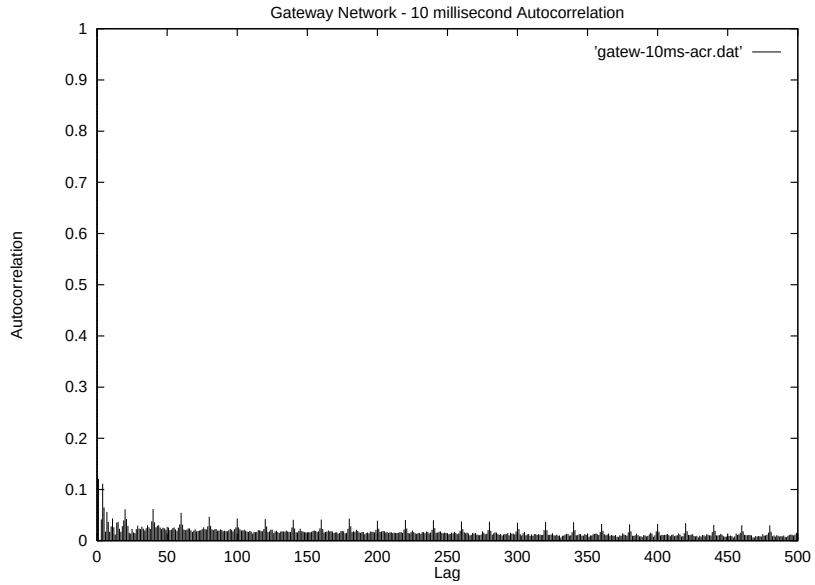

Figure 4.15: Gateway network autocorrelation with time interval 10 milliseconds

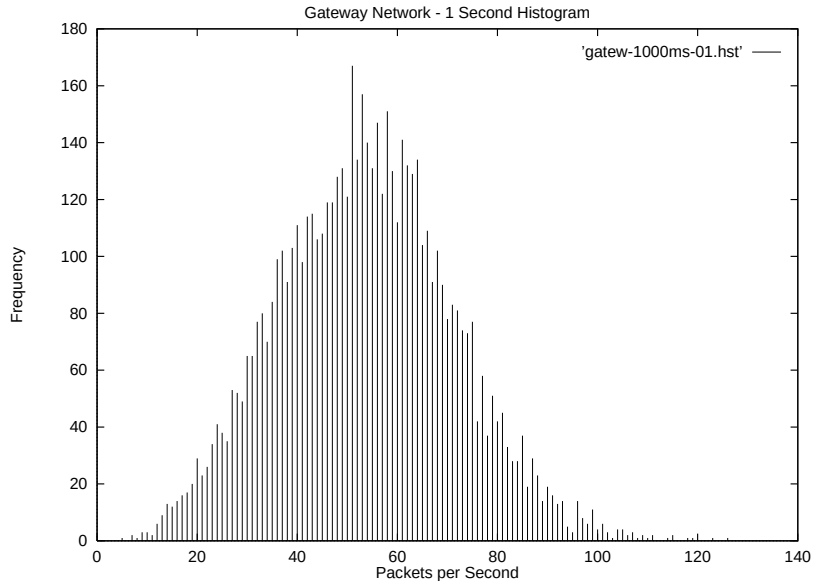

Figure 4.16: Gateway network histogram with time interval 1 second

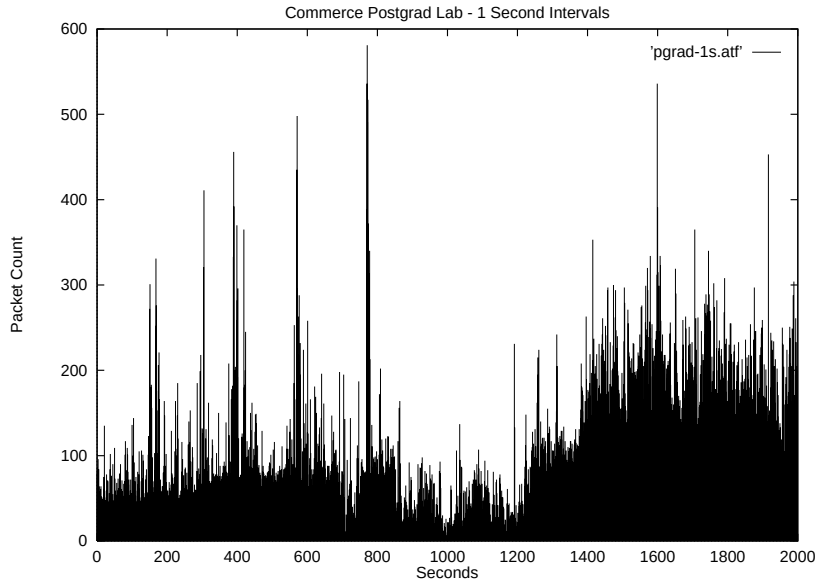

Figure 4.17: Commerce Postgraduate Lab with time interval 1 second

smaller in comparison to the other segments, with virtually all the traffic consisting of TCP/IP. The 1 second autocorrelation (figure 4.14) shows that there exists a dependency that decays steadily with lag until it becomes noise around the 10 minute lag time. The 10 millisecond autocorrelation (figure 4.15) shows periodic behaviour every 200 milliseconds. I am uncertain as to its exact cause but it could well be Network Time Protocol (NTP) [14].

The histogram (figure 4.16) shows very steady traffic flows without any large bursts. This link is connected to a 128 kbit connection to the outside world so the size of bursts would be limited in any case but the unimodal, approximately symmetric shape of the histogram most likely comes from the steady flow of USENET traffic and the congestion control used by TCP.

4.4.2.4 Commerce postgraduate teaching laboratory

The Commerce Postgraduate teaching laboratory consists of about twenty personal computers running Windows For Workgroups. They connect to a Windows NT server using the NetBIOS LAN protocol.

This segment has a reasonably heavy load and only has NetBIOS traffic. The down loading of large windows applications (over two megabytes or more) creates large bursts of traffic.

NetBIOS has no routeing information and no directory system so there is minimal periodic background traffic. The figures 4.17 and 4.18 show a steady level of traffic with only a few notable peaks. This behaviour is echoed in the histogram (figure 4.21) showing that most of the traffic occured as steady flows.

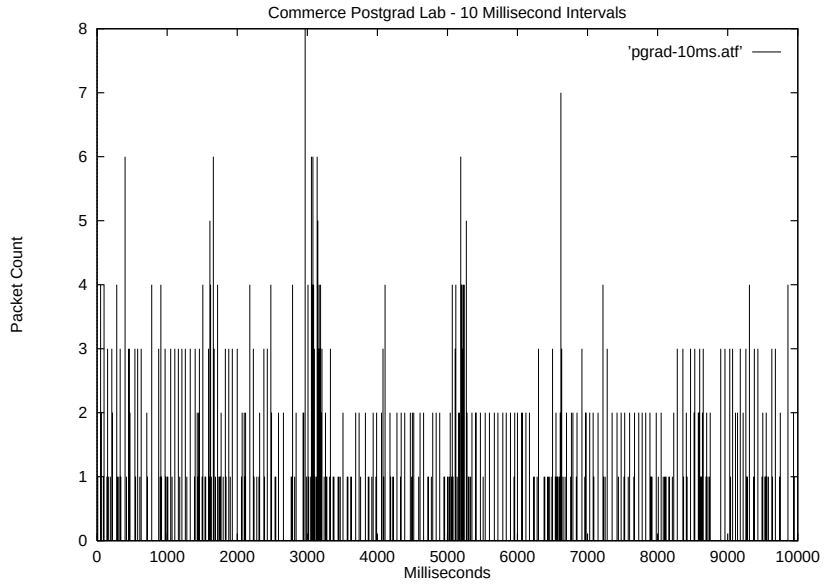

Figure 4.18: Commerce Postgraduate Lab with time interval 10 milliseconds

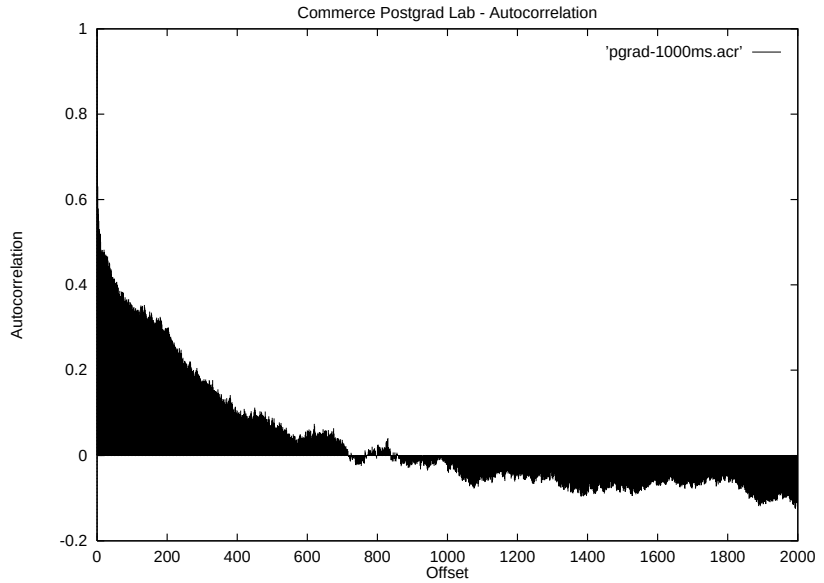

Figure 4.19: Commerce Postgraduate Lab autocorrelation with time interval 1 second

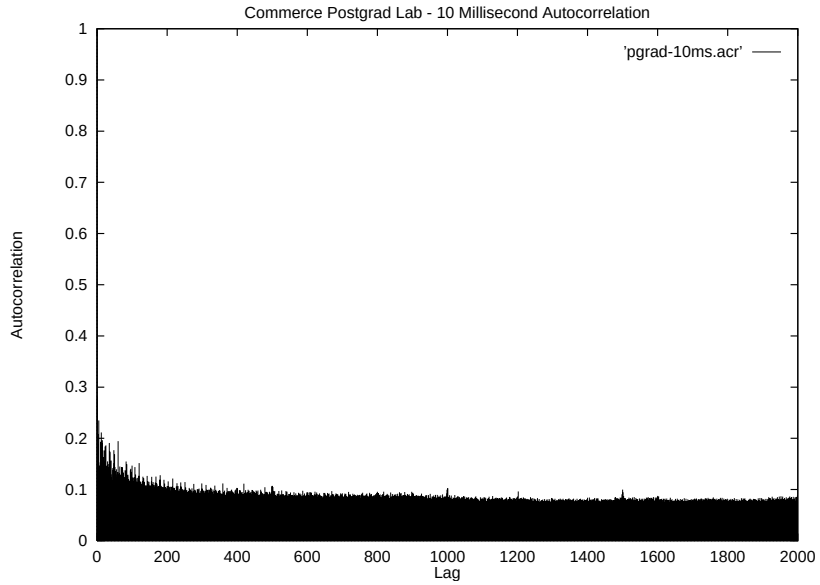

Figure 4.20: Commerce Postgraduate Lab autocorrelation with time interval 10 milliseconds

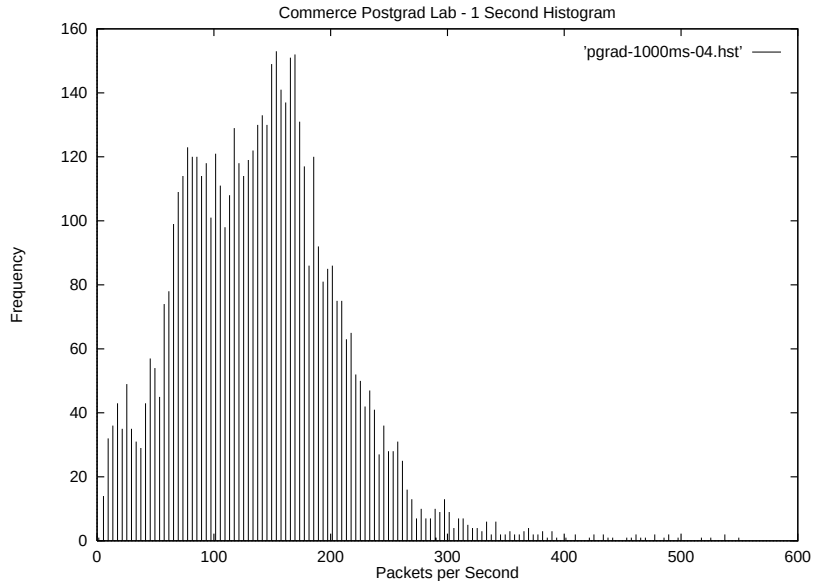

Figure 4.21: Commerce Postgraduate histogram Lab with time interval 1 second

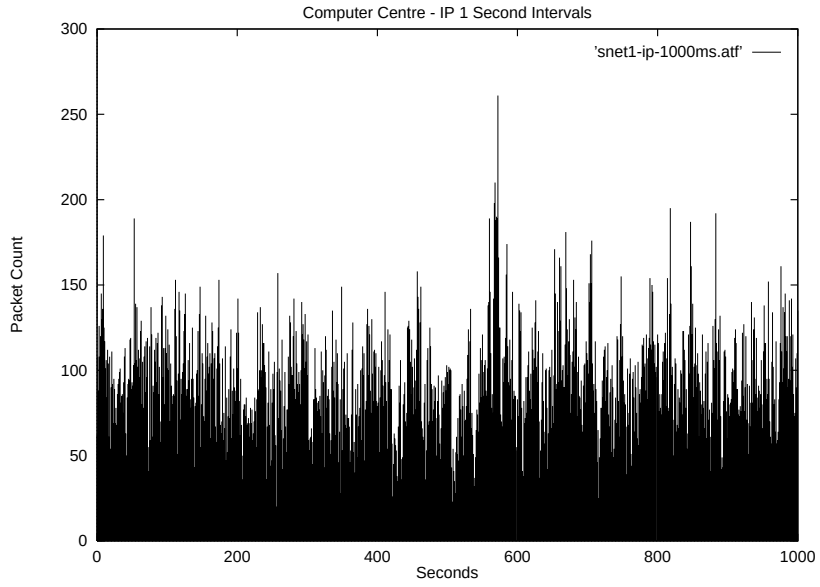

Figure 4.22: IP traffic on Computer Centre network with time interval 1 second

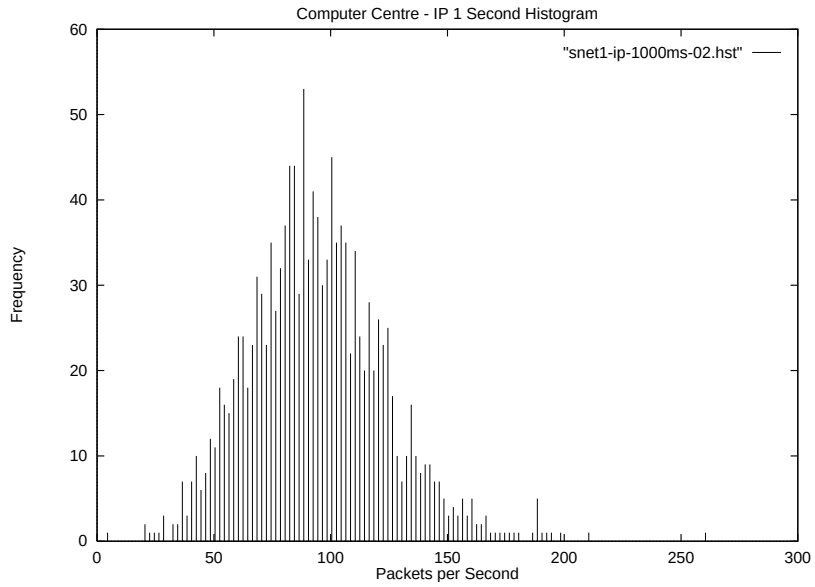

Figure 4.23: Histogram of IP traffic on Computer Centre network with time interval 1 second

4.4.3 Graphs by protocol

Below are three graphs showing the most prevalent protocols used around the university, that is IP (§1.2.1, AppleTalk (§1.2.2) and Netware IPX (§1.2.3).

4.4.3.1 Internet Protocol

Figure 4.22 shows Internet Protocols traffic. This graph shows a reasonable (with respect to the peak maximums) amount of background traffic. This is most likely a result of a steady level of remote login and mail traffic.

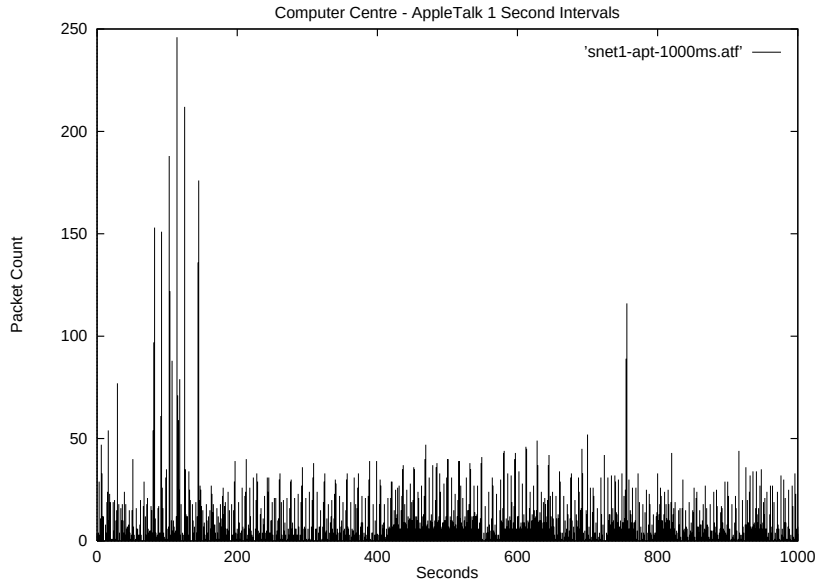

Figure 4.24: AppleTalk traffic on Computer Centre network with time interval 1 second

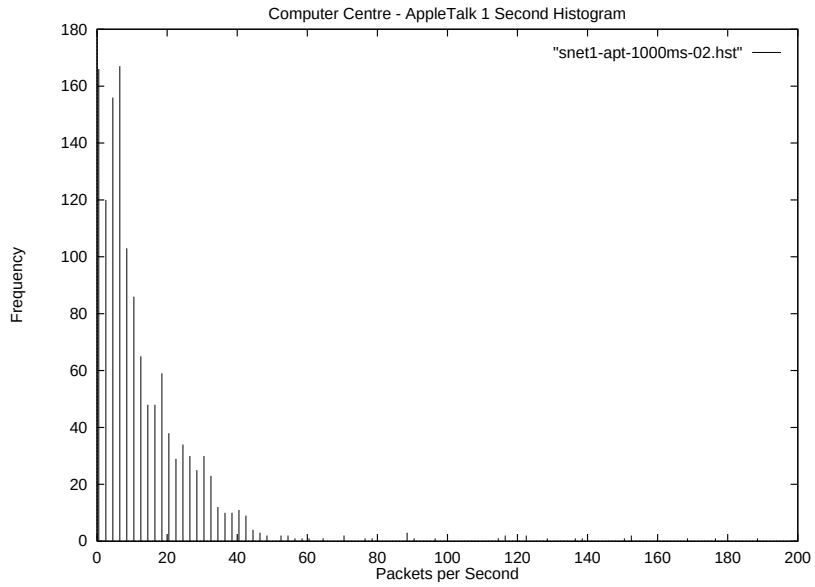

Figure 4.25: Histogram of AppleTalk traffic on Computer Centre network with time interval 1 second

4.4.3.2 AppleTalk

Figure 4.24 shows AppleTalk traffic on the Computer Centre network. Because there is very little AppleTalk traffic on subnet 1 events become more pronounced and noticeable. AppleTalk has periodic broadcasts to relay routeing and directory services (zone maps) information.

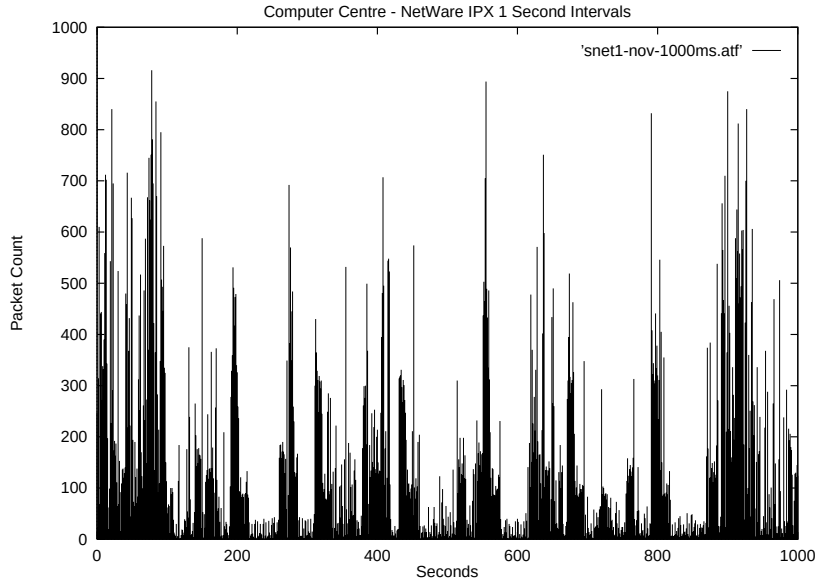

Figure 4.26: IPX traffic on Computer Centre network with time interval 1 second

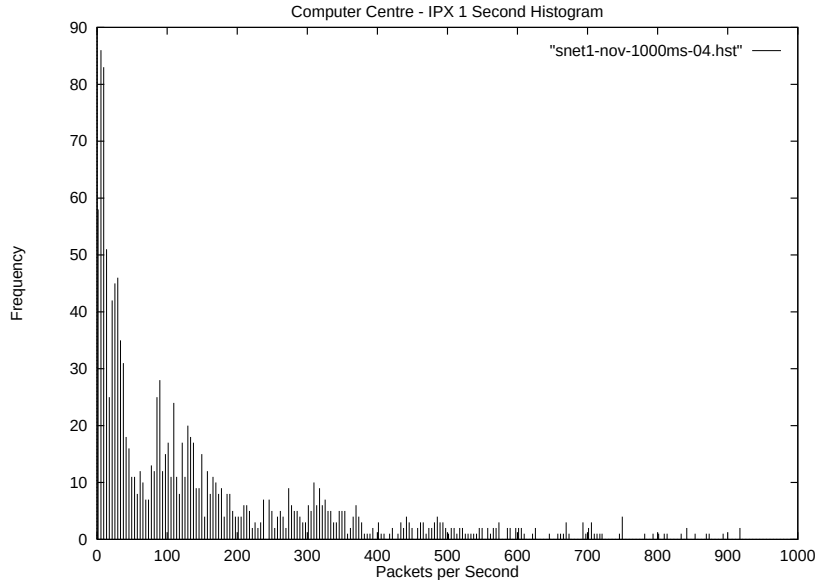

Figure 4.27: Histogram of IPX traffic on Computer Centre network with time interval 1 second

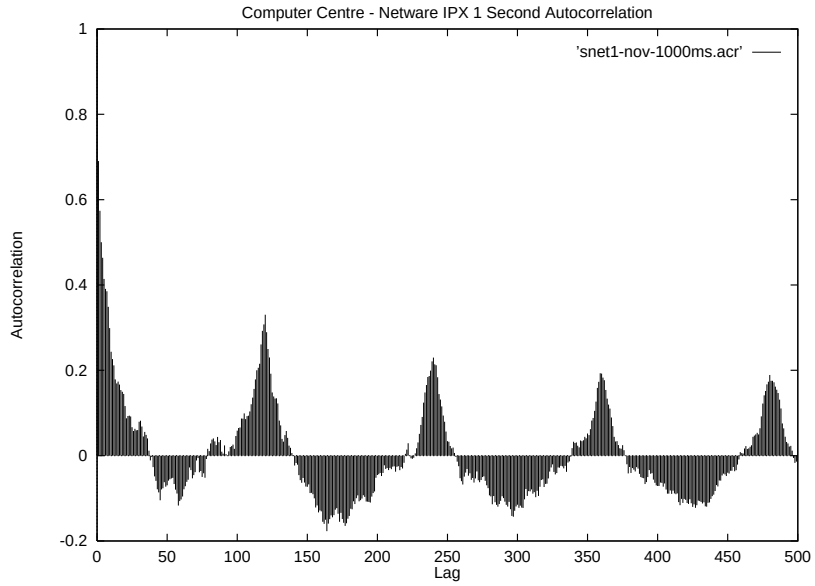

Figure 4.28: Autocorrelation of IPX traffic on Computer Centre network with time interval 1 second

4.4.3.3 Netware IPX

Figure 4.26 show Netware traffic on the Computer Centre network. While it looks as though there is considerable IPX traffic in fact there is little file transfer traffic. The large bursts of traffic come from Netware's service announcements. Every two minutes Netware capable routers broadcast all available Netware services, such as file servers, printing spoolers and mail exchanges. Due to the number of these services around the university it amounts to over 500 kilobytes of information. This is clearly seen in the autocorrelation (figure 4.28).

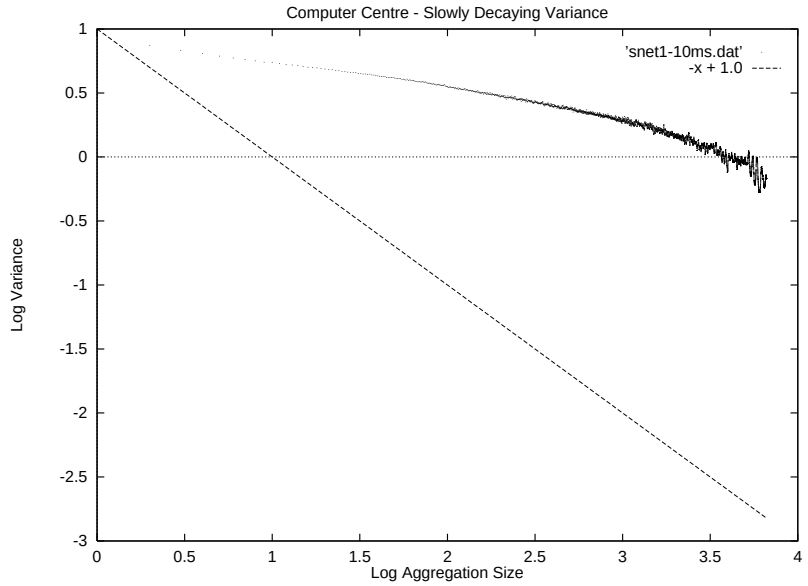

Figure 4.29: Slowly decaying variance plot of the Computer Centre network

4.5 Slowly decaying variance

Figures 4.29, 4.30, 4.31 and 4.32 all show the slowly decaying variance associated with self-similar behaviour. The plots are created from the output of `stat` program (§4.2.5. The plots use the logarithm base 10 output from `stat`. This is the same as the slowly decaying variance plots in the Bellcore papers [1] [9]. Each plot includes a line with slope $y = -x$ for reference (the theoretical slope for a general renewal is $y = x^{-1}$, so taking the logarithm gives $y = -x$). While each plot is unique they all have a similar shape and clearly show slowly decaying behaviour.

While is obvious that the shapes are different, because of the differences in the samples there is no simple explanation as to how the shape of the curve relates to the underlying traffic behaviour.

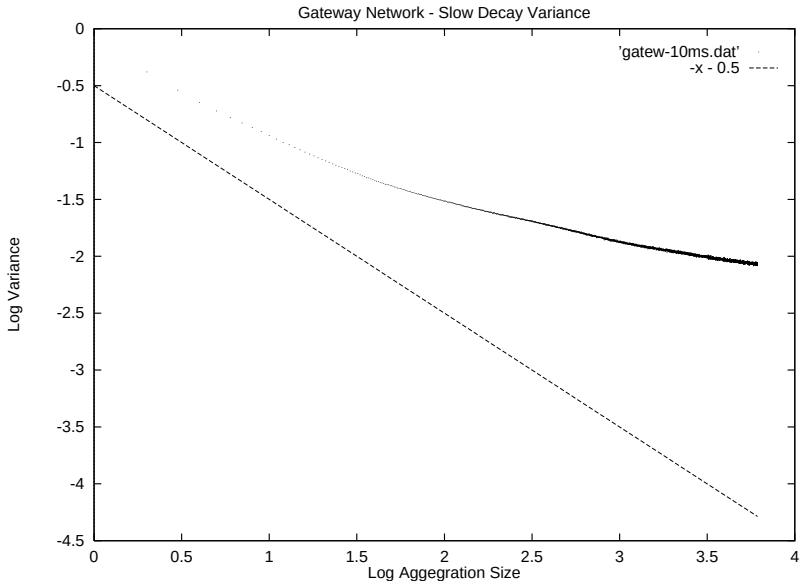

Figure 4.30: Slowly decaying variance plot of the Gateway network

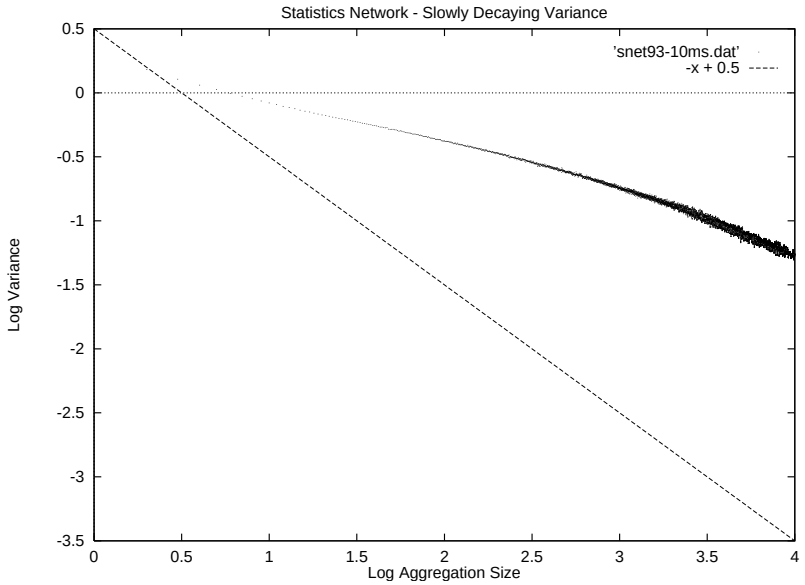

Figure 4.31: Slowly decaying variance plot of the Statistics network

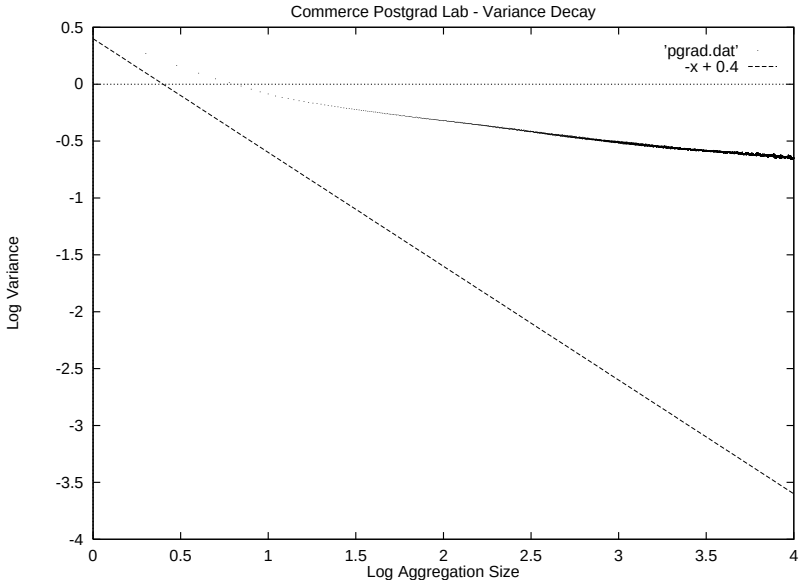

Figure 4.32: Slowly decaying variance plot of the Commerce Postgraduate network

Chapter 5

Simulation

5.1 Introduction

A series of simulation experiments was performed. Below is the list of nine experiments. Each experiment generated a sample trace of length 1,000 seconds.

1. Poisson process simulation.
2. Renewal process with scaled uniform inter-renewal distribution.
3. A 3-state modulated renewal process.
4. Renewal process with Pareto (infinite variance) inter-renewal distribution.
5. Renewal process with $t_2 - distribution$ (infinite variance) inter-renewal distribution.
6. Renewal process with Cauchy inter-renewal (infinite variance) distribution.
7. Single renewal process with $t_2 - distribution$ inter-renewal distribution.
8. 10 super-imposed independent renewal processes with $t_2 - distribution$ inter-renewal distributions.
9. 100 super-imposed independent renewal processes with $t_2 - distribution$ inter-renewal distributions.

The simulations were done using a single chain of recurrent events, ordered by time. Each event is independent of all others on the chain, and knows when in the future it will occur. The chain also has a notion of the current time. When an event reaches the head of the chain it occurs and is recorded into the trace file. The current time is set to the time of the event and a new time for it to occur is generated. This event (now to occur sometime in the future) is placed back into the chain in chronological order.

The events are self contained and may have simple (samples from some distribution) or complex (with multiple internal states) recurrent behaviour. All that is required is that it generates a new recurrence time in the positive future.

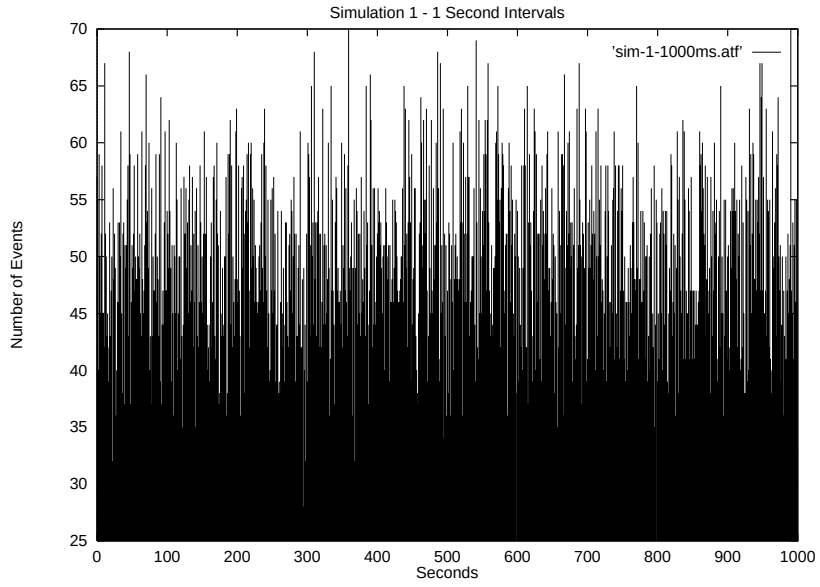

Figure 5.1: Poisson Process Simulation with Exponential Distribution

5.2 Simulation of simple Poisson process

Using the simplest model available, that is exponentially distributed inter-arrival times, we can produce sample traces to examine. This simulation is not meant to be comparable to the real traces, but rather a counter example to show that such a model does not fit Ethernet traffic patterns.

It is also intended to show that the simulation techniques work properly and that the tools developed to produce the results are robust.

5.2.1 Exponential arrival distribution

The first simulation *sim-1* is a Poisson Process with rate $\lambda = 0.05$ packets per millisecond, simulated over a period of 1,000 seconds ($\sim$ 16 minutes). This results in an exponentially distributed inter-arrival time with mean $\mu = 20$. A sample trace can be seen in figure 5.1 along with the packet frequency histogram (figure 5.2) and slowly decaying variance plot (figure 5.3). As expected the plot is a straight line with a slope of -1, hence the variance decays order n^{-1} with averaging aggregation, in line with the theoretical model.

5.2.2 Single state with uniform inter-arrival time distribution

The second simulation *sim-2* is a renewal process which has inter-arrival times that are uniformly distributed with mean $\mu = 20$. The time series can be seen in figure 5.4 along with its histogram (figure 5.5) and the slowly decaying variance plot (figure 5.6). Again the variance plot is a straight line with slope -1, that is it shows no sign of self-similar behaviour.

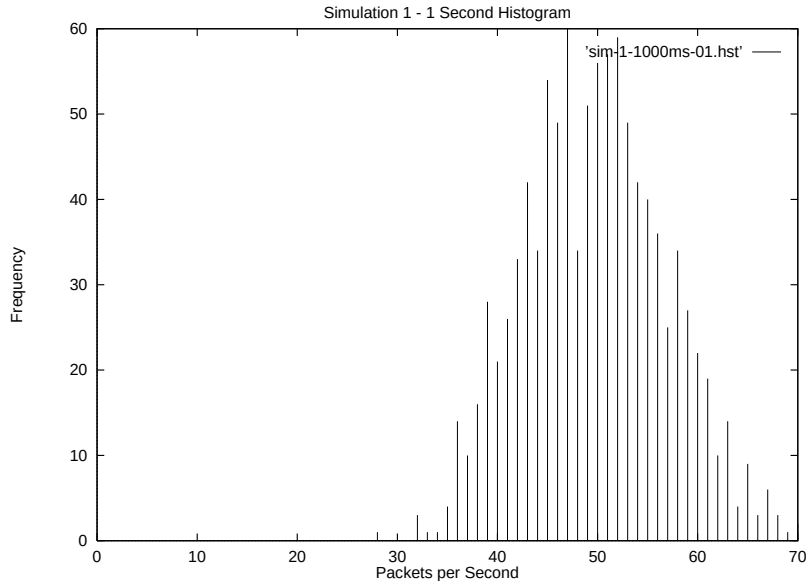

Figure 5.2: Histogram of a Poisson Process Simulation

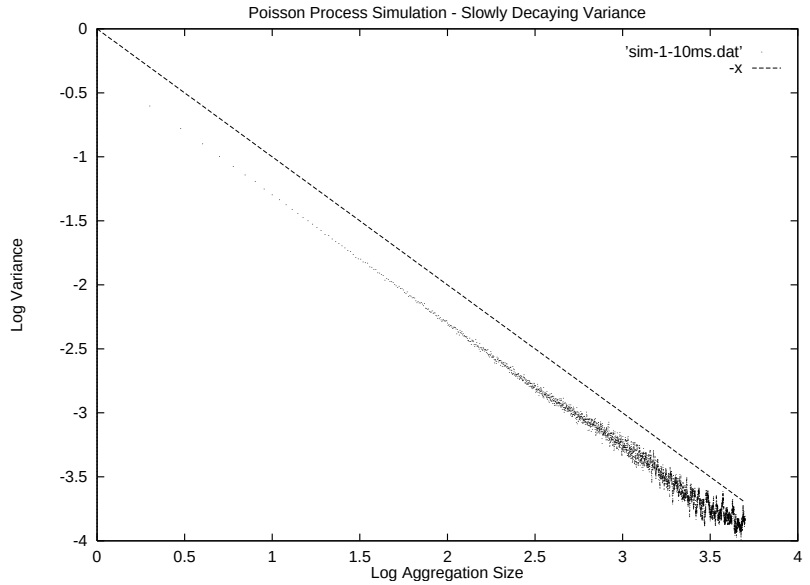

Figure 5.3: Slowly decaying variance plot of a Poisson Process Simulation

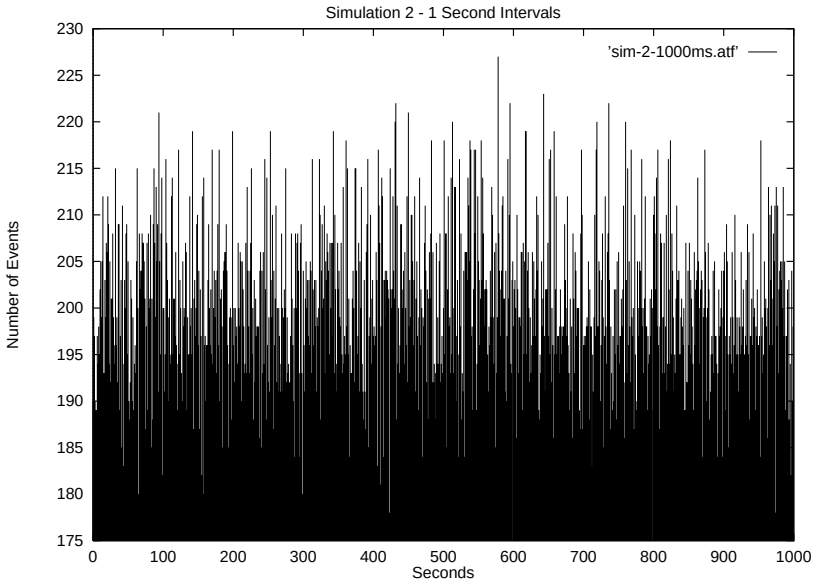

Figure 5.4: Uniform Distribution Renewal Process Simulation

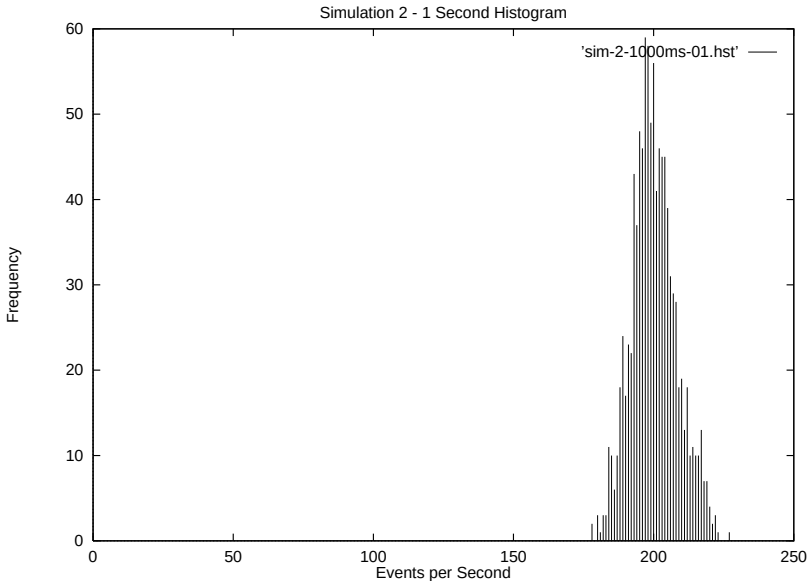

Figure 5.5: Histogram of Uniform Distribution Renewal Process Simulation

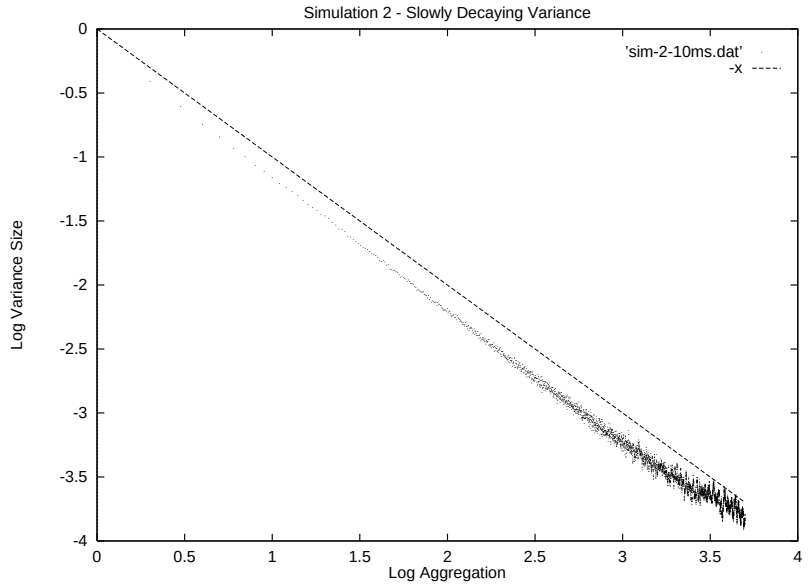

Figure 5.6: Slowly decaying variance plot of Uniform Distribution Renewal Process Simulation

5.3 Generalised modulated renewal process

5.3.1 The model

The mathematical model for this simulation is discussed in the models chapter (§3.3.2).

5.3.2 The program

The program reads in a parameter file and produces a trace file (Figure 2.1).

```
simulate [-t time in seconds] parameter files
```

5.3.2.1 An example

For an example I have chosen a three state process.

State	Inter-Event Distribution		Lifetime Distribution	
	Distribution	Mean	Distribution	Mean
0	Exponential	10	Exponential	100
1	Deterministic	2	Uniform	40
2	Deterministic	1	Deterministic	15

The results of the simulation *sim-3* are shown in figures 5.7, 5.8, 5.9 and 5.10. The time series (figure 5.7) looks bursty but the histogram (figure 5.8 shows that the number of large bursts is small and that most of the flows are in a limited range (200 – 350 events per second).

The autocorrelation (figure 5.9) shows the existence of a definite short term correlation, caused by the deterministic behaviour within two of the states. This short term correlation

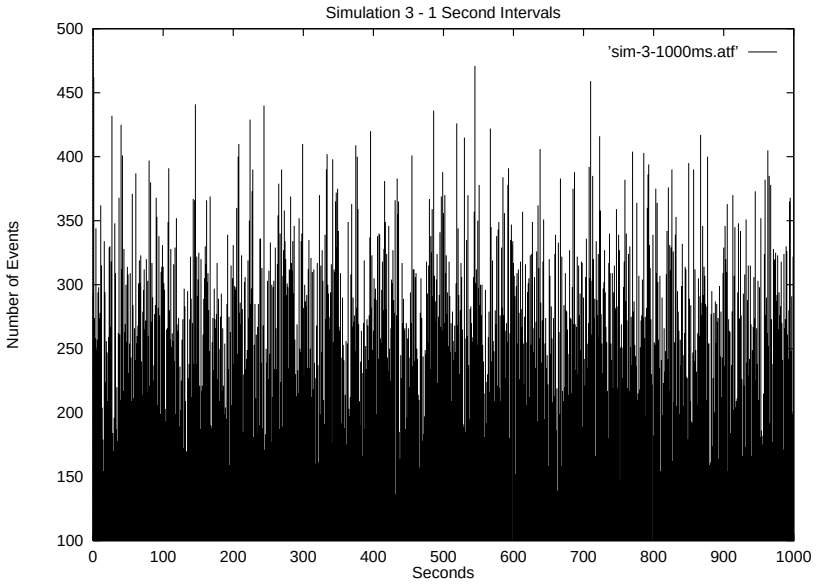

Figure 5.7: Modulated Renewal Process Simulation

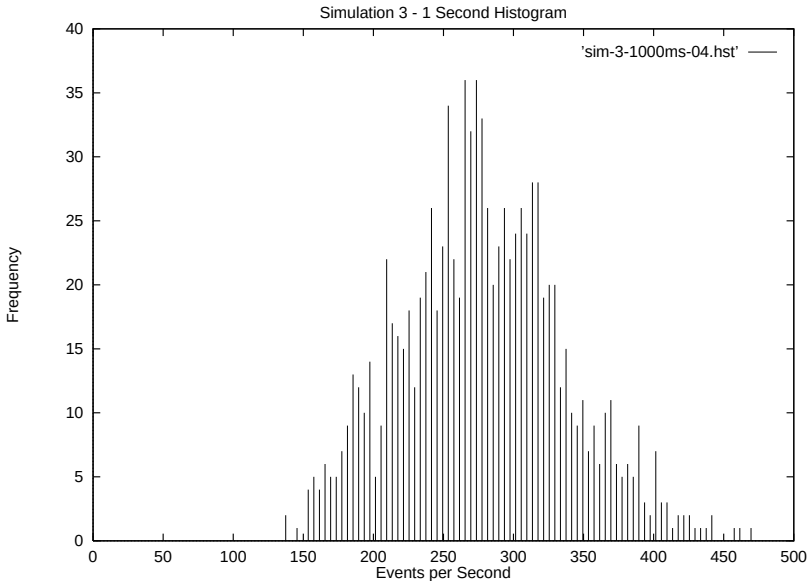

Figure 5.8: Histogram of Modulated Renewal Process Simulation

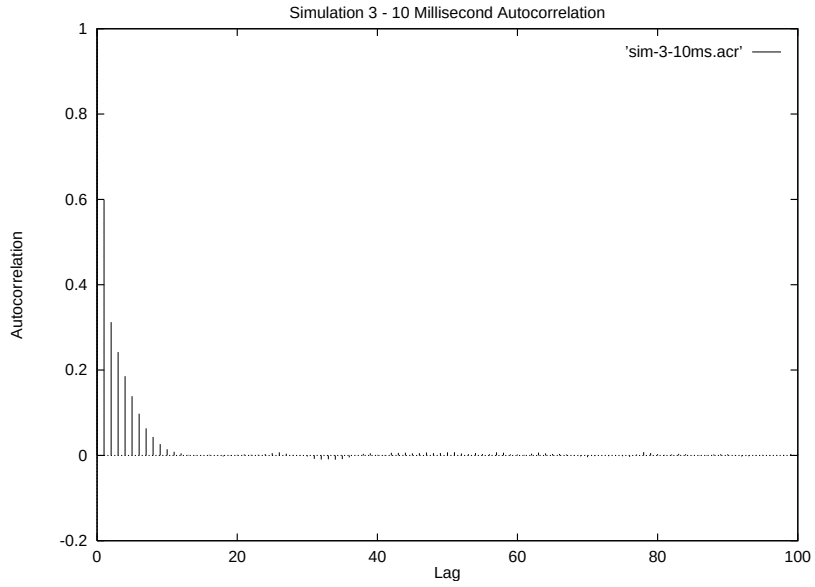

Figure 5.9: Autocorrelation of Modulated Renewal Process Simulation

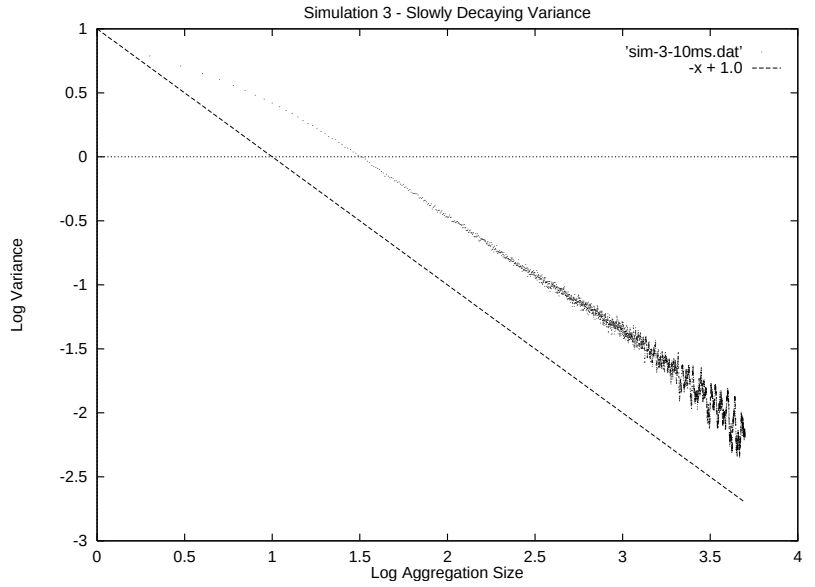

Figure 5.10: Slowly decaying variance plot of Modulated Renewal Process Simulation

falls away in relation to the expected life time of the deterministic states to white noise caused by the Poisson distributed state and life time.

The slowly decaying variance graph (figure 5.10) shows that this model is not what we are looking for, and that a more complex model does not help.

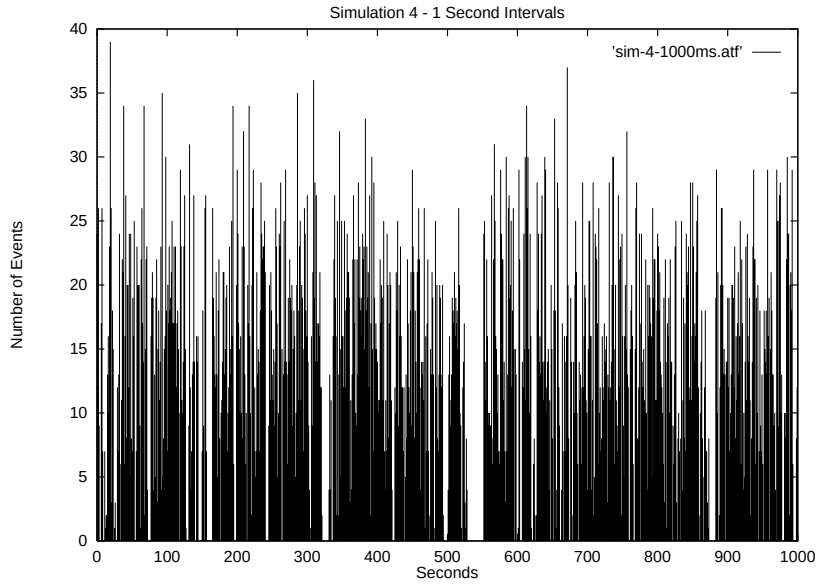

Figure 5.11: Pareto Distributed Renewal Process Simulation

5.4 Infinite variance renewal processes

5.4.1 Introduction

Below are the results of three simple renewal processes, similar to the simple Poisson process (§5.2) except that the inter-renewal distributions have infinite variance. This class (having infinite variance) of distributions is often known as heavy-tailed distributions.

The three distributions used are the Pareto distribution, a non symmetric distribution that is defined on $\mathbb{R}^+$. This distribution was used because of its reference in [10] (§3.2.3, Page 19). The Pareto distribution is a stable distribution making further mathematical analysis a little simpler than other distributions.

The other two distributions come from Student's t distribution. For degrees of freedom 1 and 2 it has infinite variance. The $t_1 - distribution$, commonly known as the Cauchy distribution, also has infinite mean. The $t_\nu - distribution$ is a flattened normal, symmetric and defined on all of $\mathbb{R}$. For the simulations the distributions are folded around the y-axis so that only positive inter-renewal times exist.

5.4.2 Pareto distribution

The time series for *sim-4* is shown in figure 5.11. The large number of gaps and lack of "background" traffic are notable features of this plot (the $t - distribution$ plots also show this behaviour). The histogram (figure 5.12) shows a wide range of traffic levels. Note that although it cannot be seen there is bar of height 143 at 0 events per second.

The autocorrelation (figure 5.13) displays the important positive autocorrelation, indicating long range dependence. The slowly decaying variance (figure 5.14) shows the decay occuring at a slower rate than x^{-1}, verifying that infinite variance renewal processes do exhibit fractal behaviour.

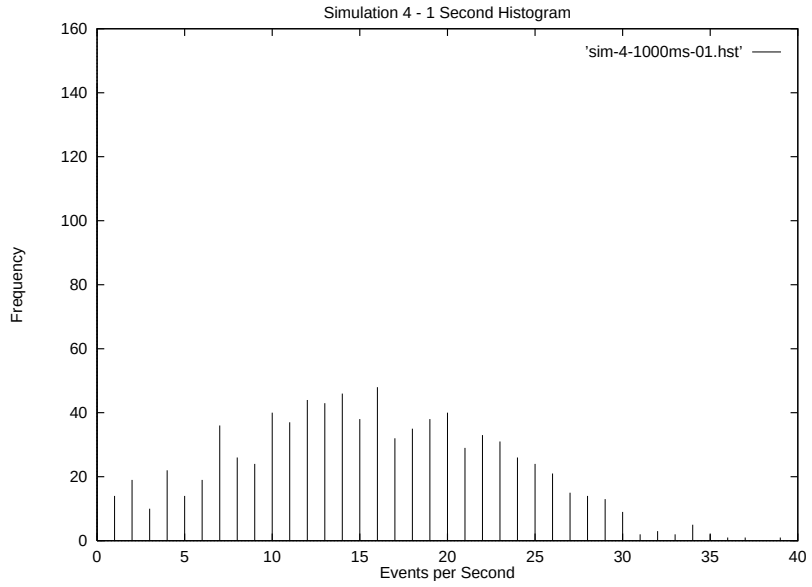

Figure 5.12: Histogram of Pareto Distributed Renewal Process Simulation

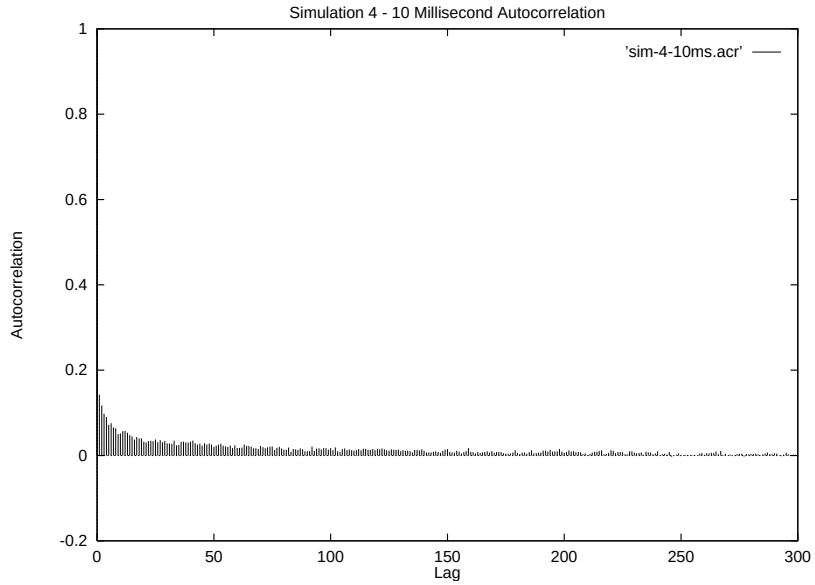

Figure 5.13: Autocorrelation of Pareto Distributed Renewal Process Simulation

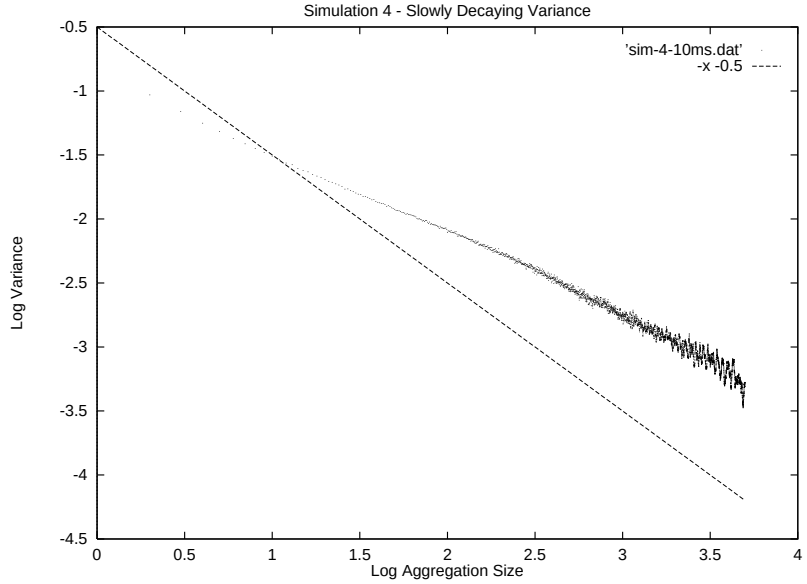

Figure 5.14: Slowly decaying variance plot of Pareto Distributed Renewal Process Simulation

5.4.3 Cauchy and t distributions

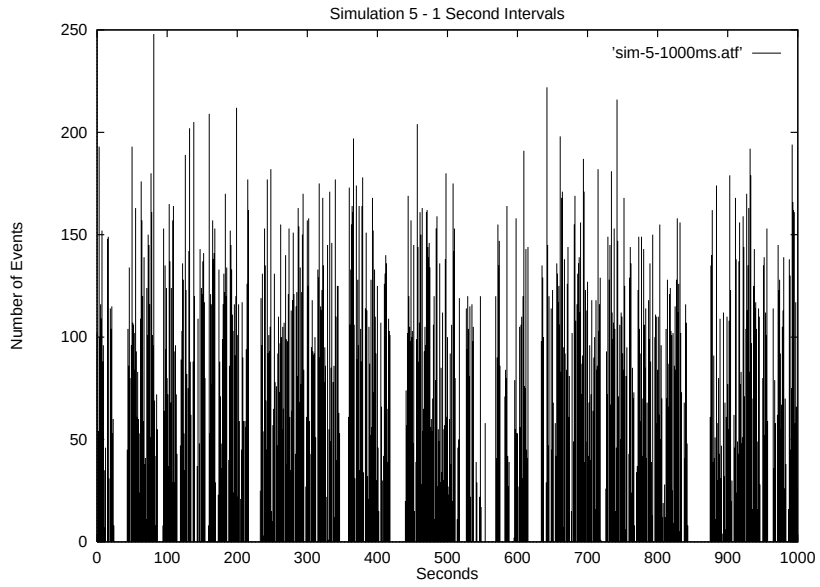

Figure 5.15: $t_2 - distribution$ Distributed Renewal Poisson Simulation

Figures 5.15, 5.16 and 5.17 are the result from simulating a general renewal process with a $t_2 - distribution$ inter-renewal distribution (*sim-5*).

Figures 5.18, 5.19 and 5.20 are the result from simulating a general renewal process with a Cauchy ($t_1 - distribution$) inter-renewal distribution (*sim-6*).

The Cauchy distribution simulation generated very few (in comparison with the other simulations and real samples) events so that the analysis is less reliable. The $t_2 -$

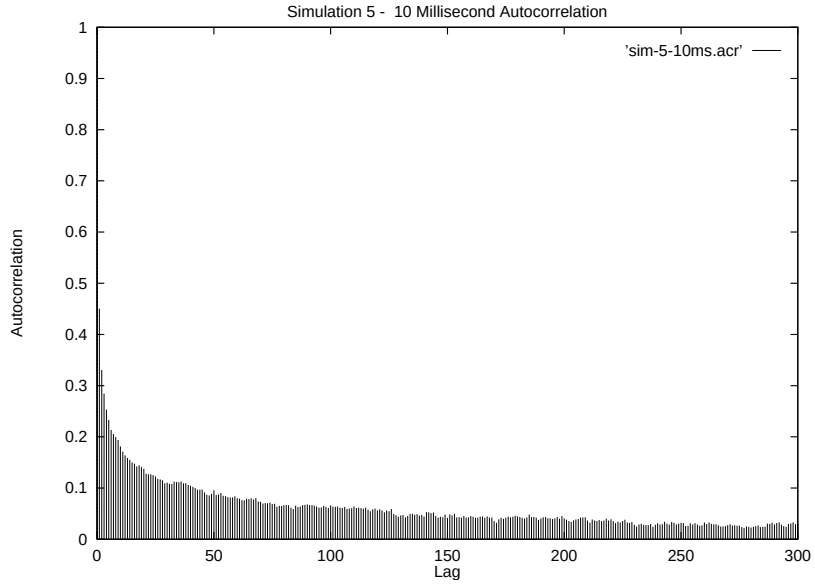

Figure 5.16: Autocorrelation of $t_2 - distribution$ Distributed Renewal Process Simulation

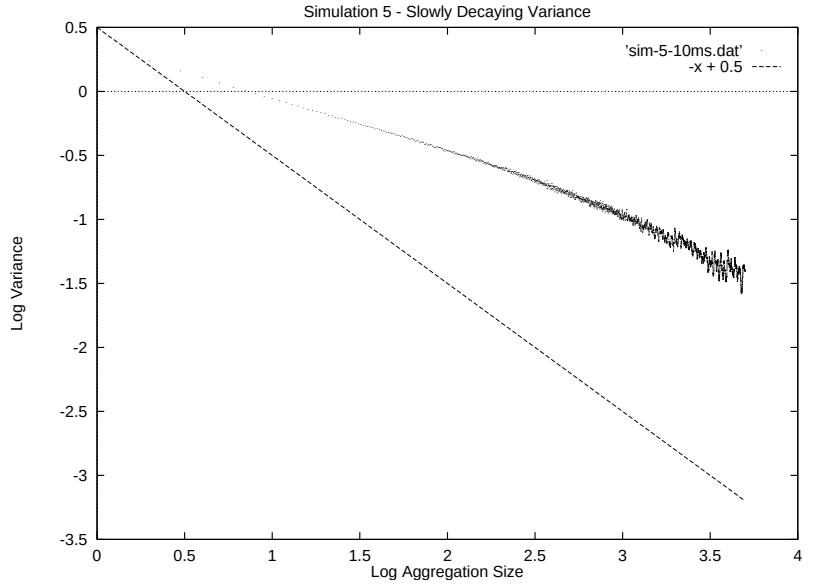

Figure 5.17: Slowly decaying variance plot of $t_2 - distribution$ Distributed Renewal Process Simulation

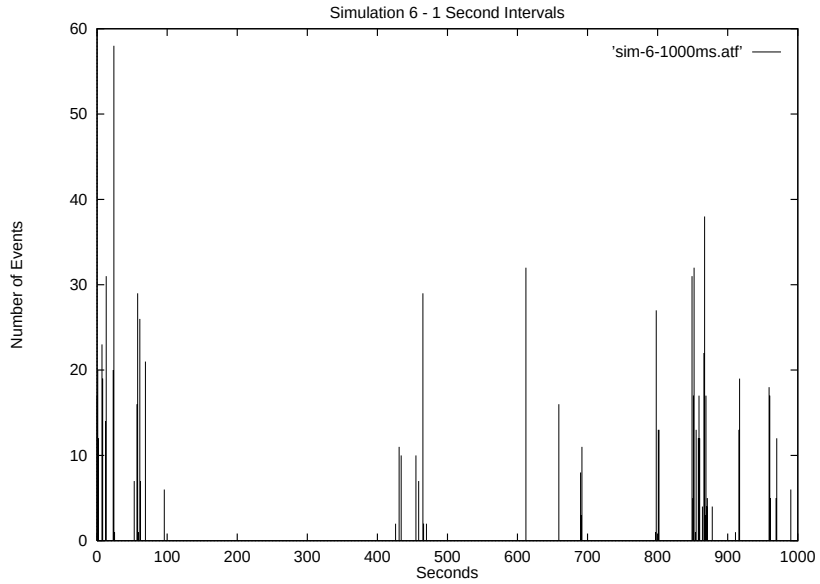

Figure 5.18: Cauchy Distributed Renewal Poisson Simulation

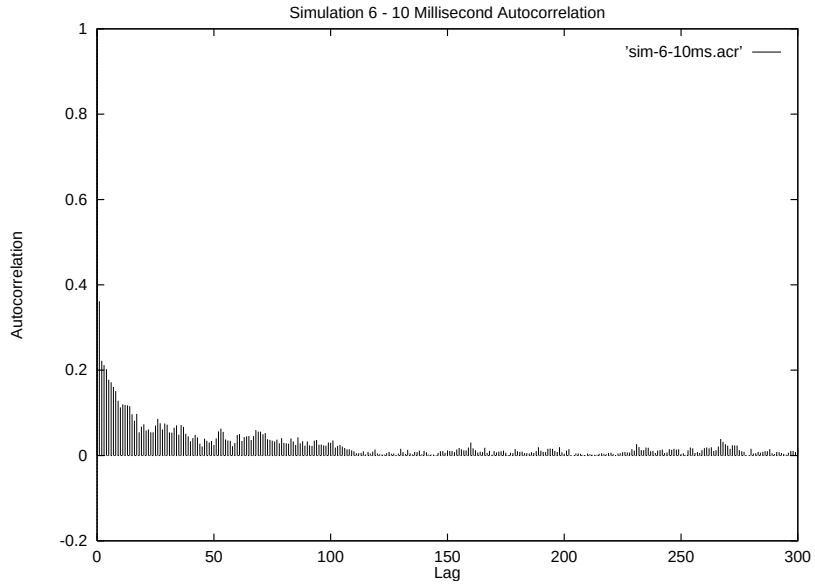

Figure 5.19: Autocorrelation of Cauchy Distributed Renewal Process Simulation

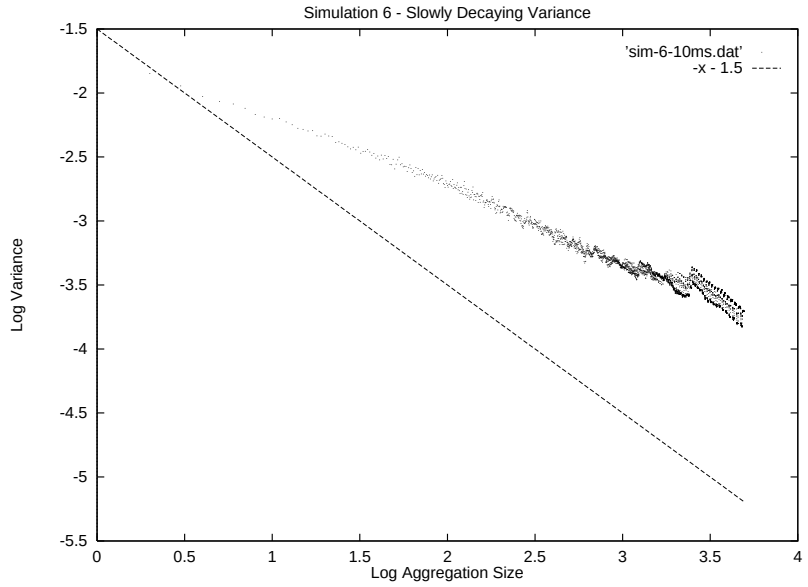

Figure 5.20: Slowly decaying variance plot of Cauchy Distributed Renewal Process Simulation

distribution results show strong positive autocorrelation and noticeable slowly decaying variance.

5.4.4 Merged $t_2 - distribution$ renewal processes

Below are the results for the superimposed (or merged) renewal processes. The full definition can be found in the models chapter (§3.4). The base distribution is a $t_2 - distribution$. This was used as it produced clear results in the earlier simulations. The choice of the $t_2 - distribution$ rather than Pareto was mainly one of personal opinion and my impression that it gave a clearer results with respect to slowly decaying variance and long range dependence (the autocorrelation). Earlier experiments show that the Pareto is a perfectly acceptable distribution for these experiments and produces similar conclusions.

A single process is repeated (identical in distribution to simulation 5) as a control (*sim-7*). 10 and 100 merged processes were then produced (*sim-8* and *sim-9* respectively). Both of these produced a large number of total events giving decisive results.

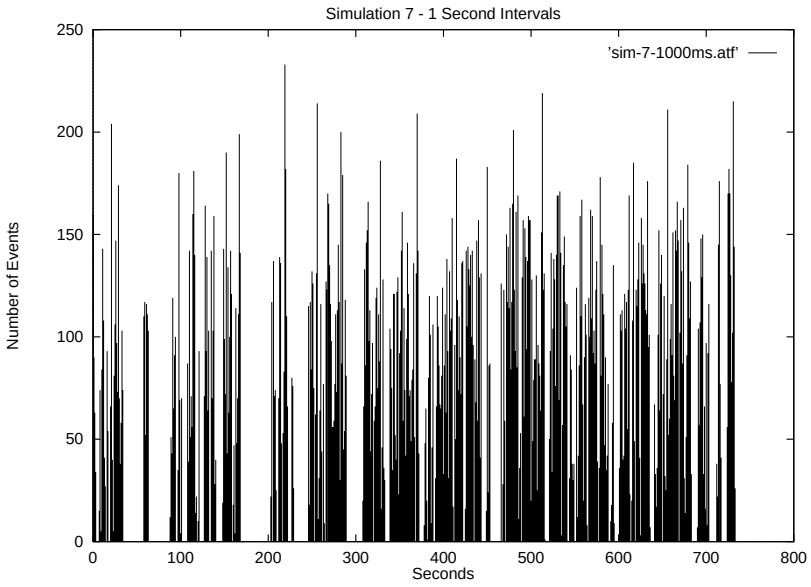

Figure 5.21: Single $t_2 - distribution$ Distributed Renewal Process Simulation

The results show that superimposing does affect slowly decaying variance and that the number of merged processes directly influences the rate of variance decay.

The simulations make it plain that visual inspection of the trace is not enough. This is clearly seen in comparing simulation 3 (general modulated renewal process) with simulation 8 (10 merged $t_2 - distribution$ renewal processes). While their time series (figures 5.7 and 5.25) and histograms (figures 5.8 and 5.26) look similar this is clearly shown to be superficial as simulation 8 displays marked fractal properties, whereas simulation 3 shows little beyond a Poisson process.

Although the slope of the slowly decaying plot decreases as the number of merged processes increases, these suggest that a single process (as for figures 5.14, 5.17, 5.20 and 5.24) is sufficient to produce time series having self-similar behaviour. This is a much simpler simulation than that suggested in reference [1] [9] [8] [10] [5].

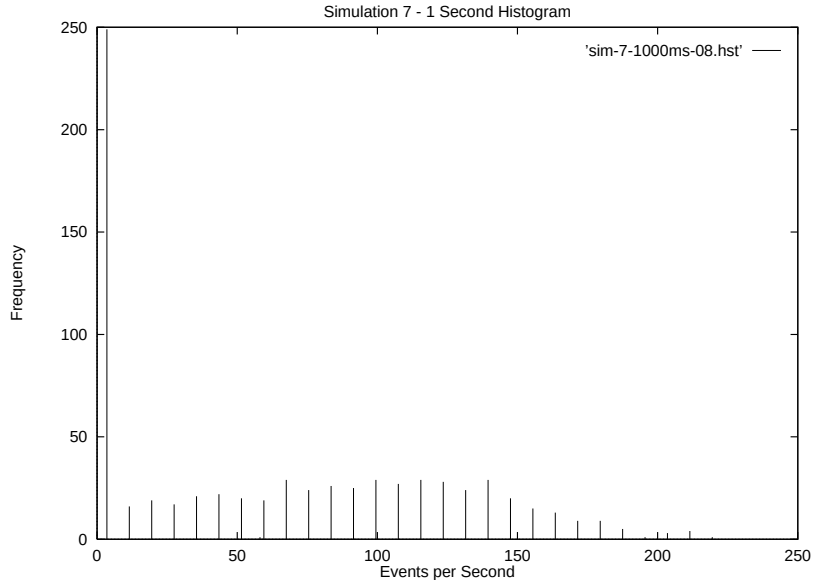

Figure 5.22: Histogram of Single $t_2 - distribution$ Distributed Renewal Process Simulation

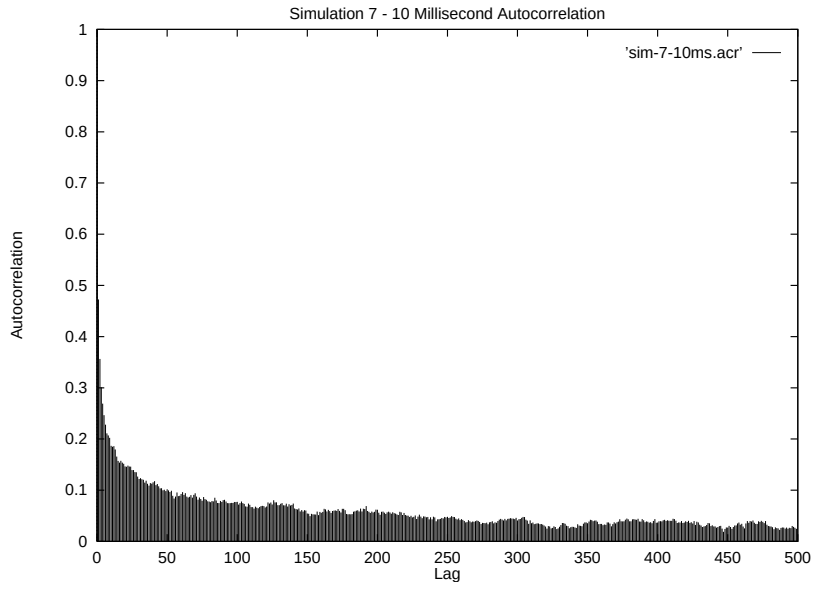

Figure 5.23: Autocorrelation of Single $t_2 - distribution$ Distributed Renewal Process Simulation

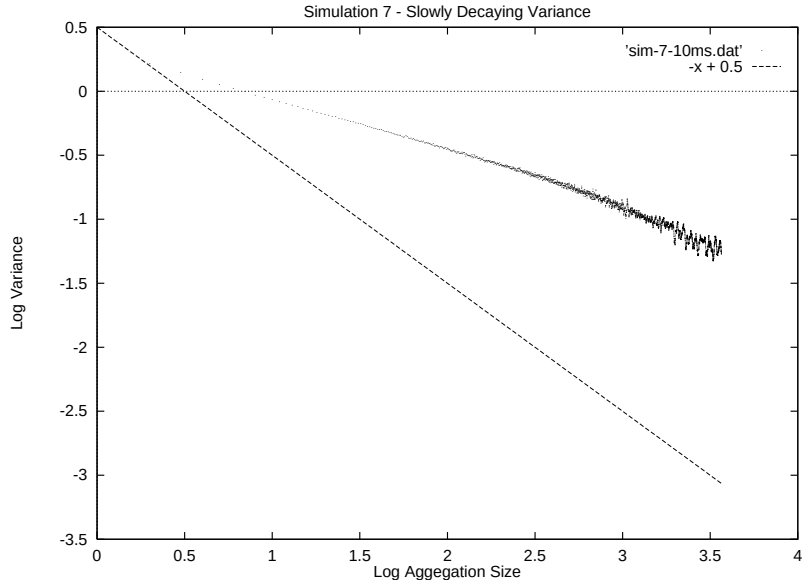

Figure 5.24: Slowly decaying variance plot of Single $t_2 - distribution$ Distributed Renewal Process Simulation

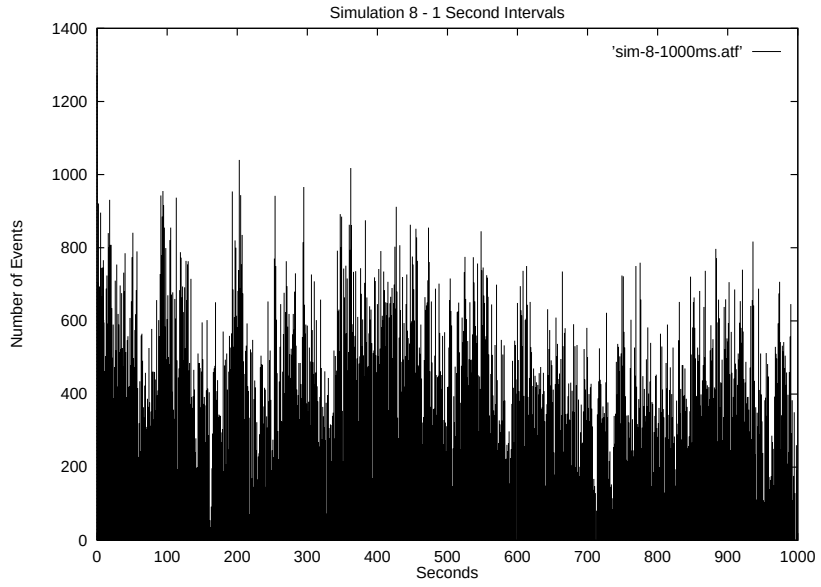

Figure 5.25: 10 Superimposed $t_2 - distribution$ Distributed Renewal Process Simulation

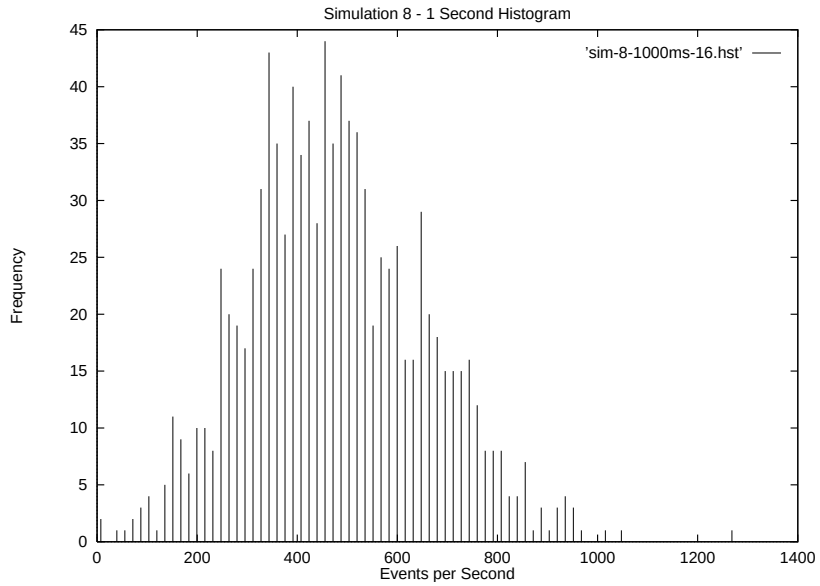

Figure 5.26: Histogram of 10 Superimposed $t_2 - distribution$ Distributed Renewal Process Simulation

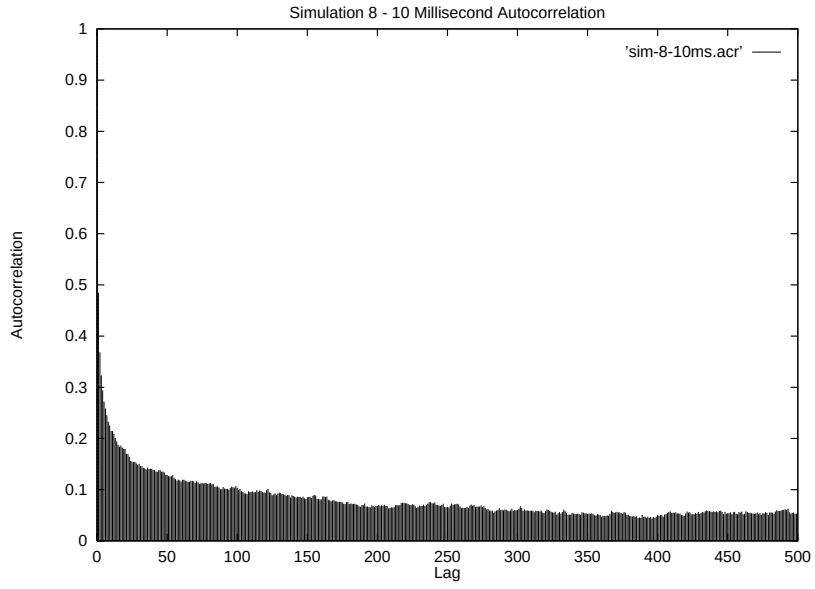

Figure 5.27: Autocorrelation of 10 Superimposed $t_2 - distribution$ Distributed Renewal Process Simulation

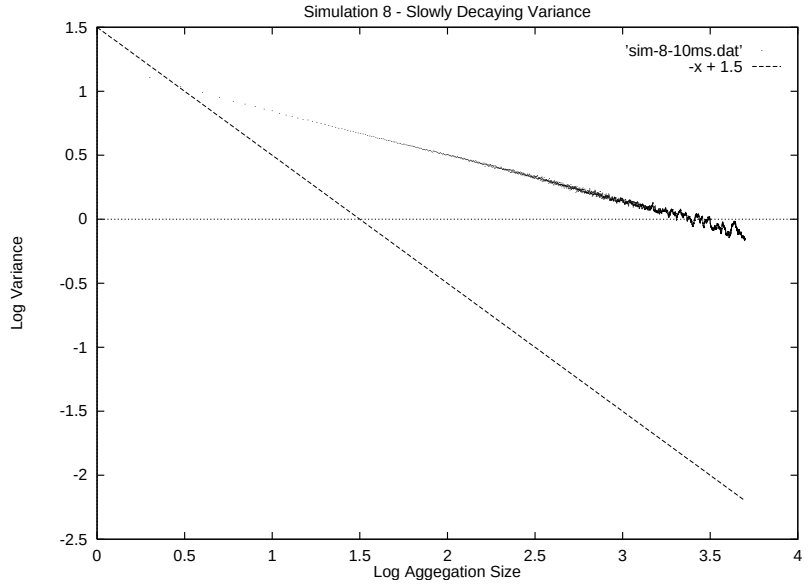

Figure 5.28: Slowly decaying variance plot of 10 Superimposed $t_2 - distribution$ Distributed Renewal Process Simulation

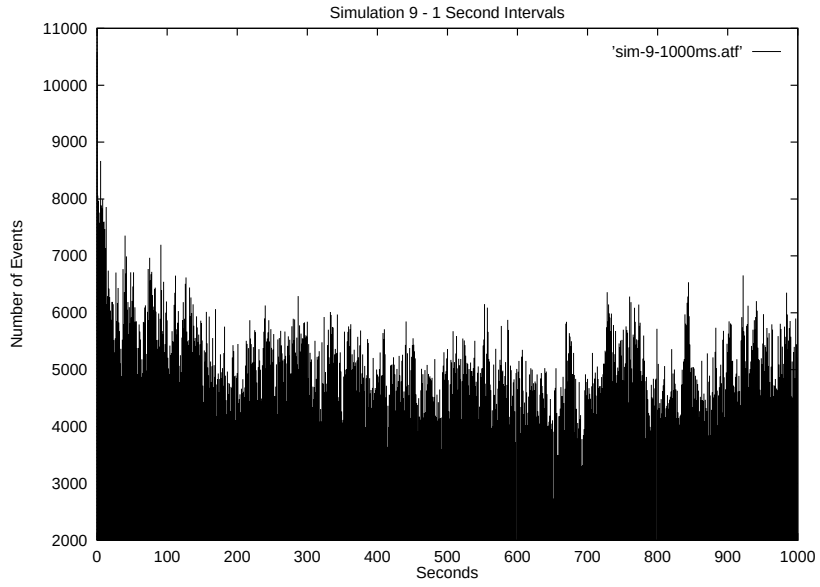

Figure 5.29: 100 Superimposed $t_2 - distribution$ Distributed Renewal Process Simulation

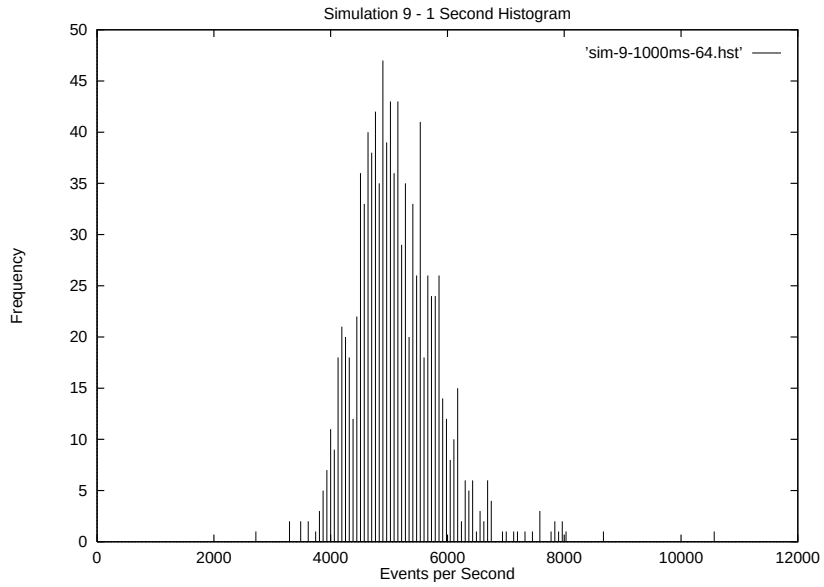

Figure 5.30: Histogram of 100 Superimposed $t_2 - distribution$ Distributed Renewal Process Simulation

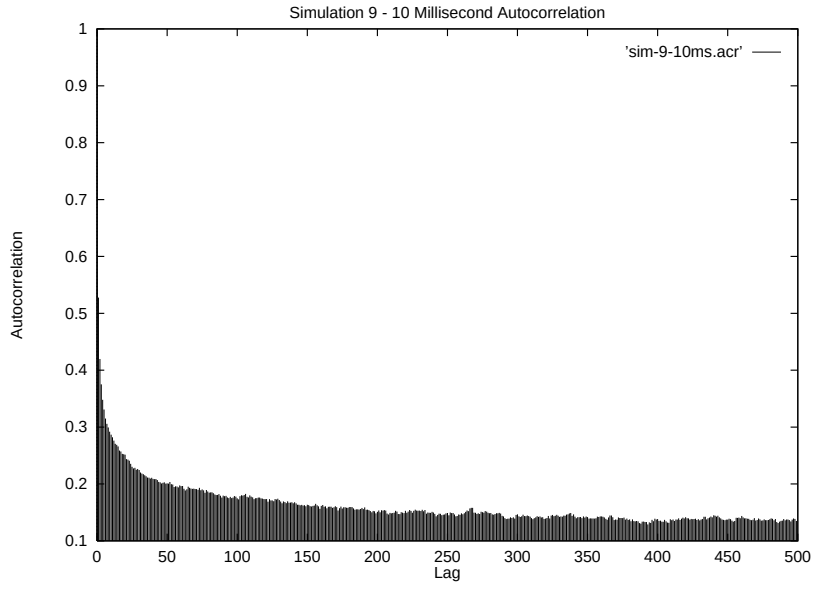

Figure 5.31: Autocorrelation of 100 Superimposed $t_2 - distribution$ Distributed Renewal Process Simulation

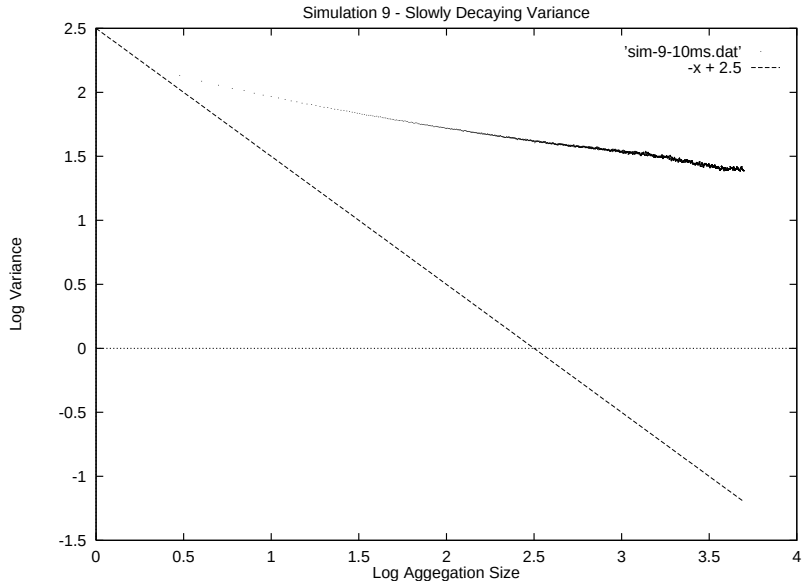

Figure 5.32: Slowly decaying variance plot of 100 Superimposed $t_2 - distribution$ Distributed Renewal Process Simulation

Chapter 6

Conclusion

6.1 What was achieved

6.1.1 Modest resources suffice

The results produced in the thesis were done using very limited hardware. This shows that most people with modest hardware can reproduce these results for their own networks, widening the applicability and usefulness of research in this area.

The software used to produce the results is also of modest proportions. Apart from the graph plotting, which was done using GNUplot, all the analysis software was written by myself and is compilable on any standard C++ compiler under Unix. This avoids the need for specialised statistical packages and helps with making the tools portable to other operating systems and platforms.

While "super computer" processing power is not needed, the calculating of the slowly decaying variance and autocorrelation plots is computationally intensive and a minimum of a 80486 processor running at 66 MHz is recommended.

A large amount of storage space is also important, especially if you want to keep the results even for a short time. While it is possible to reproduce all the results from the trace file it is a tedious process. Some of the simulations, most notably the superimposed processes, also produce very large (tens of megabytes) trace files.

6.1.2 Samples available for future reference

A set of traces were taken from around the University campus. This provided us with a snapshot of the network in 1994, and allowed for comparisons between segments and protocols.

This work could usefully be repeated in – say – 2 years, so as to observe whether the traffic behaviour changes as the network matures and traffic levels increase.

6.1.3 The Bellcore results are reproducible

By reproducing the work done at Bellcore [1] this thesis has strengthened both pieces of work and given strong support to the conclusions they made. The existence of fractal (self-similar) behaviour within Ethernet traffic has strong evidence to support it. This should

cause a reassessment of many of the classic models currently used for congestion prediction in data networks.

Both the mathematics in chapter 3 and the simulations in chapter 5 tie the theoretical models to the observed results. The evidence clearly supports the notion that heavy-tailed renewal processes (renewal processes that have a heavy-tailed inter-renewal distribution) and superimposed processes produce self-similar (fractal) behaviour.

6.1.4 Producing self-similar behaviour through simulation is not difficult

The work in chapter 5 shows that it is possible to produce packet traces with self-similar behaviour without difficult or large amounts of computer processing power.

This gives us a tool to empirically examine self-similar traffic from various mathematical models. It also enables people to gain an understanding of how heavy-tailed renewal processes behave (and insight into infinite moment distributions). Such simulation could be used as an effective learning tool.

6.2 What was not achieved

6.2.1 Bugs in the tracing program

There were several bugs in the tracing program. They were not removed because of time restraints and because they did not affect the results. As this program will be used in the future to continue to collect network traffic data around the University these bugs need to be removed.

The tracing program is also very rudimentary in its user interface. If this program is to be distributed beyond the experimental stage and into public use then documentation is required and the user interface should be improved.

6.2.2 Unknown physical processes

While a strong theoretical model for the observed results now exists there is still little knowledge about how or why the self-similar behaviour occurs. The merging of multiple hosts sending onto the network is one explanation but without some process corresponding to a heavy-tailed renewal process it is only a weak conjecture.

This problem was encountered by the Bellcore researchers and they did not manage to give a conclusive answer. Without the ability to relate the theoretical results back to known physical processes it will not be possible to influence the traffic flow behaviour. This could be important when trying to deal with congestion at a source level.

6.3 Possible follow up work

6.3.1 Effects of self-similar traffic on network controls

The existence of self-similar behaviour may require an examination of current congestion and flow control mechanisms. As the amount of network traffic increases and data networks

become even more common it is important that the problems with packet delivery and error correction using flow controls are addresses.

With the emergence of very busty traffic as normal event on data networks, which such services as compressed video and music on demand, congestion control and timely delivery become more and more important. The current methods of determining buffering within packet switches and re-transmission ordering may have to examined in light of the advanced in the theoretical models of self-similar traffic.

6.3.2 Looking at network protocol behaviour

This involves looking at the behaviour bottom up by attempting to construct accurate models of how network protocols behave. Work needs to be done on discovering where the self-similar behaviour originates. Network protocol behaviour is extremely complicated in real life because of the complex interaction within operating systems. A simple model of a network and protocols which produces self-similar behaviour would be a valuable tool in understanding what is happening.

Looking at each individual protocol was only done superficially in this thesis, leaving room for research into how the traffic of each protocol differs and the possible reasons behind such differences.

6.3.3 Further references

Below are a collection of references that may be of interest [11] [2] [12] [20] [19]. On important reference which was only recently become available (as of February 1995 in New Zealand) is [21]. One of its authors also co-wrote some of the Bellcore papers [9] [8] [10].

Appendix A

Underlying Mathematics

A.1 Random variables

A.1.1 Definition

A random variable is a variable which can take a selection of values, either countable (whether finite or infinite), or uncountable, and is described as *discrete* or *continuous* respectively.

Associated with every random variable is a collection of probability functions. For all random variables there is the *probability distribution function* which is defined as $F_X(x) = P(X \leq x)$. For discrete random variables there is also the *probability function* defined as $f_X(x) = P(X = x)$. The continuous analogue is the *probability density function* f_X. These are related in the following way. For discrete random variables

$$F_X(x) = P(X \leq x) = \sum_{j=-\infty}^{x} P(X = j)$$

and for continuous random variables

$$F_X(x) = P(X \leq x) = \int_{-\infty}^{x} f_X(t)dt$$

A.1.2 Moments of random variables

Moments are values which describe the behaviour of a random variable. These include the *mean* and *variance*. These are calculated from the probability function or probability density function. The first moment is the mean μ, which gives the *average* or expected output of a random variable.

$$\mu = E[X] = \sum_{t=-\infty}^{\infty} t f_X(t)$$

for discrete random variables and for continuous random variables

$$\mu = E[X] = \int_{-\infty}^{\infty} t f_X(t)dt$$

The next moment is the variance $\text{Var}[X] = \sigma^2$ where σ is the *standard deviation*. This is a measure of the *spread* of a distribution given by how far from the mean the values lie.

This is calculated by $E[(X-\mu)^2]$. It is also known as the second moment. Higher order moments are calculated by expectation of higher orders of X.

$$\sigma^2 = E[(X-\mu)^2] = E[X(X-1)] - \mu(\mu-1) = \sum_{t=-\infty}^{\infty} x(x-1)f_X(t) - \mu(\mu-1)$$

for discrete random variables and for continuous random variables

$$\sigma^2 = E[(X-\mu)^2] = E[X^2] - \mu^2 = \int_{-\infty}^{\infty} t^2 f_X(t)dt - \mu^2$$

A.1.3 Collections of random variables

Often more than one random variable is required. For this we construct vectors of random variables $X = (X_1, X_2, \ldots, X_{n-1}, X_n)$. It is common for the X_i's to be *independent and identically distributed (iid).*

A.1.4 Moment generating functions

A.1.4.1 Definition

A *Moment generating function* (mgf) is a transform on a probability function or probability density function for discrete and continuous distributions respectively. It is defined by

$$\mathrm{M}_X(t) = \mathrm{E}[e^{Xt}]$$

For discrete distributions this equates to

$$\mathrm{M}_X(t) = \mathrm{E}[e^{Xt}] = \sum_{x=-\infty}^{\infty} e^{xt} f_X(x)$$

and for continuous distributions

$$\mathrm{M}_X(t) = \mathrm{E}[e^{Xt}] = \int_{-\infty}^{\infty} e^{xt} f_X(x)dx$$

A.1.4.2 Sums of independent random variables

If $X_i : i = 1 \ldots n$ are independent random variables and $Y = \sum_{i=1}^{n} X_i$ then the moment generating function of Y is

$$M_Y(t) = M_{X_1+\cdots+X_n}(t) = \prod_{i=1}^{n} M_{X_i}(t)$$

A.2 Poisson processes

A.2.1 Definition

The Poisson process is a model for events (arrivals/occurrences) that happen randomly in time or space. Suppose that arrivals occur randomly in the interval $(0, t]$. Let N(t) be the number of events by time t.

A Poisson process with intensity (or rate) $\lambda, 0 < \lambda < \infty$ is a process

$$N = N(t) : t \geq 0 \text{ taking values in } S = 0, 1, 2, 3, \ldots$$

- $N(0) = 0$; if $s \leq t$ then $N(s) \leq N(t)$.
- If $s < t$ then $N(t) - N(s)$ in the interval $(s, t]$ is independent of the number and times of arrivals during $(0, t]$.
- The number of events in any interval of length t is distributed as a Poisson(λt) random variable where

$$P(N(t) = j) = f_X(j) = \begin{cases} \frac{e^{-\lambda t}(\lambda t)^j}{j!} & j \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

  with $\mathrm{E}[X] = \lambda t$ and $\mathrm{Var}[X] = \lambda t$.

Poisson processes often arise in operations research and are commonly used as a first approximation to an arrival process. The behaviour of the Poisson process is very well understood.

A.2.2 Inter-arrival times

The number of events within a given time period has a Poisson distribution. It can be shown that the times between subsequent events are exponentially distributed.

$$f_X(x) = \begin{cases} \lambda e^{-\lambda x} & x > 0 \\ 0 & \text{otherwise} \end{cases}$$

with $\mathrm{E}[X] = \frac{1}{\lambda}$ and $\mathrm{Var}[X] = \frac{1}{\lambda^2}$.

Using this it is possible to generate sample realisations of the Poisson process for statistical analysis.

A.2.3 Simple model using Poisson distribution

In the simplest model we just generate pseudo packets with exponential inter-arrival times. This is done by transforming a uniform $U(0, 1)$ distribution, which is just the standard C *random()* function, to an exponential distribution Exp(λ).

A.2.4 Moment Generating Function

The moment generating function of a Poisson distribution X_i with parameter $\lambda_i t$ is

$$\mathrm{M}_X(s) = \mathrm{E}[e^{Xs}] = e^{\lambda t(e^s - 1)}$$

hence the aggregation of several Poisson distributions is

$$\mathrm{M}_X(s) = \prod_{i=1}^{n} e^{\lambda_i t(e^s - 1)} = e^{\lambda^* t(e^s - 1)}$$

where $\lambda^* = \sum_{i=1}^{n} \lambda_i$, giving us a Poisson distribution with parameter $\lambda^* t$.

A.3 Common distribution functions

A.3.1 Exponential

The exponential(λ) distribution has the density function

$$f_X(x) = \begin{cases} \lambda e^{-\lambda x} & x \geq 0 (\lambda > 0) \\ 0 & x < 0 \end{cases}$$

and distribution function

$$F_X(x) = \begin{cases} 1 - e^{-\lambda x} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

A.3.2 Gamma

The gamma(k, λ) distribution has the density function

$$f_X(x) = \begin{cases} \frac{\lambda^k}{\Gamma(k)} x^{k-1} e^{-\lambda x} & x \geq 0 (\lambda > 0, k > 0) \\ 0 & x < 0 \end{cases}$$

A.3.3 Chi-Squared

The chi-squared with ν degrees of freedom χ^2_ν has the density function

$$f_X(x) = \begin{cases} \frac{1}{2^{\frac{\nu}{2}}\Gamma(\frac{\nu}{2})} x^{\frac{\nu}{2}-1} e^{-\frac{x}{2}} & x \geq 0 (\nu > 0) \\ 0 & x < 0 \end{cases}$$

A.3.4 Normal

The normal(μ, σ^2) has the density function

$$f_X(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

with no closed integral form except when the bounds are trivial $(0, -\infty, +\infty)$.

A.3.5 Student's t

Student's t-Distribution with ν degrees of freedom t_ν has the density function

$$f_T(t) = \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\sqrt{\nu\pi}\Gamma\left(\frac{\nu}{2}\right)\left(1 + \frac{t^2}{v}\right)^{\frac{\nu+1}{2}}} \quad -\infty < t < \infty$$

A.3.6 Cauchy

A Cauchy distribution is a T distribution with $\nu = 1$

$$f_T(t) = \frac{1}{\pi(1+t^2)} \quad -\infty < t < \infty$$

A.4 Example of aggregated Poisson processes

Aggregating multiple Poisson random variables $(X_1, X_2, \ldots)$ gives another Poisson distribution. Let $\bar{X} = \sum_{i=1}^{n} \frac{X_i}{n}$. Using moment generating functions we see

$$\mathrm{M}_{\bar{X}}(s) = \left[\mathrm{M}_X(\frac{s}{n})\right]^n$$

so with Poisson random variables we get

$$\mathrm{M}_{\bar{X}}(s) = \left[e^{\lambda t(e^{\frac{s}{n}}-1)}\right]^n = e^{n\lambda t(e^{\frac{s}{n}}-1)}$$

Taking the log of $\mathrm{M}_{\bar{X}}(s)$ we get

$$C_{\bar{X}}(s) = n\lambda t(e^{\frac{s}{n}} - 1)$$

Differentiating this once gives us

$$C'_{\bar{X}}(s) = \lambda t e^{\frac{s}{n}}$$

and again to give

$$C''_{\bar{X}}(s) = \frac{\lambda t}{n} e^{\frac{s}{n}}$$

hence

$$\mu_{\bar{X}} = C'(0-) = \lambda t$$

and

$$\sigma^2_{\bar{X}} = C''(0-) = \frac{\lambda t}{n}$$

Appendix B

Source Code

B.1 The Network tracer

B.1.0.1 common.h

```
#ifndef _common_
#define _common_

#include "queue.h"
#include <time.h>

/* ------------------------------------------------------------------------
-  essential defined
------------------------------------------------------------------------ */
#define TRUE 1
#define true TRUE
#define FALSE 0
#define false FALSE

typedef unsigned long longword; /* 32 bits */
typedef unsigned short word; /* 16 bits */
typedef unsigned char byte; /*  8 bits */

#define htons(x) intel16(x)
#define htonl(x) intel(x)
#define ntohs(x) intel16(x)
#define ntohl(x) intel(x)
#define checksum(p, len) inchksum(p, len)

/* ------------------------------------------------------------------------
-  packet driver defines
------------------------------------------------------------------------ */
#define MAX_INTF 6
#define MAX_PCOLS 6

#define PD_USABLE 1

#define PD_CLASS_DIX 1
#define PD_CLASS_SLIP 6
#define PD_CLASS_8023 11

#define IP_ADDR_LEN 4
#define ENET_ADDR_LEN 6
#define SLIP_ADDR_LEN 0
#define PPP_ADDR_LEN 0
#define MAX_ADDR_LEN 10

#define ENET_HDR_LEN 14
#define ENET_TLR_LEN 2
#define ENET_MTU 1500

#define SLIP_HDR_LEN 2
#define SLIP_TLR_LEN 2
#define SLIP_MTU 500

#define PPP_HDR_LEN 4
#define PPP_TLR_LEN 2
#define PPP_MTU 1500
```

```
#define MAX_KEEP_PKT 58
#define MAX_KEEP_CNT 400

/* ---------------------------------------------------------------------
-  packet driver type defintions
--------------------------------------------------------------------- */
typedef struct {
  int intf, index, flags;
  word type;
  word handle;
  int addrLen;
  byte addr[MAX_ADDR_LEN];
  byte bcastAddr[MAX_ADDR_LEN];
  long pktRcv, pktSent;
  queue rcvQ;
  } protoRec;

typedef struct {
  word vector;
  word PDNumber;
  int number;
  int functionality;
  int class; /* Packet driver class */
  int version;
  int type;
  int flags; /* flags */
  int nProto; /* # protocols attached */
  int intfType;
  int hardware_type; /* ARP hardware type */
  int hdrLen; /* size of header in octets */
  int tlrLen; /* size of trailer in octets */
  int maxPktLen; /* MTU, 1500 ethernet */
  int functions; /* unicast, broadcast, multicast */
  } intfRec;

typedef struct {
  int nIntf;
  int maxHdr, maxMTU;
  intfRec intf[MAX_INTF];
  protoRec pcol[MAX_INTF][MAX_PCOLS];
  queue rcvQ[MAX_PCOLS];
} PDRec;

/* ---------------------------------------------------------------------
-  assembly prototypes
--------------------------------------------------------------------- */
extern unsigned long intel(unsigned long val);
extern unsigned int intel16(unsigned int val);
extern word inchksum(void *ptr, int len);

/* ---------------------------------------------------------------------
-  packet driver prototypes
--------------------------------------------------------------------- */
extern PDRec *PDInit();
extern word PDAddProtocol(int intf);
extern void PDRemoveProtocol(int intf, word handle);
extern int PDCheckQueue(void);

#endif
```

B.1.0.2 asmpkt.asm

```
;
;
;  Usage :
;     -pktentry
;           _pktasminit( void far * buffers, int maxbufs, int buflen)
;
;  (c) 1991 University of Waterloo,
;           Faculty of Engineering,
;           Engineering Microcomputer Network Development Office
;
;  version
;
;    0.1    22-May-1992   E. P. Engelke
;
;
        include masmdefs.hsm
```

```
        include model.hsm

ORIGINAL  equ  0         ; AU

cextrn pkt_rcv_1
cextrn pkt_rcv_2
cextrn test1
cextrn test2
codedef ASMPKT
datadef

datastart
STACK_FILL      equ     0CC33H  ; Stack for AU interrupt handler
ISTACK_SIZE     equ     512     ; bytes
        dw      256 dup (STACK_FILL)
END_ISTACK      equ     $
dataend

cstart  ASMPKT

helper_ds DW ?
helper_ss DW ?
helper_sp DW ?
interface DW ?

cproc pktentry0       ; AU: packet receive code from pcip
pushf
cli
mov interface, 0
jmp entry
cendp pktentry0

cproc pktentry1       ; AU: packet receive code from pcip
pushf
cli
mov interface, 1
jmp entry
cendp pktentry1

proc main
ASSUME DS:NOTHING, SS:NOTHING, ES:NOTHING
; Squirrel away the packet driver's critical registers...

entry: mov helper_ds,ds
mov helper_ss,ss
mov helper_sp,sp

or ax,ax ; First or second call?
; DON'T DO ANYTHING TO CHANGE FLAGS
; UNTIL THE JNZ SECOND_CALL, BELOW

; Get a stack and a way to reach our data segment...
; MOV instructions don't affect the flags
mov ax,DGROUP
mov ss,ax
ASSUME SS:DGROUP
mov ds,ax
ASSUME DS:DGROUP
mov sp,OFFSET DGROUP: END_ISTACK

; Save some of the packet driver's other registers...
pushf ; Preserve direction-flag and int-flag

cld ; Microsoft C needs this, doesn't
; harm the flags set by the or ax,ax
; above

jnz short second_call ; Check result of or ax,ax
; way above
push cx ; len
ccall pkt_rcv_1 ; First call returns a far pointer
; in dx (segment) & ax (offset)
add sp,4 ; Remove the parameters from the stack
mov es,dx ; segment of far ptr returned
mov di,ax ; offset of far ptr returned
help_ret:
popf ; Restore int status and direction flag
mov ds,helper_ds
ASSUME DS:NOTHING
mov ss,helper_ss
ASSUME SS:NOTHING
```

```
mov sp,helper_sp

popf
retf

ASSUME DS:DGROUP, SS:DGROUP
second_call:
push helper_ds ; Segment part of buffer
push si ; Offset part of buffer
push cx ; len
push bx ; handle
push interface ; interface
ccall pkt_rcv_2 ; Nothing returned
add sp,10 ; Remove the parameters from the stack
jmp short help_ret
endp    main

cend    ASMPKT
        end
```

B.1.0.3 hwtimer.asm

```
;  2100, Tue 19 Apr 94
;
;  HWTIMER.ASM:  High-resolution time-of-day routines
;
;  Copyright (C) 1994,  Nevil Brownlee,
;  Computer Centre, The University of Auckland
;
;  void hwt_init(void);  /* Initialise timer routines */
;  void hwt_fin(void);   /* Restore BIOS timer routines */
;
;  unsigned long hw_time(void);  /* Time in 1/256 clock tick units */
;
include ndos.mac ; Assembler Macros
environ test
SETX
begdata hwtimer
;
t0_st dw 0 ; 0ABCH = initialised
;
enddata hwtimer
begcode hwtimer
;
old_t0_int  dd  ? ; Stored in code segment
; ;    This is because we don't know
; ;    where ds will be pointing when we
; ;    call the BIOS timer0 handler!
;
t0count  dw 0 ; Nbr of timer interrupts
db 0
;
t0int proc near
inc word ptr cs:t0count ; Our IRQ0 (timer 0) handling
jnz int_exit
inc byte ptr cs:t0count+2
;
int_exit:
pushf
call cs:old_t0_int ; Goto BIOS time-of-day handler
iret
t0int endp
;
;
subrt   hwt_init
; ===== ========
cmp     [t0_st],0ABCH    ; Interrupts already initialised?
je      hwtiset
;
push    ds              ; Save ds
mov     ax,0            ; Point to bottom of memory
mov     ds,ax
mov     bx,0020H ; Address for timer0 interrupt
mov     cx,offset t0int
mov     dx,cs
cli                     ; Disable interrupts
xchg    [bx],cx         ; Set kb interrupt offset
xchg    [bx+2],dx       ;    and segment
sti                     ; Enable interrupts
```

```
pop     ds              ; Restore ds
mov     word ptr cs:old_t0_int,cx ; Save kb interupt offset
mov     word ptr cs:old_t0_int+2,dx ;    and segment
mov     [t0_st],0ABCH
;
hwtiset:
mov al,24H ; Counter 0, MSB only, mode 2
out 43H,al ;   Decr in ones, interrupt every cycle
mov al,0
out 40H,al ; Set MSB
;
xor ax,ax ; Zero our counter
mov     word ptr cs:t0count,ax
mov byte ptr cs:t0count+2,al
endsub  hwt_init
;
;
subrt   hwt_fin
; ===== =======
cmp     [t0_st],0ABCH    ; Interrupts initialised?
jne     hwtrst
;
mov     cx,word ptr cs:old_t0_int
mov     dx,word ptr cs:old_t0_int+2
push    ds              ; Save ds
mov     ax,0            ; Point to bottom of memory
mov     ds,ax
mov     bx,0020H ; Address for timer0 interrupt
cli                     ; Disable interrupts
mov     [bx],cx         ; Restore MSDOS kb interrupt offset
mov     [bx+2],dx       ;    and segment
sti                     ; Reenable interrupts
pop     ds              ; Restore ds
mov     [t0_st],0000H
;
hwtrst:
mov al,36H ; Counter 0, LSB+MSB, mode 3
out 43H,al ;   Decr in ones, interrupt every second cycle
mov al,0
out 40H,al ; Set LSB
out 40H,al ;    and MSB
endsub  hwt_fin
;
;
function  hw_time
;       ========  =======
mov al,00H ; Counter 0, Counter Latch
cli ; Disable interrupts
out 43H,al
in al,40H ; MSB in al
mov ah,byte ptr cs:t0count
mov dx,word ptr cs:t0count+1
sti ; Enable interrupts
;
mov bl,al
mov al,0FFH ; Counter MSB in 4th byte
sub al,bl
add al,2
endfn  hw_time ; Result returned in dx,ax
;
;
endcode hwtimer
end
```

B.1.0.4 driver.c

```
/* ************************************************************************
*
*  Packet driver interface code
*
*  version 0.0.1
*    Written by Ross Alexander, 17 Dec 1992
*    Modefied for a general packet receiver on 29 March 1994
*
************************************************************************** */

#pragma inline

#include <dos.h>
```

```
#include <string.h>
#include <stdio.h>
#include <time.h>

#include "queue.h"
#include "common.h"

#define MAX_BUFFER 50
#define GRAB_SIZE 40

/* --------------------------------------------------------------------
-packet driver defines -
-------------------------------------------------------------------- */
#define CARRY 1

#define INT_FIRST 0x60
#define INT_LAST 0x80

#define PKT_LINE "PKT DRVR"
#define MAX_DRIVERS 6
#define MAX_PROTOCOLS 5

#define PD_DRIVER_INFO 0x01FF
#define PD_ACCESS_TYPE 0x0200
#define PD_RELEASE_TYPE 0x0300
#define PD_SEND_PACKET 0x0400
#define PD_GET_ADDRESS 0x0600
#define PD_SET_RCV_MODE 0x1400

#define PD_FN_BASIC 1
#define PD_FN_EXT 2
#define PD_FN_HP 5
#define PD_FN_EXT_HP 6

#define PD_RCV_OFF 1
#define PD_RCV_UNI 2
#define PD_RCV_BCAST 3
#define PD_RCV_LIM_MCAST 4
#define PD_RCV_ALL_MCAST 5
#define PD_RCV_ALL 6
/* --------------------------------------------------------------------
-  external function prototypes
-------------------------------------------------------------------- */
extern long hw_time(void);

/* --------------------------------------------------------------------
-packet driver external prototypes -
-------------------------------------------------------------------- */
extern void pktentry0();
extern void pktentry1();
extern void pktentry2();
extern void pktentry3();
/* --------------------------------------------------------------------
-packet driver global variables
-------------------------------------------------------------------- */
void  (*pkentry[])() = { pktentry0, pktentry1 };
PDRec pd;

long cur_bf, bf_cnt = 0, pkt_count = 0, init_tick;
int refresh = 0;
byte *buffer[2];

/* ********************************************************************
*
* pkt_rcv_1
*
*  Function used to received packets from the packet drivers.
*
*  Inputs:
*    unsigned int handle - the handle the packet was received on (each
*                          protocol/port combination is given a handle by
*                          the packet drivers)
*    unsigned int len - the length of the incoming packet
*
*    Outputs:
*      unsigned char* - a pointer to a buffer where the can be stored
*                       in memory
*
******************************************************************** */
byte far *pkt_rcv_1(word len)
{
```

```
  byte *tmp;
  if (!(bf_cnt < MAX_KEEP_CNT))
    {
      bf_cnt = 0;
      refresh = cur_bf + 1;
      cur_bf = 1 - cur_bf;
    }
  tmp = buffer[cur_bf] + (sizeof(clock_t) + sizeof(int) + MAX_KEEP_PKT)*bf_cnt;
  *((long*)(tmp)) = htonl(hw_time());
  *((int*)(tmp + sizeof(clock_t))) = htons(len);
  if (MAX_KEEP_PKT)
    return tmp + (sizeof(clock_t) + sizeof(int));
  else
    {
      bf_cnt++;
      pkt_count++;
      return NULL;
    }
}

/* ************************************************************************
*
*  pkt_rcv_2
*
*  Called once packet has been stored in main memory.  Currently the code
*  does a linear search through the ports and protocols to match the handle.
*  Because of the limited number of ports and protocols (not more than twenty
*  at most), there is little reason to make the search more efficient.
*
*  Inputs:
*    word intf - the interface the packet has arrived on
*    word handle - the handle the packet was received on
*    word len - the length of the incoming packet
*    byte *buff - the address of the buffer
*
************************************************************************ */
void pkt_rcv_2()
{
  bf_cnt++;
  pkt_count++;
}

/* ************************************************************************
*
*  CheckDrivers
*
*  Runs through the list of available packet drivers and finds those ones
*   which are usable.  Currently on Dec Intel Xerox ethernet cards are usable.
*
*  Output:
*    int - the number of usable packet drivers.
*
************************************************************************ */
int CheckDrivers(void)
{
  int i;
  struct REGPACK regs;
  word packetInt;
  longword far *interrupts;
  char far *temp;

  interrupts = 0L; /* a PC hack looking through low memory */
  i = 0;

  for (packetInt = INT_FIRST; packetInt <= INT_LAST; packetInt++)
    {
      temp = (char far *)(interrupts[packetInt]);
      if (!strcmp(&temp[3], PKT_LINE))
{
  pd.intf[i].vector = packetInt;
  regs.r_ax = PD_DRIVER_INFO;
  intr(pd.intf[i].vector, &regs);
  if (regs.r_flags & CARRY)
    exit(1);
  pd.intf[i].functionality = regs.r_ax & 0xFF;
  pd.intf[i].version = regs.r_bx;
  pd.intf[i].class = regs.r_cx >> 8;
  pd.intf[i].type = regs.r_dx;
  pd.intf[i].PDNumber = regs.r_cx & 0xFF;

  printf("Packet driver at 0x%0X type %d\n",pd.intf[i].vector,
```

```
          pd.intf[i].class);
    switch(pd.intf[i].class)
    {
    case PD_CLASS_DIX:
      pd.intf[i].flags = PD_USABLE;
      pd.intf[i].hdrLen = ENET_HDR_LEN;
      pd.intf[i].tlrLen = ENET_TLR_LEN;
      pd.intf[i].maxPktLen = ENET_MTU;
      i++;
      break;
    case PD_CLASS_SLIP:
      pd.intf[i].flags = PD_USABLE;
      pd.intf[i].hdrLen = SLIP_HDR_LEN;
      pd.intf[i].tlrLen = SLIP_TLR_LEN;
      pd.intf[i].maxPktLen = SLIP_MTU;
      i++;
      break;

    default:
      pd.intf[i].flags = 0x00;
      break;
    }
}
    }
  return i;
}

/*
*/
int DriverAdd(int i)
{
  struct REGPACK regs;
  int error, result, addr_len;
  word temp, handle;
  byte addr[MAX_ADDR_LEN];
  temp = ntohs(0x0800);

  regs.r_ax = PD_ACCESS_TYPE | PD_CLASS_DIX;
  regs.r_bx = 0xFFFF; /* pd.intf[i].type; */
  regs.r_cx = 0;
  regs.r_dx = pd.intf[i].PDNumber;
  regs.r_ds = FP_SEG(&temp);
  regs.r_si = FP_OFF(&temp);
  regs.r_es = FP_SEG(pkentry[i]);
  regs.r_di = FP_OFF(pkentry[i]);
  intr(pd.intf[i].vector, &regs);
  handle = regs.r_ax;

  error = regs.r_flags & CARRY;
  if (error)
    {
      printf("Error in ACCESS_TYPE\n");
      return 0;
    }
  regs.r_ax = PD_GET_ADDRESS;
  regs.r_bx = handle;
  regs.r_cx = MAX_ADDR_LEN;
  regs.r_es = FP_SEG(addr);
  regs.r_di = FP_OFF(addr);
  intr(pd.intf[i].vector, &regs);
  addr_len = regs.r_cx;

  switch(pd.intf[i].functionality)
    {
    case PD_FN_EXT:
    case PD_FN_EXT_HP:
      {
regs.r_ax = PD_SET_RCV_MODE;
regs.r_bx = handle;
regs.r_cx = PD_RCV_ALL;
intr(pd.intf[i].vector, &regs);
break;
      }
    case PD_FN_BASIC:
    case PD_FN_HP:
    default:
      break;
    }
  return handle;
}
```

```
void DriverRemove(int i, word handle)
{
  struct REGPACK regs;
  regs.r_ax = PD_RELEASE_TYPE;
  regs.r_bx = handle;
  intr(pd.intf[i].vector, &regs);
}

/* ************************************************************************
*
*  PDInit
*
************************************************************************ */
PDRec *PDInit()
{
  int numDrivers, maxHdr, maxLen, store_len;
  register int i, temp;

  pd.nIntf = CheckDrivers();

  for (maxHdr = maxLen = i = 0; i < pd.nIntf; i++)
    {
      temp = pd.intf[i].maxPktLen + pd.intf[i].hdrLen + pd.intf[i].tlrLen;
      maxLen = maxLen > temp ? maxLen : temp;
      maxHdr = maxHdr > pd.intf[i].hdrLen ? maxHdr : pd.intf[i].hdrLen;
    }
  pd.maxMTU = maxLen;
  pd.maxHdr = maxHdr;
  maxLen += sizeof(clock_t) + sizeof(int);
  store_len = MAX_KEEP_PKT + sizeof(clock_t) + sizeof(int);
  buffer[0] = malloc(MAX_KEEP_CNT * store_len + maxLen);
  buffer[1] = malloc(MAX_KEEP_CNT * store_len + maxLen);
  cur_bf = 0;
  return &pd;
}
```

B.1.0.5 main.c

```
/* ************************************************************************
*
*  main
*
*  version 0.0.1
*    27 March 1993
*    Add hwtimer code for accurate timings based on the Interval counter.
*
*
************************************************************************ */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <time.h>
#include <bios.h>
#include <dos.h>
#include "common.h"

extern void hwt_init(void);  /* In hwtimer.asm */
extern void hwt_fin(void);
extern unsigned long hw_time(void);

static PDRec *pd;

extern int refresh, cur_bf;
extern long pkt_count;
extern long bf_cnt;
extern byte *buffer[2];

int main(int argc, char *argv[])
{
  FILE *stream;
  int cnt = 0, i, store_len;
  time_t seconds;
  char *fname = "trace.icr";
  word handle;
  word hdr[4];

  for (i = 0; i < argc; i++)
    if (argv[i][0] == '-')
```

```
      switch (argv[i][1])
        {
        case 'o':
          fname = argv[++i];
          break;
        }

  pd = PDInit();
  hwt_init();
  if (!pd->nIntf)
    {
      printf("No usable packet drivers were found.\n");
      return 1;
    }
  handle = DriverAdd(0);

  stream = fopen(fname, "ab");
  hdr[0] = htons(sizeof(clock_t));
  hdr[1] = htons(sizeof(int));
  hdr[2] = htons(MAX_KEEP_PKT);
  hdr[3] = htons(0);
  fwrite(hdr, sizeof(word), 4, stream);

  store_len = MAX_KEEP_PKT + sizeof(clock_t) + sizeof(int);
  do
    if (refresh)
      {
        fwrite(buffer[refresh-1], store_len, MAX_KEEP_CNT, stream);
        refresh = 0;
        cnt++;
      }
  while (!kbhit() && (cnt < 1000));
  DriverRemove(0, handle);
  fwrite(buffer[cur_bf], store_len, bf_cnt, stream);
  fclose(stream);
  hwt_fin();
  printf("\n---------------------------\n");

/*
  for (i = 0; i < pkt_count; i++)
    {
      int j;
      for (j = 0; j < 36; j++)
        printf("%02X", buffer[i*store_len + j]);
      printf("\n");
    }
*/
  printf("Pkt count = %ld\n", pkt_count);
  return 0;
}
```

B.1.1 Time error correction program

B.1.1.1 correct.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <netinet/in.h>

#define HWT_TICK 0.2145527

typedef struct {
  short time, count, data, blank;
  int total;
} file_hdr;

void get_hdr(FILE *stream, file_hdr *hdr)
{
  fread(&hdr->time, 1, sizeof(short), stream);
  fread(&hdr->count, 1, sizeof(short), stream);
  fread(&hdr->data, 1, sizeof(short), stream);
  fread(&hdr->blank, 1, sizeof(short), stream);

  hdr->total = 0;
  hdr->total += hdr->time = ntohs(hdr->time);
  hdr->total += hdr->count = ntohs(hdr->count);
  hdr->total += hdr->data = ntohs(hdr->data);
```

```
}

void put_hdr(FILE *stream, file_hdr *hdr)
{
  short time, count, data, blank;
  time = htons(hdr->time);
  count = htons(hdr->count);
  data = htons(hdr->data);
  blank = htons(hdr->blank);

  fwrite(&time, 1, sizeof(short), stream);
  fwrite(&count, 1, sizeof(short), stream);
  fwrite(&data, 1, sizeof(short), stream);
  fwrite(&blank, 1, sizeof(short), stream);
}

void fix(FILE *stream, FILE *output, file_hdr *hdr, int test)
{
  unsigned char *bf;
  unsigned long send_time = 0, send_old, recv_time = 0, recv_old;
  int cnt = 0;

  bf = malloc(hdr->total);
  while (fread(bf, 1, hdr->total, stream))
    {
      int i;
      send_old = send_time;
      recv_old = recv_time;
      recv_time = ntohl(*(long*)(bf));
      send_time = ntohl(*(long*)(bf + 24));

      if (send_old > send_time)
send_time += 256;
      if (recv_old > recv_time)
recv_time += 256;

      recv_time = recv_time * HWT_TICK;
      send_time = send_time * HWT_TICK;

      (*(long*)(bf)) = recv_time;
      if (test) (*(long*)(bf + 24)) = send_time;

      fwrite(bf, 1, hdr->total, output);
    }
  free(bf);
}

extern int optind;
extern char *optarg;

int main(int argc, char *argv[])
{
  int ch, speed_test = 0;
  file_hdr hdr;
  char *output_name = "correction.out";

  while ((ch = getopt(argc, argv, "so:")) != EOF)
    switch(ch)
      {
      case 'o':
output_name = optarg;
break;
      case 's':
speed_test = 1;
break;
      }
  for (;optind < argc; optind++)
    {
      FILE *input, *output;
      char tmp[256], fname[256];
      int length;

      length = strlen(argv[optind]);
      if ((length > 4) && !strcmp(argv[optind] + length - 4, ".icr"))
length -= 4;
      strncpy(fname, argv[optind], length);
      fname[length] = '\0';

      strcpy(tmp, fname);
      strcat(tmp, ".icr");
      if (!(input = fopen(tmp, "rb")))
```

```
break;

      strcpy(tmp, fname);
      strcat(tmp, ".tra");
      if (!(output = fopen(tmp, "wb")))
break;

      get_hdr(input, &hdr);
      put_hdr(output, &hdr);
      fix(input, output, &hdr, speed_test);
      fclose(input);
      fclose(output);
    }
}
```

B.2 Analysis programs

B.2.1 Common code

B.2.1.1 mathlib.h

```
/* R Mathematical Function Library Header */
/* Copyright 1994, Robert Gentleman and Ross Ihaka */
/* All Rights Reserved */

#include <math.h>
#include <limits.h>
#include <float.h>
#include <errno.h>

#ifndef drand48
extern "C" double drand48();
#endif

#ifndef srand48
extern "C" void srand48(long);
#endif

extern double snorm(void);
extern double sunif(void);
extern double sexp(void);

extern double rgamma(double, double);
extern double rnorm();

/* 30 Decimal-place constants computed with bc */

#ifndef M_1_SQRT_2PI
#define M_1_SQRT_2PI        0.398942280401432677939946059934
#endif

#ifndef M_1_SQRT_2
#define M_1_SQRT_2        0.707106781186547524400844362105
#endif

#ifndef M_PI
#define M_PI         3.141592653589793238462643383276
#endif
```

B.2.1.2 common.h

```
#include <stdio.h>

typedef unsigned char byte;

class TraceFile {
private:
  FILE *stream;

public:
  int lineLen;
  short time_len;
  short size_len;
  short pkt_len;
  short blank;

  TraceFile(FILE *f);
```

```
  ~TraceFile();
  void ReadHdr();
  void WriteHdr();
  int ReadLine(unsigned char *ptr);
};

typedef struct {
  short time_len;
  short size_len;
  short pkt_len;
  short blank;
} file_hdr;


typedef struct {
  int rows, columns;
  void *ptr;
} s_data;

// ---------------------------------------------------------------------------

#define D_EXP 1
#define D_UNI 2
#define D_DET 3
#define D_GAM 4
#define D_CHI 5
#define D_TDI 6
#define D_NOR 7

union DistParam {
  struct {
    double lambda;
  } Exponential;
  struct {
    double alpha;
    double beta;
  } Pareto;
  struct {
    double mean;
  } Uniform;
  struct {
    double value;
  } Deterministic;
  struct {
    double lambda;
    double k;
  } Gamma;
  struct {
    double mean;
    double deviation;
  } Normal;
  struct {
    double freedom;
  } ChiSquared;
  struct {
    double mean;
    double deviation;
    double freedom;
  } T;
  struct {
    double A;
    double B;
    double D;
  } CutOff;
};

extern double uniform();
extern double DistExp(DistParam &);
extern double DistUniform(DistParam &);
extern double DistDet(DistParam &);
extern double DistPareto(DistParam &);
extern double DistNormal(DistParam &);
extern double DistGamma(DistParam &);
extern double DistChiSq(DistParam &);
extern double DistT(DistParam &);

// ---------------------------------------------------------------------------

#ifdef __cplusplus
#include <string.h>
```

```
#define EXT_DEFAULT 0x01

struct Ext {
  char ext[10];
  int id;
  int flags;
};

extern Ext* CheckExtension(char *fname, char *rtn, Ext *exts[]);
#endif

extern void get_hdr(FILE*, file_hdr*);
extern void put_hdr(FILE*, file_hdr*);
extern int ReadFileText(int, s_data*, FILE*);
extern int ReadFileTextFloat(int, s_data*, FILE*);
extern void ReadFileBinary(s_data*, FILE*);
extern void DumpMtx(s_data*);
extern void DumpMtxFloat(s_data*);
```

B.2.1.3 vector.h

```
class Vector {
public:
  Vector();
  Vector(int);
  ~Vector();
  Vector(Vector&);
  int length();
  double &operator[](int);
  Vector operator()();
  Vector operator()(int);
  Vector operator()(int, int);
  Vector &operator=(Vector&);
  void map(double (*)(double));
  double fold(double (*)(double), double (*)(double, double), double);
  Vector aggregate(int);
  void dump();

  double sum_ident();
  double sum_square();

private:
  int size, ref, *refptr;
  double *ptr;
};

extern double f_identity(double);
extern double f_square(double);
extern double f_add(double, double);
```

B.2.1.4 common.cc

```
#include <stdio.h>

typedef unsigned char byte;

class TraceFile {
private:
  FILE *stream;

public:
  int lineLen;
  short time_len;
  short size_len;
  short pkt_len;
  short blank;

  TraceFile(FILE *f);
  ~TraceFile();
  void ReadHdr();
  void WriteHdr();
  int ReadLine(unsigned char *ptr);
};

typedef struct {
  short time_len;
  short size_len;
  short pkt_len;
```

```
  short blank;
} file_hdr;


typedef struct {
  int rows, columns;
  void *ptr;
} s_data;

// ---------------------------------------------------------------------------

#define D_EXP 1
#define D_UNI 2
#define D_DET 3
#define D_GAM 4
#define D_CHI 5
#define D_TDI 6
#define D_NOR 7

union DistParam {
  struct {
    double lambda;
  } Exponential;
  struct {
    double alpha;
    double beta;
  } Pareto;
  struct {
    double mean;
  } Uniform;
  struct {
    double value;
  } Deterministic;
  struct {
    double lambda;
    double k;
  } Gamma;
  struct {
    double mean;
    double deviation;
  } Normal;
  struct {
    double freedom;
  } ChiSquared;
  struct {
    double mean;
    double deviation;
    double freedom;
  } T;
  struct {
    double A;
    double B;
    double D;
  } CutOff;
};

extern double uniform();
extern double DistExp(DistParam &);
extern double DistUniform(DistParam &);
extern double DistDet(DistParam &);
extern double DistPareto(DistParam &);
extern double DistNormal(DistParam &);
extern double DistGamma(DistParam &);
extern double DistChiSq(DistParam &);
extern double DistT(DistParam &);

// ---------------------------------------------------------------------------

#ifdef __cplusplus
#include <string.h>

#define EXT_DEFAULT 0x01

struct Ext {
  char ext[10];
  int id;
  int flags;
};

extern Ext* CheckExtension(char *fname, char *rtn, Ext *exts[]);
#endif
```

```
extern void get_hdr(FILE*, file_hdr*);
extern void put_hdr(FILE*, file_hdr*);
extern int ReadFileText(int, s_data*, FILE*);
extern int ReadFileTextFloat(int, s_data*, FILE*);
extern void ReadFileBinary(s_data*, FILE*);
extern void DumpMtx(s_data*);
extern void DumpMtxFloat(s_data*);
```

B.2.2 Arrival

B.2.2.1 arrival.cc

```
#include <stdio.h>
#include <stdlib.h>
#include <alloca.h>
#include <string.h>
#ifdef sun
#include <sys/types.h>
#endif
#include <netinet/in.h>
#include <unistd.h>

#include "common.h"

#define INTERVAL 10.0 // 10 ms
#define SLOT_MAX 2000

// ---------------------------------------------------------------------------

struct o_struct {
  int time, count, octets;
};

struct Param {
  int verbose;
  int binary;
  int cut;
  double cuttime;
  double slotsize;
  FILE *output;
};

// ---------------------------------------------------------------------------

void CutAggregate(TraceFile &hdr, char *fname, Param &p)
{
  int read = 1, eof = 0, octets = 0, count = 0, i = 1, j = 0, k = 0, tm;
  register double tmp;
  double slot_mod = p.slotsize >= 1000 ? p.slotsize / 1000 : p.slotsize;
  o_struct o;
  FILE *output;
  char filetmp[256];
  byte *bf = new byte [hdr.lineLen];

  while (!eof)
    {
      if (p.binary)
        {
          strcpy(filetmp, fname);
          sprintf(filetmp, "%s-%dms-%d.abf", fname, (int)(p.slotsize), i);
          if (!(output = fopen(filetmp, "wb")))
            return;
        }
      else
        {
          strcpy(filetmp, fname);
          sprintf(filetmp, "%s-%dms-%d.atf", fname, (int)(p.slotsize), i);
          if (!(output = fopen(filetmp, "w")))
            return;
        }
      if (p.verbose) fprintf(stderr, "Opening output file %s\n", filetmp);
      octets = count = 0;
      while (1)
        {
          if (read)
            if (eof = !hdr.ReadLine(bf))
              break;
            else
```

```
            ;
          else
            read = 1;

          tm = *((int*)bf);
          tmp = tm;
          if (tmp > i * p.cuttime)
            {
              read = 0;
              break;
            }
          while (1)
            if (tmp > (j + 1) * p.slotsize)
              {
         if (p.binary)
           {
             o.time = (int)(j * slot_mod);
             o.count = count;
             o.octets = octets;
             fwrite(&o, sizeof(o_struct), 1, output);
           }
         else
           fprintf(output, "%9d %5d %7d\n", (int)(j * slot_mod), count, octets);
         j++;
         k++;
         octets = count = 0;
              }
            else
              {
         octets += ntohs(*(short*)(bf + hdr.time_len));
         count++;
         break;
              }
        }
      octets = count = 0;
      while (i * p.cuttime > (j + 1) * p.slotsize)
        {
          if (p.binary)
            {
              o.time = (int)(j * slot_mod);
              o.count = count;
              o.octets = octets;
              fwrite(&o, sizeof(o_struct), 1, output);
            }
          else
            fprintf(output, "%9d %5d %7d\n", (int)(j * slot_mod), count, octets);
          j++;
          k++;
        }
      if (p.binary)
        {
          int a = 3;
          o.time = (int)(j++ * slot_mod);
          o.count = count;
          o.octets = octets;
          fwrite(&o, sizeof(o_struct), 1, output);
          fwrite(&++k, sizeof(int), 1, output);
          fwrite(&a, sizeof(int), 1, output);
        }
      else
        {
          fprintf(output, "%9d %5d %7d\n", (int)(j++ * slot_mod), count, octets);
        }
      if (p.verbose) fprintf(stderr, "Line count = %d\n", ++k);
      fclose(output);
      i++;
      k = 0;
    }
}

// ---------------------------------------------------------------------------
void Aggregate(TraceFile &hdr, char *fname, Param &p)
{
  int octets = 0, count = 0, j = 0, tm;
  register double tmp;
  double slot_mod = p.slotsize >= 1000 ? p.slotsize / 1000 : p.slotsize;
  o_struct o;
  byte *bf;
  FILE *output;
  char filetmp[256];
```

```
  if (p.binary)
    {
      strcpy(filetmp, fname);
      sprintf(filetmp, "%s-%dms.abf", fname, (int)(p.slotsize));
      if (!(output = fopen(filetmp, "wb")))
        return;
    }
  else
    {
      strcpy(filetmp, fname);
      sprintf(filetmp, "%s-%dms.atf", fname, (int)(p.slotsize));
      if (!(output = fopen(filetmp, "w")))
        return;
    }
  p.output = output;
  if (p.verbose) fprintf(stderr, "Opening output file %s\n", filetmp);

  bf = new byte [hdr.lineLen];

  while (hdr.ReadLine(bf))
    {
      tm = *((int*)bf);
      tmp = tm;
      while (1)
        if (tmp > (j + 1) * p.slotsize)
          {
            if (p.binary)
              {
           o.time = (int)(j * slot_mod);
           o.count = count;
           o.octets = octets;
           fwrite(&o, sizeof(o_struct), 1, p.output);
                }
              else
               fprintf(p.output, "%9d %5d %7d\n", (int)(j * slot_mod), count, octets);
              j++;
              octets = count = 0;
           }
        else
          {
            octets += ntohs(*(short*)(bf + hdr.time_len));
            count++;
            break;
          }
    }
  if (p.binary)
    {
      int a = 3;
      o.time = (int)(j++ * slot_mod);
      o.count = count;
      o.octets = octets;
      fwrite(&o, sizeof(o_struct), 1, p.output);
      fwrite(&j, sizeof(int), 1, p.output);
      fwrite(&a, sizeof(int), 1, p.output);
    }
  else
    {
      fprintf(p.output, "%9d %5d %7d\n", (int)(j++ * slot_mod), count, octets);
    }
  if (p.verbose) fprintf(stderr, "Line count = %d\n", j);
  fclose(output);
}

// ----------------------------------------------------------------------------

extern char *optarg;
extern int optind;


int main(int argc, char *argv[])
{
  int c;
  Param p = {0, 0, 0, 0.0, INTERVAL, NULL};

  while ((c = getopt(argc, argv, "vbt:s:")) != -1)
    switch(c)
      {
      case 't':
        p.slotsize = atof(optarg);
        break;
      case 'v':
```

```
        p.verbose = 1;
        break;
      case 'b':
        p.binary = 1;
        break;
      case 's':
        p.cut = 1;
        p.cuttime = atof(optarg) * 1000.0;
        break;
      }

  for (; optind < argc; optind++)
    {
      FILE *input;
      char tmp[256], fname[256];
      Ext exttra = {"tra", 1, 1};
      Ext *rtn, *exts[] = {&exttra, NULL};

      rtn = CheckExtension(argv[optind], fname, exts);
      switch(rtn->id)
        {
        case 1:
          strcpy(tmp, fname);
          strcat(tmp, ".");
          strcat(tmp, rtn->ext);
          input = fopen(tmp, "rb");
          break;
        default:
          input = NULL;
          break;
        }
      if (!input) break;
      if (p.verbose) fprintf(stderr, "Opening input file %s\n", tmp);
      TraceFile trace(input);
      trace.ReadHdr();
      if (p.cut)
        CutAggregate(trace, fname, p);
      else
        Aggregate(trace, fname, p);
      fclose(input);
    }
}
```

B.2.3 Stat

B.2.3.1 `stat.cc`

```
#include <stdlib.h>
#include <stdio.h>
#include <alloca.h>
#include <string.h>
#include <math.h>
#include <unistd.h>

#include "common.h"
#include "vector.h"

// ----------------------------------------------------------------------------

Ext eatf = {"atf", 1, 0}, eabf = {"abf", 2, 0};
Ext *exts[] = {&eatf, &eabf, NULL};

extern int optind;
extern char *optarg;

int main(int argc, char *argv[])
{
  s_data mtx;
  int i, logout = 0, offset = 1, limit = 10000, c, width = 2, verbose = 0;

  while ((c = getopt(argc, argv, "vlm:w:")) != EOF)
    switch(c)
      {
      case '?':
      printf("%s:  [-w width] [-v verbose] [-l log output] [-m maximum]\n", argv[0]);
        break;
      case 'w':
        width = atoi(optarg);
        break;
```

```
      case 'v':
        verbose = 1;
        break;
      case 'l':
        logout = 1;
        break;
      case 'm':
        limit = atoi(optarg);
        break;
      }

  Ext *rtn;
  exts[0]->flags = EXT_DEFAULT;

  for (; optind < argc; optind++)
    {
      FILE *input, *stdoutput, *logoutput;
      char tmp[256], fname[256];
      int count;
      mtx.rows = -1;
      rtn = CheckExtension(argv[optind], fname, exts);
      switch(rtn->id)
        {
        case 1:
          strcpy(tmp, fname);
          strcat(tmp, ".");
          strcat(tmp, rtn->ext);
          if (!(input = fopen(tmp, "r")))
            break;
          count = ReadFileText(width, &mtx, input);
          mtx.columns = width;
          mtx.rows = count;
          fclose(input);
          break;
        case 2:
          strcpy(tmp, fname);
          strcat(tmp, ".");
          strcat(tmp, rtn->ext);
          if (!(input = fopen(tmp, "rb")))
            break;
          ReadFileBinary(&mtx, input);
          count = mtx.rows;
          fclose(input);
        }

      if (mtx.rows != -1)
        {
          Vector freq (mtx.rows);
          for (int i = 0; i < mtx.rows; i++)
            freq[i] = ((int*)mtx.ptr)[i * mtx.columns + offset];
          delete mtx.ptr;
          if (verbose)
            {
            fprintf(stderr, "count = %d freq.length = %d\n", count, freq.length());
              fprintf(stderr, "Loaded freq Vector.\n");
            }
          strcpy(tmp, fname);
          strcat(tmp, ".sta");
          if (!(stdoutput = fopen(tmp, "w")))
            break;
          if (logout)
            {
              strcpy(tmp, fname);
              strcat(tmp, ".dat");
              if (!(logoutput = fopen(tmp, "w")))
         break;
            }
          for (i = 1; (i <= freq.length() / 20) && (i < limit); i++)
            {
#ifdef DEBUG
              fprintf(stderr, ".");
#endif
              register s = freq.length() / i;
              double sum;
              Vector tmp(s);
              for (register int j = 0; j < s; j++)
         {
           sum = 0;
           for (register int k = 0; k < i; k++)
             sum += freq[j * i + k];
           tmp[j] = sum / i;
```

```
         }
              double n = tmp.length();
              double mean = tmp.sum_ident() / n;
              double x_squared = tmp.sum_square();
              double var = (x_squared - n * mean * mean) / (n-1);
              fprintf(stdoutput, "%d %lf %lf\n", i, mean, var);
              if (logout) fprintf(logoutput, "%lf %lf\n", log10(i), log10(var));
#ifdef DEBUG
              fflush(output);
#endif
            }
#ifdef DEBUG
          fprintf(stderr, "\n");
#endif
          fclose(stdoutput);
          if (logout) fclose(logoutput);
        }
    }
}
```

B.2.4 Hist

B.2.4.1 `hist.cc`

```
#include <stdlib.h>
#include <stdio.h>
#include <alloca.h>
#include <string.h>
#include <math.h>
#include <unistd.h>

#include "common.h"
#include "vector.h"

// ---------------------------------------------------------------------------

Ext eatf = {"atf", 1, 0}, eabf = {"abf", 2, 0};
Ext *exts[] = {&eatf, &eabf, NULL};

extern int optind;
extern char *optarg;

int main(int argc, char *argv[])
{
  s_data mtx;
  int *hist, max, i, smooth = 1, offset = 1, c, width = 2, verbose = 0;

  while ((c = getopt(argc, argv, "vs:w:")) != EOF)
    switch(c)
      {
      case '?':
        printf("%s:  [-s smooth] [-w width] [-v verbose]\n", argv[0]);
        break;
      case 'w':
        width = atoi(optarg);
        break;
      case 's':
        smooth = atoi(optarg);
        break;
      case 'v':
        verbose = 1;
        break;
      }

  Ext *rtn;
  exts[0]->flags = EXT_DEFAULT;

  for (; optind < argc; optind++)
    {
      FILE *input, *stdoutput;
      char tmp[256], fname[256];
      int count;
      mtx.rows = -1;
      rtn = CheckExtension(argv[optind], fname, exts);
      switch(rtn->id)
        {
        case 1:
          strcpy(tmp, fname);
```

```
          strcat(tmp, ".");
          strcat(tmp, rtn->ext);
          if (!(input = fopen(tmp, "r")))
            break;
          count = ReadFileText(width, &mtx, input);
          mtx.columns = width;
          mtx.rows = count;
          fclose(input);
          break;
        case 2:
          strcpy(tmp, fname);
          strcat(tmp, ".");
          strcat(tmp, rtn->ext);
          if (!(input = fopen(tmp, "rb")))
            break;
          ReadFileBinary(&mtx, input);
          count = mtx.rows;
          fclose(input);
        }

      if (mtx.rows != -1)
        {
          max = 0;
          Vector freq (mtx.rows);
          for (i = 0; i < mtx.rows; i++)
            {
              freq[i] = ((int*)mtx.ptr)[i * mtx.columns + offset];
              if (freq[i] > max) max = (int)(freq[i]);
            }
          delete mtx.ptr;
          if (verbose)
            {
            fprintf(stderr, "count = %d freq.length = %d\n", count, freq.length());
              fprintf(stderr, "Loaded freq Vector.\n");
            }
          sprintf(tmp, "%s-%02d.hst", fname, smooth);
          if (!(stdoutput = fopen(tmp, "w")))
            break;

          hist = new int[++max];
          for (i = 0; i < max; i++) hist[i] = 0;
          for (i = 0; i < freq.length(); i++)
            hist[(int)(freq[i])]++;

          i = 0;
          while (i < max)
            {
              double x = 0.0, y = 0.0;
              for (int j = 0; (j < smooth) && (i+j < max); j++)
         {
           x += i+j;
           y += hist[i+j];
         }
              fprintf(stdoutput, "%6.3lf %6.0lf\n", x/j, y);
              i += smooth;
            }
          fclose(stdoutput);
        }
    }
}
```

B.2.5 Single-event

B.2.5.1 `single-event.cc`

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <limits.h>
#include <unistd.h>

#include "common.h"
#include "mathlib.h"

// ----------------------------------------------------------------------

struct Params {
```

```
  double limit;
  int verbose;
  int nodes;
};

// ------------------------------------------------------------------------

class Node {
private:
  double (*renewalDist)(DistParam &p);
  DistParam renewalParam;

public:
  class Node *next;
  double lastEvent, nextEvent;

  Node()
    {
      next = 0;
      lastEvent = nextEvent = 0.0;
    };

  void SetRenewalParam(DistParam &p) { renewalParam = p; };
  void SetRenewalDist(double (*d)(DistParam &p)) { renewalDist = d; };

  void GenerateEvent()
    {
      double evt = (*renewalDist)(renewalParam);
      lastEvent = nextEvent;
      nextEvent += evt;
    };
};

class Chain {
private:
  class Node *head;
public:
  double currentTime;

  Chain()
    {
      head = 0;
      currentTime = 0.0;
    };
  AddNode(Node *node)
    {
      Node **back = &head, *tmp = head;
//    double time = currentTime;
      if (node->nextEvent <= currentTime)
        node->GenerateEvent();
      while (tmp && (tmp->nextEvent < node->nextEvent))
        {
          back = &tmp->next;
          tmp = tmp->next;
        }
      node->next = tmp;
      *back = node;
    };
  Node *RemoveHead()
    {
      Node *tmp = head;
      head = tmp->next;
      return tmp;
    };
  void UpdateTime(double t) { currentTime = t; };
};

// ------------------------------------------------------------------------

void OutputBinary(double t, void *d)
{
  short hdr_size = 0;
  long tmp = (long)t;
  FILE *stream = (FILE*)d;

  fwrite(&tmp, sizeof(long), 1, stream);
  fwrite(&hdr_size, sizeof(short), 1, stream);
}

// ------------------------------------------------------------------------
```

```
double DistAbsT(DistParam &p)
{
  return fabs(DistT(p));
}

double DistCutOff(DistParam &p)
{
  register double uni = sunif();
  return exp(log((pow(p.CutOff.B,-p.CutOff.D) - pow(p.CutOff.A,-p.CutOff.D) *
           uni) + pow(p.CutOff.A,-p.CutOff.D))/ -p.CutOff.D);
}

// ------------------------------------------------------------------------

extern int optind;
extern char* optarg;

int main(int argc, char *argv[])
{
  int ch;
  Params p = {100000, 0,1};
  file_hdr hdr = {sizeof(long), sizeof(short), 0, 0};
  srand48(time(NULL));

  while ((ch = getopt(argc, argv, "hvt:n:")) != -1)
    switch(ch)
      {
      case 'h':
      case '?':
        fprintf(stderr, "%s:  -v [-n nodes] [-t time in seconds]\n", argv[0]);
        exit(1);
        break;
      case 't':
        p.limit = 1000 * atof(optarg);
        break;
      case 'v':
        p.verbose = 1;
      case 'n':
        p.nodes = atoi(optarg);
        break;
      }

  for (; optind < argc; optind++)
    {
      FILE *stream;
      char f_tmp[256];
      sprintf(f_tmp, "%s.tra", argv[optind]);
      stream = fopen(f_tmp, "wb");
      put_hdr(stream, &hdr);

      DistParam d;

      //      d.Exponential.lambda = 0.05;
      //      d.Uniform.mean = 20;

      d.Pareto.alpha = 1.5;
      d.Pareto.beta = 40.0;

      d.T.mean = 0;
      d.T.deviation = 1;
      d.T.freedom = 4;

      //      d.CutOff.A = 10.0;
      //      d.CutOff.B = 1000000.0;
      //      d.CutOff.D = 1.5;

      Chain chain;
      Node n, nlist[500];
      //      n.SetRenewalDist(DistExp);
      //      n.SetRenewalDist(DistUniform);
      //      n.SetRenewalDist(DistPareto);
      n.SetRenewalDist(DistAbsT);
      //      n.SetRenewalDist(DistCutOff);

      n.SetRenewalParam(d);
      for (int j = 0; j < p.nodes; j++)
        {
          nlist[j] = n;
          chain.AddNode(&nlist[j]);
        }
```

```
      if (p.verbose)
        fprintf(stderr, "Starting simulation on file %s\n", f_tmp);

      while (chain.currentTime < p.limit)
        {
          Node *tmp = chain.RemoveHead();
          chain.UpdateTime(tmp->nextEvent);
          if (chain.currentTime < p.limit)
            OutputBinary(chain.currentTime, stream);
          chain.AddNode(tmp);
        }
      fclose(stream);
    }
}
```

Bibliography

[1] Walter Willinger Ashok Erramilli, James Gordon. Applications of fractals in engineering for realistic traffic processes. *International Tele-traffic Conference*, 14:35 – 44, 1994.

[2] Hans-Werner Braun, Bilal Chinoy, Kimberly C Claffy, and George C Polyzos. Analysis and modeling of wide-area networks: Annual status report. Technical report, Applied Network Research, San Diago Supercomputer Center, February 1993.

[3] Novell Corp., editor. *NetWare Application Notes*, pages 32 – 50. Novell Corp., September 1990.

[4] Xerox Corp. Digital Corp., Intel Corp. The Ethernet: A local area network data link layer and physical layer specifications, version 2.0. Technical report, Digital Corp., Intel Corp., Xerox Corp., February 1982.

[5] Ashok Erramilli and Walter Willinger. Fractal properties in packet traffic measurements. Unpublished draft, 1993.

[6] D R Stirzaker G R Grimmett. *Probability and Random Processes*. Oxford Science publications, 1982.

[7] Alan B Oppenheimer Gursharan S Sidhu, Richard F Andrews. *Inside AppleTalk*. Addison Wesley, 1989.

[8] W E Leland, M S Taqqu, W Willinger, and D V Wilson. On the self-similar nature of ethernet traffic. *SIGCOMM Symposium on communications Architectures Protocols*, pages 183 – 193, September 1993.

[9] W E Leland, Walter Willinger, M S Taqqu, and D V Wilson. Statistical analysis and stochastic modeling of self-similar data traffic. *International Tele-traffic Conference*, 14:319 – 328, 1994.

[10] W E Leland, Walter Willinger, Murad S Taqqu, and Daniel V Wilson. Ethernet traffic is self-similar: Stochastic modeling of self-similar data traffic. Unpublished draft, March 1993.

[11] Ying-Dar Jason Lin, Tzi-Chieh Tsai, San-Chiao Huang, and Mario Gerla. HAP: A new model for packet arrivals. *ACM SIGCOMM*, pages 212 – 223, 1993.

[12] S B Lowen. Fractal renewal processes. *Transactions on Information Theory*, 39(5):1669 – 1671, September 1993.

[13] Benoit Mandelbrot. Self-similar error clusters in communications systems and the concept of conditional stationarity. *IEEE Transactions on Communication Technology*, 13:71 – 90, March 1965.

[14] D Mills. RFC 1305: Network time protocol (version 3) specification, implementation and analysis. Technical report, Internet Engineering Task Force, March 1992.

[15] The Institution of Electrical and Electronic Engineers. Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications. Technical report, The Institution of Electrical and Electronic Engineers, 1985. American National Standards Institute ANSI/IEEE Std. 802.3-1985.

[16] The Institution of Electrical and Electronic Engineers. Token ring access method and physical layer specifications. Technical report, The Institution of Electrical and Electronic Engineers, 1985. American National Standards Institute ANSI/IEEE Std. 802.5-1985.

[17] Jon Postel. RFC 791: The internet protocol (IP). Technical report, Internet Engineering Task Force, September 1981.

[18] J Romkey. RFC 1055: Nonstandard for transmission of IP datagrams over serial lines (SLIP). Technical report, Internet Engineering Task Force, June 1988.

[19] M C Teich S B Lowen. Fractal renewal processes generate $1/f$ noise. *Physical Review E*, 47(2), February 1993.

[20] Malvin C Teich S B Lowen. Doubly stochastic poisson point process driven by fractal shot noise. *Physical Review A*, 43(8), April 1991.

[21] Gennady Samorodnitsky and Murad S Taqqu. *Stable Non-Gaussian Random Processes*. Chapman & Hall, 1994.

[22] William Simpson. RFC 1661: The point-to-point protocol (PPP). Technical report, Internet Engineering Task Force, July 1994.