Prediction of ATM multiplexer performance by simulation and analysis of a model of packetized voice traffic

Tom Corcoran B.Sc.

Supervised by Dr. N. G. Duffield

Dublin City University
School of Mathematical Sciences
February 1994

M.Sc. Thesis by Research
Submitted in partial fulfilment of the requirements
for the degree of Master of Science in Applied Mathematical Sciences
at Dublin City University

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Master of Science in Applied Mathematical Sciences is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

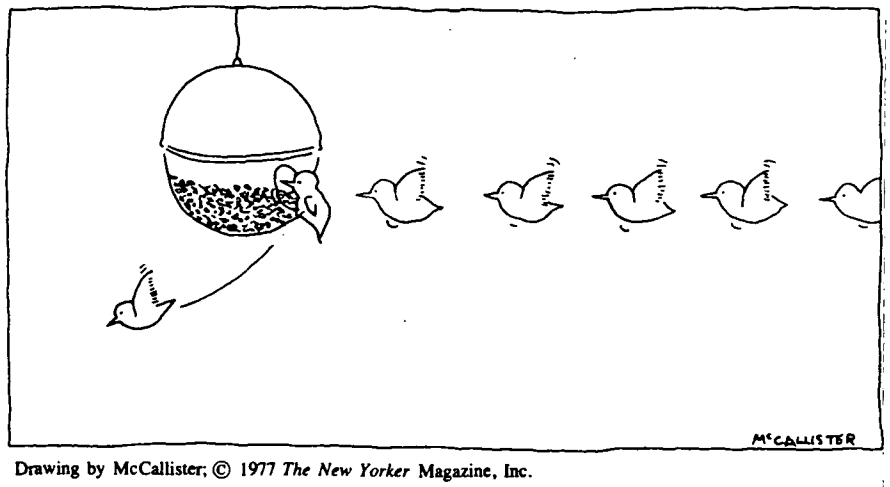
Signed: Tom Corcoran

Candidate

Date: 5 February 1994

Acknowledgements

Thanks to Dr. Nick Duffield, for hanging in there. Also, thanks to the lads on the telecommunications project, Paul Farrell and Robert Tucker. Thumbs up to all the maths postgrads, especially Kieran Murphy. I would also like to acknowledge help from Dr. Emmanuel Buffet and Aishling Walsh, computer support from Dr. Tommy Curran and partial funding from Eolas. Finally, thanks to everyone who kept me sane during the write up period.



Contents

1 Introduction **1**
- 1.1 Queueing Theory . . . 1
- 1.2 What is a queue? . . . 1
- 1.3 Queueing characteristics . . . 2
- 1.4 Report objectives . . . 2
- 1.5 Report outline . . . 3

2 Modelling an ATM multiplexer **6**
- 2.1 ATM Multiplexing: general concepts and preliminaries . . . 6
- 2.2 Modelling statistics of traffic on a single line . . . 9
  - 2.2.1 Terminology and assumptions . . . 9
  - 2.2.2 Markov Chains: definitions and limiting behaviour . . . 12
  - 2.2.3 Defining a Markov process . . . 14
- 2.3 Multiplexed system: superposition of sources and modelling the arrival process . . . 15
  - 2.3.1 Lindley equation with a Markovian workload . . . 15
  - 2.3.2 Superposition of sources . . . 17
  - 2.3.3 Stability condition for multiplexed system . . . 17
  - 2.3.4 Cell and burst level traffic . . . 18
- 2.4 Block Level Model . . . 19

3 Simulation of an ATM multiplexer **22**
- 3.1 Introduction . . . 22
- 3.2 Discrete and continuous simulation . . . 22
- 3.3 Next-event simulation . . . 23
  - 3.3.1 Outline of next-event simulation . . . 23
  - 3.3.2 Event definition . . . 23
  - 3.3.3 Event actions . . . 24
- 3.4 Simulation system . . . 25
  - 3.4.1 System description . . . 25
  - 3.4.2 Representation of time . . . 27
  - 3.4.3 System performance and limitations . . . 27
- 3.5 System objects . . . 29
  - 3.5.1 Introduction . . . 29
  - 3.5.2 Class node_c . . . 29
  - 3.5.3 Class line_list_c . . . 29

3.5.4 Class probabilities_c 30
3.5.5 Class queue_c 32
3.5.6 Class mlcg32_c 33
3.5.7 Class stop_watch_c 33
3.5.8 Class string_c 33
3.6 Main program and flowcharts 33
3.7 Start-up policy 40
3.8 Simulation duration 43

4 Simulation performance: results and evaluation **44**
4.1 Introduction 44
4.2 Explanation of simulation parameters 44
4.3 Simulation graphs 47
4.4 A model for the buffer queue distribution 51
4.5 Regression analysis 52
4.6 Presentation of results 56
4.6.1 Introduction 56
4.6.2 Analysis of model parameters 56
4.6.3 $\tilde{M}(x)$ versus $M(x)$ 58
4.6.4 Proportion of time in burst level congestion and the knee . . . 60

5 Conclusion **61**
5.1 Prediction of large ATM systems 61
5.2 QoS for large ATM systems 62
5.3 Further Study 62

Bibliography **63**

Appendices

A Object-oriented programming and C++ **A-1**
A.1 Introduction A-1
A.2 Object-oriented design versus top-down design A-1
A.3 Understanding object-oriented design A-2
A.4 Data abstraction and object-oriented programming in C++ A-4
A.4.1 Classes A-4
A.4.2 Inheritance and virtual member functions A-4
A.4.3 Friend functions A-4
A.4.4 Operator name overloading A-5
A.4.5 Inline functions A-5
A.4.6 Encapsulation A-5
A.5 Conclusion A-5

B Pseudorandom numbers and generators **B-1**
B.1 Introduction B-1
B.2 Pseudorandom numbers B-2
B.3 Generating pseudorandom numbers B-2

B.4 Multiplicative linear congruential generator B-3
B.5 Generator used in the simulation system B-3

C ATM simulator source code **C-1**
C.1 Class definitions and header files C-1
C.2 Class member functions C-14
C.3 Main program C-42
C.4 Join program C-68
C.5 Disc index C-76

Abstract

A multiplexer is used to group and carry multiple channels on a single transmission line. Asynchronous transfer mode (ATM) is a high speed packet switching or multiplexing technique. Due to the possible statistical fluctuations in the arrival process to an ATM multiplexer, it must be equipped with a buffer, which stores temporarily the excess arrivals, until they can be processed. The buffer must be of a size large enough to guarantee a sufficiently small probability of overflow. In this thesis bursty voice sources are modelled using a Markov chain. The inherent correlations of the generated arrival process make exact analysis intractable. An object oriented simulation system developed in C++ is used to obtain empirical queue length distributions of the buffer for a number of multiplexer loads. The shape of the distribution highlights two forms of congestion due to low and heavy traffic, occurring at small and large buffer sizes, respectively, and motivates the choice of a phenomenological model. The parameters of the phenomenological model are fitted by analysis of the simulation model. In practice for large ATM systems, the number of input lines, l, makes simulation infeasible. We show how loss probabilities for large l can be predicted by simulating for smaller values of l and using the phenomenological models in conjunction with the simulation data to fit the parameters for the phenomenological model of the large system itself.

List of Figures

2.1 Sample traffic on a single line 9
2.2 Probability distribution of the times between cells 11
2.3 Cell level Markov model for a single line source 14
2.4 Relationship between variables in a queueing system 15
2.5 Sample traffic for a superpositional arrival process 17
2.6 Block level Markov model for a single line 20

3.1 The main program 34
3.2 Procedure *control_simulation* 35
3.3 Creating a probabilities object 36
3.4 Setting up a linked list 37
3.5 The simulation loop 38
3.6 Moving a node in the list 39
3.7 Effect of different initial conditions 41
3.8 Simulation run with periodic output 43

4.1 $\tilde{M}(x)$ for various l with $\rho = 0.82$ 48
4.2 Enlargement of knee area for $\rho = 0.82$ 48
4.3 $\tilde{M}(x)$ for various l with $\rho = 0.78$ 49
4.4 Enlargement of knee area for $\rho = 0.78$ 49
4.5 $\tilde{M}(x)$ for various l with $\rho = 0.85$ 50
4.6 Enlargement of knee area for $\rho = 0.85$ 50
4.7 $\tilde{M}(x)$ for various l with $\rho = 0.9$ 50
4.8 Enlargement of knee area for $\rho = 0.9$ 51
4.9 u_2 versus left cutoff point for $\rho = 0.82$ and $l = 100$ 54
4.10 Intercept plotted against l for chosen ρ 57
4.11 u_2 plotted against l for chosen ρ 57
4.12 u_1 plotted against l for chosen ρ 58
4.13 Plot of $\tilde{M}(x)$ and $M(x)$ for $\rho = 0.85$ and $l = 100$ 59
4.14 Enlargement of knee region for figure 4.13 59
4.15 Plot of $\tilde{M}(x)$ and $M(x)$ for $\rho = 0.9$ and $l = 200$ 59
4.16 Enlargement of knee region for figure 4.15 59
4.17 % of time in burst level congestion for the chosen ρ 60

List of Tables

3.1 Comparing simulation lengths 28
3.2 % time buffer non-empty 41

4.1 β depending on l 46
4.2 s, C and $\overline{I}$ for the chosen ρ 46
4.3 σ required for the chosen ρ 46
4.4 Cutoff points used in calculation of u_1 and u_2 53
4.5 u_2, intercept and λ with 95 % confidence interval 54
4.6 u_1 with 95 % confidence interval 55

Abbreviated terms

ATDM	Asynchronous Time Division Multiplexing
ATM	Asynchronous Transfer Mode
BBP	Backwards busy period
CTMC	Continuous time Markov chain
FIFO	First in First out
ISDN	Integrated Services Digital Network
MLCG	Multiplicative linear congruential generator
MMP	Markov modulated process
OOP	Object-oriented programming
QoS	Quality of Service
SRO	Service in random order
UDDT	User defined data type
VBR	Variable bit rate

Chapter 1

Introduction

1.1 Queueing Theory

Queueing theory is a branch of Applied Mathematics utilizing concepts in the fields of stochastic processes and applied probability and is concerned with the study of the behaviour of waiting lines. The theory can provide us with among others: predictions about waiting times, the length of a busy period and the overflow distribution; it can give us a better understanding of queues with the aim being to improve the system performance and achieve greater control.

We will begin by giving a brief overview of the characteristics of queueing systems, followed by a summary of our objectives and finally an outline of the report.

1.2 What is a queue?

Americans call it waiting in line, but regardless of name everybody knows what queues are. Queues are becoming more and more prevalent in our increasingly congested and urbanized society and many are inevitable. Most queueing systems involve human beings and every moment a person spends waiting for some type of service they are part of a queue. Everyday people queue for service in restaurants, supermarkets, shops, banks, post offices, passport offices, theatres, cinemas, pubs, train stations, bus stops, taxi-ranks, labour exchanges and hospitals.

Queues are not always this obvious and they do not have to involve people. A queue does not have to be visible, it is simply a group of people, tasks, or objects that have requested, but not yet received, service. Other examples of queues are: pedestrians waiting to cross the road; waiting lists for telephone installation, specialist surgery, council houses and court cases; a suit waiting to be dry-cleaned; vehicles queueing at service stations or stuck in bottlenecks; aircraft waiting in a holding pattern to land; boats waiting to pass locks and bridges; assembly lines and typing pools; dam operations and climbers waiting to do a classic route.

In the field of communications there are queues due to telegrams, telephones, computers (jobs waiting for CPU time) and other data communications. Queues occur at different levels, for example, telephone calls wait to get through an exchange and packets of data wait in the **buffer** of a **multiplexer**.

It is the later queue which we are interested in. A multiplexer takes in many channels and allows each in turn access to an output link. A buffer store is interposed between the set of incoming links and the output link, ie. in the multiplexer. Its general purpose is to receive and store (queue) digitized information until it can be processed.

1.3 Queueing characteristics

The elements of a queueing system are the *customers* waiting for service, the *server*, the *queue* itself and the *output*. All the queues described in section 1.2 possess these elements but they are not always obvious, for example, at a supermarket checkout, the customer may be either the shopper in line or the items being purchased - both are waiting for service.

The *arrival process* depicts the timing of customer arrivals at the queue. We need to know the rate of customer arrival and if they are independent of each other. If all the servers are busy, then the arriving customer joins a queue where they remain until a server becomes available. The *service rate* represents the time taken to serve a customer, another important aspect of the server is its configuration, ie. the number of parallel servers, consequently we talk about single server and multi server queues.

Queue discipline or scheduling represents the order in which the customers in the queue are served, ie. the disposition of the blocked units, eg. service in random order (SRO) or first in first out (FIFO). Another factor of the queue discipline is *priority*, ie. we need to know if different customers receive different priority and if so, what is the system for assigning it (triage).

We are concerned with voice traffic as the input to the multiplexer, which operates as a single server queue, all the traffic has equal priority and is served on a first come first served basis.

1.4 Report objectives

We consider l input voice channels to a multiplexer with a single buffer. Speech from a telephone (active voice source) is digitized and is fed by the input source to the common queue (buffer) in the form of periodic cells (packets) of information, of constant deterministic length, the periodicity of which is due to the constant sampling rate of the source. Arrivals at the buffer are regular during active periods (talking) and there are no arrivals during periods of inactivity (listening). We describe a source as bursty if correlations exist between the activity of the source at different times. The superposition of a large number of sources constitutes a voice packet arrival process which possesses positive correlations, due to the periodicity of the sources, making exact analysis intractable [9]. The random and unpredictable traffic flow causes unavoidable buffer queues and it is the analysis of these which is the objective.

The scenario of interest is an arrival process constituting of a large number of independent voice sources, which asynchronously alternate between the transmitting and idle states. The packets are fed into the multiplexer buffer, which has unlimited waiting room, on a FIFO basis, and the server removes them for transmission over

a shared communications channel. The queue length distribution of the buffer is important in accessing the performance of the multiplexed system, which is reflected by the Quality of Service (QoS), which is described by the cell loss probability and by the buffer delay. The cell loss probability depends on the workload arriving at the multiplexer and on the size of the buffer. Buffer or queueing delays occur when a statistical variation in the arrival process is such that the arrivals exceed the service capacity for some period, the excess is then stored in the buffer.

There has been some work on developing analytic models to reflect the statistical properties of the voice sources and approximate the queue length distribution for the packet voice statistical multiplexing system. Heffes and Lucantoni worked on approximating the superposition by a correlated Markov modulated Poisson process (MMPP) in which the durations of the active and inactive periods are exponentially distributed [24]. Daigle and Langford [9] developed a continuous time Markov chain (CTMC) model where the amalgamated arrival process is Poissonian and also has exponentially distributed burst and silence periods [9], Sriram and Whitt have have done similar work [42].

We are interested in large systems prediction, ie. predicting systems where the number of input lines, l, and the sampling frequency, s, of the multiplexer are scaled to infinity. Buffet and Duffield have analyzed the queue due to an arrival process which is a 2-state Markov process and have shown that the probability of the queue length exceeding x is bounded above by a function of the form $z^{-l}y^{-x}$, where $z > 1$ and $y > 1$ (exponential decay rate of the queue length distribution) are functions of s/l, if the average number of arrivals at the multiplexer per unit time is held constant [4]. If this holds more generally then predictions can be made for larger systems by analyzing the behaviour of smaller, easier to simulate, systems. In this thesis the behaviour of the queue length distribution, as l scales with s/l kept constant, is investigated. We use a modification of the CTMC model developed by Daigle and Langford, in discrete time, where the periods of activity and inactivity are modelled by a geometric distribution.

1.5 Report outline

This report looks at a certain implementation of a statistical multiplexer using a purpose built simulation system. The mathematical theory regarding the model being simulated is presented so as to give a full understanding of the problem in hand. Conclusions regarding the prediction of large ATM systems are drawn from the conjunction of simulation results and a phenomenological model.

The simulation process follow three main steps: modelling, simulation and interpretation - chapters 2, 3 and 4 follow these steps in the main. Modelling identifies the relevant system features and the applicable assumptions and simplifications necessary. Simulation runs the specifications for a variety of system parameters relevant to the purpose of the simulation. interpretation extracts information from the output of the simulation and evaluates and analyzes it with respect to the task at hand.

Chapter 2 collects several topics that are needed for the study of the problem, including details on ATM multiplexing, Markov chains and renewal processes.

The Markov approach can be applied to the random behaviour of a system provided the behaviour is characterized by a lack of memory, future states are independent of all past states except the immediately preceding one, ie. probability density functions (pdf) that are conditioned on several previous time instants always reduce to a pdf that is conditioned only on the most recent time instant.

We model the arrival process of a single input voice channel as a Markov chain. So, the input to the multiplexer is a superposition of a number of these Markov processes, each of which we take to be independent. Consequently, the service requirement at each integral time is the sum of a number of random variables, each of which is the state of an independent Markov chain. The sources are typically bursty, in the sense that their activity is highly correlated into bursts rather than occurring independently at different times, this reflects the reality of a telephone conversation which is made up of alternating periods of speech and silence.

Congestion will occur if there are more customers in the system than the server can handle simultaneously. Heuristically, we can identify two types of congestion, namely cell level congestion and burst level congestion. Cell level traffic exists when the queue was empty more recently than the typical correlation time of arrivals, which in this case is the periodicity of transmission of an active source. This cell level congestion resembles that due to Poissonian arrivals of the same rate as the rate of each source. Burst level congestion is due to the persistence of arrivals from the l bursty sources which result in long periods when the queue is non-empty. Burst level congestion is determined by the correlations in the arrivals, which do not contribute to cell level congestion.

We describe the 2-state Markov model of Buffet and Duffield, which is a simplification of the Markov Chain model for a single voice source and we also draw comparisons between the two models. We introduce the upper bound for the queue length distribution obtained by Buffet and Duffield, which is used as a basis for choosing some of the parameters for the simulation of the modified CTMC model.

We are looking at one particular implementation of a statistical multiplexer, where the buffer queue is measured from the point of view of an arriving cell of information. Chapter 3 introduces the idea of simulation and outlines the event-to-event discrete simulation process. A specification of the system designed to simulate the above situation is then presented. The development took place using an object oriented approach and was written in C++; the source code and sample output is included in section C.

The system is made up of objects designed to pattern the behaviour of the real system being simulated. Each line source is an object and another object represents the aggregate input. The other main objects are the buffer queue, the random number generator and the system probabilities. The components of these objects are outlined, as well as the main program, which interacts with the system objects. The main program is described briefly including the simulation algorithm, which is the procedure that is repeated until some limit on the length of the simulation is reached. A simulation run can be described as a non-terminating process, ie. there is no definite climatic event which stops everything; it can be stopped and started at arbitrary times and can be run indefinitely. However, simulation is only feasible for relatively high cell loss probabilities, in the order of 10^{-4}, as otherwise the simulation

length would need to be extremely long so as to sample the rare events. The chapter concludes by discussing the effect of initial conditions and the duration of simulation runs.

Chapter 4 begins with an explanation of the particular choice of numerical values for the parameters used in the simulations. The main simulation output is an empirical probability distribution of queue length of the multiplexer buffer (which has a correlated arrival process), versus the buffer size. We present and discuss the simulation graphs for the chosen parameters. The shape of the output graph motivates the choice of the following phenomenological model and reflects the two types of congestion.

$$\Pr\{\text{queue length} \geq x\} = \lambda e^{-u_1 x} + (1-\lambda)e^{-u_2 x}$$

The first term models the queue length due to cell level congestion and the second term that due to burst level congestion.

We discuss how we use linear regression to fit the parameters of this model, u_1, u_2 and λ. The main results of this thesis are the plots of the intercept, u_2 and u_1 against the number of inputs l. We also compare the fitted model with the simulation model. Finally we present simulation statistics which give the proportion of time spent in burst level traffic.

Chapter 5 draws some conclusions regarding the prediction and QoS of large systems. We show how observed dependence of the quantities, λ, u_1 and u_2 on the parameters of the simulated model can be used to make predictions about the performance of ATM multiplexers in which the number of inputs is too large to simulate.

Chapter 2

Modelling an ATM multiplexer

2.1 ATM Multiplexing: general concepts and preliminaries

For the purpose of our study, only a general understanding of statistical multiplexing is needed. What we need to understand is how digital communications (of any type) share a single common broadband (high speed) transmission link. Although some of these terms were introduced in sections 1.4 and 1.5, this question still leaves us with some terms which need to be explained before the question itself is answered.

The term broadband covers speed ranges from 1 Mb/s (10^6 bits per second) to 100 Mb/s and beyond. In our case, the sources of digital communications are l input telephone lines, which must share the same transmission link (line/channel), ie. communication bandwidth. Telephone circuits are designed to pass a certain limited bandwidth (see below), this permits efficient transmission of the voice and signal frequencies; signals outside this range and below a certain threshold are suppressed, effectively being treated as silence for the purpose of transmission. Bandwidth is the difference between the upper and lower limit of the band.

Nowadays telephone systems are moving towards an all-digital network, but up to the late 1960's telephone networks were largely analog. However, electrical communication in its earliest stage was digital in the form of telegraphy, which takes one of two discrete amplitudes, namely on and off. In digital networks, voice - speech generated at the telephone - and signalling information (both analog) are converted to digital signals for transmission along the network. In telephony, once the digital signals are transmitted they must then be reconverted into analog form to generate the speech at the called telephone.

One of the features of voice digitization is that the encoded signals are divorced from the analog waveforms of the source. The digital transmission and switching equipment of a voice network is then inherently capable of servicing any traffic of a digital nature. It is this property that is exploited to provide Integrated Services Digital Network (ISDN) services, which are an entirely digital implementation. Since all data can be represented in digital form, all data (regardless of source) can travel along the same link, hence an integrated approach can be taken to communication networking [1].

Digital technology samples a continuous time signal at discrete instants and the sampled value is represented in digital format, ie. a series of zeros and ones; therefore, it is a method for encoding a signal. Each source has a maximum bandwidth of 4 kHz, this takes into account that most conversations involving humans happen in the 300 Hz $\rightarrow$ 3.4 kHz range and also the frequency of some signalling transmissions. An analog signal needs to be sampled at twice it's highest frequency to obtain an accurate digital representation of the information content of the signal. The standard sampling rate used in digital telephony is 8 khz, since a sample is 8 bits this means sampling at a constant bit rate of 64 kb/s. Consequently, the data is partitioned or segmented into fixed length **packets**, or to use more modern terminology, **cells**, with a fixed length of 53 bytes (424 bits), 48 bytes for the data and 5 bytes for the header (containing identification and routing data).

Multiplexing can be used to group and carry multiple channels on a single transmission line, whose inherent bandwidth is greater than that needed for a single channel source, thus transmitting them simultaneously. A sequence of time slots, of duration equal to the duration of a single packet (integral length), are established on the transmission medium bandwidth, during which individual sources, each of which is connected to a caller, can transmit signals. The bandwidth is shared by multiplexing the bit streams in the time slots to form a single digital stream for transmission on the common communication channel. There are a number of terms used to describe this method of sharing: Asynchronous Time Division Multiplexing (ATDM), more commonly known as Asynchronous Transfer Mode (ATM) of packet switching [25, 37].

ATM is a fast packet switching technique with the principle characteristic that it can support cell traffic generated by variable bit rate (VBR) sources. So, ATM is a high speed multimedia network, ie. it can support a broad spectrum of traffic classes - voice, video, videotelephony, colour facsimile, LAN (local area network) interconnections and other data communications - at a variety of transmission speeds (different bit rates). If the ATM channel (transmission line) has a capacity of C Mb/s, therefore, the time taken to transmit each cell is a constant $424/C$ - due to the fixed cell size. Conceptually, time is divided into slots corresponding to this cell transmission time which is known as a **tick**. A tick can be looked on as the period at which the output of the multiplexer operates, ie. the multiplexer can process one cell per tick. All cells are required to arrive at the beginning of each time slot, also, cells are cleared to the output line by the end of each time slot.

ATM carries bursty traffic efficiently and a voice source is a well known example of a sporadic or bursty source. This is true as long as the coding scheme employs speech activity detection and silence suppression. A packet switched system does not need to transmit silence and so the silence "is removed" [25]. For intervals when a caller is speaking, otherwise known as a talkspurt, a bursty voice source emits a periodic stream of constant length cells in serial form. Talkspurts are interspersed by silent periods during which no packets arrive. Both talkspurts and silent periods are variable in length, which reflects the fact that a proportion of the average telephone conversation constitutes silence. A cell stream from a single voice source can be modelled by arrival streams, with individual cells separated by a fixed interval during talkspurts, or by no arrivals during silence. The fixed length between cells in a burst

is because of the periodicity at which each individual source is sampled and is known as the sampling or packetization period. So, in other words, the cell interarrival time during a talkspurt is constant, otherwise it is one packetization period plus a silence. This sampling period is measured in units of the output period, ie. ticks, and is labelled as of length s; s can be seen as the ratio of the sampling period to the output.

The input traffic to the multiplexer is taken to be a superposition of a finite population of l packetized voice sources, each of which is characterized as above. The completion of a service by the multiplexer constitutes the insertion of a call into the transmission medium - an actual physical channel - which is modelled as a single server. The queueing or service discipline of the buffer in the multiplexer is governed by a FIFO policy in order to guarantee call sequence integrity. Each source is assigned an identifier, which is incorporated in a cell prefix or header. So, a cell is a labelled block of transmitted information and as each cell is formed it is given the next available time slot (bandwidth on demand). When the cells reach the head of the queue they are sent down the output line and when they emerge from the far end of the transmission line, the data is transmitted to its destination according to the cell header. This allows data from many conversations to be interleaved in any sequence without mutual interference and permits individual conversations to be retrieved by demultiplexing.

So, a statistical multiplexer is used to share a transmission line and in this case, to gain efficiency in a superposition of bursty sources; thereby reducing the number of transmission links needed. It takes advantage of the statistical variations in the incoming traffic to perform statistical multiplexing and save communication bandwidth. Its purpose is to maximize utilization by using all the available bandwidth. It does this by utilizing the sampling period, s between cell arrivals (on a single input line) and the periods of silence in bursty sources, to support other active sources.

A multiplexer and its buffer can be seen as a single server, constant deterministic service time queueing system. Queues of cells arise when the available capacity of the ATM is overallocated using statistical multiplexing. This congestion occurs due to the dynamic nature of bursty traffic - the fluctuations in the number of bursts and hence the number of cells arriving. From time to time a number of sources emit bursts simultaneously, consequently producing a transient cell arrival rate greater than the multiplexer capacity.

The performance of a network - our inputs, multiplexer and output line form one part of such - is generally measured in terms of throughput and delay. In order to minimize the inevitable delays in a queue there is generally a limit on the length of the queue; however, we are not concerned with this particular problem and assume that the multiplexer has an infinite capacity, ie. all packets are allowed in. Consequently the probability of cell loss is not an issue and throughput is not degraded. In synchronous packet networks messages arrive with varying degrees of urgency and hence priority. Priority is established by the need of transmission or by the importance of transmission, for example, in voice and data synchronous systems, voice may get priority over data in order to have a reasonable QoS; different cell types may have higher or lower maximum admissible cell loss probabilities. We are dealing with voice traffic only and so priority is not in question; although the fact that the packets

are coming from speech signals on voice channels is not important in modelling the arrival process.

So, we model the buffer in the multiplexer as an infinite capacity single server queue with a constant service time and with no priority. By the end of the chapter we will have presented the necessary background and discussed the problem at hand, as well as outlining our motivations for simulating.

2.2 Modelling statistics of traffic on a single line

2.2.1 Terminology and assumptions

Firstly, we will examine the characteristics of the traffic for a single bursty voice source.

- **Burst.** A burst or talk spurt is packetized into a series of fixed length packets (see section 2.1) and continues until a silence longer than the overhang time in encountered, ie. a burst includes the overhang time. Since there are s ticks between cells, a burst can be described as a grouping of s ticks, where the last s ticks are the overhang.
- **Overhang.** The overhang is a deterministic period of time after the talk spurt has ended; it is a waiting period to see if another cell of information arrives or if silence has begun.
- **Silence.** This is a period of time during which there is no speech activity and represents the time where a caller is listening and not talking; it continues until the next burst starts.

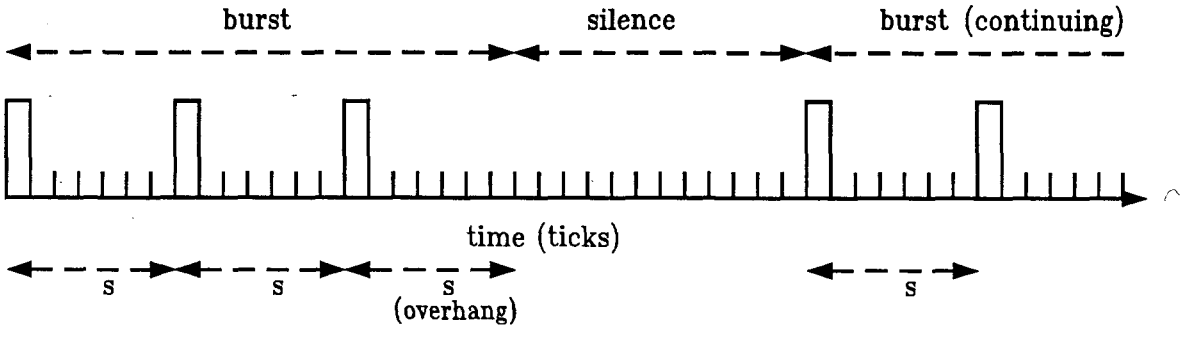


Figure 2.1: Sample traffic on a single line

Human speech is modelled as alternative bursts and silences of variable length and sample traffic for a single voice source is shown in figure 2.1. A superposition of l bursty sources is how the traffic is presented to the multiplexer, which regulates the inputs and multiplexes them onto a single transmission line.

Before we continue we need to define the following probabilities which characterize the distributions of the burst and idle periods:

$$\alpha = \Pr\{\text{a burst continues}\} \tag{2.1}$$

ie. an active line stays active. So, $(1-\alpha)$ is the probability that an active source becomes inactive, in other words, that the burst stops. If the burst stops then the last grouping of s ticks was the overhang.

$$\beta = \Pr\{\text{the silence continues for another tick}\} \tag{2.2}$$

ie. an inactive line stays inactive. So, $(1-\beta)$ is the complimentary probability that an inactive source becomes active, ie. another talk spurt begins. See section 4.2 for a more detailed description of α and β.

For each voice source we make the following assumptions:

A1. The telephones is continuously off-hook, ie. continually busy.

A2. The traffic (on each line) is independent.

A3. The burst lengths are independent and identically distributed (i.i.d.) and are geometrically distributed.

A4. The silent lengths are i.i.d. and are geometrically distributed.

Assumption A1 means that there is no silence between phone calls, only silence between bursts of speech, words and syllables (see section 2.1) When a person is talking there are two possible situations, they could be actually speaking or having a slight, small, pause. So, although we describe a talkspurt or burst as the period when a person is talking, it really is the period that data is being sent, ie. when the person is really talking.

The telephones transmit only when there is speech activity; an absence of a cell means that either the input is not active or that it is between packets, since a packet of data is only transmitted every s ticks.

Since the packet interarrival times can be regarded as i.i.d, we assume that the successive talkspurts and silence periods form an alternating renewal process (regenerative process). This is a sequence of i.i.d. random variables which represent the lifetimes of a burst or a silent period. It is described as alternating since the system can be in one of two possible states. The time between the cells, D, measured in ticks, is never less than s, ie. $D \geq s$. We define V to describe the distribution of the times between cells, ie. $V(d) = \Pr\{D \leq d\}$. From equations 2.1 and 2.2 we know that $\Pr\{D = s\} = \alpha$ and $\Pr\{D = s+1\} = (1-\alpha)(1-\beta)$. Now using the fact that $V(s+x) = \sum_{i=1}^{x} \Pr\{D = s+i\}$ we can write

$$V(d) = \alpha + (1-\alpha)(1-\beta)^{d-s}, \quad d \geq s \tag{2.3}$$

From assumptions A3 and A4 we know that the burst and silence lengths are random variables with geometric distributions.

$$\Pr(\text{burst} = j \text{ cells}) = (1-\alpha)\alpha^{j-1} \tag{2.4}$$

$$\Pr(\text{silence} = s \text{ ticks}) = (1-\beta)\beta^{s-1} \tag{2.5}$$

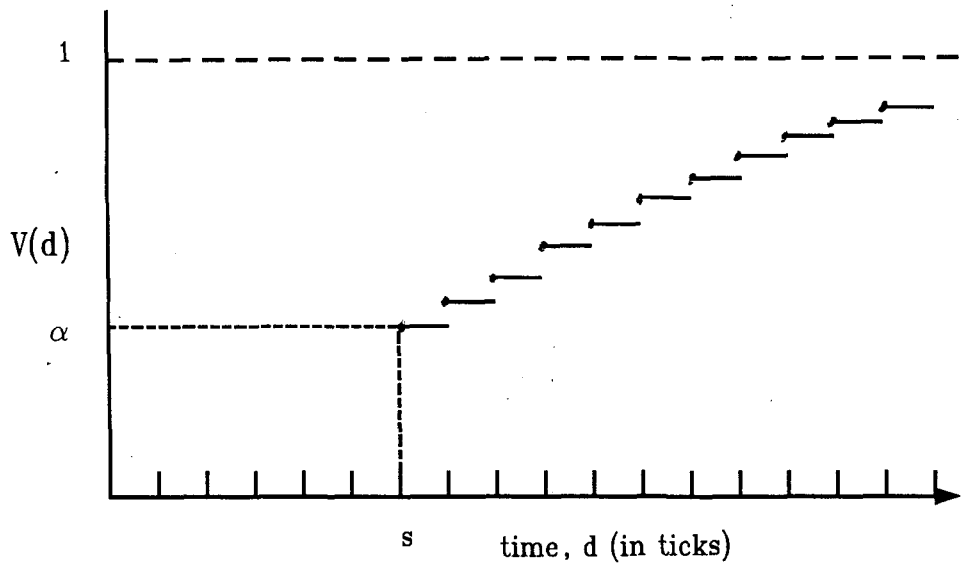


Figure 2.2: Probability distribution of the times between cells

Note we speak of bursts in terms of cells and silence in terms of ticks, since this is what characterizes them. Also, we assume that there is at least one cell in a burst and at least one tick in a silence. Now, it is straightforward to show the following

$$\Pr(\text{burst} > c \text{ cells}) = (1-\alpha)\sum_{j=c+1}^{\infty} \alpha^{j-1} = \alpha^c \tag{2.6}$$

$$\Pr(\text{burst} \geq c \text{ cells}) = \alpha^{c-1} \tag{2.7}$$

$$\Pr(\text{silence} > s \text{ ticks}) = \beta^s \tag{2.8}$$

$$\Pr(\text{silence} \geq s \text{ ticks}) = \beta^{s-1} \tag{2.9}$$

The burst length is geometrically distributed, so the expected mean number of cells in a burst, E_B, is by definition

$$E_B = \frac{1}{(1-\alpha)} \tag{2.10}$$

Similarly, the expected mean number of ticks in a silence (idle period), E_I, is

$$E_I = \frac{1}{(1-\beta)} \tag{2.11}$$

We define $\overline{B}$ and $\overline{I}$ to be the actual mean burst and mean silence lengths respectively. Then, if T_W is the tick width (see section 2.1), we can write the following

$$\alpha = 1 - \frac{sT_W}{\overline{B}} \quad \text{and} \quad \beta = 1 - \frac{T_W}{\overline{I}} \tag{2.12}$$

The values of E_B and E_I used in the simulations are discussed in section 4.2.

We can calculate the activity rate, A_R, ie. the probability that a line source is active, to be

$$A_R = \frac{\overline{B}}{\overline{B} + \overline{I}} \tag{2.13}$$

This also represents the proportion of time in the long run, that the process is in burst mode, the complimentary probability giving the proportion of time spent in silent mode.

2.2.2 Markov Chains: definitions and limiting behaviour

A **stochastic process** is a collection of random variables defined on a common probability space; we consider the discrete stochastic process $\{X_n, n \in N\}$ with a countable state space E. **Markov processes** are an important class of stochastic processes with the property that given the present state of the stochastic process the future (process evolution) is independent of the past (evolution process). A Markov process can be thought of as the sequence of states entered by a system over time and is called a **Markov chain** if it's state space is discrete. Then, the above stochastic process is called a Markov chain (of order one) provided that

$$\Pr\{X_{n+1} = j \mid X_0, \ldots, X_n\} = \Pr\{X_{n+1} = j \mid X_n\} \tag{2.14}$$

$\forall\, j \in E$.
We will consider only the Markov chain for which the **transition probabilities**, $P(i,j)$, ie. the probability of transition from state $i \rightarrow j$ in one-step, are independent of the time variable, ie.

$$P(i,j) = \Pr\{X_{n+1} = j \mid X_n = i\}, \qquad i,j \in E \tag{2.15}$$

So, the Markov chain has stationary probabilities, ie. it is time-homogeneous. A markov chain is characterized by it's **transition matrix** (square) which is made up of the probabilities $P(i,j)$, $\forall\, i,j \in E$. The transition matrix of a Markov chain is a Markov (stochastic) matrix, which means that all the entries are non-negative and that all the rows sum to unity, ie. $0 \leq P(i,j) \leq 1, i,j \in E$ and $\sum_{j \in E} P(i,j) = 1$, for every $i \in E$.

The joint distribution of $X_0, \ldots, X_m$ is completely specified for every $m \in N$ once the initial distribution, $\psi(i_0) = \Pr\{X_0 = i\}$, and the transition matrix are known.

$$\Pr\{X_0 = i_0, X_1 = i_1, \ldots, X_m = i_m\} = \psi(i_0)P(i_0, i_1), \ldots, P(i_{m-1}, i_m) \tag{2.16}$$

The m-step transition probabilities can be computed using the **Chapman - Kolmogorov** equation, which is

$$P^{m+n}(i,j) = \sum_{k=0}^{\infty} P^m(i,k)P^n(k,j) \quad \text{for every } i,j \in E \tag{2.17}$$

So, the process must be in some intermediate state k after m steps and the probability of reaching state j does not depend on how state k was reached. It can be easily shown

that P^n, which contains elements $P^n(i,j)$, each of which is the probability of going from state i to state j in exactly n transitions, is also a Markov matrix.

State j of the Markov chain $\{X_n\}$ is reachable from state i if it is possible to reach state j from state i in a finite number of steps, ie. $P^n(i,j) > 0$ for some $n \geq 0$. If every state can be reached from every other state the Markov chain is **irreducible**, this means that the only closed set is the set of all states. For each state we define $T_j^{(n)}$ to be the probability that the first return visit to state j occurs after n transitions (after leaving j), ie. $T_j^{(n)} = \Pr\{X_n = j, X_{n-1} \neq j, \ldots, X_1 \neq j \mid X_0 = j\}$. Then the probability of ever returning to state j is given by $T_j = \sum_{n=1}^{\infty} T_j^{(n)}$. If the probability of returning to state j is 1, ie. $T_j = 1$, then state j is said to be **recurrent**, otherwise, if $0 \leq T_j < 1$, the j is a **transient** state. If we define M_j to be the time of the first visit to state j, then if $E[M_j] = \infty$ a recurrent state j is called **null**, otherwise, if $E[M_j] < \infty$, the same state j is called **non-null**. A recurrent state j is said to be **periodic** with period $d \geq 2$ if $P^{(n)}(i,j) = 0$ unless $n = vd$ is a multiple of d, and d is the largest integer with this property. If no such d exists then the state j is called **aperiodic**.

Feller [15] describes recurrent non-null, aperiodic states as **ergodic**. If all the states are ergodic, this means that a unique stationary distribution exists (see below). An ergodic theorem gives conditions under which an average over time of a stochastic process will converge as the number of observed periods becomes large, eg. the strong law of large numbers or the following conditions for calculating the limiting probabilities of a Markov chain.

Now, we define the probability that the Markov chain $\{X_n\}$ is in the state j at the n^{th} step by $\pi_j^{(n)}$, ie.

$$\pi_j^{(n)} = \Pr\{X_n = j\} \tag{2.18}$$

Thus, the initial distribution of the states is given by $\pi_j^{(0)}$, for every $j \in E$. The matrix P^n converges to the steady state vector $\pi = (\pi_1, \pi_2, \ldots)$ when n becomes large, which contains the probability of being in each state at any time, independent of the initial state, ie.

$$\lim_{n \to \infty} \pi_j^{(n)} = \pi_j \tag{2.19}$$

The Markov chain is said to have **stationary** or **steady-state probabilities** (probabilities do not change with time), π, if the matrix equation $\pi = \pi P$ is satisfied, ie.

$$\pi_j = \sum_i \pi_i P_{ij}, \qquad \text{for every } j \in E \tag{2.20}$$

This system of linear equations is solved by a standard algorithm which takes into account that the parameters π_j must also have the properties that

$$\pi_j > 0 \text{ and } \sum_j \pi_j = 1 \tag{2.21}$$

The limiting probabilities, π_j, always exist and are independent of the initial distribution if the Markov chain is irreducible, aperiodic and time homogeneous. If all the states are transient or recurrent null then $\pi_j = 0$ for all j and the Markov chain is not stationary (invariant). Otherwise, if all states are recurrent non-null a unique stationary distribution exists [6].

that P^n, which contains elements $P^n(i,j)$, each of which is the probability of going from state i to state j in exactly n transitions, is also a Markov matrix.

State j of the Markov chain $\{X_n\}$ is reachable from state i if it is possible to reach state j from state i in a finite number of steps, ie. $P^n(i,j) > 0$ for some $n \geq 0$. If every state can be reached from every other state the Markov chain is **irreducible**, this means that the only closed set is the set of all states. For each state we define $T_j^{(n)}$ to be the probability that the first return visit to state j occurs after n transitions (after leaving j), ie. $T_j^{(n)} = \Pr\{X_n = j, X_{n-1} \neq j, \ldots, X_1 \neq j \mid X_0 = j\}$. Then the probability of ever returning to state j is given by $T_j = \sum_{n=1}^{\infty} T_j^{(n)}$. If the probability of returning to state j is 1, ie. $T_j = 1$, then state j is said to be **recurrent**, otherwise, if $0 \leq T_j < 1$, the j is a **transient** state. If we define M_j to be the time of the first visit to state j, then if $E[M_j] = \infty$ a recurrent state j is called **null**, otherwise, if $E[M_j] < \infty$, the same state j is called **non-null**. A recurrent state j is said to be **periodic** with period $d \geq 2$ if $P^{(n)}(i,j) = 0$ unless $n = vd$ is a multiple of d, and d is the largest integer with this property. If no such d exists then the state j is called **aperiodic**.

Feller [15] describes recurrent non-null, aperiodic states as **ergodic**. If all the states are ergodic, this means that a unique stationary distribution exists (see below). An ergodic theorem gives conditions under which an average over time of a stochastic process will converge as the number of observed periods becomes large, eg. the strong law of large numbers or the following conditions for calculating the limiting probabilities of a Markov chain.

Now, we define the probability that the Markov chain $\{X_n\}$ is in the state j at the n^{th} step by $\pi_j^{(n)}$, ie.

$$\pi_j^{(n)} = \Pr\{X_n = j\} \tag{2.18}$$

Thus, the initial distribution of the states is given by $\pi_j^{(0)}$, for every $j \in E$. The matrix P^n converges to the steady state vector $\pi = (\pi_1, \pi_2, \ldots)$ when n becomes large, which contains the probability of being in each state at any time, independent of the initial state, ie.

$$\lim_{n \to \infty} \pi_j^{(n)} = \pi_j \tag{2.19}$$

The Markov chain is said to have **stationary** or **steady-state probabilities** (probabilities do not change with time), π, if the matrix equation $\pi = \pi P$ is satisfied, ie.

$$\pi_j = \sum_i \pi_i P_{ij}, \qquad \text{for every } j \in E \tag{2.20}$$

This system of linear equations is solved by a standard algorithm which takes into account that the parameters π_j must also have the properties that

$$\pi_j > 0 \text{ and } \sum_j \pi_j = 1 \tag{2.21}$$

The limiting probabilities, π_j, always exist and are independent of the initial distribution if the Markov chain is irreducible, aperiodic and time homogeneous. If all the states are transient or recurrent null then $\pi_j = 0$ for all j and the Markov chain is not stationary (invariant). Otherwise, if all states are recurrent non-null a unique stationary distribution exists [6].

2.2.3 Defining a Markov process

We now construct the following discrete time Markov process, which represents the possible states entered by a single line source. The state space for a line will be $E = \{0, 1, 2, \ldots, s\}$ and the state of a line j, $X_j(t) \in E$, is given by

$$X_j(t) = \min(s, \text{ number of ticks since last cell arrival}) \tag{2.22}$$

Note that $X_j(t) = 0$ means that a cell has arrived. If the number of ticks between cell arrivals is $> s$ then a silent period has been entered; as is evident from figure 2.3, this means that the line source stays in state s, until a burst begins - at which point it moves to state 0. It is clear also, that there is no probability content in moving from state 0 to state $s-1$, they are deterministic transitions - a transition occurs every tick - as nothing further can happen for a period of s ticks; the system thus traverses from state 0 to state s with probability 1. Then, either a cell arrives with a probability of α and it is back to the start of the chain or silence begins with a probability of $1-\alpha$. If $X_j(t) = 0$ this means that an arrival has taken place.

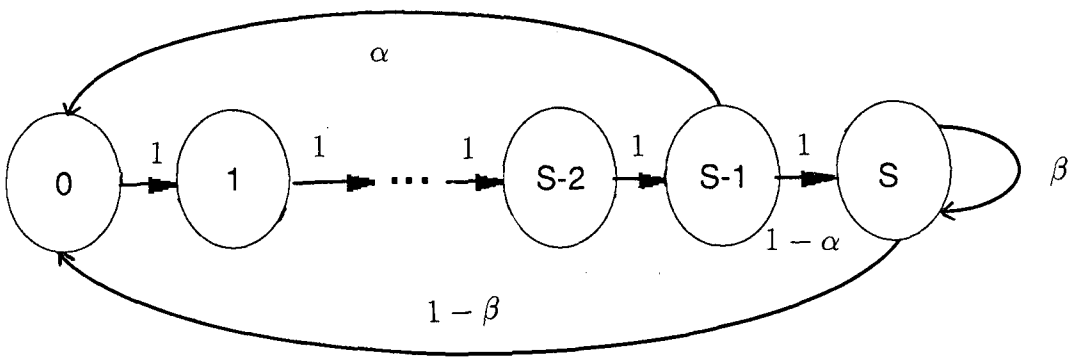


Figure 2.3: Cell level Markov model for a single line source

The process shown in figure 2.3 is Markov because the probability distribution of the line at time n is determined only from it's distribution at time $n-1$, according to equation 2.14. Therefore, a transition matrix, 2.23, can be written for the $s+1$ state Markov model for a single line as graphically represented in figure 2.3. Q is a sparse $(s+1) \times (s+1)$ transition matrix and is made up of the transition probabilities Q_{xy}, see equation 2.15.

$$Q = \begin{pmatrix} 0 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1 & 0 \\ \alpha & 0 & 0 & 0 & \cdots & 0 & 0 & 1-\alpha \\ 1-\beta & 0 & 0 & 0 & \cdots & 0 & 0 & \beta \end{pmatrix} \tag{2.23}$$

The Markov chain has a finite state space and the states form one irreducible closed set and are all recurrent non-null aperiodic. This is clear from an examination of

the transition diagram in figure 2.3: all initial states have a non-zero probability to pass through any other state within s ticks. The invariant distribution is obtained by solving the system of linear equations given by equation 2.20 and the unique solution obtained is

$$\left(\frac{1}{f}, \frac{1}{f}, \ldots, \frac{1}{f}, \frac{\frac{1-\alpha}{1-\beta}}{f} \right) \tag{2.24}$$

where $f = s + \frac{1-\alpha}{1-\beta}$. The reciprocal of f is multiplied by the particular solution, which is the stationary state of the arrival process on each line, $(1, 1, \ldots, 1, \frac{1-\alpha}{1-\beta})$, ie. the vector is normalized, so that the properties 2.21 are satisfied.

The probability that a line is active, P_A, is equal to the stationary probability that the chain is in state 0 (cell arrival), therefore, from equation 2.24 we can write

$$P_A = \frac{1}{s + \frac{1-\alpha}{1-\beta}} \tag{2.25}$$

which is of course equal to the activity rate, A_R, given in equation 2.13.

2.3 Multiplexed system: superposition of sources and modelling the arrival process

2.3.1 Lindley equation with a Markovian workload

Before we talk about the multiplexed system we need to introduce the *Lindley equation*, which is a recursive relation for calculating the waiting time of a customer in a queueing system, we will make the following definitions:

- u_n Instant of arrival of the n^{th} customer.
- v_n Interarrival time of the n^{th} customer, where $t_n = u_{n+1} - u_n$, ie. the time between the arrival of the $(n+1)^{th}$ and the n^{th} customer.
- s_n Service time of the n^{th} customer.
- w_n Waiting time of the n^{th} customer.

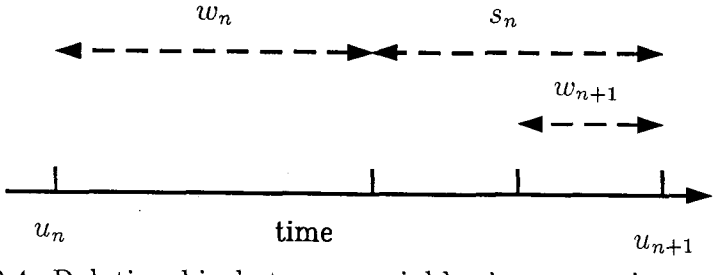


Figure 2.4: Relationship between variables in a queueing system

From figure 2.4 it can be shown for a single server queueing system that the waiting time W_{n+1} of the $(n+1)^{th}$ customer is got from the following recurrence relation

$$w_{n+1} = \max(0, w_n + s_n - v_n) \tag{2.26}$$

Equation 2.26 is called Lindley's equation.

Moving on to the multiplexed system: as a consequence of the multiplexing techniques we can look at the l lines superimposed over a period of s ticks. At each tick of the clock any cells present at the input lines are emptied into the buffer in the multiplexer and the cells are then removed at the rate of one per tick, as described in section 2.1. If there are customers in the buffer this means that there is no gap in the output, ie. the output line is being fully utilized. Also, there must be some protocol to order the different cells arriving simultaneously, this is discussed in section 3.3.2.

Let us examine the virtual queue, which we define to be the queue as seen by the multiplexer - any cells present at a given tick are emptied into the buffer. To specify the collective state of the l input sources we define the following vector of states in $E^{\times l}$

$$\underline{X}(t) = (X_1(t), X_2(t), \ldots, X_l(t)) \tag{2.27}$$

where $X_j(t)$ is defined by equation 2.22. Thus, $\underline{X}(t)$ represents a state of the input lines of the multiplexing system, which is the sum of l Markov processes. Further, we define $A(\underline{X}(t))$ to be the number of cells emptied into the buffer - workload brought by customers - at time t (measured in ticks), ie.

$$A(\underline{X}(t)) = \sum_{i=1}^{l} \delta_0(x_i) = \#\{i \mid X_i = 0\} \tag{2.28}$$

where δ is an indicator (kronecker delta) function, such that

$$\delta_0(m) = \begin{cases} 1 & n = m \\ 0 & \text{otherwise} \end{cases}$$

Now, if we define $q(t)$ to be the number of cells in the buffer at time t, we can relate $q(t+1)$ to $q(t)$ in the same way as the waiting times are related in equation 2.26 as follows

$$q(t+1) = \max(0, q(t) + A(\underline{X}(t)) - 1) \tag{2.29}$$

$A(\underline{X}(t))$ plays the role of the service time and the interarrival time is 1, ie. the output rate of the multiplexer per tick.

In the simulated implementation of the queue we are interested in the queue from the point of view of an arriving packet of data as opposed to the virtual queue. Now, we define $\tilde{q}(t_n)$ to be the number buffer places occupied (cells in buffer) when the n^{th} arrival occurs at time t_n, including the n^{th} arrival itself, ie. the n^{th} arrival is counted as a waiting cell. Similar to equation 2.29, we can make the following relation

$$\tilde{q}(t_{n+1}) = \max(0, \tilde{q}(t_n) + 1 - t_n) \tag{2.30}$$

The one arrival plays the role of the service time and the interarrival time is t_n. We implement equation 2.30 in our simulation and this is discussed in section 3.3.3. Since each arrival is a function of a Markov chain, equations 2.30 and 2.29) are essentially Lindley's equation with a Markovian workload.

2.3.2 Superposition of sources

The multiplexed system is a superposition of the l bursty sources as outlined in section 2.1. Now, in section 2.2.3 we described the packet arrival process for a single line as an alternating renewal process. However, the superpositional arrival process can not be modelled as a renewal process due to its burstiness (high variability) and also because the only renewal process whose superposition is also a renewal process is the Poisson process [26]. Because of this bursty nature the aggregate arrival process is highly correlated between different times [42]. We can describe the resultant queueing system as $lV/D/1$, where there are l sources, the service time is deterministic and there is one server; V is the distribution of the alternating renewal process for one line as given in equation 2.3.

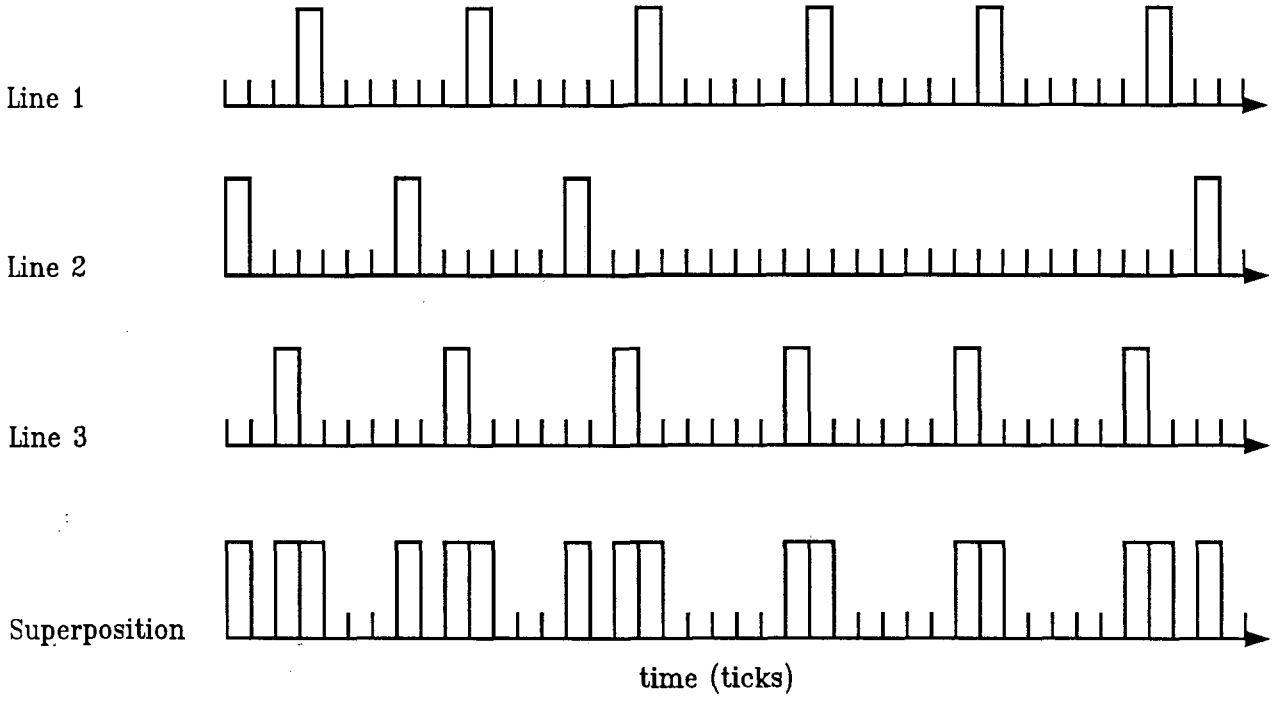


Figure 2.5: Sample traffic for a superpositional arrival process

2.3.3 Stability condition for multiplexed system

From equation 2.25 we know that with l input lines, the average number of active lines is lP_A; this must be less than the service rate, S_R, otherwise the number of cells in the system will grow without bound.

$$\frac{lP_A}{S_R} \leq 1 \tag{2.31}$$

The **offered load** or **traffic intensity**, ρ, is a measure of the demand made on the system and can be defined as the expected number of cells (active lines) arriving at

the multiplexer during a service time. We know that the multiplexer can service one packet per tick, therefore $S_R = 1$. Thus, from equation 2.31 we get the following stability (non-overloaded) condition:

$$\rho < 1 \tag{2.32}$$

with

$$\rho = \frac{1}{\sigma + \frac{1-\alpha}{\phi}} \tag{2.33}$$

where

$$\sigma = \frac{s}{l} \tag{2.34}$$

and

$$\phi = (1 - \beta)l \tag{2.35}$$

$1 - \beta$ is the probability that a silent source will become active, therefore, intuitively we can say that ϕ represents the expected number of inactive lines becoming active.

2.3.4 Cell and burst level traffic

Due to the burstiness of the l sources the queue can be described by two kinds of congestion, namely burst level congestion and cell level congestion. These were discussed briefly in section 1.5.

If we examine a block of s ticks and find that the buffer is never empty during the block (no idle period), then we can describe the congestion as burst level. Burst level congestion is associated with heavy traffic and is determined by the correlations in the arrivals. The queue length at the end of the i^{th} block, q_i, depends only on the queue length at the start of the block, q_{i-1}, and the number of arrivals in the block, A_i, but not their specific arrival times within the block. Now, we adapt equation 2.29 to get the following relationship for the queue lengths from block to block.

$$q_i = q_{i-1} + A_i - s \tag{2.36}$$

The multiplexer can only process s ticks per block, so in the burst level queueing regime there always will be a surplus of cells in the queue at each step. We define the **backwards busy period** (BBP) as the period which began the last time the buffer was empty. In burst mode the BBP extends into the previous block and perhaps beyond, the queue length then depends on a number of correlated arrivals. If $A_0 > s$, this means that $q_1 > q_0$; the probability that an active line stays active, α, is very close to 1 (see section 4.2), consequently $A_1 \approx A_0$, $A_2 \approx A_1$, etc. This correlation of the arrival process is the main reason that the amalgamated arrival process can not be modelled as a renewal process (see section 2.3.2). So, it is very likely that $q_2 > q_1$, $q_3 > q_2$ and so on; consequently, the buffer grows over a large time.

So, in burst level congestion it is clear that the queue at cell level is unimportant. We make the heuristic that a 2-state Markov model, called the block level model, which is a simplification of the cell level model, should be appropriate for burst level

traffic. The queue length is calculated using the following iterative which is based on the equations 2.29 and 2.36.

$$q_i = \max(0, q_{i-1} + A_i - s) \tag{2.37}$$

The block model is discussed briefly in section 2.4.

Cell level congestion relates to the low traffic regime and reflects the fact that it is less likely that customers will arrive at the same time. It means that the queue is empty at some point in the block prior to the given tick, therefore, the (correlated) arrivals from the next previous block do not influence the queue length. So, to keep an accurate track of the queue length distribution the details at cell level are important; equation 2.29 is used to record the queue length in the buffer. The BBP is typically $< s$, which means that the correlations between arrivals are not manifested in the queue, since what happened before the queue was last zero is of no consequence to the current queue length. To characterize the arrivals we define the following proposition.

Proposition 1 *The superposition of l i.i.d renewal processes, each with rate $\frac{1}{\eta}$, tends to a Poisson process of rate $\frac{1}{\eta}$ as l tends to infinity* [26].

Section 4.2 discusses the parameters chosen for simulation in detail, however it is necessary to begin the discussion here. If we fix the traffic at the burst level by fixing α, and scale l and s to infinity, we are scaling the rate at the cell (tick) level. This corresponds to the conditions set by proposition 1. For low level traffic, using proposition 1 we can make the heuristic that the arrival process converges to a Poisson distribution (see section 4.3) as s and l scale, this amounts to ignoring the correlations between successive blocks. For block level congestion proposition 1 is true but it is not relevant, since due to the correlations between arrivals it is not sufficient to only look at the arrivals at the tick time scale.

2.4 Block Level Model

The cell level model for a single line as shown in figure 2.3 can be greatly simplified by considering the behaviour of a line source at block level. Similarly to equation 2.22 we can represent the state space for a single line for block j as $Y_j \in \{0, 1\}$, where this is essentially a projection of the cell level model.

$$Y_j = \begin{cases} 0 & \text{no cell arrival in block } j \\ 1 & \text{(one) cell arrival in block } j \end{cases} \tag{2.38}$$

We wish to establish a relationship between the two processes - described in equations 2.22 and 2.38. We further define equation 2.22 so that X_{jk} represents the state of the line at the jth tick in the kth block. Then, if $X_{(s-1)k} = s$, the line is silent, its not in the overhang or between cells. The next tick is the first of the next block represented by $X_{(1)(k+1)}$ and the last cell arrived in some previous block, ie. no cell arrived in the jth block. Similarly, if $X_{(s-1)k} < s$, a cell arrived in the jth block. So, the relation between the two processes is as follows

$$Y_j = 0 \leftrightarrow X_{(s-1)j} = s \tag{2.39}$$

$$Y_j = 1 \leftrightarrow X_{(s-1)j} < s \tag{2.40}$$

It can be shown that this new block process as represented in equation 2.38 is not Markov.

Now, we define a 2 state Markov model, $\tilde{Y}$ which is represented by the state transition diagram in figure 2.6.

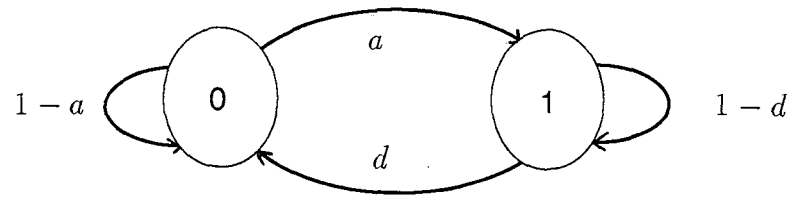


Figure 2.6: Block level Markov model for a single line

The transition matrix, P, for this Markov chain can be written as follows

$$P = \begin{pmatrix} 1-a & a \\ d & 1-d \end{pmatrix} \tag{2.41}$$

where a and d are defined as follows:

$$a = \Pr\{\tilde{Y}_{j+1} = 1 \mid \tilde{Y}_j = 0\} \tag{2.42}$$

$$d = \Pr\{\tilde{Y}_{j+1} = 0 \mid \tilde{Y}_j = 1\} \tag{2.43}$$

Now, from equation 2.42 we know that a is the probability that a silent line becomes active, when we compare this with the cell level model it means that $a = \Pr\{\text{silence} \leq s\}$. Then, using equation 2.9 we can write $a = 1 - \beta^s$. $1 - \beta$ is a very small value since β is close to 1 (see section 3.3.3). Using the binomial theorem we can approximate $(1 - (1 - \beta))^s$ by $1 - s(1 - \beta)$) and we get the following relation.

$$a = s(1 - \beta) \tag{2.44}$$

Similarly comparing equation 2.43 with equation 2.1 we get the following relation.

$$d = 1 - \alpha \tag{2.45}$$

If we fill equations 2.44 and 2.45 into the formula for the load (see equation 2.33) we get the following expression for the load in the block level model.

$$\rho = \frac{a}{\sigma(a+d)} \tag{2.46}$$

From equation 2.32 we get the following relationship for a stable regime.

$$\sigma > \frac{a}{a+d} \tag{2.47}$$

As was discussed in section 2.3.4 the 2 state Markov block model can be viewed as an approximation for the burst level component of the queue of an ATM multiplexer. It has been shown that the upper bound for the tail (indicates heavy traffic) of the queue length distribution is of the following form [4].

$$\Pr\{\text{queue length} \geq b\} \leq z^{-l} y^{-b} \quad \text{for any } b > 1 \tag{2.48}$$

where $z > 1$ and $y > 1$ are given in terms of the parameters of the model, a, d and σ. In section 4.2 we describe how we use the upper bound to choose values for the parameters in our simulation runs.

Chapter 3

Simulation of an ATM multiplexer

3.1 Introduction

First of all we will define a system as a collection of interacting parts, a system is either deterministic or stochastic. Stochastic systems, unlike deterministic systems involve random, unpredictable events and this is the kind of system which interests us. Also, we will define a model as an abstract representation of a physical system.

Imagine the following situation - some new system has been proposed and we wish to examine the implications of the system for different parameters. From an analytic point of view, the problem may be too difficult to solve or the level of detail provided by the analytic results may not be comprehensive enough for our needs. To learn more about a particular problem one possible solution is to put the concept into practice on an experimental basis and see what results are produced. However, constraints such as money, technology, risk and time often make such experiments infeasible, indeed only rarely does one have an opportunity to perform large scale *in situ* experiments in the real world.

The analyst thus is unable to use an analytic approach or to perform a real-time experiment, a solution then would be to simulate the system, which make a systematic study of the problem possible. Simulation has been around a long time, armies the world over have simulated real battle situations in the save haven of their training grounds, airline pilots have used flight simulators, etc.

Simulation has been used to analyze both theoretical problems in Science and practical real-world problems, its success has been in parallel with the advance of the digital computer. Simulation is most useful when analytic and numerical techniques cannot supply answers; it is essentially a statistical sampling experiment that is used, in conjunction with a model, to obtain approximate answers for multi-parameter probabilistic problems, by generating external stimuli and observing how the system reacts over time.

3.2 Discrete and continuous simulation

Simulation models analyze the behaviour of systems as a function of time and therefore we talk about discrete simulation and continuous simulation. In discrete simu-

lation, the system is examined only at definite points in time, whereas in continuous simulation the system is monitored at every point in time.

In both types of simulation the end objective is to collect statistics which describe the behaviour of the simulated system and the way in which these are collected is the main factor in determining if a system is discrete or continuous. In discrete systems the statistics are collected only when certain events take place and as will be evident from section 3.3.3 it is discrete asynchronous (events can occur at any time) system simulation that we are interested in.

One of the attractions of simulation is that once the initial conditions and final states for a particular simulation run are known, then various runs can be rerun or replicated. For example, if after analyzing the output of a simulation run it was decided that certain unrecorded data might explain the results obtained, then the simulation could be rerun with the same initial conditions once the extra data collection was added to the system. This can be useful in the development stage. More useful is the ability to replicate any experiment, in other words rerun it with selected changes made to the parameters.

3.3 Next-event simulation

3.3.1 Outline of next-event simulation

Next-event scheduling is one of the common approaches to discrete simulation and was the approach used in developing the simulation system. The implementation proceeds as follows: select the first event on the time scale and perform all associated actions including putting any newly generated events in their proper order on the time scale. With the events now updated on the time scale, choose the next event and repeat until a desired simulation period is covered.

3.3.2 Event definition

In section 2.1 we introduced the concept of l telephone lines sharing a transmission line. The sharing of the communication bandwidth is scheduled by ATM multiplexing where each terminal transmits short fixed length cells, see section 2.1. Also in this section we learnt that if a telephone line is firing (talk spurt) this means that packets of data are being transmitted and if it is not firing this means that the talk spurt has ended and that there is silence on the line.

Our simulation start-up policy (see section 3.7) positions the l input lines over a block of s ticks and marks them as firing or not firing. The position a line takes is reserved for the packet of data that the line will transmit if it is firing or when it starts firing. Another way of looking at it is that if the line is inactive, the cell contains silence and the probability of speech activity in the next packet is deterministic, with probability γ, see equation 3.2. The set of numbers indicating activity and firing time for every input represents the state of the system and is sometimes known as the **system image** [21]. An event is something which can make changes to this system image. Now, we ask ourselves, what activities can cause events? If we keep in

mind that every line in the system is examined every s ticks and less when an inactive line becomes active, we conclude that the activities which can cause events are:

(i) A packet arriving at the multiplexer forms part of a burst for one input; also, the packet may be the last cell in the burst in which case the activity flag (see section 3.5.2) will change from firing to not firing.

(ii) Á line which is currently silent may begin speech activity, in which case the activity flag will change from not firing to firing and the firing time will also change.

We mentioned in section 2.3.1 that we must have some protocol for cells arriving simultaneously, ie. at the same tick. It is essentially random in that whichever line gets into position first will be dealt with before other cells at the same tick. The system cycles from line to line in the order of which they are positioned in time, until some limit on the simulation length is reached (see section 3.4.1); this is the essence of next event scheduling.

3.3.3 Event actions

This section describes the actions which occur when an event takes place and is outlined in detail in the flowchart shown in figure 3.5. These actions are the heart of the simulation loop and form the algorithm that is repeatedly executed to carry out the simulation.

The status of the input line is examined, it is either active (firing) or silent (not firing). If it is firing the following actions take place in sequence.

Action 1: Increment the buffer, B, in the multiplexer.

Action 2: Packet processing.

Action 3: Check to see is the talk spurt is going to end.

When action 1 occurs it indicates that the packet is ready for processing; the queue length is then stored so as to build a probability distribution of the buffers queue length (data collection). In section 2.1 we described how we were modelling a buffer of infinite capacity. For practical reasons, ie. due to restrictions on RAM size (see section 3.4.3) a maximum value, M_B, is given for the buffer, which is a finite approximation of an infinite buffer. For high load values (see section 4.2) the queue grows very large and so we wish the buffer (distribution) to be truncated for modelling or simulation purposes. If the queue length overflows the given maximum value at any stage a count is kept of the number of cells lost, ie.

$$B = \min(B + 1, M_B) \tag{3.1}$$

Referring to action 2, one packet can be processed per tick so we work out out many ticks have passed since the last processing (interarrival time) and decrement the buffer by this amount, this is outlined by equation 2.30. So, at each tick any cells

arriving at the multiplexer from any input line are emptied into the buffer. The cells are the removed from the multiplexer buffer at a rate of one per tick. Actions 1 and 2 constistute the implementation of equation 2.30.

In section 2.2.1 we gave the probability of an inactive line becoming active as $1-\alpha$. Action 3 involves sampling against this probability and if the result is that the talkspurt ends, then we set the activity flag to not firing (indicates a departure from the system).

An inactive input line means that currently that input is experiencing a period of silence, so if it is inactive we:

Action 4: Check to see if a talk spurt is about to begin.

If the line is to become active within the next s ticks, it is removed from its current position on the time scale and reinserted at some point within the next s ticks, see flowchart 3.6. We can conclude that the events are a subset of the activities in the system and that the events when data collection takes place are a further subset.

As mentioned above, we check against the probability γ to see if an inactive line will start to fire within the next s ticks, practical values for γ are discussed in section 4.2. If the answer is yes, then we must decide the criteria for where to position the line. From equation 2.9 we can write the probability that the silence is less than s ticks, γ, as

$$\gamma = 1 - \beta^{s-1} \tag{3.2}$$

Using equations 2.5 and 3.2 we can write

$$\Pr\{\text{silence} = t \mid \text{silence} \leq s\} = \frac{\Pr\{\text{silence} = t\}}{\Pr\{\text{silence} \leq s\}}$$

$$= \frac{(1-\beta)\beta^{t-1}}{1-\beta^s} \tag{3.3}$$

Later, in section 4.2, we will show that β is always near 1 - this also means that γ is very small and consequently that the average silent period is large (in terms of ticks). Applying L'hopital's rule to equation 3.3 we can write

$$\Pr\{\text{silence} = t \mid \text{silence} \leq s\} = \frac{1}{s} \tag{3.4}$$

Therefore, we can approximate the silent period by a uniform distribution.

3.4 Simulation system

3.4.1 System description

In section 3.3.2 we mentioned that the simulation system cycles from line to line, essentially what this means is that the simulation process is carried out block by block. For each block a doubly linked list of the inputs, representing the superposition of l inputs over a block, is traversed. The system counts time by counting the block number and within a block it counts tick by tick. So, in the way that was described

in section 3.3.3, each line is continuously examined every block and the appropriate actions are taken.

The system is designed so that all input and output takes place through ASCII character files. The first input file contains the following parameters:

- *in_f_name.* This is the main input file name (see below).
- *summary_f_name.* This file is where the summaries for all the simulations in the batch will be written.
- *start_seed_f_name.* This file contains the starting seeds for the random number generator, see section A.

The file *in_f_name*, contains the parameters which fully describe the inputs to the multiplexer, the initial conditions for the system and the output file names; they are as follows:

- l. The number of input line sources to the multiplexer.
- s. The sampling period, which is the same for all the line sources.
- ρ. The load, ie. the mean number of arrivals per tick.
- ϕ. Used to fix traffic: $l(1-\beta)$, see equation 2.35.
- *sim_run.* The inputed simulation length is in the form of the number of blocks.
- *max_queue_size.* The given limit for the infinite buffer, see section 3.3.3.
- *initial_queue.* This specifies the initial number of cells present in the buffer when the simulation begins.
- *last_process_time.* This represents the time ($\in \{0,\ldots,s\}$, for reasons given above) when the multiplexer last processed the system. This will be non zero if a previous simulation is being continued and zero otherwise.
- *initial_flag.* This boolean flag indicates whether initial conditions are given.
- *seed_flag.* This boolean flag indicates whether alternative seeds are given for the random number generator, which will take precedence over the seeds given in *start_seed_f_name.*
- *output_f_name.* This is the name of the file which will contain the resultant probability distribution from the simulation, all other parameters and results are contained in *summary_f_name.*
- *initial_conditions_f_name.* If *initial_flag* is true then this file contains the initial system image (see section 3.3.2). The system image is dependent on the length of the queue, so if *initial_queue* is non-zero then the initial image of the system must be given, otherwise the system generates an initial image as described in section 3.7.

- *seed_f_name.* If *seed_flag* is true then this file contains the seeds to reset the random number generator.

- *sys_state_f_name.* This file is where the final system image will be recorded.

These parameters are duplicated for every simulation to be run in the batch.

As we mentioned in section 3.3.3, we measure the queue length every time a packet arrives at the multiplexer, therefore, we are measuring the queue length as seen by the arriving packets. On average there will be $s.\rho$ measurements per block. This is different from the virtual queue length which would be obtained if we measured the queue length every tick, which would mean s measurements per block, regardless of activity.

We can run a batch of simulations which in reality is just one long simulation with the results outputed at various intervals. This can be done so as to examine how the probability distribution builds up over time (see section 3.8), or to facility the running of longer simulations. The simulation outputs can then be joined (see section C.4) and the output viewed as if the simulations were run as one.

The main output we want to obtain is a probability distribution of the queue length of the buffer in the multiplexer, resulting from the arrival process of the l bursty inputs. The data recorded in file *out_f_name* is the log of the probability distribution that the queue length, q, is greater than or equal to the buffer size, ie.

$$\log[\Pr\{q \geq b\}], \quad \forall\, b \in [1, M_B]$$

Some statistics describing the behaviour of the system are also produced: total number of cells arriving at the multiplexer, probability of cell loss (if the buffer happens to have overflowed), percentage of blocks where the buffer was continuously non-zero (see section 4.6.4). For the run lengths discussed in section 3.8 the total number of arrivals at the multiplexer were between the order of 10^8 and 10^9. The source code for the system is included in C and the executable code and help files are included on the accompanying disc, an index of which is included in section C.5.

3.4.2 Representation of time

The term *simulated time* refers to how many units of simulated time have passed since the beginning of the simulation. In our system a unit of simulated time is a tick, which represents the service time of the multiplexer. The tick width (length in real time) is decreased as we increase s, this is discussed in section 4.2. There is of course no connection between the simulation time and the real time (actual time taken for simulation), is the *real time* is dependent on how many events occur. In our system the simulation could easily take many times longer that the actual operations being simulated or it could also take shorter time, it depends on the number of inputs; see table 3.1 below.

3.4.3 System performance and limitations

The simulation system was developed using object oriented design and programming methods. There is a possible run-time overhead associated with the object oriented

l	ρ	s	Blocks	Real time	Sim. time	Clock
50	0.82	20	10^5	2	2.6596	33
10000	0.82	3979	10^5	630	2.6596	33
50	0.82	20	10^7	162	265.96	33
50	0.82	20	10^7	108	265.96	50
50	0.9	15	10^7	158	265.96	33
200	0.9	58	10^7	642	265.96	33
400	0.9	116	10^7	1336	265.96	33
2000	0.9	579	2.10^5	1497	51.1911	33

Table 3.1: Comparing simulation lengths

design, this is a trade-off with robustness, maintenance, extendibility and compatibility (see A).

The main limitations of the system are the memory available on the host machine and also the clock speed of the computer which can really affect the real time of the simulation. From our experience, a user intending to run simulations using the developed system would need a machine with a minimum of a 80486 dx processor and a 33 MHz clock, otherwise simulations run will just take too long to be practical. Simulations were also run on a machine with a motherboard speed of 50 MHz and the corresponding performance gain was approximately 35%. The size of the multiplexer buffer is limited by the amount of RAM available as mentioned in section 2.1. We found that a 4 Mb RAM system had enough memory for a maximum buffer of size 7500, whereas moving up to 16 Mb, we did not experience any such problems. So, 4 Mb should be considered a minimum requirement, otherwise the maximum buffer may overflow, due to its limited size, and we will not be able to make accurate predictions.

The real time for the simulation increases linearly with the number of inputs, l, and the simulation length in blocks as shown in table 3.1; time is (measured in minutes). The value for the load or the mean lines active per tick, ρ, does not affect the simulation length that much, ie. for practical values of ρ (see section 4.2), the number of active lines will not change so much as to make a significant difference on the number of operations necessary. The number of operations per block depends on the number of input lines and the total number of operations depends on the number of blocks. For long simulation runs the cycle length of the random number generator should be kept in mind (see section B). Table 3.1 also shows how the the real time is affected as we vary the clock speed (MHz) of the CPU. The only parameter affecting the simulated time (measured in minutes) is the number of blocks being simulated, this is explained in more detail in section 4.2.

3.5 System objects

3.5.1 Introduction

The data structure of the system is directly patterned on the objects whose behaviour is being simulated. C++ classes provide the system structuring mechanism and each class corresponds to a meaningful data abstraction (see A). The system makes use of seven classes and the following sections describe them.

3.5.2 Class node_c

An input object or an instance of class node_c represents one input line. For any given simulation there will be l inputs and consequently l node_c objects. An input object is characterized by the following private data members:

- *line_no.* Each input line is given a line number (between 1 and l) for identification purposes.
- *firing_flag.* A flag which is either 0 or 1 depending on whether the line is active or inactive respectively.
- *firing_time.* This is a value between 1 and s. If *firing_flag* $= 1$ this represents the time at which a packet will arrive at the multiplexer. If *firing_flag* $= 0$ this represents the time at which the line will be examined to check to see if it will become active within the next s ticks.
- *prior.* A pointer to the previous line in the block.
- *next.* A pointer to the next line in the block.

Class line_list_c is a friend of node_c which means that an instance of line_list_c has access to the above members. The main service available is a constructor function which is used to link the input lines together.

3.5.3 Class line_list_c

One instance of this class is created in the main program and represents the state of the system (system image) over a block of s ticks. The class has the following attributes:

- *head.* A pointer to the first node in the list.
- *foot.* A pointer to the last node in the list.
- *last_process_time.* The last tick at which the system was processed.

There are a number of routines which may be implemented on the class instance:

- *Constructor function.* This is used to create instances for head and foot, initialize last_process_time to zero and call connect_two_nodes().

- *connect_two_nodes()*. This function links two nodes and is used to link head and foot initially.

It should be noted that the linked list is made up of instances of node_c and is setup in the main program using the constructor function of node_c, see flowchart 3.4.

- *sort_linked_list()*. A routine used to sort the linked list in order of firing time. If random initial conditions are set then the list will initially not be in order on the time scale so it needs to be sorted.

- *one_block_simulation()*. This is the most important member function and is defined as a virtual function (see B). It carries out the procedure described in 3.3.2 and is outlined in figure 3.5. This function is a friend of class queue_c,which stores the queue length distribution, and accesses that data structure so as to build up the queue statistics. The number of arrivals at the multiplexer is counted for each block and a running total is kept and at the end of the simulation this total used to build the probability distribution (see section 3.5.5).This function uses the next procedure to control the random behaviour of the inputs.

- *move_line()*. If a line enters a bursty mode after a period of inactivity it is first removed from the linked list and then reinserted at the appropriate position (see figure 3.6). This module controls this moving of a node in the list and makes use of the following other functions.

- *remove_head()*. Removes the head of the list.

- *remove_foot()*. Removes the foot of the list.

- *remove_line()*. Removes a node in the list other than the head or foot.

- *clear_pointers()*. Removes the pointers from a (removed) node.

- *new_head()*. Inserts a node at the head of the list.

- *new_foot()*. Inserts a node at the foot of the list

- *insert_mid_before()*. Inserts a node before a given node.

- *insert_mid_after()*. Inserts a node after a given node.

3.5.4 Class probabilities_c

This user defined data type completely defines the traffic parameters of the input objects and it has the following attributes:

- l. The number of input source lines.

- s. The sampling period of the multiplexer (also the number of ticks in a block).

- ρ. The mean number of active lines in a block, otherwise known as the load.

- α. The probability an active line stays active, see equation 2.1.
- β. The probability an inactive line stays inactive, see equation 2.2.
- γ. The probability that the silence is $< s$ ticks, see equation 3.2.

Once the above probabilities are set they are constant throughout the simulation run. The following set of operations are provided:

- *Constructor functions.* Three initializers are provided which initialize the above data members depending on the input given. Figure 3.3 shows how an object is created. If the user specifies a value for ϕ (see equation 2.35) then either the sampling period s or the load must be given. The remaining parameter is calculated using equation 2.33.
- *return_sigma().* See equation 2.34.
- *return_prob_line_active().* See equation 2.25.

There are functions which set and return functions the above variables and the following functions are used in the initializing of the variables:

- *fix_alpha().* Alpha is set as the default value (see section 4.2) if no argument is supplied, this default value was used for all the simulations carried out. The function calls calc_beta().
- *fix_beta().* Facility for fixing beta, which also calls calc_alpha().
- *calc_alpha().* Facility to work out α, if it is not fixed, using equation 2.33.
- *calc_beta().* Works out β using equation 2.33.

These functions calculate the respective variable values using equation 2.35.

- *calc_s_given_phi().* Used if ϕ and ρ given.
- *calc_rho_given_phi().* Used if ϕ and s given.
- *calc_beta_given_phi().* Used if ϕ given.

The member function of line_list_c, *one_block_simulation()* is a friend of this class for ease of performance.

3.5.5 Class queue_c

There is one instance of this class for each simulation and it has the following attributes:

- *queue.* A pointer to a dynamic array (created at run-time). This array is used to store the frequency of different queue lengths in the multiplexer buffer. The simulation loop (see figure 3.5) included in *one_block_simulation()* builds up this array, we also keep track of how frequently the buffer overflows, if at all.
- *max_queue_size.* The maximum buffer size, defined at run-time.
- *over_flow_size.* This is used to index the last element in the array which keeps track of the number of cells lost.
- *actual_queue_size.* This is known once the simulation is finished and may equal max_queue_size.
- *total_activity.* The number of packets that have arrived at the multiplexer and have been processed or are in the queue.
- *buffer.* The size of the queue at any given moment; if this is bigger than max_queue_size then then queue array element which is incremented is indexed by over_flow_size.

The member functions which operate on the data are:

- *Constructor function.* This creates the dynamic array and initializes it.
- *add_total_activity.* This is used by the function *one_block_simulation()* to total the cells handled by the multiplexer.

The following functions are used once the simulating is over to calculate the final probability distribution:

- *setup_pdf().* This creates a probability density function from the array of queue lengths by normalizing the data using *total_activity.*
- *build_cdf().* A cumulative density function is then built which represents the $Prob[q < b], \forall\, b \in [1, M_B]$.
- *build_one_minus_cdf().* This calculates the $Prob[q \geq b], \forall\, b \in [1, M_B]$.
- *log_queue().* This is used to get the $\log_{10}$ of the array so we can make a graphical presentation of the log probability distribution.

3.5.6 Class mlcg32_c

This class represents the random number generator (see appendix A) and has two attributes which are the generator seeds. Two constructors allow the seeds to be read from a file or manually set. There is also a reset function which has a default of unity for both seeds. The starting and finishing seeds are always saved for each simulation so that a simulation can be rerun, replicated or continued at some future point (see section 3.7).

3.5.7 Class stop_watch_c

This class controls the timing of the simulation in real-time and displays the start time, end time and simulation time.

3.5.8 Class string_c

An instance of this class is created for every character string used in the system and is used for better memory management.

3.6 Main program and flowcharts

The main program is shown in diagrams 3.1, 3.2, 3.3 and 3.4. The main procedure is shown in diagram 3.1, it calls the module *control_simulation()* shown in figure 3.2, which, as the name suggests, controls the simulation and the collection and recording of data.

Initially it inputs the data from the main input file as described in section 3.4.1 and it is outlined in some detail in figure 3.2. An instance of a list object is created with initially only the head and the foot of the list linked. Instances of the queue object and the probabilities object (see figure 3.3) are then created. Once all the conditions are checked a list is set up for the l inputs, linking from the head to the foot. Class member function *one_block_simulation()*, which traverses the input lines for one block, is then called for the given number of blocks. The simulation output is recorded in a summary file and the queue data is written to a data file. The process is repeated for every simulation in the batch.

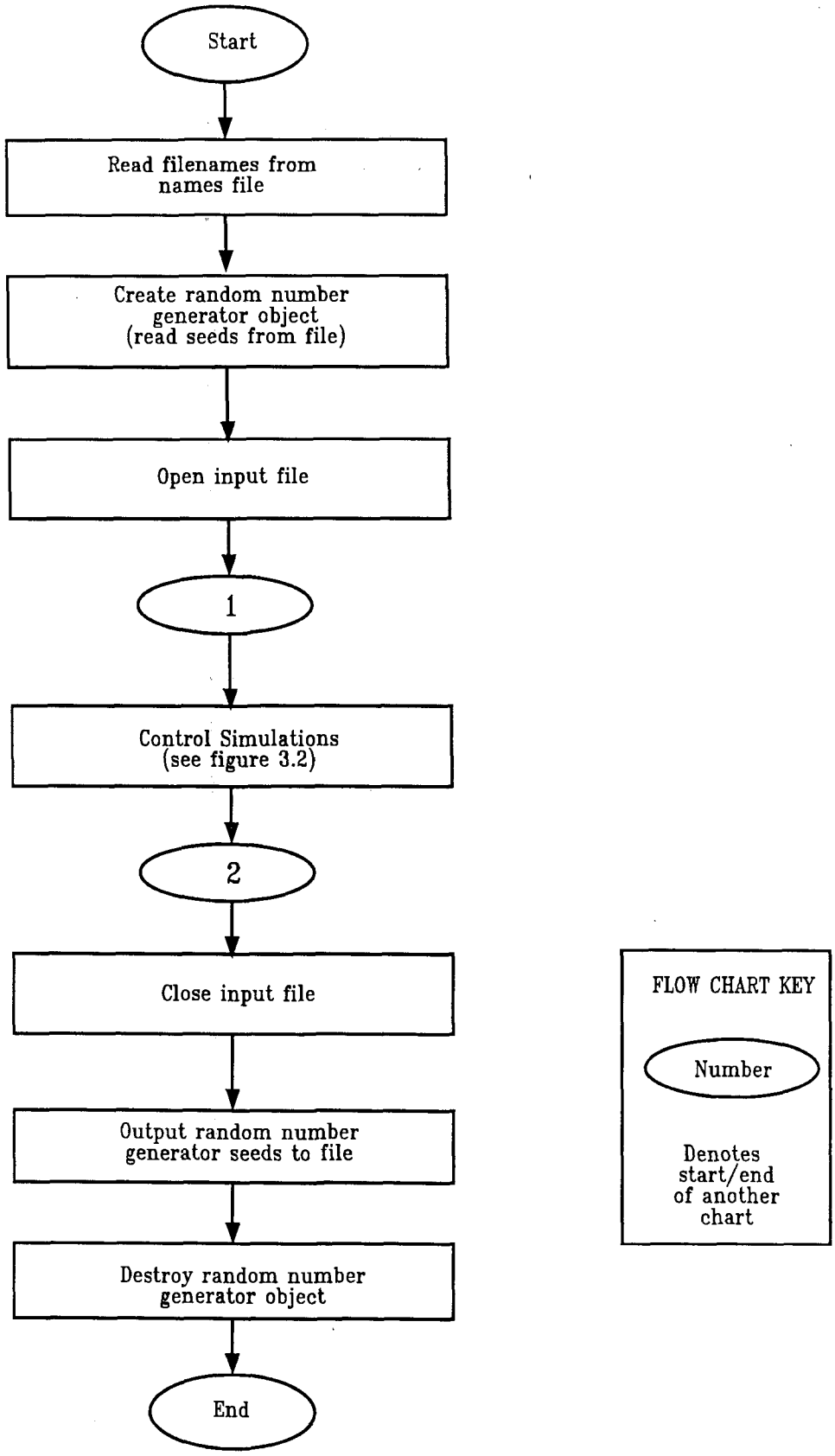


Figure 3.1: The main program

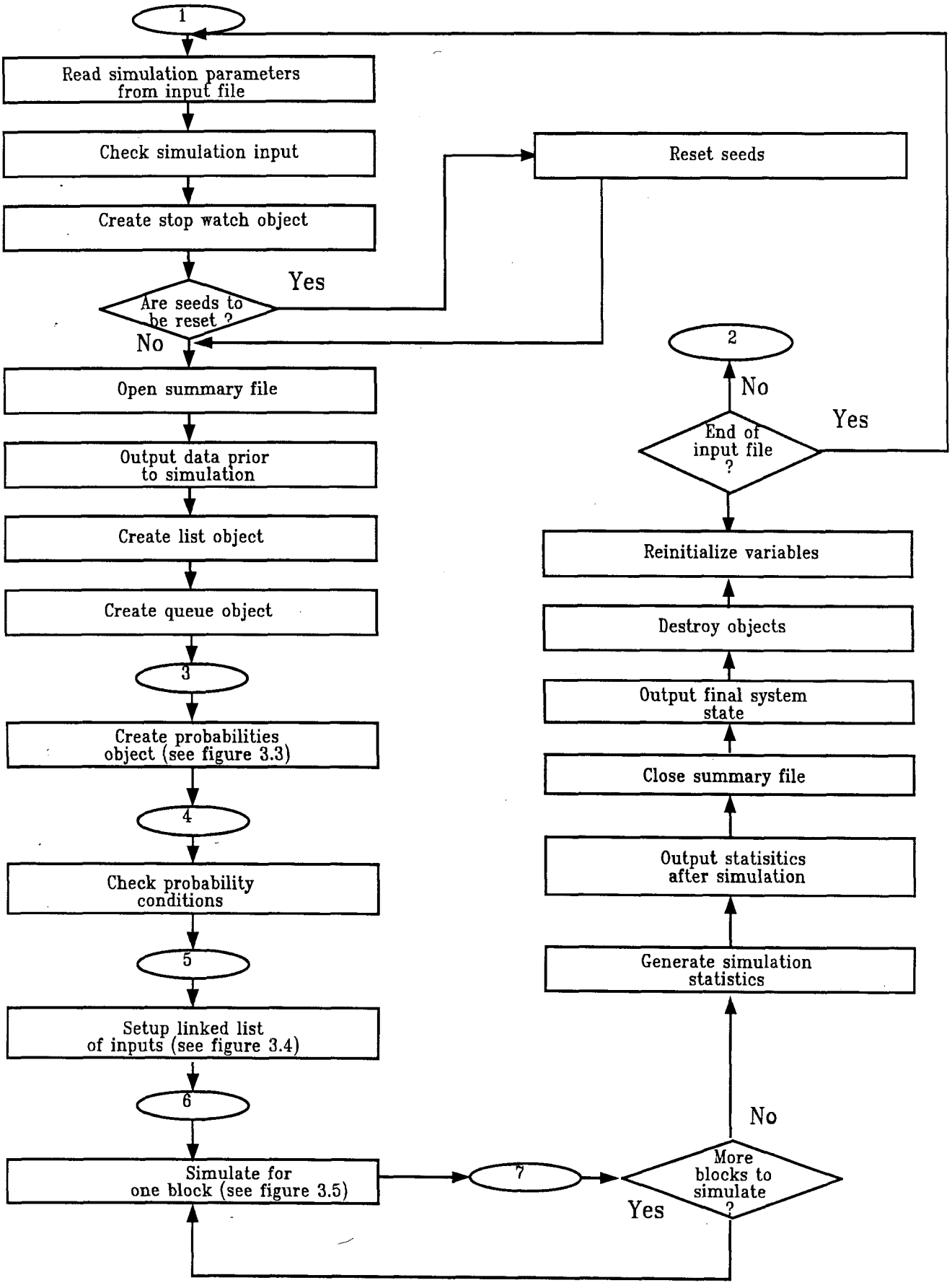


Figure 3.2: Procedure *control_simulation*

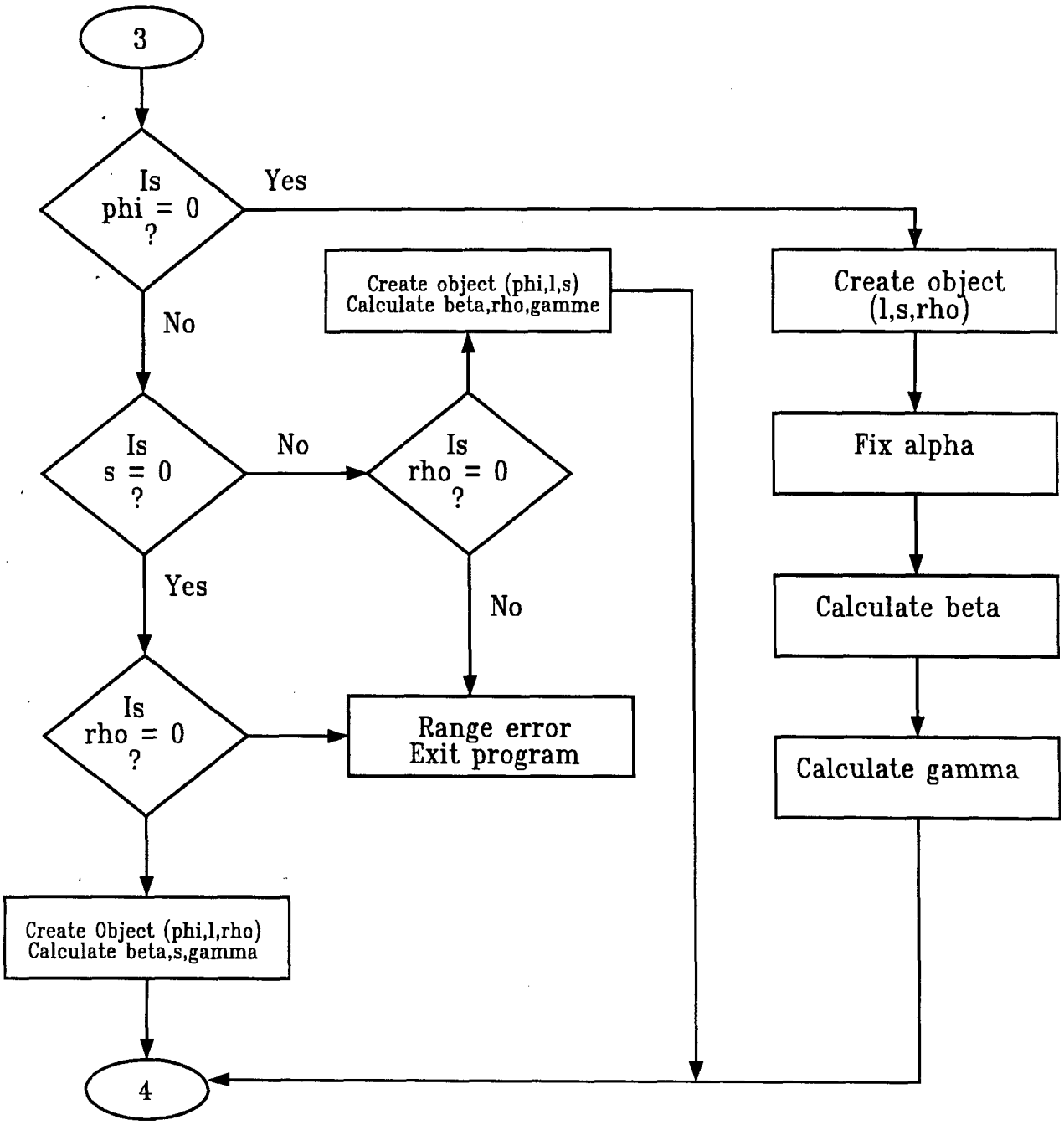


Figure 3.3: Creating a probabilities object

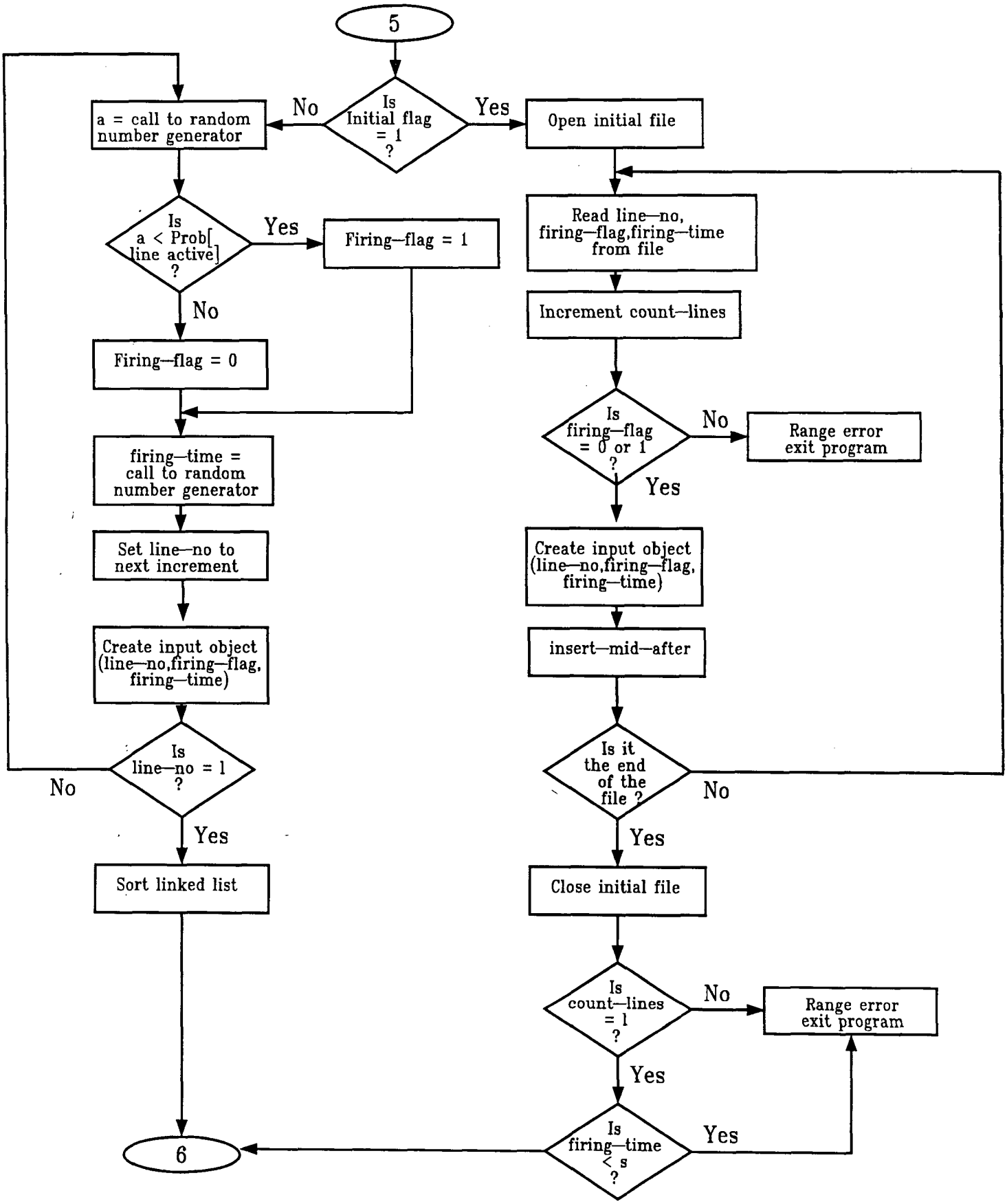


Figure 3.4: Setting up a linked list

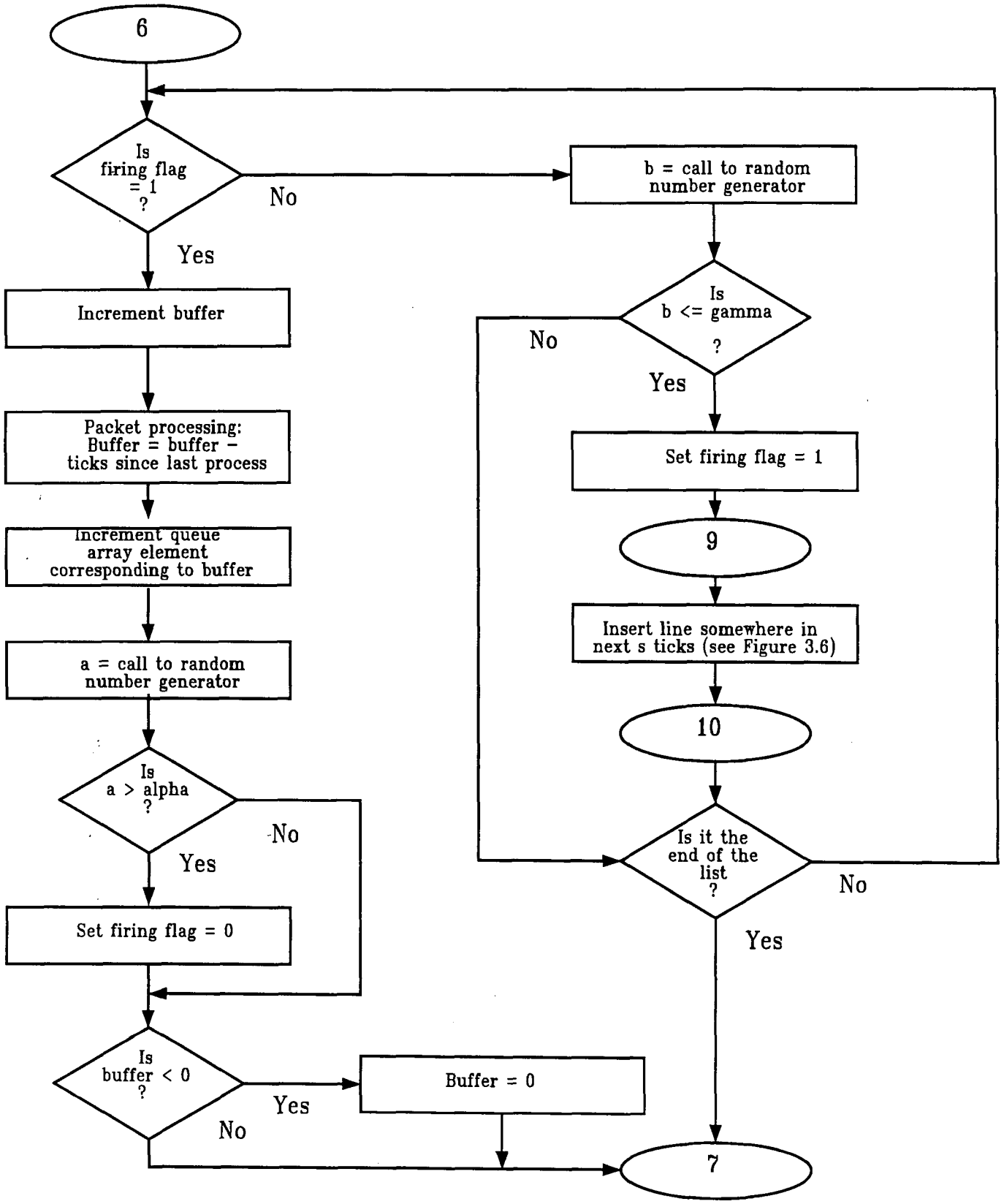


Figure 3.5: The simulation loop

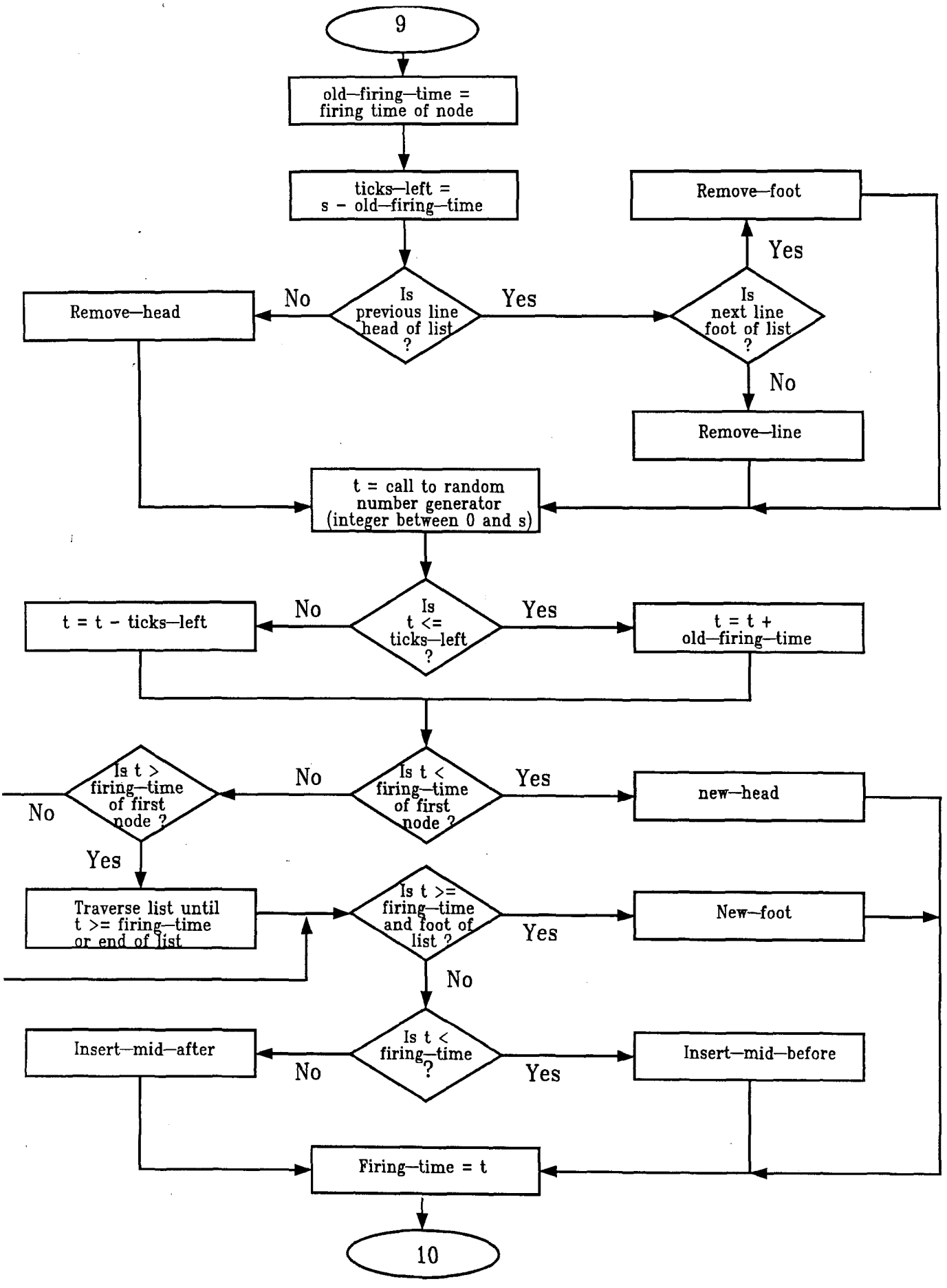


Figure 3.6: Moving a node in the list

3.7 Start-up policy

In section 3.3.2 we mentioned that our start-up policy positions the l input lines over a block of s ticks and marks them as firing or not firing. This was further outlined in the flowchart of figure 3.4 and the relevant code can be found in the listing for the main program in C.3. We now clarify and expand on how the simulation system is started.

Some problems which are frequently encountered, when using a computer simulation model, are [7, 21]

1. How to start the model?
2. How to obtain measurements that are not biased by the starting or stopping of the model?
3. Whether it is better to make a single continuous long run or to make an equivalent number of independent (different random stream) and shorter runs, ie. replications.

None of these issues can be dealt with in isolation as they are all interconnected.

The assumption concerning the initial conditions for starting the model is important and its validity needs to be accessed before one can provide answers for the other two problems. For many models the problem is that it takes some time for the simulation to overcome the artificiality introduced by the abrupt start-up, in other words to "warm-up". The performance of the simulation in some initial period is then distorted. We have to decide if this initial bias exists and if so how it can be eliminated. There are two general approaches which can be used to reduce the bias [21].

(i) The system can be started in a state which is more representative than the empty and idle state (defined below).

(ii) Ignore the data produced from some initial period.

The most common approach is method (ii), ie. to eliminate an initial section of the run, perhaps choosing starting conditions which make the necessary excluded interval as short as possible. in our case, we used the first method to reduce the initial bias and a description of the method and the reasons for using it follow.

Probably the most common way for starting a simulation is in the empty and idle state, in other words all queues are empty and all facilities are idle. In our case the facilities means the input lines and the multiplexer server. However, since we are simulating voice traffic for l input lines it is extremely unlikely that all the lines will be inactive in reality, also, our assumption A1 (see section 2.2.1 was that the phones are continuously off-hook. The system then needs to be started in a state which is reasonable and more representative of the system, this can be done from our em a prior knowledge of the system.

Equation 2.25 gives us the probability that a line is active, P_A, per tick. The probability that a line is active during a block is then sP_A, ie. $\sigma\rho$; an input line is

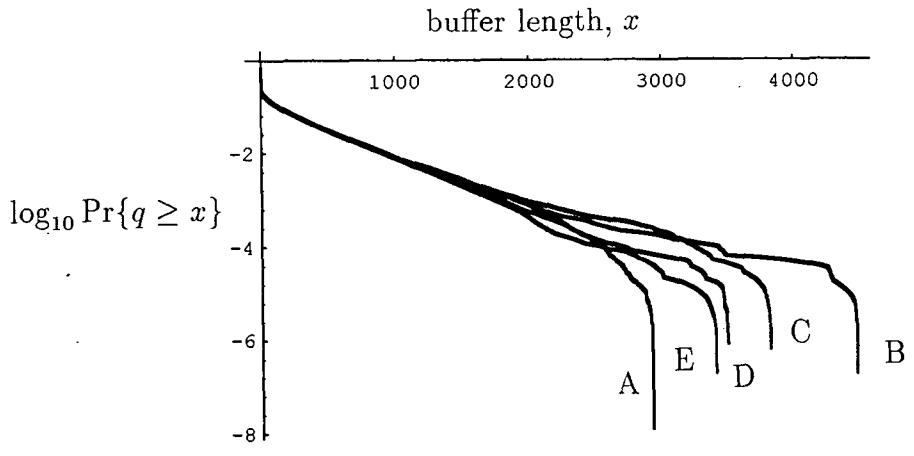


Figure 3.7: Effect of different initial conditions: see table 3.2 for a key to the plots

Plot	l	ρ	s	Blocks	% time non-empty
A	50	0.82	20	10^6	17.44
B	50	0.82	20	10^6	17.44
C	50	0.82	20	10^6	17.42
D	50	0.82	20	10^6	17.18
E	50	0.82	20	10^6	17.35

Table 3.2: Simulation parameters for figure 3.7 and % time buffer non-empty

marked as firing or not firing by sampling against this probability. Next, we have to decide how the l input lines should be positioned over the block of s ticks (see section 3.3.2). The "silence" before the first packet from each active line arrives at the multiplexer, must be less than s ticks (since the lines are active). Therefore, we know that it is uniformly distributed, ie. the first arrival on each line is uniformly distributed by assumption. So, to position each input line over the block of s ticks we generate a random number between 1 and s inclusive (see flowchart 3.4). We start the buffer in the empty and idle state.

Previously, in section 3.4.1, we said that the system image is dependent on the queue length. This comes from the fact that when the system has a particular system image, it also has an associated queue length for the multiplexer buffer. Figure 3.7 shows the results of 5 pilot simulation runs we conducted, using the same sequence of random numbers in each case, our objective was to test the above heuristic regarding initial conditions. Plot A shows the probability distribution obtained from a simulation begun with the default initial system image and empty queue. Plots B, C, D and E result from initial conditions comprising of system images and associated queue lengths obtained from the final values of the previous simulations, A, B, C and D respectively.

We note that initial conditions obtained from previous simulations may be unrepresentative in that the particular simulation may have ended when the system was

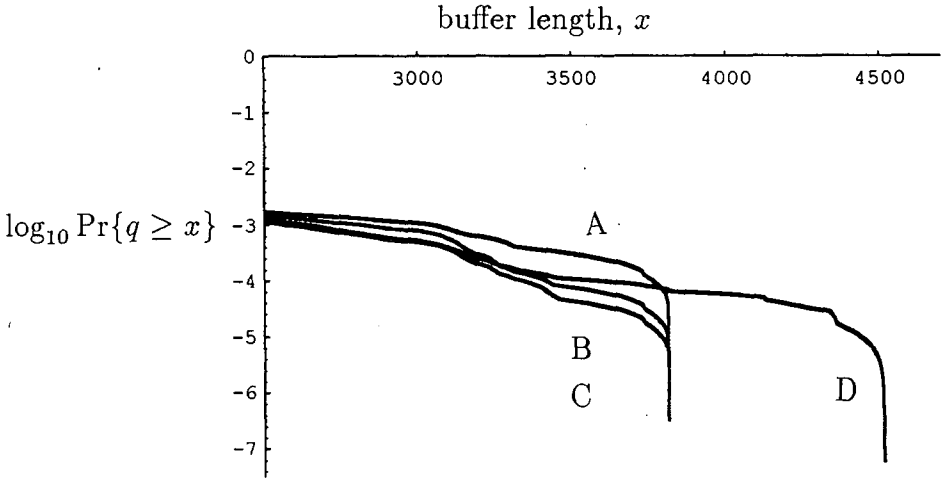
in an unusually busy burst period or in an unusually quite silent period. Examining figure 3.7 we can see that the initial conditions do not decrease the level of noise at the tail probabilities. So, we conclude from figure 3.7 that the default initial conditions are representative of the system. Our second problem is also then answered: by choosing representative initial conditions the initial bias is reduced and consequently we can begin collecting simulation data from the very start.

The percentages of time that the queue was non-empty in a block was also collected for each simulation shown in figure 3.7 and are shown in table 3.2; this table shows that a queue length of zero is very likely indeed. Thus, our assumption that as long as the initial system state and queue length are representative, it does not make that much difference on the simulation output, is backed up.

We then had to resolve the last problem, whether to make a single continuous run or to make an equivalent number of independent and shorter runs. From figures 3.7 and 3.8 it is clear that calculating the average data from a number of runs has the same affect as making one long run and so we decided to go with the latter. The next issue was to decide for what length the simulations should be run.

3.8 Simulation duration

We needed to decide what constituted a satisfactory simulation run length (in blocks). We made some pilot runs with periodic interval reporting, which amounted to building a probability distribution of the queue length at each point, one of the runs is shown in figure 3.8. Our criteria for judging the necessary length was the distribution of the tail probabilities.



Plot	l	ρ	s	Blocks
A	50	0.85	18	10^5
B	50	0.85	18	4×10^5
C	50	0.85	18	7×10^5
D	50	0.85	18	10^6

Figure 3.8: Simulation run with periodic output

Plots A (1 million blocks), B (4 million blocks) and C (7 million blocks) have the same maximum buffer length, the tail distributions are noisy due to correspondingly low probabilities (see sections 1.5 and 4.3) and decrease slightly in turn. By the time the simulation length is 10 million blocks (plot D) the resulting probability distribution is a little less noisy at the end. A maximum simulation length had to be decided on for practical reasons (time constraint on the running time, see section 3.4.3) and we decided on a length of 10 million blocks. The simulations outlined in figures 3.7 and 3.8 were run using a fixed value for α and $\phi = l(1 - \beta)$ as described in section 4.2.

Chapter 4

Simulation performance: results and evaluation

4.1 Introduction

In this chapter we present and analyze the results of our simulations. Initially, we explain how the characteristics of the input traffic to the multiplexer are kept constant and how the parameters used in the simulations were chosen; these parameters are then given in full. We ran a batch of five simulations, for $l = 50$, 100, 200, 300 and 400, for a number of different loads and we present and comment on the main simulation output, which is an empirical probability distribution of the buffer queue resulting from the chosen simulation parameters. A phenomenological model of the probability distribution, based on the shape of the simulation graph, is defined and we demonstrate how the model parameters are fitted using regression analysis. The model parameters are plotted against the number of inputs, l, and the fitted model is compared with the simulation graph for a selection of the chosen simulation parameters. Finally we present other results obtained from the simulations, detailing the proportion of time spent in the burst level traffic regime.

4.2 Explanation of simulation parameters

In section 1.4 we described our objective as being to make predictions for large systems, where the number of input lines, l, is scaled, also, $\sigma = s/l$ (see equation 2.34), and hence the load, ρ, is kept constant. Thus, we will keep the traffic characteristics of the activity and burst length constant. α (see equation 2.1) characterizes the mean burst length, which we will fix, so we need to decide what is a valid choice for α.

We determine the choice of the individual line characteristics by reference to the block model described in section 2.4. Since we are interested in large systems, we choose $l = 1000$ for the purpose of fixing α. The stability condition for the block level model, given by equation 2.47, states that σ must be greater than the activity (see equation 2.13), ie. the percentage of time, in the long run, that a caller is speaking. We assume an activity of 35% (a commonly accepted value, see [9]) so $\sigma > 0.35$ for stability. The load is equal to the activity divided by σ as given by equation 2.46.

We wish to have a high load (close to 1, see equation 2.32) so as to get burst level congestion in the simulations. Otherwise, the probability of burst level congestion is too small, meaning that it occurs to rarely to be manifested in our simulations. Consequently, we choose $\sigma = 0.4$ and since $\sigma = s/l$ this gives us $s = 400$. In translating from the block level model to the cell level model, the interpretation of s and l is the same. We assume a mean burst length, $\overline{B}$ (see equation 2.2.1), of 352 ms and a multiplexer capacity of 106 Mbs [37]. The corresponding multiplexer service time or tick width is 3.989 μs. Since $s = 400$, if we fill these values into equation 2.12, we get $\alpha = 0.995466$.

For the cell level model we have, from equation 2.33, a multiplexer load represented by

$$\rho = \frac{1}{\sigma + \frac{(1-\alpha)}{l(1-\beta)}}$$

The traffic is characterized by α, which we have fixed, and β. Now, if we fix $\phi = l(1-\beta)$, then a given value for the load will determine the multiplexer sampling period, s. Note, that fixing ϕ amounts to fixing the mean silent period (see equation 2.11), measured in blocks of s ticks. Thus, we keep the traffic characteristics constant at the block time scale even though we vary s and l. To fix ϕ we choose $\rho = 0.82$, which gives us $\phi = 0.00553175$. Consequently, β is a function of l and the its values are shown in table 4.1. Also, the mean silence length for each choice of ρ is approximately constant as shown in table 2.11. These values for ϕ and α are then used for all the simulations conducted, with the desired load being obtained by varying σ, see table 4.3.

To decide on what other load values to simulate for, we turned to the upper bound of the block model, as described by equation 2.48. From preliminary simulations we found that noise due to the sampling of rare events set in at an empirical probability of 10^{-4}. We found that for values of $l = 200$ or less the lowest load suitable for simulating was $\rho = 0.7$, for larger l and smaller ρ, noise makes simulations impractical due to time constraints. Our aim was to simulate for l up to 400 and we found, using the upper bound, that the lowest acceptable load for this l was $\rho = 0.78$. As ρ approaches 1 the queue size in the buffer grows increasingly large (and correspondingly the percentage of time in burst level congestion), especially if the output rate of the multiplexer is "slow", ie. for small l (σ constant). Using the upper bound we found that the highest value of the load worth simulating was 0.9. We also choose to simulate with a load of 0.85, as the difference between 0.82 and 0.9 was quite large. Table 4.3 shows the values of σ (constant for all l) for the chosen loads, these then determine s.

At this stage it is perhaps worth relating the chosen parameters to reality. β is very close to 1, which means that the probability of an inactive line starting to fire within the next s ticks, γ (see equation 3.2), is of the order 2×10^{-3}. Since α is also close to 1, we can conclude that if a line is active or inactive, it is very likely that it will stay that way. Consequently the mean burst and mean silence lengths will be large, relative to the tick size. From equation 2.12 we have the tick width, $T_W = (\overline{B}(1-\alpha))/s$, which decreases as we scale l and keep σ constant, consequently, the multiplexer capacity, c, increases. By increasing ρ we are decreasing

l	β
50	0.999889
100	0.999945
200	0.999972
300	0.999982
400	0.999986

Table 4.1: β depending on l

ρ	s	C (Mbs)	$\overline{I}$ (ms)
0.82	20	5.3147	719
	40	10.6283	725
	80	21.2567	712
	120	31.8850	739
	160	42.5134	712
0.78	23	6.1113	625
	46	12.2226	631
	92	24.4452	619
	138	36.6668	642
	185	49.1561	616
0.85	18	4.7828	799
	35	9.2998	829
	71	18.8653	803
	106	28.1651	836
	142	37.7306	803
0.9	14	4.7828	1027
	29	7.7056	1000
	58	15.4111	983
	87	23.1166	1019
	116	30.8222	983

Table 4.2: s, C and $\overline{I}$ for the chosen ρ

ρ	σ
0.78	0.46254
0.82	0.4
0.85	0.35696
0.9	0.2916

Table 4.3: σ required for the chosen ρ

s, the sampling period of the multiplexer, for each value of l, this means that the multiplexer is working slower (lower capacity) and block level congestion is much more likely. Equation 2.12 also gives us the mean silence, $\overline{I} = T_W/(1-\beta)$, we list the values for $\overline{I}$ and C, for each value of ρ, in table 4.2. Similar values for the mean silence length have been used in other work [37]. We also note that by by fixing α and $\overline{B}$ we are keeping the actual length of a block constant at 1.598 ms; this means that the simulated time is determined by the simulation length and not by the number of inputs, see table 3.1.

4.3 Simulation graphs

As explained, we fix the traffic, which is characterized by α and β, by setting $\alpha = 0.995466$ and $l(1-\beta) = 0.00553175$; the corresponding values of β, for each l, are given in table 4.1. The desired loads, 0.78, 0.82, 0.85 and 0.9, are obtained by varying σ, as listed in table 4.3. For each load we run simulations for $l = 50$, 100, 200, 300 and 400. The main simulation output was described in section 3.4.1 and consists of a probability distribution of the buffer queue, $\tilde{M}(x) = \log_{10}[\Pr\{\text{queue length} \geq x\}]$. Figures 4.1 and 4.2 show $\tilde{M}(x)$ for a load of 0.82.

Examining figure 4.1, we can see that $\tilde{M}(x)$ decreases as l increases. We notice, as mentioned in section 4.2, that all the graphs become very noisy at a threshold probability of 10^{-4}. This is because the calculated probabilities beyond this threshold are based on events too rare to be sampled reliably during the length of our simulations. The part of the graphs visible in figure 4.1 represent burst level congestion (see section 2.3.4), which occurs when there are long periods when the queue is not empty. In order to see adequately the part of the graphs relating to cell level congestion, the left most part of the graphs are blown up and shown in figure 4.2, where the initial steep decay represents cell level congestion. This figure highlights the area of the graph which we term the **knee**, this is the region of the graph which is the boundary between cell level congestion and burst level congestion. It is clear from figure 4.2 that it is extremely hard to define exactly where the knee begins and where it ends (see section 4.6.3).

We include the graphs of $\tilde{M}(x)$ against the buffer size, x, for the other loads, the full versions are displayed in figures 4.3, 4.5 and 4.7 for the loads 0.78, 0.85, and 0.9 respectively. The enlarged knee sections are shown in figures 4.4, 4.6 and 4.8. The graphs are linear locally right of the knee and it is also apparent that the log probability at which the knee occurs, varies roughly linearly with l, and that the graphs are parallel, at least locally, to the right of the knee.

It is evident from the observation of all the graphs, that $\tilde{M}(x)$ is increasing in ρ. This is unsurprising since as the load increases we expect congestion to be more likely. Concerning the variation of $\tilde{M}(x)$ with l, we can distinguish between the region of the graph to the left of the knee, corresponding to cell level congestion, and the region to the right of the knee, corresponding to burst level congestion. To the left of the knee $\tilde{M}(x)$ appears independent of l, agreeing with the heuristic that the cell level queue corresponds to that of independent arrivals having activity determined by the load, independent of l. To the right of the knee $\tilde{M}(x)$ decreases with l. Recall

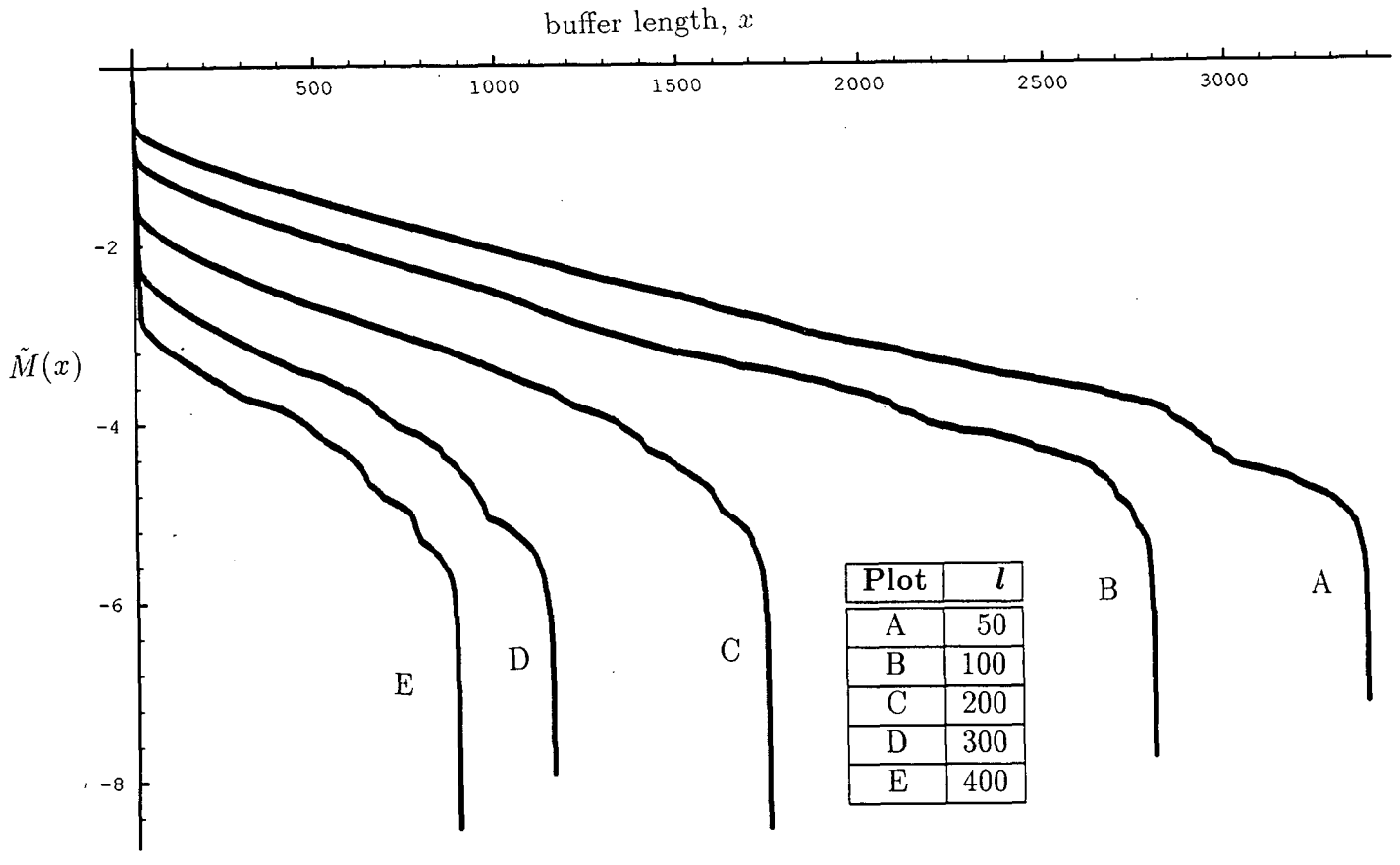


Figure 4.1: $\tilde{M}(x)$ for various l with $\rho = 0.82$

our heuristic that burst level congestion requires the BBP to extend a minimum of s prior to a block of s ticks. This requires that at least the fraction $\sigma = s/l$ of the lines be active in the block (otherwise the multiplexer can process all cell arrivals in the block), an event which will become less likely as l increases.

The effect of noise is apparent on all the full graphs (figures 4.1, 4.3, 4.5 and 4.7), this comes about in the probability range of 10^{-3} to 10^{-4}. This was most apparent in the combinations of low load $\rho = 0.78$ and a large number of input lines $l = 400$. In this case most of the graph fell in the noise region - see figure 4.3. This run was at what we regarded to be the limit of convenience for simulation, with a running time of over 24 hours (see section 4.2).

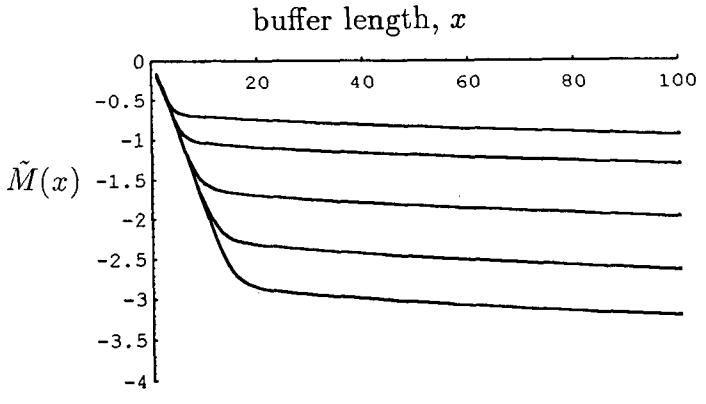
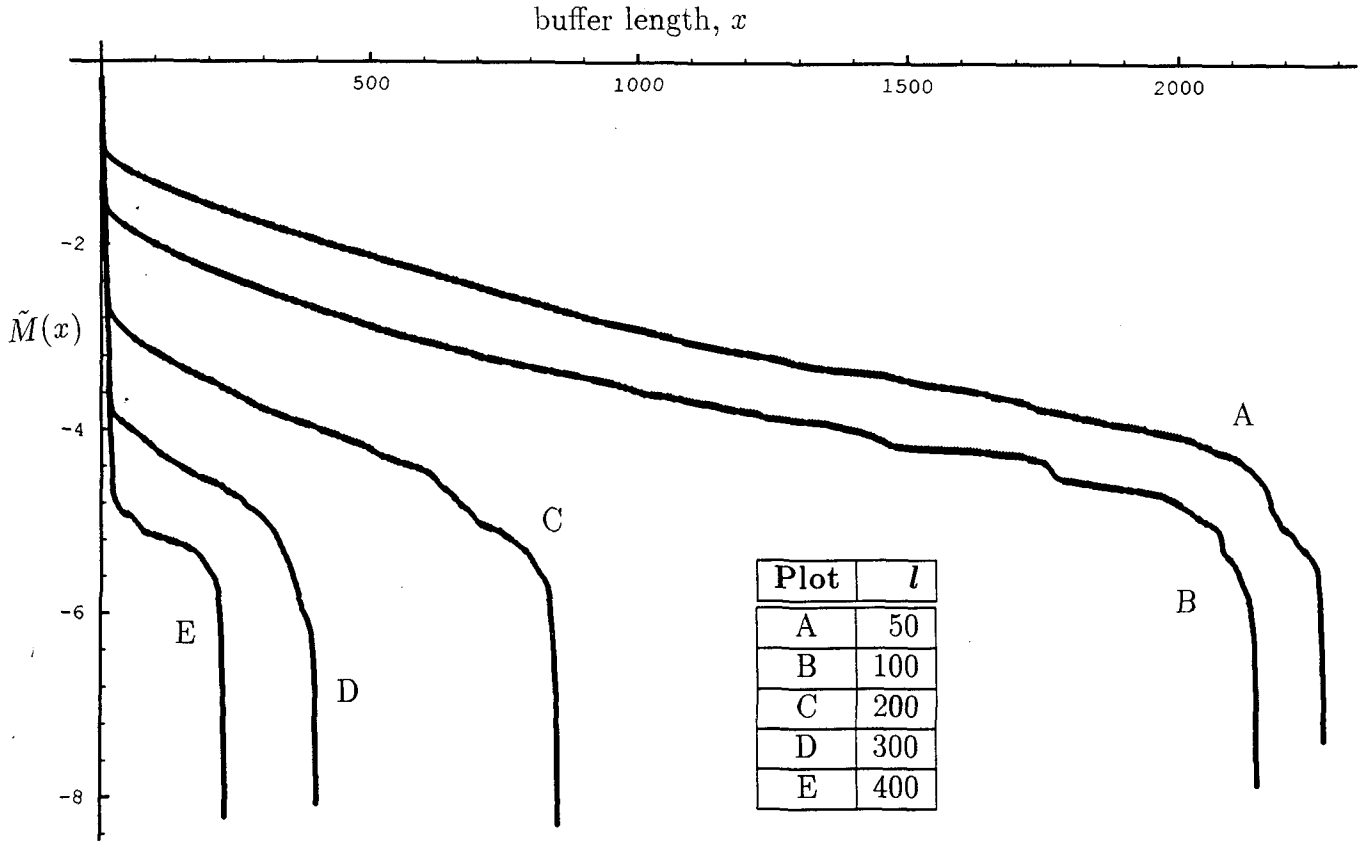


Figure 4.2: Enlargement of knee area for $\rho = 0.82$



Plot	l
A	50
B	100
C	200
D	300
E	400

Figure 4.3: $\tilde{M}(x)$ for various l with $\rho = 0.78$

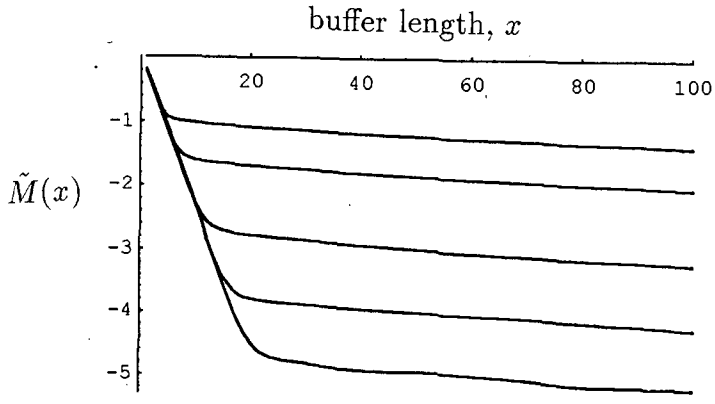
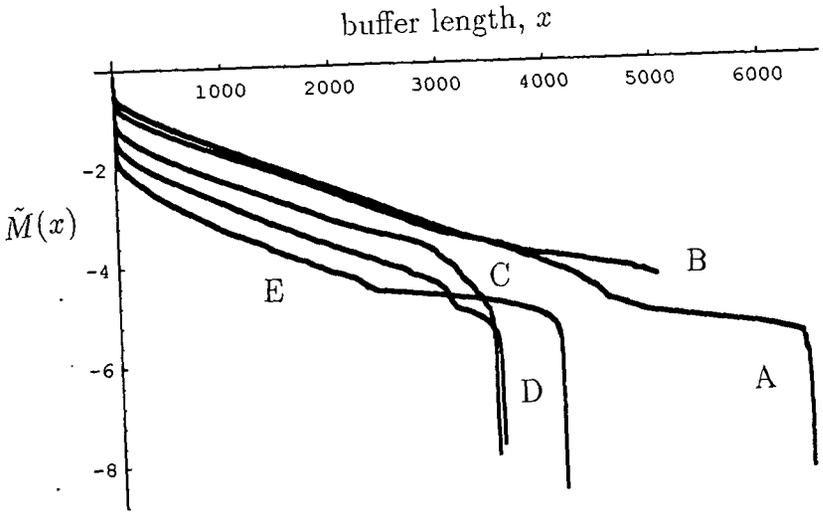


Figure 4.4: Enlargement of knee area for $\rho = 0.78$



Plot	l
A	50
B	100
C	200
D	300
E	400

Figure 4.5: $\tilde{M}(x)$ for various l with $\rho = 0.85$

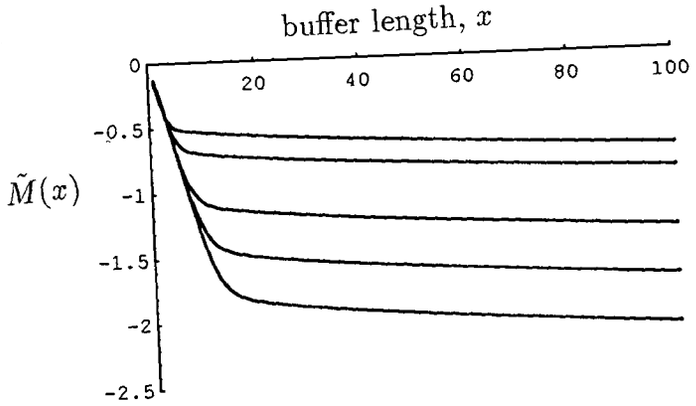
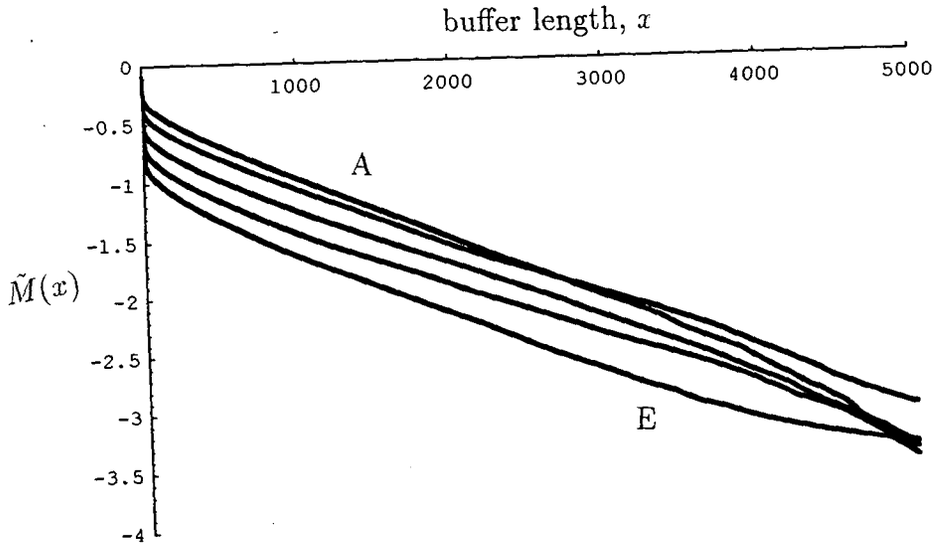


Figure 4.6: Enlargement of knee area for $\rho = 0.85$



Plot	l
A	50
B	100
C	200
D	300
E	400

Figure 4.7: $\tilde{M}(x)$ for various l with $\rho = 0.9$

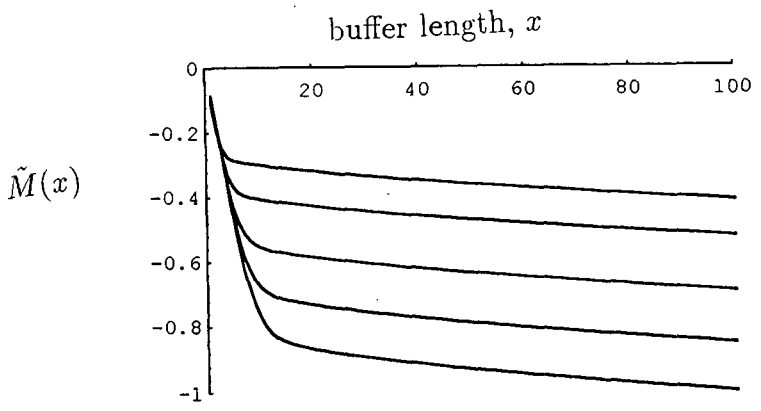


Figure 4.8: Enlargement of knee area for $\rho = 0.9$

4.4 A model for the buffer queue distribution

We introduced a phenomenological model based on the shape of the simulation graph in section 1.5. Examining the shape of the simulation graphs of the probability distribution of the buffer's queue length, allows us to break the graphs into 3 parts. The first is characterized by a sharp decay, before moving into the second part, which we termed the knee of the graph in section 4.3. The third part, to the right of the knee, is also characterized by a decay constant, although less sharp than the first.

We define λ to be a weighting constant and $u_1 \gg u_2$, where u_1 and u_2 are the decay constants for the parts of the graph to the left, and asymptotically to the right of the knee, respectively. Now, we can write the following phenomenological model for the probability distribution of the buffer queue length, q.

$$\Pr\{q = x\} = \lambda u_1 e^{-u_1 x} + (1 - \lambda) u_2 e^{-u_2 x} \tag{4.1}$$

where $\lambda \in [0, 1]$. The model described by equation 4.1 is defined for all $x > 0$, therefore we can write $M(x) = \Pr\{q \geq x\}$, where

$$\begin{aligned} M(x) &= \int_x^\infty \Pr\{q = x\}\, dx \\ &= \lambda e^{-u_1 x} + (1 - \lambda) e^{-u_2 x} \end{aligned} \tag{4.2}$$

The origin of this model is explained by looking at the eigenfunction expansion of the buffer queue probability distribution, $F(x) = \Pr\{q \geq x\}$, ie.

$$F(x) = \sum_{j=0}^{k} c_j e^{-y_j x} \tag{4.3}$$

where $\sum_{j=0}^{k} c_j = 1$. This expansion can in principle be explained using the matrix geometric solution method [36], although the calculations of c_j and y_j are intractable in general and impracticably lengthy even for particular choices of model parameters. The eigenvalues in this expansion, y_j, are labelled such that $0 < y_0 < y_1 < \ldots < y_k$, where $k = \#(y_j)$. For a large buffer size, x, $F(x) \approx c_0 e^{-y_0 x}$, and for a small buffer size,

x, $F(x) \approx c_k e^{-y_k x} + \sum_{j=0}^{k-1} c_j$. Comparing this with the phenomenological model, given by equation 4.2, we have $c_o = 1-\lambda$, $y_0 = u_2$, $c_k = \lambda$, $y_k = u_1$ and $\sum_{j=0}^{k-1} c_j = 1-\lambda$. This means that $c_2 = c_3 = \ldots = c_{k-1} = 0$. Therefore, we can regard the phenomenological model as using the lowest and highest eigenvalues in expansion 4.3, consequently the model is only good at the extremes, $x \to 0$ and $x \to \infty$ (see section 4.6.3).

Returning to model 4.2, for small buffer sizes, $M(x) \approx \lambda e^{-u_1 x} + (1 - \lambda)$, since $u_1 \gg u_2$. We define this approximation to be $M_1(x)$ and it corresponds to cell level congestion. Now, defining $Q_1(x) = \log_{10}[M_1(x)]$, we can compare $Q_1(x)$ directly with a simulation graph of $\tilde{M}(x)$ (see section 4.3). To compare the slope R_1 of the part of the graph left of the knee and the decay constant u_1, we take the limit as $x \to 0$, of the derivative of $Q_1(x)$, ie.

$$R_1 = \lim_{x \to 0} Q_1'(x) = \frac{-u_1 \lambda}{\log_e 10} \tag{4.4}$$

For large buffer sizes $M(x)$ is approximated by $(1 - \lambda)e^{-u_2 x}$. We define $M_2(x) = (1 - \lambda)e^{-u_2 x}$ and $Q_2(x) = \log_{10}[M_2(x)]$. The slope R_2 to the right of the knee is calculated by taking the limit as $x \to \infty$, of the derivative of $Q_2(x)$, ie.

$$R_2 = \lim_{x \to \infty} Q_2'(x) = \frac{-u_2}{\log_e 10} \tag{4.5}$$

To work out the weighting constant λ we calculate the intercept for the right part of the graph, I_0, where $I_0 = Q_2(0)$, since at the intercept the buffer length is zero. This gives us the following expression for λ:

$$\lambda = 1 - e^{I_0 \log_e 10} \tag{4.6}$$

To fit the model we need to estimate the slopes R_1 and R_2 and obtain the intercept for the right part of the graph, this is described in the next section.

4.5 Regression analysis

The model described by equation 4.2 is intrinsically non linear and therefore it can not be transformed into a linear model by regular techniques such as a log transformation. Non linear estimation was attempted by approximating a linear expansion of the model using Taylor's series. To estimate the parameters we needed to obtain a solution for the normal equations, however this is difficult to obtain analytically and iterative methods must be employed [13]. Instead a quicker, simpler method was used which we now describe.

We obtain estimates of the parameters by using linear regression to fit a least squares line for the parts of the graph to the left and right of the knee. u_1 and u_2 are multiplicative constants times R_1 and R_2 respectively, and are calculated using equations 4.4 and 4.5.

We return to the figure 4.1, which plots $\tilde{M}(x)$ for $\rho = 0.82$. At some point between a probability loss of 10^{-3} and 10^{-4}, the graphs become visibly noisy, this is consistent with the graphs for the other load values shown in figures 4.3, 4.5 and 4.7.

ρ	l	u_2 left	u_2 right	u_1 cutoff
0.82	50	2284	2800	4
	100	1385	2100	4
	200	292	1300	8
	300	185	800	12
	400	60	700	15
0.78	50	1040	2000	4
	100	992	1500	6
	200	285	600	10
	300	107	300	12
	400	79	180	16
0.85	50	2835	4200	2
	100	2105	3000	3
	200	1446	2200	4
	300	1747	2800	7
	400	1422	2100	10
0.9	50	463	4450	2
	100	2069	4000	3
	200	885	4000	5
	300	2871	4000	6
	400	3558	4000	7

Table 4.4: Cutoff points used in calculation of u_1 and u_2

We have to decide what points should be included in obtaining a least squares fit. For the right hand part of the graph, we decided to fix a right cut off point and obtain a least squares fit as we varied the left cutoff point. This right cutoff point is picked to be the largest value of the buffer before the graph is deemed to have become noisy. A list of the right hand cutoffs used is given in table 4.4. We now have to decide what left cutoff to choose.

We obtain a least squares fit with the right cutoff fixed and the left cutoff varying. For each least squares fit the value of u_2 is calculated using equation 4.5, we then plot u_2 against the left cutoff point. To choose u_2 we examine this graph and use the heuristic of choosing the largest (least negative) u_2, before the estimate becomes noisy due to the decreasing number of points used in the fit, as we move the left cutoff rightwards. Our reasoning uses the hypothesis that $\tilde{M}(x)$ should be asymptotically linear for large buffer sizes. Since $\tilde{M}(x)$ is convex, the above heuristic should provide the best estimate of the limiting slope from our data; u_2 is just a multiplicative constant times this slope. The chosen left cutoffs resulting from this heuristic are listed in table 4.4.

ρ	l	$u_2 \pm 95\%$CI	Intercept$\pm 95\%$CI	λ
0.82	50	$0.00192193 \pm 1.3434 \times 10^{-5}$	$-1.47452 \pm 1.485 \times 10^{-2}$	0.966536
	100	$0.00221279 \pm 1.5647 \times 10^{-5}$	$-1.76019 \pm 1.193 \times 10^{-2}$	0.982630
	200	$0.00351096 \pm 1.1489 \times 10^{-5}$	$-1.90886 \pm 4.220 \times 10^{-3}$	0.987665
	300	$0.00474721 \pm 3.8460 \times 10^{-5}$	$-2.46860 \pm 8.740 \times 10^{-3}$	0.996601
	400	$0.00580844 \pm 7.0704 \times 10^{-5}$	$-2.91241 \pm 1.298 \times 10^{-2}$	0.998777
0.78	50	$0.00246813 \pm 1.2422 \times 10^{-5}$	$-1.84515 \pm 8.340 \times 10^{-3}$	0.985716
	100	$0.00252016 \pm 3.0897 \times 10^{-5}$	$-2.45699 \pm 1.683 \times 10^{-2}$	0.996509
	200	$0.00513300 \pm 2.7047 \times 10^{-5}$	$-3.09473 \pm 5.310 \times 10^{-3}$	0.999196
	300	$0.00773395 \pm 1.1273 \times 10^{-4}$	$-3.89744 \pm 1.033 \times 10^{-2}$	0.999873
	400	$0.00499003 \pm 1.2148 \times 10^{-4}$	$-4.93494 \pm 7.020 \times 10^{-3}$	0.999988
0.85	50	$0.00174327 \pm 4.795 \times 10^{-6}$	$-1.04643 \pm 7.370 \times 10^{-3}$	0.910139
	100	$0.00190143 \pm 2.138 \times 10^{-6}$	$-0.94687 \pm 2.386 \times 10^{-3}$	0.886987
	200	$0.00183684 \pm 5.514 \times 10^{-6}$	$-1.53338 \pm 4.390 \times 10^{-3}$	0.970717
	300	$0.00183281 \pm 4.675 \times 10^{-6}$	$-2.04305 \pm 4.660 \times 10^{-3}$	0.990944
	400	$0.00193700 \pm 7.832 \times 10^{-6}$	$-2.48260 \pm 6.030 \times 10^{-3}$	0.996708
0.9	50	$0.00125641 \pm 1.326 \times 10^{-6}$	$-0.419442 \pm 1.565 \times 10^{-3}$	0.619322
	100	$0.00100742 \pm 2.029 \times 10^{-6}$	$-0.696548 \pm 2.711 \times 10^{-3}$	0.798881
	200	$0.00110693 \pm 7.540 \times 10^{-7}$	$-0.769986 \pm 8.500 \times 10^{-4}$	0.830170
	300	$0.00096406 \pm 2.630 \times 10^{-6}$	$-1.094710 \pm 3.940 \times 10^{-3}$	0.919593
	400	$0.00088971 \pm 4.592 \times 10^{-6}$	$-1.563260 \pm 7.540 \times 10^{-3}$	0.972664

Table 4.5: u_2, intercept and λ with 95 % confidence interval

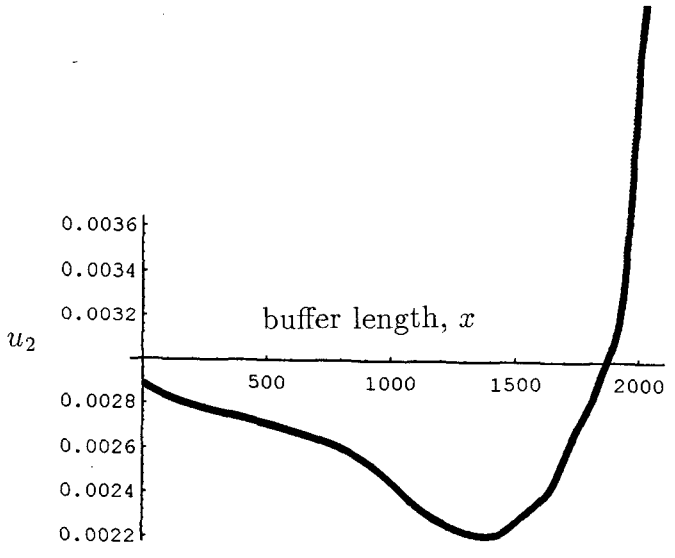


Figure 4.9: u_2 versus left cutoff point for $\rho = 0.82$ and $l = 100$

As an example we will look at plot B ($l = 100$) in figure 4.1, the right hand cutoff was set at 2100. Figure 4.9 plots the value of u_2, obtained from the least squares fit, against the left cutoff point. The lowest point occurs for a buffer size of 1385, thus we pick the value of u_2 which corresponds to this point. Once we have chosen u_2 a confidence interval can be obtained and using the regression equation for u_2, which gives the intercept, λ can be obtained from equation 4.6.

u_1 is worked out by first picking a point left of where the knee stops. Linear regression is then used to estimate the slope R_1 and consequently u_1. In our example ($\rho = 0.82$, $l = 100$) we use a right cutoff point of 4, linear regression is then carried out to fit the four points. Because of the small number of points used to fit a line the error will be relatively high, for example, if we examine the enlarged knee section for a load of 0.82 (figure 4.2) it is evident that the error in the calculation of R_1 will be high for $l = 50$. The same can be said for the other loads (figures 4.4, 4.6 and 4.8, in that the error will be high for low values of l and high values of ρ. This is reflected in section 4.6.3, when the fitted model, $M(x)$, is compared with the simulation model, $\tilde{M}(x)$.

ρ	l	$u_1 \pm 95\%$CI
0.82	50	$0.346334 \pm 1.53277 \times 10^{-1}$
	100	$0.424258 \pm 3.57201 \times 10^{-2}$
	200	$0.414494 \pm 2.04946 \times 10^{-2}$
	300	$0.413946 \pm 1.62087 \times 10^{-2}$
	400	$0.423931 \pm 1.25103 \times 10^{-2}$
0.78	50	$0.488622 \pm 1.24585 \times 10^{-1}$
	100	$0.529873 \pm 3.33545 \times 10^{-2}$
	200	$0.546772 \pm 9.75732 \times 10^{-3}$
	300	$0.550050 \pm 9.49095 \times 10^{-3}$
	400	$0.555527 \pm 7.96766 \times 10^{-3}$
0.85	50	$0.387249 \pm$ indeterminate
	100	$0.370817 \pm 1.06872 \times 10^{-1}$
	200	$0.357120 \pm 6.22726 \times 10^{-3}$
	300	$0.344720 \pm 6.79923 \times 10^{-3}$
	400	$0.343001 \pm 7.24042 \times 10^{-3}$
0.9	50	$0.341795 \pm$ indeterminate
	100	$0.260539 \pm 1.86491 \times 10^{-1}$
	200	$0.237418 \pm 3.41750 \times 10^{-2}$
	300	$0.221699 \pm 1.84414 \times 10^{-2}$
	400	$0.215059 \pm 1.23758 \times 10^{-2}$

Table 4.6: u_1 with 95 % confidence interval

The values of u_2, the intercept, λ and the corresponding confidence intervals, for all the load values, are listed in table 4.5. The confidence interval for λ is omitted. The values of u_1 and there respective confidence intervals are listed in table 4.6. It

is noticed that there are two instances of indeterminate confidence intervals: in these cases there were only two points being fitted (see table 4.4).

4.6 Presentation of results

4.6.1 Introduction

In this section we present the main results of this thesis: the intercept of the limiting slope, u_1 and u_2 plotted as we scale the number of inputs l. We make some brief comments about these graphs and we draw our conclusions in section 5.1. We compare the fitted model $\tilde{M}(x)$ with the simulated model $M(x)$ for a choice of loads and inputs. These choice are made from the analysis of the the plot of u_2 against l. Finally we present simulation statistics detailing the percentage of time spent in burst level congestion. We expect that this proportion should be approximately equal to the probability at which the knee occurs.

4.6.2 Analysis of model parameters

Figure 4.10 shows the intercept, I_0, plotted against the number of inputs, l, for the different loads simulated. As discussed in section 4.3, the intercept is increasing in ρ. From figure 4.10 we can see that the intercept scales approximately linearly with l, this is analyzed in more detail in section 5.1. The intercept gives us the approximate threshold probability, T_P, beyond which the effect of the correlations in the arrivals can no longer be neglected, this is because the intercept roughly marks the left boundary of the knee, the beginning of the region of transition between cell and burst level congestion (the precise boundaries of the knee are indeterminable, see section 4.6.3).

Figure 4.11 shows the value of u_2 plotted against l for the different load values. The anomaly in the 0.78 graph for $l = 400$ can be attributed to noise, as was explained in section 4.3. We recall the simulation graphs of $\tilde{M}(x)$ for loads of 0.82 and 0.78, figures 4.1 and 4.3 respectively. We notice that as l increases the number of points in the tail to the right of the knee, prior to the onset of sampling noise, decreases. The right non-noisy boundary of the tails is roughly at a probability of 10^{-4}. Observing the graphs for loads of 0.85 and 0.9, figures 4.5 and 4.7 respectively, we can see that the buffer lengths are substantially larger before the noisy region is entered (although for a load of 0.9, see figure 4.7, the noisy region boundary as such has shifted to a higher probability of 10^{-3}).

The variation in the tail lengths for loads of 0.78 and 0.82 is reflected in figure 4.11. Because of the decrease in tail length the least squares fit (see section 4.5) of our data provides us with a less accurate estimation of the limiting slope in these cases. For this reason in section 5.1, where we draw our conclusions, we will confine ourselves to the higher loads of 0.85 and 0.9.

Figure 4.12 shows u_1 plotted against the number of inputs, l, for our choice of load values. If we examine table 4.4, which gives the cutoffs used in obtaining the slope R_1 (from the least squares fit), of which u_1 is a multiplicative constant, we can

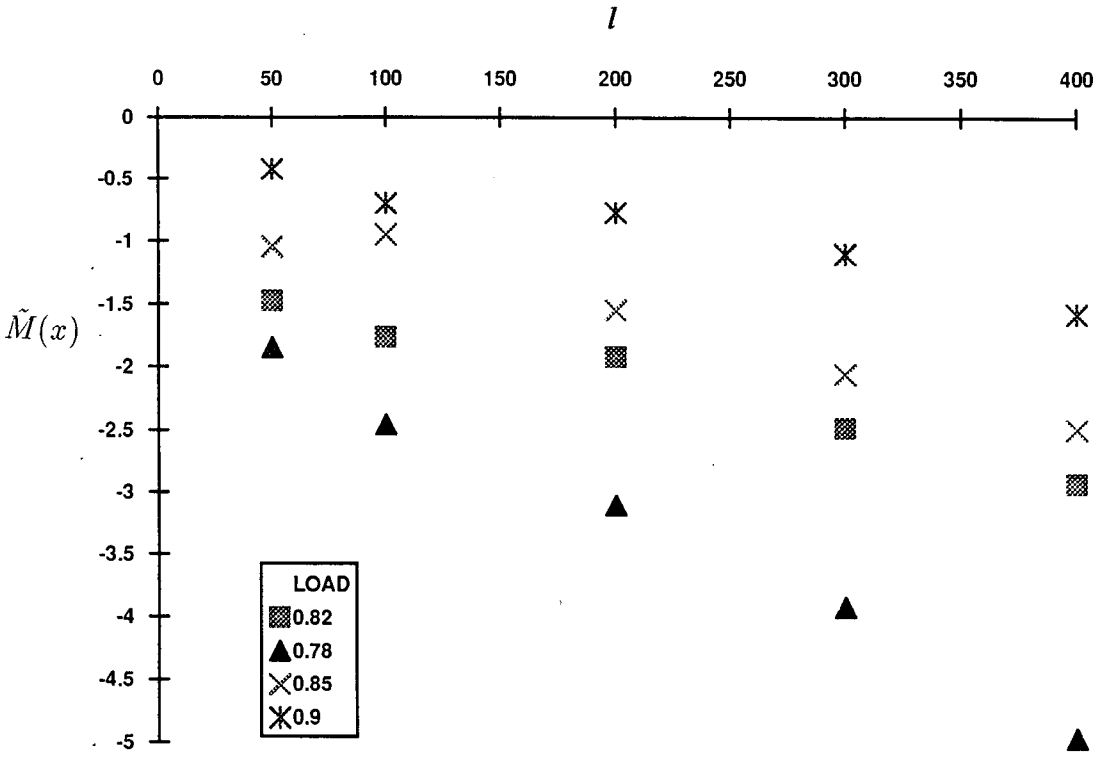


Figure 4.10: Intercept plotted against l for chosen ρ

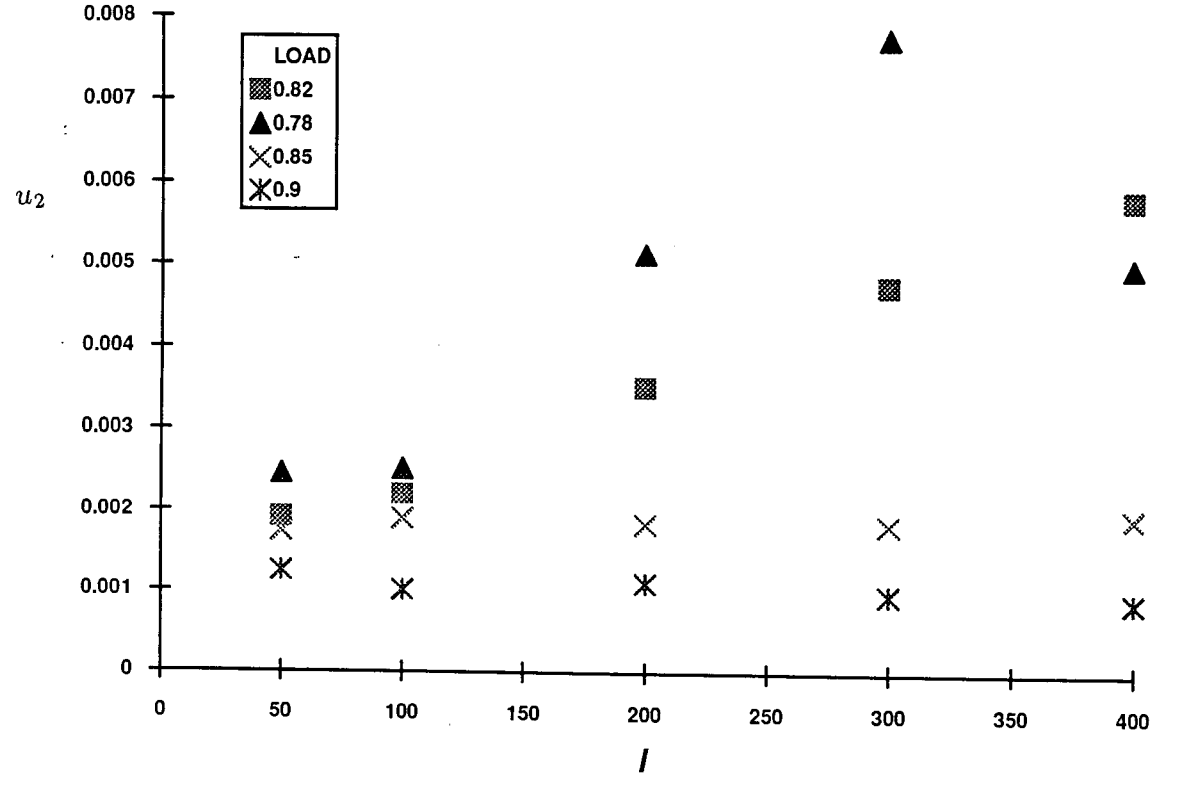


Figure 4.11: u_2 plotted against l for chosen ρ

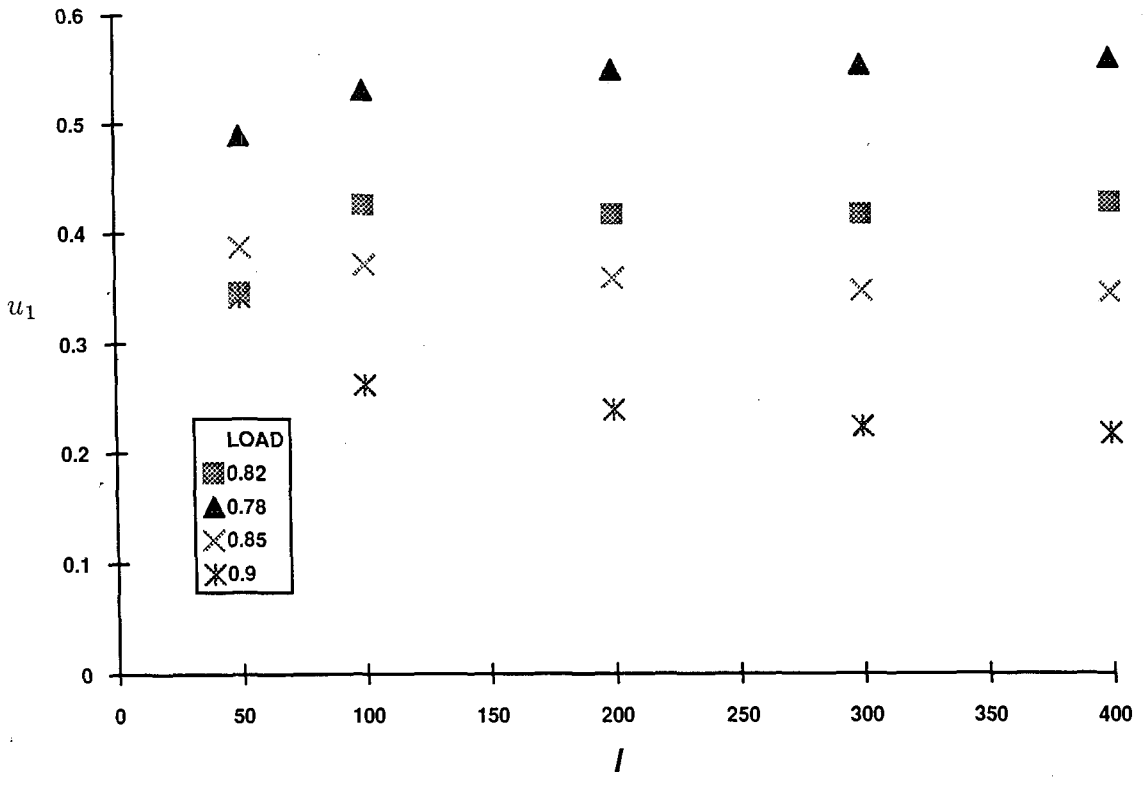


Figure 4.12: u_1 plotted against l for chosen ρ

see that the higher the load, ρ, and the lower the number of inputs, l, the smaller the cutoff point and consequently the wider the confidence intervals. The estimates get more accurate with larger l, as a result we discount the values for u_1 up to and including $l = 100$. We will discuss the implications of this graph in section 5.1.

4.6.3 $\tilde{M}(x)$ versus $M(x)$

In section 4.6.2 we concluded that the fitted values of u_2 for loads of 0.85 and 0.9 were more reliable estimates of the limiting slope than values obtained from loads of 0.78 and 0.82. We now plot a fitted model $M(x)$, for each of these loads, superimposed with the appropriate simulation model $\tilde{M}(x)$. Figure 4.13 shows the two models for a load of 0.85 and $l = 100$, with the knee region enlarged in figure 4.14. It is evident from these two plots that the model is accurate for the cell level congestion and burst level congestion. This is backed up from the evidence of figure 4.15 which shows the two models for a load of 0.9 and $l = 200$, with the knee region enlarged in figure 4.16.

There is a certain discrepancy about the knee region in evidence from the enlargements (see figures 4.14 and 4.16), but this is to be expected, as a model with only two decay constants can not be expected to accurately model the unwieldy knee. Recall equation 4.3 in section 4.4 which gave the eigenfunction expansion of the buffer queue length probability distribution. We discussed that the phenomenological model used the lowest and highest eigenvalues in this expansion, consequently the model is only good at the extremes and does not accurately model the knee region.

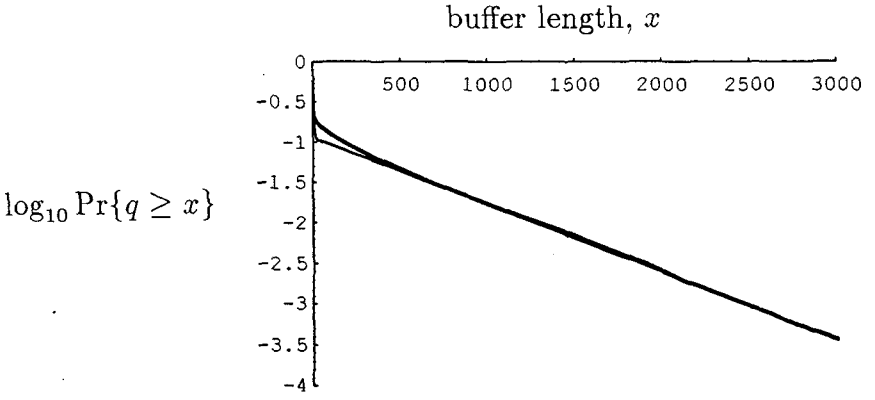


Figure 4.13: Plot of $\tilde{M}(x)$ and $M(x)$ for $\rho = 0.85$ and $l = 100$

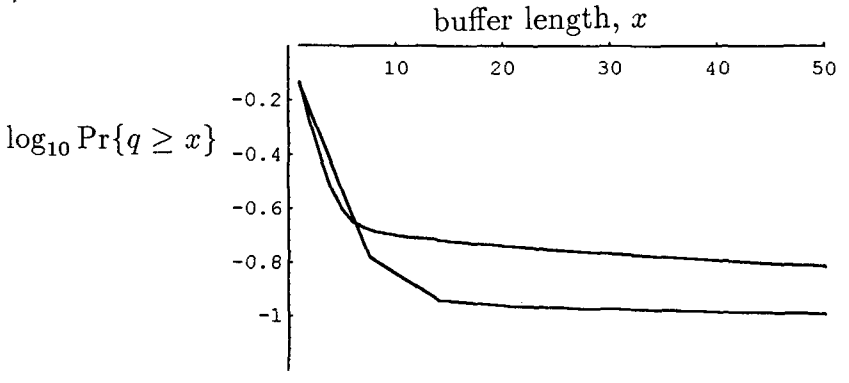


Figure 4.14: Enlargement of knee region for figure 4.13

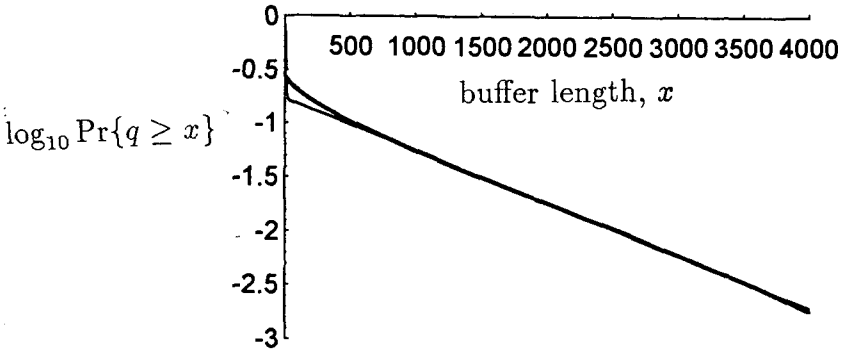


Figure 4.15: Plot of $\tilde{M}(x)$ and $M(x)$ for $\rho = 0.9$ and $l = 200$

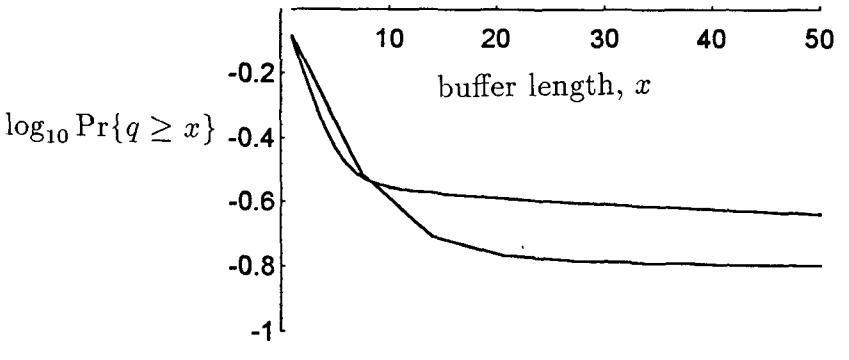


Figure 4.16: Enlargement of knee region for figure 4.15

4.6.4 Proportion of time in burst level congestion and the knee

We collected statistics during the simulation run, outlining the proportion of time spent in the burst level traffic regime. Referring to section 2.3.4, this means the proportion of blocks in which the queue was never empty. Figure 4.17 graphs the results obtained. and highlights that the time spent in burst level traffic decreases as the number of inputs, l, is scaled. Also, the higher the load, the more time that is spent in heavy traffic, relative to l.

Returning to the original simulation graphs of $\tilde{M}(x)$, figures 4.1, 4.3, 4.5 and 4.7, we know that the position of the knee drops in l. We note the apparent similarities of the height of the knee with the percentage of time the buffer is non-empty. In section 4.6.2 we discussed that the intercept gives the approximate threshold probability of the left boundary of the knee. It has been mentioned that the unwieldy nature of the knee does not lend itself to close analysis, however, it seems that the proportion of time in burst level congestion may correspond to the threshold probability of the right knee boundary, ie. the point at which burst level congestion begins. This can not be substantiated, but we note its possible usefulness as a rule of thumb.

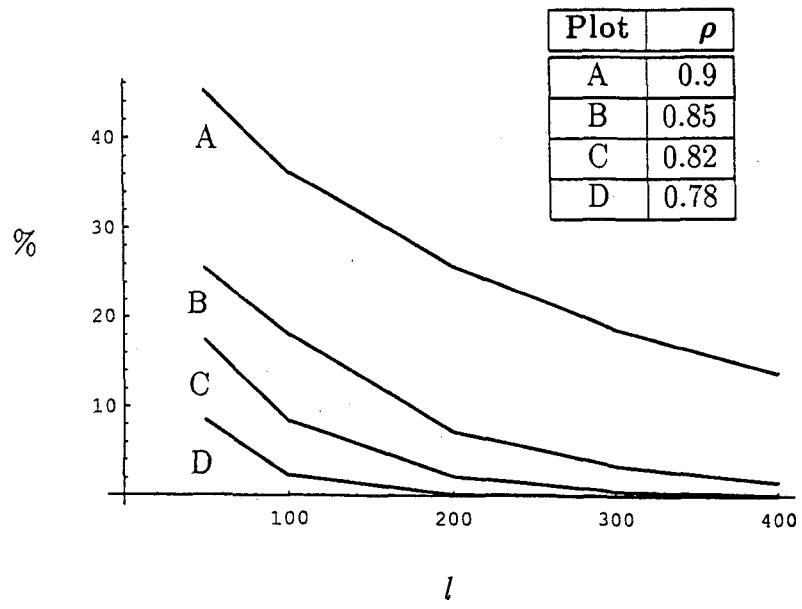


Figure 4.17: % of time in burst level congestion for the chosen ρ

Chapter 5

Conclusion

5.1 Prediction of large ATM systems

In section 4.6.2 we presented a graph of u_2 against the number of inputs, l, and the load, ρ (see figure 4.11). After analysis we decided to confine any conclusions regarding the u_2 values, to the results obtained using loads of 0.85 and 0.9, the data for $\rho = 0.78$ and 0.82 being unreliable due to simulation noise. Re-examining figure 4.11 we see that u_2 is to a good approximation independent of l. We also found that the intercept of the regression line is affine in l (see figure 4.10). A plot of u_1 against l is shown in figure 4.12. We decided, due to the small number of sample points in these cases, to discount the values of u_1 for $l = 50$ and $l = 100$. If we re-examine figure 4.12 we can conclude that u_1 is also independent of l.

The following summarizes how the parameters of the phenomenological model

$$M(x) = \lambda e^{-u_1 x} + (1-\lambda)e^{-u_2 x}$$

behave after analysis of the simulation results obtained from the graphs of $\tilde{M}(x)$. For fixed α, $\phi = l(1-\beta)$ and $\sigma = s/l$:

- u_1 is independent of l.
- u_2 is independent of l.
- The intercept $\frac{\log_{10}(1-\lambda)}{\log_e 10}$ is an affine function of l, ie.

$$\frac{\log_{10}(1-\lambda)}{\log_e 10} = A + Bl$$

for some constants A and B.

Now, we illustrate how to make predictions for large systems. Suppose we wish to analyze the queueing problems for an ATM multiplexer for given values of α, β, s and l, where the number of inputs l make simulation infeasibly lengthy. We fix α, $\phi = l(1-\beta)$ and $\sigma = s/l$ for the given values and we estimate the constants u_1 and u_2 by performing simulations for lower values of l. A minimal way of calculating A and B is to simulate for two values l_1 and l_2, where $l_1, l_2 \ll l$ and estimate a straight line on a plot of the intercepts against the chosen l. Consequently, we can fit a phenomenological model $M(x)$ for the large system.

5.2 QoS for large ATM systems

In section 5.1 we discussed the prediction of large systems by simulating smaller, easier to simulate ones, now we will apply what we have learned to the issue of QoS, which is an important issue in the design of large multiplexed systems.

We mentioned in section 1.4 that QoS is described by queueing delay and cell loss probability. Consequently, QoS is a function of the buffer size and the load, so the choice of buffer size is an important issue in guaranteeing a recommended QoS. Therefore, it is desirable to have a buffer size large enough to have the required cell loss probability and to have a low probability of burst level congestion, so as to avoid queueing delays.

Recall the projected phenomenological model, $M(x)$, of the hypothetical large system discussed in section 5.1. If the plot of $M(x)$ is examined we can work out the approximate threshold probability of burst level congestion and the probability of the buffer overflowing for a given buffer size can be approximated. We can also measure approximately, the buffer length corresponding to the knee, b_k, by getting the intercept of the two regression lines obtained in calculating u_1 and u_2. For large systems it is clear that the probability of the queue exceeding b_k is low (how low depends on the load), consequently the probability of burst level congestion is low. These are all issues affecting QoS decisions.

Since $u_1 \gg u_2$, the gain in loss probability obtained by increasing the buffer size, x, up to b_k is far greater than the probability gained by increasing the buffer size by the same amount beyond b_k. Thus, on economic grounds one could suggest the rule that the buffer size should be chosen to be b_k, since the return in terms of the loss probability per unit buffer size is greatest for this choice.

5.3 Further Study

Ongoing work by another member of the telecommunications project, involves the comparison of the limiting slope of $\tilde{M}(x)$ obtained for the burst level traffic regime (see section 4.6.2), with the theoretical slope obtained using martingale techniques [14]. This work also includes comparing the $\log_{10}[\Pr\{\text{queue length} = 0\}]$ obtained using large deviation methods to the intercept, I_0.

This project concentrated on keeping the traffic constant by fixing α and $l(1-\beta)$. We study relatively small systems by scaling s and keeping σ constant, our motivation being to predict large systems. However, the simulation system was developed, such that it is possible to fix any combination of the simulation parameters. Also, the system was designed with reusability, integration and further development in mind (see section A.1), and should be of use to other ATM simulation projects.

Bibliography

[1] J. Bellamy, *Digital Telephony*, J. Wiley, New York, 1991.

[2] J. T. Berry, *The Waite group's C++ programming*, H. W. Sams, Indianapolis, 1988.

[3] P. Bratley, B. L. Fox, L. Schrage, *A guide to simulation*, Springer-Verlang, New York, 1987.

[4] E. Buffet, N. G. Duffield, "Exponential upper bounds via martingales for multiplexers with markovian arrivals", *DIAS-STP-92-16.*

[5] C. H. Cheng, "Note on the effect of initial conditions on a simulation run", *Operational Research Quarterly*, Vol. 27, no. 2, ii, pp. 467-470, 1976.

[6] E. Çinlar, *Introduction to stochastic proccesses*, Prentice-Hall, Englewood cliffs, 1975.

[7] R. W. Conway, "Some tactical problems in digital simulation", *Management Science*, Vol. 10, No. 1, pp. 47-61, October 1963.

[8] B. Cox, *Object-oriented programming: an evolutionary approach*, Addison-Wesley, Reading, 1986.

[9] J. N. Daigle, J. D. Langford, "Models for analysis of packet voice communications systems", *Journal on Selected Areas in Communications*, Vol. SAC-4, No. 6, pp. 847-855, September 1986.

[10] S. C. Dewhurst, K.T. Stark, *Programming in C++*, Prentice-Hall, Englewood Cliffs, 1989.

[11] R. Drake, "Object-oriented programming in C++", *Personal Computer World*, February 1989.

[12] R. Drake, "Better C programming in C++", *Personal Computer World*, March 1989.

[13] N. R. Draper, S. Smith, *Applied regression analysis*, Wiley, New York, 1981.

[14] N. G. Duffield, "Exponential bounds for Markovian queues", *DIAS-STP-93-01.*

[15] W. Feller, *An introduction to probability theory and its applications Volume I*, Wiley, New York, 1968.

[16] T. Ferguson, "Operator overloading in C++?", *Journal of Object-Oriented Programming*, pp. 42-48, March/April 1991.

[17] G. Fishman, *Principles of discrete event simulation*, Wiley, Chichester, 1973.

[18] G. Fishman, "Achieving specific accuracy in simulation", *Communications of the ACM*, Vol. 20, no. 5, pp. 310-315, May 1977.

[19] G. Fishman, "Statistical analysis for queueing systems", *Management Science*, Vol. 20, no. 3, pp. 363-369, November 1973.

[20] A. G. Fraser, "Early experiments with asynchronous time division networks", *IEEE Network*, pp. 12-14, January 1993.

[21] G. Gordon, *System simulation*, Prentice-Hall, Englewood Cliffs, 1978.

[22] K. E. Gorlen, S. M. Orlow, P. S. Plexico *Data abstraction and object-oriented programming in C++*, Wiley, Chichester, 1990.

[23] T. L. Hansen, *The C++ answer book*, Addison-Wesley, Reading, 1990.

[24] H. Heffes, D. M. Lucantoni, "A Markov Modulated characterization of packetized voice and data traffic and related statistical multiplexer performance", *Journal on Selected Areas in Communications*, Vol. SAC-4, No. 6, pp. 856-868, September 1986.

[25] J. Y. N. Yui, *Switching and traffic theory for integrated broadband networks*, Kluwer Academic Publishers, Boston, 1990.

[26] S. Karlin, H. M. Taylor, *A first course in stochastic processes*, Academic Press, New York, 1975.

[27] B. W. Kernighan, D.M. Ritchie, "The state of C", *Byte*, pp. 205-210, August 1988.

[28] A. Koening, "Classes that keep track of themselves", *Journal of Object-Oriented Programming*, pp. 62-65, November/December 1992.

[29] A. Koening, "Understanding constructer initializers in C++", *Journal of Object-Oriented Programming*, pp. 40-46, November/December 1991.

[30] D. E. Knuth, *The art of computer programming Vol 2*, Addison-Wesley, Reading, 1973.

[31] S. B. Lippman, *C++ Primer*, Addison-Wesley, Reading, 1989.

[32] P. L'Ecuyer, "Efficient and portable combined random number generators", *Communications of the ACM*, Vol. 31, no. 6, pp. 742-749, June 1988.

[33] B. Meyer, *Object-oriented software construction*, Prentice-Hall International, London, 1988.

[34] A. Navab, E. Chiariotti, "A note on the definition of visibility for protected class members in C++?", *Journal of Object-Oriented Programming*, pp. 35, March/April 1992.

[35] J. Neter, W. Wasserman, M. H. Kutner, *Applied linear statistical models; regression, analysis of variance, and experimental designs*, R. D. Irwin, Homewood, 1974.

[36] M. F. Neuts, *Solutions in stochastic models: an algorithmic approach*, The John Hopkin University Press, Baltimore, 1981.

[37] I. Norros, J. W. Roberts, A. Simonian, J. T. Virtamo, "The superposition of variable bit rate sources in an ATM multiplexer", *IEEE Journal on Selected Areas in Communications*, Vol. 9, No. 3, pp. 378-387, April 1991.

[38] I. Pohl, *C++ for Pascal programmers*, Benjamin/Cummings Pub. Redwood City, 1989.

[39] A B. Pritsker, *Introduction to simulation and Slam II*, Wiley, New York, 1986.

[40] D. R. Reed, G. Wyant, "How safe is C++?", *Journal of Object-Oriented Programming*, pp. 69-72, May 1992.

[41] H. Schildt, *Teach yourself C++*, Osborne McGraw-Hill, Berkeley, 1992.

[42] K. Sriram, W. Whitt, "Characterizing superposition arrival processes in packet multiplexers for voice and data", *IEEE Journal on Selected Areas in Communications*, Vol. SAC-4, No. 6, pp. 833-846, September 1986.

[43] A. Stevens, *Teach yourself C++*, MIS Press, New York, 1991.

[44] B. Stroustrup, *The C++ programming lannguage*, Addison-Wesley, Reading, 1991.

[45] B. Stroustrup, "What is object-oriented programming?", *IEEE Software*, pp. 10-20, May 1988.

[46] B. Stroustrup, "A better C?", *Byte*, pp. 215-216, August 1988.

[47] H. A. Taha, *Operations research: an introduction*, Collier Macmillian. London, 1987.

[48] R. S. Wiener, L.J. Pinson, *An introduction to object-oriented programming and C++*, Addison-Wesley, Reading, 1988.

Appendix A

Object-oriented programming and C++

A.1 Introduction

This appendix is a brief overview of the object-oriented approach to software design and to the programming language C++. It is included so as to give the reader an understanding of the design philosophy behind the simulation program (see appendix B) and also as to why a programming language supporting object-oriented methods was used rather than a standard structured programming language. All references to examples in this appendix refer to the source code in appendix C.

Initially in designing the simulation program normal top-down design was used, ie. the repetitive process of decomposing problem into several sub-problems. As the more detailed design phase was entered an object-oriented approach was incorporated into the methodology.

One of the main motivation for using object-oriented programming (OOP) was the time that would be saved as different versions of the software were developed, object-oriented software being a lot more amenable to change. Also, using inheritance (see section A.3 and A.4.2) it is possible that other researchers on the telecommunications project will be able to reuse the software developed to build their own simulation software.

A.2 Object-oriented design versus top-down design

Using top-down design a well developed system tends to be modular, in other words the system consists of a collection of modules or subprograms, ie. procedures or functions which should be separately understood. The developer examines a system and decides what operations are taking place, these operations are then modeled using procedures or functions which form the basis of the system.

What is wrong with this method one may ask? The most noteworthy problems are that the data structure aspect of the system is neglected and software reusability

is not promoted, indeed top-down design is essentially the contrary of reusability. Reusable software implies that the system is developed by combining existing components, which is essentially bottom-up design [31].

The difference between traditional design methods and the object-oriented approach is whether the system should be based on the actions or on the data structure. The key to object-oriented design is the Law of Inversion which states that if there is too much data transmission in your routines, then put your routines into your data [42]. Of course, if object-oriented techniques are incorporated into the design from the start, then this law is not needed. The basis of the approach is: do not ask what the system does but instead ask what does it do it to?

Object-oriented programming has similarities to data abstraction as well as having subtle differences. Data abstraction is programming with abstract data types or more correctly user-defined data types (UDDT). A UDDT specifically describes a data structure not by an implementation but by the list of operations that are available (to the outside world), which service the data structure; thus it can be viewed as a sort of a black box - once it has been defined it does not really interact with the rest of the program [42]. So in other words, abstraction means that the essential features of something are represented without the background or inessential detail being included.

Object-oriented programming can be seen as an extension of data abstraction because it allows you to define new UDDTs which derive properties from those data types already declared. This is the principle of inheritance. The open-closed principle states that a module is open if it is available for extension and it is closed it it is available for use by other modules. With classical approaches to design there is no way to write modules that are both open and closed. Inheritance solves the apparent dilemma with the open-closed principle [31].

A.3 Understanding object-oriented design

In system development there are both internal quality and external quality factors. Internal factors include modularity and readability and are only perceptible to computer professionals, therefore only external factors really matter in the end. However internal factors are the key to ensuring that external factors are satisfied and object-oriented design is then a technique for obtaining internal quality. The main external factors are: correctness, robustness, extensibility, reusability and compatibility and these all benefit from object-oriented design.

An object consists of some data and a set of methods or operations that can be performed on those data. The operations and data are defined together and are a single entity. An object can be understood as a class or a UDDT or an instance of a class. We have essentially said that object-oriented design is the construction of a software system as a structured collection of UDDT implementations.

The idea is that object-oriented systems are to be assembled from pre-written components with minimal effort and that the assembled system will be easy to extend without any need to change the reused components. However, it is worth bearing in mind that very few systems live up to the pure concept of object-orientation and

should not need to.

Meyer [31] promotes seven steps towards object-oriented happiness:

1. **Object-based modular structure**: Systems are broken into subprograms on the basis of their modular structure. Hiding the representation is the key to modularity [43].

2. **Data Abstraction**: A programming technique described in section A.2 and further defined in section A.4.

3. **Automatic Memory Management**: Unused objects should be deallocated without program intervention. Constructors and destructors (see section A.4.1) are used to allocate and manage memory for a class and help to make a clean separation between interface and implementation. A constructor gives a recipe for building an object of a given class; a destructor gives a recipe for undoing whatever the constructor did. The constructor and destructor functions are automatically called [26].

4. **Classes**: A class (ideally one implementation of a UDDT) describes a set of data structures (objects) characterized by common properties . Every non-simple data type should be a module and every high-level module a data type [31]. Each class is a direct representation of a concept in the program. See section A.4.2 for more detail.

5. **Inheritance**: A new class may be declared as an extension/restriction of a previously defined class. If class A is defined as an heir to class B this implies that all the features and functionality of B are applicable to A. Class A can add to or customize the features of it's parent B (see polymorphism below), only the differences need to be specified. Alternatively class A can simply reuse the implementation of class B; See section A.4.2 for more detail.

6. **Polymorphism and dynamic binding**: It is possible for class A, a descendant of B, to redefine features of it's parent B; in other words each class in the class hierarchy can implement a shared action in a way appropriate to itself. This is known as polymorphism. The ability to determine an objects class at run-time and allocate its storage is referred to as dynamic binding.

7. **Multiple and repeated inheritance**:

   A derived class can have more than one parent and other classes can be descendants of this class. Therefore, hierarchically organized classes can be specified which is one of the key features for object-oriented programming; see section A.4.1 for more detail.

A.4 Data abstraction and object-oriented programming in C++

This section includes a brief overview of some of the more important aspects of the C++ programming language:

A.4.1 Classes

We can associate variables and functions with the name of a data type. These are called members and are either private, protected or public. Members are private by default and this means that they are only accessible by a member function of the same class or a friend (see section A.4.3) of the class. Protected means that a member is accessible by members and friends of derived classes and public means that a member is accessible by clients of the class. The keywords protected and public promote the idea of encapsulation, see section A.4.6. A client can read or modify the values of the private member variables of an instance of a class indirectly, by calling the public member functions of the class. Information hiding guidelines dictates that all data within a class be private.

C++ classes have two special kinds of member functions: constructors and destructors. Constructors create new instances of the the class and take responsibility for initializing them correctly. Destructors destroy instances (dynamic storage) of the class. A constructor has the same name as the class and a destructor has also the same name but is prefixed by ~.

A.4.2 Inheritance and virtual member functions

Object-oriented programming is the process of building class hierarchies, this is achieved by the principle of inheritance, which essentially making commonality explicit. C++ supports inheritance by means of derived classes, which can differentiate from their parent classes by adding member variables or functions and/or redefining member functions inherited from their base class. The base class (parent) must be created before the descendant class can be created. A base class can declare a member function using the keyword virtual, signifying that the base classes implementation of the member function is only a default which will be used only if the derived class does not supply it's own implementation. This is used to accomplish polymorphism and dynamic binding; see the listing of list.cpp in section C.2.

Inheritance is the most important concept in realizing the goal of constructing software systems from reusable parts, rather than hard coding each system form scratch.

A.4.3 Friend functions

There needs to be a way for another class to be able to access the private data members of a class without being part of the class hierarchy. Friend functions provide a means whereby conventional functions or other classes can access private data of a class, having no particular connection to the class.

A.4.4 Operator name overloading

It is possible to use the same function name for more than one function, provided the number and/or type of the arguments are distinctive; see the listing for openfile.cpp in section C.2. This is especially useful for writing multiple constructors and is a special case of polymorphism; see the listing for node.h in section C.1.

A.4.5 Inline functions

A function can be inline implicitly or explicitly. By explicitly declaring a function inline, by using the keyword inline, each call of the function is replaced by a copy of the entire function, thus the overhead of calling a function is eliminated. Member functions can also be made implicitly inline by including the implementation details (function body) in the function declaration within the class specification; see the listing for node.h in section C.1.

A.4.6 Encapsulation

Most of the details of the implementation of a class are kept in a separate file (header file) which calls the class. This header file includes the specification of the class, in other words the information that programs that use the class (clients) need to compile successfully. The implementation file uses the scope resolution operator (::) to identify what class the member function belongs to. See list.cpp for examples of the above. This separation into specification and implementation is done to hide the implementation details from the client. This is the principle of data hiding and is known as encapsulation.

A.5 Conclusion

Software has progressed from a purely procedural application to a data driven object based approach and the emphasis has further shifted to the use of object-oriented techniques. C++ is one of the few languages that supports data abstraction, OOP and traditional programming techniques.

The object-oriented approach dictates that software be designed as a number of objects, which should be as independent of their environment (rest of the program) as possible. The fewer assumptions an object makes about it's environment and vice-versa, the easier it become to transplant it into a different environment.

OOP represents a way of thinking and a methodology for computer programming that are quite different from the usual approaches supported by structured programming languages. It would take even an experienced C programmer several months to really appreciate the power of C++ and to become class designers. One can always learn more about the subtleties of object-oriented design.

The real benefits of an object-oriented approach comes from later modification to the developed system or to a newly developed system which can make use of all the libraries of classes already in place.

Appendix B

Pseudorandom numbers and generators

B.1 Introduction

Numbers which can be chosen at random are useful in many different kinds of applications, of which simulation is one. In one sense there is no such thing as a random number, instead we speak of a sequence of independent random numbers which follow a specific distribution. In other words each number was obtained by chance and had nothing to do with the other numbers in the sequence.

How can one choose numbers at random? Possibilities include: spinning a roulette wheel, sending a current through a resistor and observing the voltage over time, using an electronic process which has an inherent randomness such as thermal noise, cosmic ray counter, geiger counter.

B.2 Pseudorandom numbers

All practical "random number" generators on computers are actually simple deterministic programs that generate a periodic sequences of numbers that should look apparently random. These computed random numbers are sometimes referred to as pseudorandom numbers as the are essentially not truly random since all the numbers are known in advance once the seed initializing the recursive computations is known (each number is completely determined by its predecessor). These functions are referred to as pseudorandom number generators.

The main advantage of being able to produce a set of numbers arithmetically, which to all intensive purposes appear random, is that the same sequence can be reproduced whenever desired once the seed is known. This is very useful when comparing simulations as you known that any difference in output is not due to experimental error, it is also very helpful when debugging a program.

B.3 Generating pseudorandom numbers

A generator is defined by a finite state space S, a function $f : S \rightarrow S$ and an initial state s_0 called the seed. The state of the generator evolves recursively to produce a sequence of non-negative integers, $Z = \{s_i\}$, where

$$s_i = f(s_{i-1}), \quad i = 1, 2, 3, \ldots \tag{B.1}$$

At this point it is worth noting that most generators are inherently pseudorandom integer generators. They can be made pseudorandom real uniform $[0, 1)$ generators indirectly by dividing by the largest possible integer (modulus). So the states s_i of the sequence Z are transformed to produce the sequence, $U = \{U_i\}$, where

$$U_i = g(s_i), \quad \forall\, i \tag{B.2}$$

where $g : S \rightarrow [0, 1)$ and the elements U_i form the pseudorandom sequence taking values in some discrete subset of $[0, 1)$. The period of the generator is the smallest positive integer p such that

$$s_{i+p} = s_i \quad \forall\, i > v \tag{B.3}$$

for some integer $v \geq 0$.

The most common method of generating pseudorandom numbers is the Lehmer linear congruential method where

$$f(s) = (as + c) \mod m \tag{B.4}$$

$$g(s) = \frac{s}{m} \tag{B.5}$$

where a, c and m are pre-chosen constants, known as the multiplier, increment and modulus respectively, where $a < m$ and $c < m$. Equation B.5 effectively normalizes the sequence [30].

Any choice of values for these constants will produce a sequence of non- negative integers that ultimately falls into a repeating pattern. The period of the ideal sequence is infinite and if the constants are chosen carefully (see below) the pattern will be large and will appear to be random. Since the current random integer s_i depends only on the previous random integer s_{i-1} (see equation B.1), once a value has been repeated the entire sequence after it must be repeated.

The maximum period (cycle length) of the above generator is clearly m as a direct consequence of the modulus operation (see equation B.5). The cycle length is an important consideration because if the simulation is long enough to cause the random numbers to repeat (as a result of the short cycle length), the simulation outputs will not be independent.

Choosing a, c and m is a science which incorporates both theoretical results and empirical tests. Since the period of the ideal sequence is infinite, the modulus m should be chosen as large as possible. Theoretical results then exist that give conditions for choosing values of the multiplier a and the increment c [28].

B.4 Multiplicative linear congruential generator

It is common practice to eliminate the addition operation by setting the increment $c = 0$, in which case the generator is called a multiplicative linear congruential generator (MLCG) and its state space is $S = \{0, 1, \ldots, m-1\}$. This is because, by definition, $x \bmod m$ is the remainder after x is divided by m. As described above, a MLCG has maximum cycle length of m.

An independent subsequence can be produced by splitting a single generator, provided the seeds can be chosen regularly spaced and far enough apart in the cycle to ensure that the sequences do not overlap. In other words, given any seed, s_i, we can compute s_{i+j}, for any $j > 0$, without generating all intermediate values.

$$s_{i+j} = (a^j s_i) \bmod m \tag{B.6}$$

which can be easily shown. This can then be implemented like any MLCG.

B.5 Generator used in the simulation system

Various methods have been proposed for combining two or more pseudorandom number generators. It has been mathematically demonstrated that this results in a much longer period. Such a generator [30] which combines 2 MLCGs is used in the simulation system developed (see the listing of mlcg32.cpp in appendix C). The first generator has modulus $m_1 = 2147483563$ and multiplier $a_1 = 40014$ and the second generator has modulus $m_2 = 2147483399$ and multiplier $a_2 = 40692$. The combined generator has period $\approx 2.30584 \times 10^{18}$. This generator has been submitted successfully [30] to a comprehensive series of statistical tests described in Knuth [28].

Appendix C

ATM simulator source code

C.1 Class definitions and header files

```
//*************************************************************************

/*                              * * * * * * * * * * * * * * * * * *
    File:    node.h             *                                   *
    Author:  Tom Corcoran       *    interface to class line_c      *
    Date:    20/7/93            *                                   *
                                * * * * * * * * * * * * * * * * * *       */

//*************************************************************************

//  includes                            //  defines

#include<stdlib.h>                      //  NULL
#include<iostream.h>                    //  ostream

//*************************************************************************

//  defines a node

class line_c {

 friend class line_list_c;              //  private members accessible by
                                        //   line_list_c (forward feference)
 private:
  int line_no;                          //  line number of node
  int firing_time;                      //  firing time of node
  int firing_flag;                      //  indicates whether node firing
                                        //   (0 = not firing,1 = firing)
  line_c *prior;                        //  pointer to previous node
  line_c *next;                         //  pointer to next node
 public:
```

```
  line_c(int num = 0,int time = 0,
         int flag = 0,
         line_c *p = NULL,
         line_c *q = NULL) {
   line_no = num;
   firing_time = time;
   firing_flag = flag;
   prior = p;
   next = q;
  };                                    //  constructor

  //  set private members

  void set_line_no(int num) {
   line_no = num;                       //  set line_no
  }
  void set_firing_time(int time) {
   firing_time = time;                  //  set firing_time
  }
  void set_firing_flag(int flag) {
   firing_flag = flag;
  }                                     //  set firing_flag

  //  inserter function is friend

  friend ostream &operator<<(ostream &stream,line_list_c *list);
  friend ostream &operator<<(ostream &stream,line_c *node) {
   stream << node->line_no << "\t" << node->firing_time << "\t"
          << node->firing_flag << endl;
   return stream;
  }
};

//*****************************************************************************

/*                              * * * * * * * * * * * * * * * * * *
    File:    list.h             *                                   *
    Author:  Tom Corcoran       *   interface to class line_list_c  *
    Date:    20/7/93            *                                   *
                                * * * * * * * * * * * * * * * * * *         */

//*****************************************************************************

//  includes                            //  defines

#include <fstream.h>                    //  ofstream
```

```
//*****************************************************************************

//  forward declarations

class line_c;
class mlcg32_c;
class probabilities_c;
class queue_c;

//*****************************************************************************

//  defines a doubly linked list made up of nodes

class line_list_c {
 private:

  /*  both a head and a foot (as opposed to just a head) are defined
      so as to make the code more readable                              */

  line_c *head;                          //  head->next points to head of list
  line_c *foot;                          //  foot->prior points to foot of list
  int last_process_time;                 //  last time list processed
 public:
  line_list_c
   (int temp_last_process_time);         //  constructor

  //  only used to link initial two nodes (head and foot) in list

  void connect_two_nodes(line_c *one,line_c *two);

  //  return functions

  line_c * return_head(void) {
   return head;
  }                                      //  returns head of list
  line_c * return_foot(void) {
   return foot;
  }                                      //  returns foot of list
  int return_last_process_time(void) {
   return last_process_time;
  }                                      //  returns last process time

  //  new_head,new_foot assume head,foot already created

  void new_head(line_c *new_head,
```

```
                line_c *old_head);      //  insert node at head of list
    void new_foot(line_c *new_foot,
                line_c *old_foot);      //  insert node at foot of list

    /*  insert_mid_before,insert_mid_after assume at least two nodes linked
        can be used to create linked list                                   */

    void insert_mid_before
        (line_c *node_a,line_c *node_b); //  insert node before another node
    void insert_mid_after
        (line_c *node_2,line_c *node_1); //  insert node after another node

    //  following member functions for use once list has been setup

    void sort_linked_list(int s_ticks);   //  sorts list in order of firing time
    void move_line(line_c *k,
                mlcg32_c *random_gen,
                int s_ticks);             //  controls moving of node in list
    void remove_head(line_c *old_head);   //  removes head of list
    void remove_foot(line_c *old_foot);   //  removes foot of list
    void remove_line(line_c *k);          //  removes node from list
    void clear_pointers(line_c *k);       //  clears pointers of node

    //  the following functions write the state of the list to file

    void print_to_f_detail(
        ofstream *file_ptr);              //  detailed write list

    void clear_list(void);                //  clears linked list

    //  performs simulation for one cycle through the list (block)

    virtual void one_block_simulation(int &empty_queue,queue_c *queue,
                                      probabilities_c *prob,
                                      mlcg32_c *random_gen);

    //  inserter function is friend (basic write list)

    friend ostream &operator<<(ostream &stream,line_list_c *list);


    ~line_list_c(void);                   //  destructor (destroys linked list,
                                          //              head,foot)
};
```

```
//*************************************************************************
/*
                                * * * * * * * * * * * * * * * * * *
        File:    prob.h         *                                 *
        Author:  Tom Corcoran   *    interface to  class prob_c   *
        Date:    20/7/93        *                                 *
                                * * * * * * * * * * * * * * * * * *      */

//*************************************************************************

//  includes                          //  defines

#ifndef LIST_H
 #define LIST_H                     //  member function of line_list_c is
 #include "list.h"                  //  declared as a friend
#endif
//*************************************************************************

//  class contains simulation input parameters and probabilities

class probabilities_c {
 private:
  int l;                            //  number of lines (nodes in list)
  int s;                            //  number of ticks in block over
                                    //   which l lines distributed
  double rho;                       //  load
  double alpha;                     //  prob(active -> active)
  double beta;                      //  prob(inactive -> active)
  double gamma;                     //  prob(silence < s ticks)
 public:

  //  constructors

  probabilities_c(void) {};         //  default constructor

  probabilities_c(int temp_l,
                  int temp_s,
                  double temp_rho) :
   l(temp_l),s(temp_s),rho(temp_rho) {
  };
  probabilities_c(double phi,int temp_l,double temp_rho);
  probabilities_c(double phi,int temp_l,int temp_s);

  //  set private members

  void set_no_lines(double temp_l) {
```

```
 l = temp_l;
}                                       //  sets number of lines
void set_no_ticks(double temp_s) {
 s = temp_s;
}                                       //  sets number of ticks
void set_load(double temp_rho) {
 rho = temp_rho;
}                                       //  sets rho/load
void set_alpha(double temp_alpha
     = 0.995466666666666) {
 alpha = temp_alpha;
}                                       //  sets alpha (default value)
void set_beta(double temp_beta) {
 beta = temp_beta;
}                                       //  sets beta
void set_gamma(void);                   //  prob(silence < s ticks)

//  return functions

int return_no_lines(void) {
 return l;
}                                       //  returns number of lines
int return_no_ticks(void) {
 return s;
}                                       //  returns number of ticks
double return_load(void) {
 return rho;
}                                       //  returns load
double return_alpha(void) {
 return alpha;
}                                       //  returns alpha
double return_beta(void) {
 return beta;
}                                       //  returns beta
double return_sigma(void);              //  return value of sigma (s/l)
double return_phi(void);                //  return value of phi
double
 return_prob_line_active(void);         //  returns probability line active

//  fix alpha/beta

void fix_alpha(double temp_alpha        //  fix alpha (default value)
     = 0.995466666666666);              //  works out beta(l,s,rho,alpha)

void fix_beta(double temp_beta);        //  fix beta
                                        //  works out alpha(l,s,rho,beta)
```

```
  //  calculate alpha,beta

  double calc_alpha(void);              //  used if beta (and rho) fixed
  double calc_beta(void);               //  used if alpha (and rho) fixed

  // .used when phi, ie. l(1 - beta), is fixed

  int calc_s_given_phi(double phi);
  double calc_rho_given_phi(double phi);
  double calc_beta_given_phi(double phi);

  //  checks that private members are consistent

  void check_conditions(void);

  //  friend declaration: function has access to private members

  friend void line_list_c::one_block_simulation(int &empty_queue,
                                                queue_c *queue,
                                                probabilities_c *prob,
                                                mlcg32_c *random_gen);
  friend ostream &operator<<(ostream &stream,probabilities_c *prob);
};

//*****************************************************************************

/*                                  * * * * * * * * * * * * * * * * * *
        File:    queue.h            *                                   *
        Author:  Tom Corcoran       *    interface to class queue_c     *
        Date:    20/7/93            *                                   *
                                    * * * * * * * * * * * * * * * * * *    */

//*****************************************************************************

//  includes                              //  defines

#include <fstream.h>                      //  ofstream
#ifndef LIST_H
 #define LIST_H                           //  member function of line_list_c
 #include "list.h"                        //   is declared as a friend
#endif

//*****************************************************************************

/*  class contains buffer array which keeps track of simulation statistics
```

```
    eg. queue[5] = 9 => buffer was of length 5 during simulation 9 times    */

class queue_c {
 private:
  int buffer;                                  //  current length of actual queue

  /* dynamic array (initially records no of times buffer is each length)
      any overflow is stored in queue[overflow_size]                          */

  double *queue;

  int max_queue_size;                          //  maximum length of array
  int overflow_size;                           //  max length of array minus one
  int actual_queue_size;                       //  actual length of array
  long int total_activity;                     //  total number of "customers served"
 public:

  //  constructors

  queue_c(void) {}                             //  default constructor
  queue_c(int temp_max_queue_size,
      int temp_initial_queue);
  queue_c(int temp_max_queue_size,
  long int temp_total_activity,
  double *temp_queue);

  //  updates private members

  void add_to_total_activity
       (long int amount);                      //  adds amount to total activity
  void set_actual_queue_size(void);            //  sets actual array size

  //  return private members

  long int return_total_activity(void) {
   return total_activity;
  }                                            //  returns total_activity
  double return_queue(int element) {
   return queue[element];
  }                                            //  returns queue[elem]
  double * return_queue_ptr(void) {
   return queue;
  }                                            //  returns pointer to queue
  int return_max_queue_size(void) {
   return max_queue_size;
  }
```

```
 int return_overflow_size(void) {
  return overflow_size;
 }                                        //  returns overflow size
 int return_actual_queue_size(void) {
  return actual_queue_size;
 }                                        //  returns actual queue size
 int return_buffer(void) {
  return buffer;
 }                                        //  returns buffer

 //  used on whole array

 void setup_pdf(void);                    //  creates PDF from buffer array
 double check_total_probability(void);    //  checks probability sums to 1
 void build_cdf(void);                    //  builds CDF from PDF,p{q < b]
 void build_one_minus_cdf(void);          //  p[q >= b]
 void log_queue(void);                    //  get log10 of array contents

 //  friend declarations: functions have access to private members

 friend void line_list_c::one_block_simulation(int &empty_queue,
                                               queue_c *queue,
                                               probabilities_c *prob,
                                               mlcg32_c *random_gen);

 friend ostream &operator<<(ostream &stream,queue_c *queue);

 ~queue_c(void) {
   delete queue;                          //  destructor: frees memory taken
 }                                        //              by queue
};

//*************************************************************************

/*                              * * * * * * * * * * * * * * * * * *
       File:   mlcg32.h         *                                   *
       Author: Tom Corcoran     *    interface to class mlcg32_c    *
       Date:   20/07/93         *                                   *
                                * * * * * * * * * * * * * * * * * *     */

//*************************************************************************

// Includes                              //  defines

#include<fstream.h>                      //  ofstream,fstream
```

```
//*************************************************************************

class mlcg32_c {
 private:
  long int seed1;                          //  first seed for ran no generator
  long int seed2;                          //  second seed for ran no generator
 public:

  //  constructors

  mlcg32_c(void) {};                       //  default constructor
  mlcg32_c(long int temp_seed1,
           long int temp_seed2) :
   seed1(temp_seed1),seed2(temp_seed2) {
  };
  mlcg32_c(fstream *seed_f);

  // reset seeds

  void reset_seeds(
   long int temp_seed1 = 1,
   long int temp_seed2 = 1);               //  given seeds
  void reset_seeds(ifstream *seed_f);      //  read seeds from file
  double random(void);                     //  generates random number in [0,1]
  int random_int(int max);                 //  returns random number in [0, int)

  //  inserter function is friend

  friend ostream &operator<<(ostream &stream,mlcg32_c *random);

};

//*************************************************************************

/*                                 * * * * * * * * * * * * * * * * * * *
        File:    timer.h           *                                     *
        Author:  Tom Corcoran      *   interface to class stop_watch_c   *
        Date:    20/7/93           *                                     *
                                   * * * * * * * * * * * * * * * * * * *  */

//*************************************************************************

//  includes                                 //  defines

#include<iostream.h>                         //  ostream
#include<time.h>                             //  time_t,tm
```

```
//*************************************************************************

class stop_watch_c {
 private:
  time_t start;                                  //  simulation start time
  time_t end;                                    //  simulation end time
  int hours;                                     //  hours of simulation
  int minutes;                                   //  minutes of simulation
  int seconds;                                   //  seconds of simulation
 public:
  stop_watch_c(void);                            //  constructor
  void set_start_time(time_t temp_start) {
   start = temp_start;
  }                                              //  sets start
  void set_end_time(time_t temp_end) {
   end = temp_end;
  }                                              //  sets end

  //  functions for displaying start/end time

  void display_start_time_to_screen(void);
  void display_end_time_to_screen(void);
  void display_time_to_screen(time_t now);

  void diff_time(void);                          //  gets differnece between
                                                 //   start and end
  //  inserter function is a friend

  friend ostream &operator<<(ostream &stream,stop_watch_c *watch);
 };

//*************************************************************************

/*                              * * * * * * * * * * * * * * * * * *
        File:    charstr.h      *                                   *
        Author:  Tom Corcoran   *   interface to class string_c   *
        Date:    20/7/93        *                                   *
                                * * * * * * * * * * * * * * * * * *      */

//*************************************************************************

//  includes                        //  defines

#include <iostream.h>               //  ostream
```

```
//*****************************************************************************

class string_c {
 private:
  char *string_ptr;
  int string_length;
 public:

  // constructors

  string_c();                          //  default
  string_c(char *temp_string_ptr);

  operator char *() {
   return string_ptr;                  //  conversion function, convert to char*
  }
  char * return_string_ptr(void) {
   return string_ptr;                  //  returns string_ptr
  }
  ~string_c(void);                     //  destructer

  //  inserter function is a friend

  friend ostream &operator<<(ostream &stream, string_c *string);
};

//*****************************************************************************

/*                                     * * * * * * * * * * * * * * * * * *
        File:    routines.h             *                                   *
        Author:  Tom Corcoran           *    interface to routines.cpp      *
        Date:    23/10/93               *                                   *
                                        * * * * * * * * * * * * * * * * * *    */

//*****************************************************************************

//  includes                            //  defines

#include<iostream.h>                    //  iostream
#include <fstream.h>                    //  ifstream


//*****************************************************************************

//  function prototypes
```

```
double round_double(double num);
int int_round_double(double num);
double * setup_dynamic_array(int array_size);
void initialize_dynamic_array(int array_size,double * new_array);
ostream &fixed_and_showpoint(ostream &stream);
ostream &error(ostream &stream);
void general_error(char *str_ptr);
void allocation_error(char *str_ptr);
void allocation_error(char *str_ptr,long int display_int);
void display_and_exit(char *error_str);


//*************************************************************************

/*                              * * * * * * * * * * * * * * * * * * *
        File: openfile.h        *                                     *
        Author: Tom Corcoran    *      interface to openfile.cpp      *
        Date: 27/11/92          *                                     *
                                * * * * * * * * * * * * * * * * * * *  */
//*************************************************************************

//  includes                                //  defines

#include<fstream.h>                         //  ifstream,ofstream,fstream

//*************************************************************************

// Function prototypes

void open_file(char *file_name,
               ifstream *fp);               //  open file for reading
void open_file_input(char *file_name,
                     fstream *fp);          //  open i/o file for reading
void open_file_io(char *file_name,
                  fstream *fp);             //  open i/o file for reading and
                                            //   writing
//  write to file and overwrite

void open_file(char *file_name,
               ofstream *fp);               //  open file for writing

void open_file_output(char *file_name,
                      fstream *fp);         //  open i/o file for writing

//  write fo file and append
```

```
void open_file_app(char *file_name,
                   ofstream *fp);          //  open i/o file for writing

//  overloaded error checks

void check_error(ifstream *fp,
                 char *file_name);         //  error check for input file
void check_error(ofstream *fp,
                 char *file_name);         //  error check for output file
void check_error(fstream *fp,
                 char *file_name);         //  error check for i/o file

void exit_program(char *file_name);        //  calls exit function

//*****************************************************************************
```

C.2 Class member functions

```
//*****************************************************************************

/*  File:    list.cpp              * * * * * * * * * * * * * *
    Author:  Tom Corcoran          *                         *
    Date:    20/7/93               *    class line_list_c    *
    Version: 6.0                   *    member functions     *
                                   *                         *
                                   * * * * * * * * * * * * * *           */

//*****************************************************************************

//  includes                       //  defines

#ifndef LIST_H
 #define LIST_H
 #include "list.h"                 //  class line_list_c
#endif
#ifndef NODE_H
 #define NODE_H
 #include "node.h"                 //  class line_c
#endif
#ifndef MLCG32_H
 #define MLCG32_H
 #include "mlcg32.h"               //  class mlcg32_c
#endif
#ifndef ROUTINES_H
 #define ROUTINES_H
```

```
 #include "routines.h"              //  allocation_exit()
#endif

//*****************************************************************************

line_list_c::line_list_c(int temp_last_process_time) {

 /*  creating new objects (need instances as use head->next,head->prior
                           foot->next,foot->prior)                          */

 if (!(head = new line_c))
  allocation_error("instance of line_c for head of list");
 if (!(foot = new line_c))
  allocation_error("instance of line_c for foot of list");

 //  initializing last time list processed

 last_process_time = temp_last_process_time;

 //  linking head and foot

 connect_two_nodes(head,foot);
}
//*****************************************************************************

void line_list_c::sort_linked_list(int s_ticks) {

 //  objective: to sort linked list in order of firing_time

 int search_tick = 1;               //  firing time
 class line_c *current_pos = head; //  node up to which list is sorted
 class line_c *find_pos = head;     //  node which is next in order
 class line_c *next_pos = NULL;     //  node which is next in sort

 while (search_tick <= s_ticks) {

  //  search list for match of firing_time (search_tick)

  while ((find_pos != foot)&&(find_pos->firing_time != search_tick))
   find_pos = find_pos->next;

  //  after leaving inner while loop => match found or at foot of list

  if (find_pos != foot) {           //  match found
   next_pos = find_pos->next;       //  search will continue from this node
   remove_line(find_pos);           //  removes matched node from list
```

```
   insert_mid_after(find_pos,
                    current_pos);  //  inserts matched node in order
   current_pos = find_pos;         //  list now sorted up to this node
   find_pos = next_pos;            //  find_pos determines from where
                                   //   search continues
  }
  else {                           //  no (other) node firing at this tick
   search_tick++;                  //   => increment search_tick
   find_pos = current_pos->next;   //  search continues from find_pos for
                                   //   first occurence of next tick
  }
 }
}
//*****************************************************************************

void line_list_c::move_line(line_c *k,mlcg32_c *random_gen,int s_ticks) {

 /*  objective: controls moving of node in linked list
                node is removed and reinserted at appropriate position     */

 int t;                                          //  firing time of node to be
                                                 //   inserted
 int old_firing_time = k->firing_time;           //  old firing time of node
                                                 //   to be moved
 int ticks_left = s_ticks - old_firing_time; //  number of ticks left in block
 line_c *find_pos = NULL;                        //  pointer to node in list

 if (k->prior == head)
  remove_head(k);
 else {
  if (k->next == foot)                           //  remove node from list
   remove_foot(k);
  else
   remove_line(k);
 }

 //  want find_pos to be structure contained at address head->next

 find_pos = head->next;

 /*  generate random time to find where node should be inserted
     returns random numner in [0,s_ticks)                                   */

 t = random_gen->random_int(s_ticks) + 1;

 //  find new firing time of node
```

```
if (t <= ticks_left)
 t += old_firing_time;
else
 t -= ticks_left;

//  check to see if node should be inserted at head

if (t < find_pos->firing_time) {
 new_head(k, find_pos);
}
else {

 /*  possibilities before loop x:
      1) t = firing_time => place node after head of list (find_pos)
      2) t > firing_time => enter next loop                              */

 if (t > find_pos->firing_time) {
  while ((t >= find_pos->firing_time) && (find_pos->next != foot))   //  {1}
   find_pos = find_pos->next;
 }

 /*  possibilities after loop x:
      3) t < firing_time (even if find_pos->next = foot)
         => place node before find_pos
      4) t >= firing_time and find_pos->next = foot
         => place node at foot of list (after find_pos)                 */

 //  check to see if node should be inserted at foot

 if ((find_pos->next == foot)&&(t >= find_pos->firing_time))
  new_foot(k, find_pos);                                            //  {4}
 else {

  //  find_pos->next may be = foot

  if (t < find_pos->firing_time)
   insert_mid_before(k,find_pos);                                   //  {3}

  //  t = find_pos->firing_time (didn't enter loop x)

  else
   insert_mid_after(k,find_pos);                                    //  {1}
 }
} //  end of first else
```

```
 //  setting firing time of moved node

 k->firing_time = t;
}
//*************************************************************************

void line_list_c::remove_head(line_c *old_head) {

 //  Objective: remove the head of the list

 (old_head->next)->prior = head;
 head->next = old_head->next;      //  reset head
 clear_pointers(old_head);         //  remove the pointers from node old_head
}
//*************************************************************************

void line_list_c::remove_foot(line_c *old_foot) {

 //  Objective: remove the foot of the list

 (old_foot->prior)->next = foot;
 foot->prior = old_foot->prior;    //  reset foot
 clear_pointers(old_foot);         //  remove the pointers from node old_foot
}
//*************************************************************************

void line_list_c::remove_line(line_c *k) {

 /*  Objective: k is the address of node to be moved, so the
                pointers are reordered

     node which was pointing to k now points to node to which k was pointing*/

 (k->prior)->next = k->next;

 //  node which k was pointing to is now pointed to by k

 (k->next)->prior = k->prior;

 //  remove the pointers from node k

 clear_pointers(k);
}
//*************************************************************************

void line_list_c::clear_pointers(line_c *k) {
```

```
 //  Objective: remove the pointers from node k

 k->prior = NULL;
 k->next = NULL;
}
//*************************************************************************

void line_list_c::connect_two_nodes(line_c *one,line_c *two) {

 //  Objective: one is to be conected to two and vice versa

 one->next = two;                    //  one is set to point (next) to two
 two->prior = one;                   //  two is set to point (prior) to one
}
//*************************************************************************

void line_list_c::new_head(line_c *new_head,line_c *old_head) {

 /*  Objective: new_head is to be inserted before old_head

     the node which used to point(prior) to head is set to point to new_head*/

 old_head->prior = new_head;

 //  new_head is set to point(next) to the node before which it is placed

 new_head->next = old_head;

 //  new_head is set to point(prior) to head and head is reset

 new_head->prior = head;
 head->next = new_head;
}
//*************************************************************************

void line_list_c::new_foot(line_c *new_foot,line_c *old_foot) {

 /*  Objective: new_foot is to be inserted after old_foot

     node which used to point(next) to foot is set to point to new_foot     */

 old_foot->next = new_foot;

 //  new_foot is set to point(prior) to the node after which it is placed
```

```
 new_foot->prior = old_foot;

 //  new_foot is set to point(next) to foot and foot is reset

 new_foot->next = foot;
 foot->prior = new_foot;
}
//*************************************************************************

void line_list_c::insert_mid_before(line_c *node_a,line_c *node_b) {

 /*  Objective: insert node_a mid list before node_b

     node after which node_a was placed is set to point(next) to node_a     */

 (node_b->prior)->next = node_a;

 //  node_a is set to point(prior) to the node after which it was placed

 node_a->prior = node_b->prior;

 //  the node before which node_a was placed is set to point(prior) to node_a

 node_b->prior = node_a;

 //  node_a is set to point(next) to node before which it was placed

 node_a->next = node_b;
}
//*************************************************************************

void line_list_c::insert_mid_after(line_c *node_2,line_c *node_1) {

 /*  Objective: insert node_2 mid list after node_1

     node before which node_2 was placed is set to point(prior) to node_2   */

 (node_1->next)->prior = node_2;

 //  node_2 is set to point(next) to node before which it was placed

 node_2->next = node_1->next;

 //  the node after which node_2 was placed is set to point(next) to node_2

 node_1->next = node_2;
```

```
 //  node_2 is set to point(prior) to the node after which it was placed

 node_2->prior = node_1;
}
//*****************************************************************************

void line_list_c::print_to_f_detail(ofstream *file_ptr) {

 //  objective: prints linked list to file (detailed)

 line_c *ptr = head;
 int count_lines_firing = 0;
 int count_lines_not_firing = 0;

 // printing out the head of the list and headings

 *file_ptr << "The state of the lines are as follows: " << "\n\n";
 *file_ptr << "Head = Line " << (ptr->next)->line_no << "\n\n";
 *file_ptr << "Line   Time   Fire"
           << "       Pointers\n";

 //  loops through each node in linked list

 do {
  ptr = ptr->next;

  //  counting lines firing

  if (ptr->firing_flag)
   count_lines_firing++;
  else
   count_lines_not_firing++;

  //  printing line_no, firing_flag, firing_time

  *file_ptr << ptr->line_no << "\t" << ptr->firing_time << "\t"
            << ptr->firing_flag << endl;

  //  printing out sequence of lines which point to each other (opt)

  *file_ptr << "\t\t" << (ptr->prior)->line_no << " <- " << ptr->line_no
            << " -> " << (ptr->next)->line_no  << "\n";
 } while (ptr->next != foot);

 //  printing out the foot of the list
```

```
 *file_ptr << "\nFoot = Line " << (ptr)->line_no << endl;

 //  printing out number of lines firing/not firing

 *file_ptr << "\nLines Firing = " << count_lines_firing
          << " Lines not firing = " << count_lines_not_firing << "\n\n";
}
//*****************************************************************************

void line_list_c::clear_list(void) {

 //  objective: clearing linked list

 line_c *pres_list_elem = head->next;
 line_c *next_list_elem = NULL;

 while (pres_list_elem != foot) {
  next_list_elem = pres_list_elem->next;
  delete pres_list_elem;
  pres_list_elem = next_list_elem;
 }
}
//*****************************************************************************

line_list_c::~line_list_c(void) {

 //  objective: deleting linked list and other pointers

 clear_list();
 delete head;
 delete foot;
}
//*****************************************************************************

ostream &operator<<(ostream &stream,line_list_c *list) {

 //  objective: prints linked list to file

 line_c *ptr = list->head;

 //  loops through each node in linked list

 do {
  ptr = ptr->next;
```

```
  // printing line_no, firing_flag, firing_time (line_c inserter function)

  stream << ptr;
 } while (ptr->next != list->foot);
 return stream;
}

//*****************************************************************************

/*                              * * * * * * * * * * * * *
        File:    prob.cpp       *                       *
        Author:  Tom Corcoran   *     class prob_c      *
        Date:    20/7/93        *   member functions    *
                                *                       *
                                * * * * * * * * * * * * *                  */

//*****************************************************************************

// includes                              // defines

#include <math.h>                       // pow()
#include <stdlib.h>                     // exit()
#include <iomanip.h>                    // setprecision()
#ifndef PROB_H
 #define PROB_H
 #include "prob.h"                      // class probabilities_c
#endif
#ifndef ROUTINESS_H
 #define ROUTINES_H
 #include "routines.h"                  // modules for rounding double,
#endif                                  //  setiosflags,error

//*****************************************************************************

probabilities_c::probabilities_c(double phi,int temp_l,double temp_rho) {

 // initializing l,s,rho,alpha,beta,gamma

 l = temp_l;
 rho = temp_rho;
 set_alpha();
 beta = calc_beta_given_phi(phi);
 s = calc_s_given_phi(phi);
 set_gamma();
}
//*****************************************************************************
```

```
probabilities_c::probabilities_c(double phi,int temp_l,int temp_s) {

 //  initializing l,s,rho,alpha,beta,gamma

 l = temp_l;
 s = temp_s;
 set_alpha();
 beta = calc_beta_given_phi(phi);
 rho = calc_rho_given_phi(phi);
 set_gamma();
}
//*****************************************************************************

void probabilities_c::set_gamma(void) {

 /*  calculating probability silence less then s ticks
     p(silence >= s ticks) = beta to the power of (s-1)                    */

 const int s_minus_one = s - 1;        //  value of power to which beta raised

 //  returning value of gamma

 gamma = (1.0 - pow(beta, double(s_minus_one)));
}
//*****************************************************************************

double probabilities_c::return_sigma(void) {

 //  returns value of sigma

 return (double(s)/double(l));
}
//*****************************************************************************

double probabilities_c::return_prob_line_active(void) {

 /*  mean lines active per block = p*s
     b = prob(line active) = (mean lines active per block)/l               */

 return ((rho*double(s))/double(l));
}
//*****************************************************************************

double probabilities_c::return_phi(void) {
```

```
 //  returns value of phi

 return (1 - alpha)/((1/rho) - return_sigma());
}
//*************************************************************************

void probabilities_c::fix_alpha(double temp_alpha) {

 //  if no arguement alpha is set to default which is declared in header

 alpha = temp_alpha;

 //  working out value of beta (function of l,s,rho,alpha)

 beta = calc_beta();
}
//*************************************************************************

void probabilities_c::fix_beta(double temp_beta) {

 //  working out value of alpha (function of l,s,rho,beta)

 alpha = calc_alpha();
 beta = temp_beta;
}
//*************************************************************************

int probabilities_c::calc_s_given_phi(double phi) {

 /*  round number of ticks to nearest integer
     round_double() first rounds double value to nearesy whole number      */

 int temp_s = int_round_double(
               ( double(1)*( (1/rho) - ((1 - alpha)/phi)) )
              );
 return temp_s;
}
//*************************************************************************

double probabilities_c::calc_rho_given_phi(double phi) {

 return ( 1/(return_sigma() + ( (1 - alpha)/phi )) );
}
//*************************************************************************

double probabilities_c::calc_beta_given_phi(double phi) {
```

```
 return (1 - (phi/double(l)));
}
//*************************************************************************

double probabilities_c::calc_alpha(void) {

 //  note: alpha = P(active -> active)

 return (1 - ((1 - beta)*(l - rho*double(s)))/rho);
}
//*************************************************************************

double probabilities_c::calc_beta(void) {

 //  note: beta = P(inactive -> inactive)

 return (1 - (1 - alpha)/((double(l)/rho) - double(s)));
}
//*************************************************************************

void probabilities_c::check_conditions(void) {

 if (l < 10) {
  cout << error << "number of lines = " << l << " (l >= 10)\n";
  exit(1);
 }
 if (s < 4) {
  cout << error << "number of ticks = " << s << " (s >= 4)\n";
  exit(1);
 }
 if (s > l) {
  cout << error << "number of ticks > number of lines\n";
  exit(1);
 }
 if ((rho < 0.05)|(rho > 1.0)) {
  cout << error << "load = " << rho << " ( rho [0.05,1.0] )\n";
  exit(1);
 }
 if ((alpha < 0.05)|(alpha >= 1.0)) {
  cout << error << "Prob[active line stays active] = " << alpha
       << " ( alpha [0.05,1.0) )\n";
  exit(1);
 }
 if ((beta < 0.05)|(beta >= 1.0)) {
  cout << error << "Prob[inactive line stays inactive] = " << beta
```

```
        << " ( beta [0.05,1.0) )\n";
    exit(1);
  }
  if ((gamma <= 0.0)|(gamma >= 1.0)) {
    cout << error << "Prob[silence < given number of ticks] = " << gamma
        << " ( gamma [0.05,1.0) )\n";
    exit(1);
  }
}
//*****************************************************************************

ostream &operator<<(ostream &stream,probabilities_c *prob) {

  stream << fixed_and_showpoint << setprecision(8)
         << "\nl = " << prob->l << " s = " << prob->s << " rho = " << prob->rho
         << " alpha = " << prob->alpha << "\nbeta = " << prob->beta
         << " gamma = " << prob->gamma << endl;
  return stream;
}

//*****************************************************************************

/*                                  * * * * * * * * * * * * *
        File:    queue.cpp          *                       *
        Author:  Tom Corcoran       *     class queue_c     *
        Date:    20/7/93            *    member functions   *
                                    *                       *
                                    * * * * * * * * * * * * *          */

//*****************************************************************************

//  includes                        //  defines

#include <stdlib.h>                 //  exit()
#include <iostream.h>               //  cout
#include <math.h>                   //  log10(),fabs()
#include <iomanip.h>                //  setiosflags()
#include <dos.h>                    //  delay()
#ifndef QUEUE_H
  #define QUEUE_H
  #include "queue.h"                //  class queue_c
#endif
#ifndef ROUTINES_H
  #define ROUTINES_H
  #include "routines.h"             //  module for error
#endif
```

```
//*************************************************************************

queue_c::queue_c(int temp_max_queue_size,long int temp_total_activity,
                 double *temp_queue) {

 // initializing private members

 buffer = 0;
 max_queue_size = temp_max_queue_size;
 overflow_size = max_queue_size - 1;
 actual_queue_size = temp_max_queue_size;
 total_activity = temp_total_activity;
 queue = temp_queue;
}
//*************************************************************************

queue_c::queue_c(int temp_max_queue_size,int temp_initial_queue) {

 /* initializing buffer,max_queue_size,overflow size,
    actual_queue_size and num_samples                                    */

 buffer = temp_initial_queue;
 max_queue_size = temp_max_queue_size;
 overflow_size = max_queue_size - 1;
 actual_queue_size = temp_max_queue_size;
 total_activity = 0;

 /*  allocating space for buffer array (queue)
     new returns zero if allocation fails                                */

 queue = setup_dynamic_array(max_queue_size);

 //  initializing buffer array

 initialize_dynamic_array(max_queue_size,queue);
}
//*************************************************************************

void queue_c::add_to_total_activity(long int amount) {

 //  adds amount to total activity

 total_activity += amount;
}
//*************************************************************************
```

```
void queue_c::set_actual_queue_size(void) {

 /*  sets: actual max buffer length if < max_queue_size
     queue[0] always equal 0 => don't worry about it
     queue[j] = 0.0 => array is of length j                          */

 register int j = 1;
 while (j <= overflow_size) {
  if (queue[j] == 0.0) {
   actual_queue_size = j;
   break;
  }
  j++;
 }
}
//*************************************************************************

void queue_c::setup_pdf(void) {

 /*  objective: normalises queue array by dividing each entry by
                total_activity (number of samples used in building array)

     queue[0] always equal 0 => don't worry about it
     queue[j] = 0.0 => array is of length j;                         */

 register int j = 1;
 while (j < actual_queue_size) {

  /*  normalize so sum of probabilities add to 1
      total_activity = sum over all j of queue[j]                    */

  queue[j] *= 1.0/double(total_activity);
  j++;
 }
}
//*************************************************************************

double queue_c::check_total_probability(void) {

 /*  note: if the actual max buffer size is very small, then the total
          probaility will not add up to 1.0 due to rounding error    */

 double check_total_probability = 0.0;

 register int j = 1;
```

```
 for (j = 1; j < actual_queue_size; j++)

  //  sum probabilities

  check_total_probability += queue[j];

 return check_total_probability;
}
//*****************************************************************************

void queue_c::build_cdf(void) {

 //  objective: build array which is the CDF, ie. Prob[q < b]

 int k = 0;

 //  calculating CDF (queue[0] = 0.0)

 for (register int j = 1; j < actual_queue_size; j++) {
  k = j - 1;
  queue[j] += queue[k];                              //  prob[q < b]
 }
}
//*****************************************************************************

void queue_c::build_one_minus_cdf(void) {

 //  objective: calculate prob[q >= b}

 for (register int j = 1; j < actual_queue_size; j++)
  queue[j] = 1.0 - queue[j];                         //  prob[q >= b]

 /*  always this value, resetting in case subtraction not exact (needs to be
     0.0 if using program join                                              */

 queue[actual_queue_size - 1] = 0.0;
}
//*****************************************************************************

void queue_c::log_queue(void) {

 /*  objective: get log of contents of array
     note: after running build_one_minus_cdf(), queue[actual_queue_size - 1]
           = 0.0 so the log of this is not got                              */
```

```
int actual_queue_size_minus_one = actual_queue_size - 1;

/*  due to truncation error (especially when using program join) queue[j]
    for large j may be negative, hence we get the absolute value
    this is not a very robust feature                                    */

for (register int j = 1; j < actual_queue_size_minus_one; j++) {
 queue[j] = log10(fabs(queue[j]));
}
}
//*************************************************************************

ostream &operator<<(ostream &stream,queue_c *queue) {

 //  output contents of array to stream

 int actual_size = queue->actual_queue_size;
 register int j = 1;

 cout << setiosflags(ios::showpoint | ios::fixed);

 while (j < actual_size) {
  stream << "\t" << j << "\t" << queue->queue[j] << endl;
  j++;
 }
 return stream;
}

//*************************************************************************

/*                                  * * * * * * * * * * * * *
        File:   mlcg32.cpp          *                       *
        Author: Tom Corcoran        *     class mlcg32_c    *
        Date:   20/07/93            *    member functions   *
                                    *                       *
        Outline:                    * * * * * * * * * * * * *

Member function random():

Multiplicative linear congruential random number generator as described
by Pierre L'Ecuyer in Communications of the ACM June 88. The version
implemented here is in 32 bits. Programmed by Enda Doyle.

Returns: Random number in range 0 - 1                                */

//*************************************************************************
```

```
// Includes                          //  defines

#include <string.h>                  //  strcpy()
#include <conio.h>                   //  clrscr()
#include <stdlib.h>                  //  exit()
#include <iostream.h>                //  ostream
#ifndef MLCG32_H
 #define MLCG32_H
 #include "mlcg32.h"                 //  class mlcg32_c
#endif
#ifndef OPENFILE_H
 #define OPENFILE_H                  //  modules for opening files
 #include "openfile.h"               //  <stdlib.h>,<fstream.h>
#endif

//*****************************************************************************

mlcg32_c::mlcg32_c(fstream *seed_f) {
 *seed_f >> seed1 >> seed2;
};
//*****************************************************************************

void mlcg32_c::reset_seeds(long int temp_seed1,long int temp_seed2) {

 //  objective: reset seeds, given seeds

 seed1 = temp_seed1;
 seed2 = temp_seed2;
}
//*****************************************************************************

void mlcg32_c::reset_seeds(ifstream *seed_f) {

 //  reset seeds, reading seeds from file

 *seed_f >> seed1 >> seed2;
}
//*****************************************************************************

double mlcg32_c::random(void) {

 //  objective: return a random number between 0 and 1

 long z,c;                           //  used to calculate random number
```

```
//  calculating random number

c  = seed1 % 53668L;
seed1 = 40014L * (seed1 - c * 53668L) - c * 12211L;
if (seed1 < 0)
 seed1 = seed1 + 2147483563L;

c  = seed2 % 52774L;
seed2 = 40692L * (seed2 - c * 52774L) - c * 3791L;
if (seed2 < 0)
 seed2 = seed2 + 2147483399L;

z = seed1 - seed2;
if (z < 1)
 z = z + 2147483562L;

return(z * 4.656613E-10);
}
//*****************************************************************************

int mlcg32_c::random_int(int max) {

 //  objective: returns a random number in [0,max)

 double ran_int = (double)max * random();

 //  note: (int)9.9 = 9  => [0,max)

 return((int)ran_int);
}
//*****************************************************************************

ostream &operator<<(ostream &stream,mlcg32_c *random) {

 stream << random->seed1 << "  " << random->seed2;
 return stream;
}

//*****************************************************************************

/*                                  * * * * * * * * * * * * * * *
        File:   timer.cpp           *                           *
        Author: Tom Corcoran        *     class stop_watch_c    *
        Date:   20/07/93            *      member functions     *
                                    *                           *
                                    * * * * * * * * * * * * * * *        */
```

```
//*****************************************************************************

//  includes                                //  defines

#include<iostream.h>                        //  ostream
#ifndef TIMER_H
 #define TIMER_H
 #include "timer.h"                         //  class stop_watch_c
#endif

//*****************************************************************************

stop_watch_c::stop_watch_c(void) {

 //  initializing private members

 hours = 0;
 minutes = 0;
 seconds = 0;
}
//*****************************************************************************

void stop_watch_c::diff_time(void) {

 /*  computes diference in time in seconds between start and end
     and converts this into hours, minutes, seconds                          */

 int time_left_in_sec;
 double diff_in_sec;

 //  difference in seconds between start and end

 diff_in_sec = difftime(end,start);

 //  number of hours (3600 seconds per hour)

 hours = int((diff_in_sec)/3600.0);

 //  total time in seconds minus hours in seconds

 time_left_in_sec = int(diff_in_sec) - (hours*3600);

 //  number of minutes (not including hours)

 minutes = time_left_in_sec/60;                    //  div operation
```

```
 //  number of seconds (not including hours,minutes)

 seconds = int(diff_in_sec) - (hours*3600) - (minutes*60);
}
//*****************************************************************************

void stop_watch_c::display_start_time_to_screen(void) {
 display_time_to_screen(start);
}
//*****************************************************************************

void stop_watch_c::display_end_time_to_screen(void) {
 display_time_to_screen(end);
}
//*****************************************************************************

void stop_watch_c::display_time_to_screen(time_t now) {

 //  tm is a structure defining the broken-down time

 struct tm *date_and_time;

 //  convert date and time to a structure

 date_and_time = localtime(&now);

 //  converts date_and_time to ASCII

 cout << asctime(date_and_time);
}
//*****************************************************************************

ostream &operator<<(ostream &stream,stop_watch_c *watch) {

 //  output simulation length (hrs:min:sec) to file

 stream << watch->hours << ":" << watch->minutes << ":" << watch->seconds;
 return stream;
}
```

```
//*************************************************************************

/*                                      * * * * * * * * * * * * *
        File:    charstr.cpp            *                       *
        Author:  Tom Corcoran           *     class string_c    *
        Date:    20/7/93                *    member functions   *
                                        *                       *
                                        * * * * * * * * * * * * *        */

//*************************************************************************

// includes                          //  defines

#include <iostream.h>                //  ostream
#include <string.h>                  //  strlen(),strcpy()
#include <stdlib.h>                  //  strlen(),strcpy()
#ifndef OPENFILE_H
 #define OPENFILE_H
 #include "openfile.h"               //  modules for opening files
#endif
#ifndef ROUTINES_H
 #define ROUTINES_H
 #include "routines.h"               //  module for overloaded operator
#endif
#ifndef CHARSTR_H
 #define CHARSTR_H
 #include "charstr.h"                //  class string_c
#endif

//*************************************************************************

string_c::string_c(char *temp_string_ptr) {

 //  calculating length of string (initializing string_length)

 string_length = strlen(temp_string_ptr);
 if (!string_length)
  general_error("attempted to setup empty string");

 /*  allocating space for character array
     new returns zero if allocation fails                                */

 if (!(string_ptr = new char[string_length + 1]))
  allocation_error("string of size ",string_length);

 //  initializing string_ptr
```

```
 strcpy(string_ptr,temp_string_ptr);
}
//*****************************************************************************

string_c::~string_c(void) {

 //   destroying object

 delete string_ptr;
}
//*****************************************************************************

ostream &operator<<(ostream &stream,string_c *string) {

 stream << string->string_ptr;
 return stream;
}

//*****************************************************************************

/*      File:   routines.cpp
        Author: Tom Corcoran
        Date:   23/10/93                                                    */

//*****************************************************************************

//  includes                        //  defines

#include <iostream.h>               //  ostream
#include <math.h>                   //  ceil,floor
#include <stdlib.h>                 //  exit()
#include <string.h>                 //  strcat()
#include <fstream.h>                //  ifstream
#ifndef ROUTINES_H
 #define ROUTINES_H
 #include "routines.h"              //  modules for setting ios flags
#endif
#ifndef CHARSTR_H
 #define CHARSTR_H
 #include "charstr.h"               //  class string_c
#endif

//*****************************************************************************

double round_double(double num) {
```

```
 double num_down = double(long(num));

 if (num >= num_down + 0.5)
  num = ceil(num);
 else
  num = floor(num);
 return num;
}
//*************************************************************************

int int_round_double(double num) {

 double ran_double = round_double(num);
 return (int(ran_double));
}
//*************************************************************************

double * setup_dynamic_array(int array_size) {

 double *new_array;

 //  allocating space for array of size array_size

 if (!(new_array = new double[array_size]))
  allocation_error("array of size ",array_size);
 return new_array;
}
//*************************************************************************

void initialize_dynamic_array(int array_size,double * new_array) {

 //  initializing arrays

 for (register int i = 0; i < array_size; i++)
  new_array[i] = 0.0;
}
//*************************************************************************

ostream &fixed_and_showpoint(ostream &stream) {

 stream.setf(ios::fixed | ios::showpoint);
 return stream;
}
//*************************************************************************
```

```
ostream &error(ostream &stream) {

 stream <<  "\nError: ";
 return stream;
}
//*****************************************************************************

void general_error(char *str_ptr) {

 string_c error_str(str_ptr);
 display_and_exit(error_str);
}
//*****************************************************************************

void allocation_error(char *str_ptr) {

 string_c head_str("allocating space for ");
 string_c error_str(str_ptr);
 strcat(head_str,error_str);
 display_and_exit(head_str);
}
//*****************************************************************************

void allocation_error(char *str_ptr,long int display_int) {

 string_c head_str("allocating space for ");
 string_c error_str(str_ptr);
 string_c display_int_str("     ");
 ltoa(display_int,display_int_str,10);        //  converts long int to string
 strcat(head_str,error_str);
 strcat(head_str,display_int_str);
 display_and_exit(head_str);
}
//*****************************************************************************

void display_and_exit(char *error_str) {

 //  displays string (memory already allocated) to screen and exits program

 cout << error << error_str;
 exit(1);
}
//*****************************************************************************
```

```
//*************************************************************************

/*      File:     openfile.cpp
        Author:   Tom Corcoran
        Date:     27/11/92

   ..   Outline: Reusable modules for opening files and checking for errors */

//*************************************************************************

// includes                          //  defines
   ..
#include <stdlib.h>                  //  exit()
#include <string.h>                  //  strcat()
#include <fstream.h>                 //  ifstream,ofstream,fstream
#ifndef OPENFILE_H
 #define OPENFILE_H
 #include "openfile.h"               //  header file
#endif
#ifndef CHARSTR_H
 #define CHARSTR_H
 #include "charstr.h"                //  class string_c
#endif
#ifndef ROUTINES_H
 #define ROUTINES_H
 #include "routines.h"               //  display_and_exit()
#endif

//*************************************************************************

//  overloaded file opening

//*************************************************************************

void open_file(char *file_name,ifstream *fp) {

 //  opening input only file

 fp->open(file_name);                //  opened for reading by default
 check_error(fp,file_name);
}
//*************************************************************************

void open_file(char *file_name,ofstream *fp) {

 //  opening output only file (overwrite)
```

```
 fp->open(file_name);                    //  opened for writing by default
 check_error(fp,file_name);
}
//*************************************************************************

void open_file_input(char *file_name,fstream *fp) {

 //  opening existing i/o file for input

 fp->open(file_name,ios::in|ios::nocreate);
 check_error(fp,file_name);
}
//*************************************************************************

void open_file_output(char *file_name,fstream *fp) {

 //  opening existing i/o file for output (overwrite)

 fp->open(file_name,ios::out|ios::nocreate);
 check_error(fp,file_name);
}
//*************************************************************************

void open_file_io(char *file_name,fstream *fp) {

 //  open i/o file for reading and writing

 fp->open(file_name,ios::in|ios::app);
 check_error(fp,file_name);
}
//*************************************************************************

void open_file_app(char *file_name,ofstream *fp) {

 //  opening existing i/o file for output (append)

 fp->open(file_name,ios::app);
 check_error(fp,file_name);
}
//*************************************************************************

//  overloaded error checks

//*************************************************************************
```

```
void check_error(ifstream *fp,char *file_name) {

 if (fp->fail())                    //  check if open failed
  exit_program(file_name);
}
//*****************************************************************************

void check_error(ofstream *fp,char *file_name) {

 if (!*fp)                          //  check if open failed
  exit_program(file_name);
}
//*****************************************************************************

void check_error(fstream *fp,char *file_name) {

 if (!*fp)                          //  check if open failed
  exit_program(file_name);
}
//*****************************************************************************

void exit_program(char *file_name) {

 //  exiting to dos when error found

 string_c error_str("opening file: ");
 strcat(error_str,file_name);
 display_and_exit(error_str);
}
//*****************************************************************************
```

C.3 Main program

```
//*****************************************************************************

/*  File:    oover16.cpp
    Author:  Tom Corcoran
    Date:    24/6/93
    Version: 8.0

    Outline:

    Simulation of a cell level model of multiplexed packetized voice traffic.

    The program uses the classes:
```

prob_c	-	probabilities, alpha and beta (function of l, s, rho)
mlcg32_c	-	random number generator
line_c	-	node of linked list (a line)
line_list_c	-	linked list (superposition of lines)
queue_c	-	buffer array
stop_watch_c	-	times simulation length
string_c	-	stores file names

They are included in the following files:

prob.h,mlcg32.h,node.h,list.h,queue.h,timer.h and charstr.h respectively.

There are four input only files:

Names file: User generated. Contains the names of the following files:

- input file
- summary file
- seeds file

Input file: User generated. Contains the following input parameters:

l	-	number of lines
s	-	number of ticks
rho	-	load
phi	-	non zero value sets l(1 - beta)
sim_run	-	length of simulation in blocks (s ticks per block)
max_queue_size	-	maximum length of buffer
initial_queue	-	initial length of queue
last_process_time	-	time list was last processed
initial_flag	-	flag indicating if initial conditions are to be inputed from file
seed_flag	-	flag indicating if seeds are to inputed from file
out_f_name	-	name of output file
in_init_f_name	-	name of initial conditions file
seed_f_name	-	name of seed file
sys_state_f_name	-	name of last system state file

A set of parameters is included for every simulation to be run.

Note: initial_queue and last_process_time are set to non

zero if a previous simulation is being continued

Initial conditions file:

User/program generated. Specifies initial state of system. Can be set to be state of lines file, which is system generated, if trying to improve initial conditions.

Seed file (in):

User generated. Specifies the starting seeds for each simulation (if seed_flag is on).

Two output (only) files are created:

Output file:

System generated. Contains the ouput empirical buffer queue length distribution, ie. Log Prob(queue >= buffer), for all values of buffer. A graph can then be drawn from this data.

Summary file:

System generated. Contains a summary of the simulation run. Displays name of initial conditions file (out). For each simulation the above input parameters are displayed as well as the simulation length, prob(queue >= max_queue_size), percentage of blocks for which queue not empty, theoritical and actual number of samples used to build queue, seed values for random number generator prior to simulation run.

Last state of system file: System generated.

The last state of the system at the end of the simulation is written to this file. A file is created for each simulation with each name given in the input file. By naming this file as the initial conditions file it can be used as the initial state in another simulation. If this is done and the last size of the queue is also entered (non zero) in the input file then the simulation for which these conditions were saved will in effect be rerun.

The following is an i/o file:

Seed file: User/system generated.

Contains the start seed values for the random number generator used

```
        by the simulation. It can be used so as to rerun the previous
        simulation only (if wished to test how different parameters effect
        results, independent of random numbers). At the very end the final
        seeds used are written out to it (these seeds are the starting
        seeds for the next batch of simulations).                          */

//*****************************************************************************

//  includes                        //  defines


#include <iostream.h>               //  ios
#include <fstream.h>                //  ofstream,ifstream
#include <string.h>                 //  strcpy(),strcmp()
#include <conio.h>                  //  clrscr()
#include <time.h>                   //  struct time_t
#include <dos.h>                    //  gettime(),getdate()
#include <math.h>                   //  log10()
#include <iomanip.h>                //  setw(),(re)setprecision(),setiosflags()
#ifndef QUEUE_H
 #define QUEUE_H
 #include "queue.h"                 //  class queue_c
#endif
#ifndef PROB_H
 #define PROB_H
 #include "prob.h"                  //  class probabilities_c
#endif
#ifndef NODE_H
 #define NODE_H
 #include "node.h"                  //  class line_c
#endif
#ifndef LIST_H
 #define LIST_H
 #include "list.h"                  //  class line_list_c
#endif
#ifndef MLCG32_H
 #define MLCG32_H
 #include "mlcg32.h"                //  class mlcg32_c
#endif
#ifndef TIMER_H
 #define TIMER_H
 #include "timer.h"                 //  class stop_watch_c
#endif
#ifndef CHARSTR_H
 #define CHARSTR_H
 #include "charstr.h"               //  class string_c
```

```
#endif
#ifndef OPENFILE_H
 #define OPENFILE_H
 #include "openfile.h"              //  modules for opening files
#endif
#ifndef ROUTINES_H
 #define ROUTINES_H
 #include "routines.h"              //  frequently used modules
#endif

//*************************************************************************

//  function prototypes

void main(void);
void control_simulation(ifstream *in_f,ofstream *out_f,fstream *start_seed_f,
                        ofstream *summary_f,ofstream *sys_state_f,
                        char *in_f_name,char *summary_f_name,
                        char *start_seed_f_name,mlcg32_c *random_gen);
probabilities_c * setup_probabilities_class(int no_lines,int no_ticks,
                                            double load,double phi);
void resetting_seeds(mlcg32_c *random,char *f_name);
void control_list_setup(int initial_flag,line_list_c *list,
                        probabilities_c *prob,mlcg32_c *random,
                        char *initial_f_name);
void setup_list_from_scratch(line_list_c *list,mlcg32_c *random_gen,
                             probabilities_c *prob);
void setup_list_from_file(line_list_c *list,ifstream *in_init_f,
                          char *in_init_f_name,probabilities_c *prob);
void overide_last_values(int initial_flag,int last_sim_last_buffer,
                         int last_sim_last_process_time,
                         int &temp_initial_queue,
                         int &temp_last_process_time);
void generate_statistics(queue_c *queue,ofstream *out_f,double &prob_max,
                         long int &num_samples);
void writing_initial_seeds(mlcg32_c *random,fstream *seed_f,
                           char *seed_f_name);
void write_before_summary(ofstream *summary_f,mlcg32_c *random,double phi,
                          long int sim_run,int initial_queue,
                          int last_process_time,int initial_flag,
                          int seed_flag,int sim_no,char *in_init_f_name);
void write_after_summary(ofstream *summary_f,probabilities_c *prob,
                         double prob_q_ge_b_max,queue_c *queue,
                         long int count_empty_blocks,long int sum_num_samples,
                         char *out_f_name,char *sys_state_f_name,
                         int final_buffer,int last_process_time,
```

```
                          long int sim_run,stop_watch_c *watch);
void initialize_char_arrays(char *temp_out_f_name,char *temp_in_init_f_name,
                            char *temp_seed_f_name,
                            char *temp_sys_state_f_name);
void display_screen(long int sim_run,int sim_no,stop_watch_c *watcher);
void start_screen(void);
void sim_screen(void);
void check_input(int sim_no,int no_lines,int no_ticks,double load,
                 double phi,int initial_queue,int last_process_time,
                 int initial_flag,int seed_flag,char *in_f_name,
                 char *in_init_f_name,char *seed_f_name,
                 char *sys_state_f_name,char *last_sys_state_f_name);
void check_flag(int flag,char *f_name);

//*****************************************************************************

void main(void) {

 //  temporary pointers to names of files

 char temp_names_f_name[32];              //  names file
 char temp_in_f_name[32];                 //  input file
 char temp_summary_f_name[32];            //  summary file
 char temp_start_seed_f_name[32];         //  starting seed file

 mlcg32_c *random_gen = NULL;             //  pointer to class mlcg32_c

 ifstream names_f;                        //  stream for names file

 //  pointer to streams for files

 ifstream *in_f = NULL;                   //  input file
 ofstream *summary_f = NULL;              //  summary file
 fstream *start_seed_f = NULL;            //  starting seed file
 ofstream *out_f = NULL;                  //  output file
 ofstream *sys_state_f = NULL;            //  pointer to state of system file

 start_screen();                          //  calls initial screen

 //  input file name entered by user

 cin >> temp_names_f_name;
 string_c names_f_str(temp_names_f_name); //  string_c class for names file

 //  opening names file,reading file names and closing file
```

```
open_file(names_f_str,&names_f);
names_f >> temp_in_f_name >> temp_summary_f_name >> temp_start_seed_f_name;
names_f.close();

//  string_c calls for each file name; setting up file names

string_c in_f_str(temp_in_f_name);
string_c summary_f_str(temp_summary_f_name);
string_c start_seed_f_str(temp_start_seed_f_name);

//  creating new objects

string_c error_header_str("pointer to stream for: ");
if (!(in_f = new ifstream))
 allocation_error(strcat(error_header_str,in_f_str));
if (!(summary_f = new ofstream))
 allocation_error(strcat(error_header_str,summary_f_str));
if (!(start_seed_f = new fstream))
 allocation_error(strcat(error_header_str,start_seed_f_str));

//  pointers created but names not known in this module

if (!(out_f = new ofstream))
 allocation_error(strcat(error_header_str,"output file"));
if (!(sys_state_f = new ofstream))
 allocation_error(strcat(error_header_str,"initial state of system file"));

//  opening starting_seed file,creating object(setting seeds),closing file

open_file_input(start_seed_f_str,start_seed_f);
if (!(random_gen = new mlcg32_c(start_seed_f)))
 allocation_error("instance of class mlcg32_c");
start_seed_f->close();

//  opening input file (other files must be opened for each simulation)

open_file(in_f_str,in_f);

//  open/header/close summary file to keep track of simulations

open_file(summary_f_str,summary_f);
*summary_f << "File: " << &summary_f_str << "\n\n";
summary_f->close();

//  perform simulations
```

```
control_simulation(in_f,out_f,start_seed_f,summary_f,sys_state_f,
                   in_f_str,summary_f_str,start_seed_f_str,random_gen);

/*  writing last seeds used to seed file and summary file
    (mlcg32_c inserter function)                                        */

open_file_app(summary_f_str,summary_f);
*summary_f << "Final seeds are: ";
*summary_f << random_gen;
summary_f->close();

open_file_output(start_seed_f_str,start_seed_f);
*start_seed_f << random_gen;
start_seed_f->close();

//  closing input file after simulations finished

in_f->close();

/*  displaying summary file and state of list file names on screen
    (string_c inserter function)                                        */

cout << "\n\t\t\tSee " << &summary_f_str
     << "\n\t\t\tfor summary of simulations\n";

//  destroying objects

delete in_f;
delete summary_f;
delete start_seed_f;
delete sys_state_f;
delete out_f;
delete random_gen;
}
//*************************************************************************

void control_simulation(ifstream *in_f,ofstream *out_f,fstream *start_seed_f,
                        ofstream *summary_f,ofstream *sys_state_f,
                        char *in_f_name,char *summary_f_name,
                        char *start_seed_f_name,mlcg32_c *random_gen) {

 int empty_queue = 0;                  //  empty queue
 int sim_no = 0;                       //  number of simulation
 int temp_max_queue_size = 0;          //  inputed max length of buffer array
 int temp_initial_queue;               //  inputed initial queue length
 int temp_last_process_time;           //  inputed last process time
```

```
int temp_l;                              //  value of l inputed from file
int temp_s;                              //  value of s inputed from file
int initial_flag;                        //  initial flag inputed from file
int seed_flag;                           //  seed flag inputed from file
int last_sim_last_buffer = 0;            //  last queue length
int last_sim_last_process_time = 0;      //  last process time of simulation
long int count_empty_blocks = 0;         //  number of empty blocks in simulation
long int sim_run;                        //  length of simulation in blocks
long int sum_num_samples;                //  total number of additions to queue
double temp_rho;                         //  value of rho inputed from file
double prob_q_ge_b_max = 0.0;            //  overflow probability
double temp_phi;                         //  determines l(1 - beta)

//  temporary file names

char temp_out_f_name[32];                //  output file
char temp_in_init_f_name[32];            //  initial system state file
char temp_seed_f_name[32];               //  seed file
char temp_sys_state_f_name[32];          //  last state of system file

char last_sys_state_f_name[32];          //  used to save file name

//  pointer to classes

line_list_c *this_list = NULL;           //  class line_list_c
probabilities_c *prob = NULL;            //  class probabilities_c
queue_c *queue = NULL;                   //  class queue_c
stop_watch_c *watch = NULL;              //  class stop_watch_c

/*  reading parameters for first simulation (non zero initial_flag indicates
    that initial conditions are to be read from file, non zero seed_flag
    indicates that seeds are to be read from file)                          */

initialize_char_arrays(temp_out_f_name,temp_in_init_f_name,temp_seed_f_name,
temp_sys_state_f_name);

*in_f >> temp_l >> temp_s >> temp_rho >> temp_phi >> sim_run
      >> temp_max_queue_size >> temp_initial_queue >> temp_last_process_time
      >> initial_flag >> seed_flag >> temp_out_f_name >> temp_in_init_f_name
      >> temp_seed_f_name >> temp_sys_state_f_name;

//  loop runs until end of input file; one cycle for each simulation

while (!in_f->eof()) {

 //  string_c class for each file name; setting up file names
```

```
string_c out_f_str(temp_out_f_name);
string_c in_init_f_str(temp_in_init_f_name);
string_c seed_f_str(temp_seed_f_name);
string_c sys_state_f_str(temp_sys_state_f_name);

// keeps track of simulation number

sim_no++;

// checking input for error

check_input(sim_no,temp_l,temp_s,temp_rho,temp_phi,temp_initial_queue,
            temp_last_process_time,initial_flag,seed_flag,in_f_name,
            in_init_f_str,seed_f_str,sys_state_f_str,last_sys_state_f_name);

// simulation start time => length of simulation can be calculated

if (!(watch = new stop_watch_c()))
 allocation_error("instance of class stop_watch_c");
watch->set_start_time(time(NULL));

// output screen displayed while simulation runs

display_screen(sim_run,sim_no,watch);

/* if the seed_flag is 1 then the simulations will run starting with the
   seeds in seed_f_name.Otherwise the simulations will begin with the seeds
   given in start_seeds_f_name which is what the seeds are initialized to*/

if (seed_flag)
 resetting_seeds(random_gen,seed_f_str);

/* write initial seeds to start seed file (so next simulation can use
   current simulation's starting seeds, by this means one long simulation
   can be monitored at regular intervals)                                */

writing_initial_seeds(random_gen,start_seed_f,start_seed_f_name);

/* check to see if temp_initial_queue and temp_last_process_time need
   to be set to values saved from last simulation                        */

overide_last_values(initial_flag,last_sim_last_buffer,
                    last_sim_last_process_time,temp_initial_queue,
                    temp_last_process_time);
```

```
/*  opening summary file (for each simulation,so if terminated data
                          for previous simulations isn't lost)
    writing starting simulation statistics to file                          */

open_file_app(summary_f_name,summary_f);
write_before_summary(summary_f,random_gen,temp_phi,sim_run,
                     temp_initial_queue,temp_last_process_time,
                     initial_flag,seed_flag,sim_no,in_init_f_str);

/*  creating new objects for this_list,queue and prob
    creating this_list also connects head and foot of list                  */

if (!(this_list = new line_list_c(temp_last_process_time)))
 allocation_error("instance of class line_list_c");
if (!(queue = new queue_c(temp_max_queue_size,temp_initial_queue)))
 allocation_error("instance of class queue_c");
prob = setup_probabilities_class(temp_l,temp_s,temp_rho,temp_phi);

/*  checking if parameters pass all conditions, exits program if any
    inconsistency is found                                                  */

prob->check_conditions();

//  setting up linked list

control_list_setup(initial_flag,this_list,prob,random_gen,in_init_f_str);

//  running simulation for sim_run number of blocks

for (register long int i = 1; i <= sim_run; i++) {

 //  runs simulation for one block

 this_list->one_block_simulation(empty_queue,queue,prob,random_gen);

 if (empty_queue) {
  count_empty_blocks++;                    //  counts no of empty blocks
  empty_queue = 0;                         //  resets empty_queue
 }
}

//  generate statistics and write to output file (sets prob_q_ge_b_max)

open_file(out_f_str,out_f);
generate_statistics(queue,out_f,prob_q_ge_b_max,sum_num_samples);
out_f->close();
```

```
//  simulation end time (gets system time in seconds from...)

watch->set_end_time(time(NULL));

//  writing message to screen (string_c inserter function)

cout << "\n\n\t\t\tSee " << &out_f_str << " for results\n\a";

/*  saving last queue length,last process time and sys_state_f_str
    in case next simulation is a continuation of this one                */

last_sim_last_buffer = queue->return_buffer();
last_sim_last_process_time = this_list->return_last_process_time();
strcpy(last_sys_state_f_name,sys_state_f_str);

//  writing statistics to summary file

write_after_summary(summary_f,prob,prob_q_ge_b_max,queue,
                    count_empty_blocks,sum_num_samples,out_f_str,
                    sys_state_f_str,last_sim_last_buffer,
                    last_sim_last_process_time,sim_run,watch);
summary_f->close();

/*  outputs state of linked list after each simulation (line_list_c
    inserter function); overwritten each time (can be used for breaking
    one long simulation into a series of shorter ones)                   */

open_file(sys_state_f_str,sys_state_f);
*sys_state_f << this_list;
sys_state_f->close();

//  deleting objects (also calls destructers)

delete this_list;
delete prob;
delete queue;
delete watch;

//  reinitialising variables for subsequent running

temp_l = temp_s = temp_max_queue_size = 0;
count_empty_blocks = sum_num_samples = sim_run = 0;
temp_phi = temp_rho = prob_q_ge_b_max = 0;
initialize_char_arrays(temp_out_f_name,temp_in_init_f_name,temp_seed_f_name,
                       temp_sys_state_f_name);
```

```
  //  reading in values for next simulation

  *in_f >> temp_l >> temp_s >> temp_rho >> temp_phi >> sim_run
        >> temp_max_queue_size >> temp_initial_queue >> temp_last_process_time
        >> initial_flag >> seed_flag >> temp_out_f_name >> temp_in_init_f_name
        >> temp_seed_f_name >> temp_sys_state_f_name;

 } //  end while loop
}
//*****************************************************************************

probabilities_c * setup_probabilities_class(int no_lines,int no_ticks,
                                            double load,double phi) {

 //  objective: create probabilities object

 probabilities_c *prob = NULL;                    //  pointer to class
                                                  //  probabilities_c

 if (!phi) {                                      //  l,s,rho all given
  if (!(prob = new probabilities_c
                  (no_lines,no_ticks,load)))
   allocation_error
    ("instance of class probabilities_c");

  /*  fix value of alpha for simulation to default, beta is then worked out
      as a function of beta,l,s,rho
      gamma = p[silence < s ticks] and is a function of beta                */

  prob->fix_alpha();
  prob->set_gamma();
 }
 else {

  //  setting traffic constant (either s or rho not given)

  if (!no_ticks) {                                //  rho given (s not given)
   if (!load)                                     //  s given (rho not given)
    general_error("both s and rho cannot"
                  " be zero\n");                  //  s = rho = zero
   else
    if (!(prob = new probabilities_c
     (phi,no_lines,load)))
     allocation_error
      ("instance of class probabilities_c");
```

```
 }
 else {
  if (load)                                          //  s given (rho not given)
   general_error("since phi != 0, then either"
                 " s = 0 or rho = 0\n");             //  s = rho != zero
  else
   if (!(prob = new probabilities_c
                    (phi,no_lines,no_ticks)))
    allocation_error
     ("instance of class probabilities_c");
 }
} //                                                 //  end main else
 return prob;
}
//*************************************************************************

void resetting_seeds(mlcg32_c *random,char *f_name) {

 ifstream *seed_f = NULL;                           //  pointer to file
 if (!(seed_f = new ifstream))                      //  creating new object
  allocation_error
   ("stream for resetting seeds file");
 open_file(f_name,seed_f);                          //  opening file
 random->reset_seeds(seed_f);                       //  reseting seeds
 seed_f->close();                                   //  closing file
 delete seed_f;                                     //  destroying object
}

//*************************************************************************

void control_list_setup(int initial_flag,line_list_c *list,
                        probabilities_c *prob,mlcg32_c *random,
                        char *initial_f_name) {

 if (initial_flag) {
  ifstream *initial_f = NULL;                       //  pointer to initial file
  if (!(initial_f = new ifstream))                  //  creating new object
   allocation_error
    ("stream for initial state file");
  open_file(initial_f_name,
            initial_f);                             //  opening initial file
  setup_list_from_file(list,initial_f,
                       initial_f_name,prob);
  initial_f->close();                               //  closing initial file
  delete initial_f;                                 //  destroying object
 }
```

```
 else {
  setup_list_from_scratch(list,random,prob);

  //  sort linked list in order of firing time

  list->sort_linked_list(prob->return_no_ticks());
 }
}
//*****************************************************************************

void setup_list_from_scratch(line_list_c *list,mlcg32_c *random,
                             probabilities_c *prob) {

 //  objective: to build and initialize linked list

 int firing_flag;
 int tick_no;
 int no_lines = prob->return_no_lines();
 int no_ticks = prob->return_no_ticks();

 /*  mean lines active per block = p*s
     prob(line active) = (mean lines active per block)/l                    */

 double prob_line_active = prob->return_prob_line_active();
 double random_a;
 line_c *node1 = NULL;
 line_c *node2 = list->return_head();

 //  while loop runs for the number of lines

 register int i = 1;
 while (i <= no_lines) {

  /*  deciding whether node fires/not fires by sampling the probabilty
      distribution of line being active                                     */

  random_a = random->random();
  if (random_a <= prob_line_active)
   firing_flag = 1;
  else
   firing_flag = 0;

  /*  generate random time to find firing time of node in list
      returns random numner in [0,no_ticks)                                 */

  tick_no = random->random_int(no_ticks) + 1;
```

```
  //  creating new node

  if (!(node1 = new line_c(i,tick_no,firing_flag)))
   allocation_error("instance of class line_c for line ",i);

  /*  insert node i in list (linking node i to node (i - 1) and node (i + 1))
      at the start head will be connected to foot and node2 = head          */

  list->insert_mid_after(node1,node2);

  node2 = node1;                          //  node2 is the last node inserted
  i++;                                    //  incrementing counter
 }
}
//*****************************************************************************

void setup_list_from_file(line_list_c *list,ifstream *in_init_f,
                          char *in_init_f_name,probabilities_c *prob) {

 //  objective: to build and initialize linked list from given input file

 int no_lines = prob->return_no_lines();
 int no_ticks = prob->return_no_ticks();
 int count_lines = 0;
 int firing_flag;
 int line_number;
 int firing_time;
 line_c *node1 = NULL;
 line_c *node2 = list->return_head();

 //  reading parameters for first node and verifying input

 *in_init_f >> line_number >> firing_time >> firing_flag;

 //  while loop runs until end of input file

 while (!in_init_f->eof()) {

  count_lines++;

  //  firing_flag = 0 or 1 only

  check_flag(firing_flag,in_init_f_name);

  //  creating new node
```

```
 if (!(node1 = new line_c(line_number,firing_time,firing_flag)))
  allocation_error("instance of class line_c for line ",line_number);

 /*  insert node i in list (linking node i to node (i - 1) and node (i + 1))
     at the start head will be connected to foot and node2 = head          */

 list->insert_mid_after(node1,node2);

 //  enables buliding of list: node2 is the last node inserted

 node2 = node1;

 //  reading parameters for first node and verifying input

 *in_init_f >> line_number >> firing_time >> firing_flag;

}                                                  //  end while loop

/*  the number of lines inputed from the file must match the given
     number of lines                                                       */

if (count_lines != no_lines)
 general_error("the initial system state does not match the given"
               "\n\tnumber of lines");

/*  the last firing time inputed from the file cannot be greater than the
    the number of ticks given                                              */

if (firing_time > no_ticks)
 general_error("the firing time of the initial system state does"
               "\n\tnot match the given number of ticks");
}
//*************************************************************************

void overide_last_values(int initial_flag,int last_sim_last_buffer,
                         int last_sim_last_process_time,
                         int &temp_initial_queue,
                         int &temp_last_process_time) {

 //  initial_flag = 1 => initial system state given

 if (initial_flag) {

 /* if temp_last_process_time and temp_initial_queue are zero then this means
    that the current simulation is a continuation of the previous simulation*/
```

```
  if ((!temp_initial_queue)&(!temp_last_process_time)) {

   /*  the parameters must be reset to the last values saved from the
       previous simulation                                              */

   temp_initial_queue = last_sim_last_buffer;
   temp_last_process_time = last_sim_last_process_time;
  }
 }
}
//*****************************************************************************

void line_list_c::one_block_simulation(int &empty_queue,queue_c *queue,
                                       probabilities_c *prob,
                                       mlcg32_c *random_gen) {

 /*  Objective: carries out simulation for one block of s ticks

     Data Dictionary:

      queue:        pointer to object of class queue_c
      prob:         pointer to object of class probabilities_c
      random_gen:   pointer to object of class mlcg32_c
      empty_queue:  if non zero indicates buffer was empty during block     */

 enum new_line_fire_type {nofire,fire};  //  defining type

 new_line_fire_type
  new_line_fire = nofire;                //  flag for line to be moved

 int count_firing = 0;                   //  number of active lines per block
 int rel_time;                           //  time since last packet processing
 double random_p,random_q;               //  random numbers
 line_c *line_ptr = head->next;          //  simulation node (cycles list)
 line_c *line_to_insert = NULL;          //  node to be moved in list
 line_c *save_pos = NULL;                //  position in list prior to a move

 while (line_ptr != foot) {

  //  checks line firing status (firing/not firing)

  if (line_ptr->firing_flag) {
   count_firing++;

   //  calculating time elapsed since last active line
```

```
if (line_ptr->firing_time >= last_process_time)
 rel_time = line_ptr->firing_time - last_process_time;
else
 rel_time = line_ptr->firing_time  + (prob->s - last_process_time);

/*  processing packets: taking time elapsed since last active line
    (last time packets processed) from buffer as a packet can be
    processed at each tick                                          */

if (rel_time) queue->buffer -= rel_time;
if (queue->buffer < 0) queue->buffer = 0;

//  buffer was empty at some stage in block if empty_queue is non zero

if (!queue->buffer) empty_queue++;

/*  incrementing buffer and gathering simulation statistics
    if the buffer overflows => count is kept of the lost cells in the
    last array element                                              */

queue->buffer++;
if (queue->buffer < queue->overflow_size)
 queue->queue[queue->buffer]++;
else
 queue->queue[queue->overflow_size]++;

//  is line going to stop firing? (test and update firing status)

random_p = random_gen->random();
if (random_p > prob->alpha) {
 line_ptr->firing_flag = 0;
}

//  line_ptr->firing time is now "last time" packets were processed

last_process_time = line_ptr->firing_time;
}
else {

/*  check if line is going to start firing; line will start firing
    at some point in the next s ticks if q <= gamma                 */

random_q = random_gen->random();
if (random_q <= prob->gamma) {
 line_ptr->firing_flag = 1;
```

```
    new_line_fire = fire;
    line_to_insert = line_ptr;

    /*  necessary here as insert will reorder pointers (firing_time)
        don't want it to effect progression through loop                    */

    save_pos = line_ptr->next;

    //  move line in linked list

    move_line(line_to_insert,random_gen,prob->s);
   }
  } //  else

  //  moving to next node in list

  if (!new_line_fire)
   line_ptr = line_ptr->next;
  else {                                  //  node has just been moved
   line_ptr = save_pos;
   new_line_fire = nofire;
  }
 }

 /*  keep track of the number of samples (lines active)
     used to normalize array of queue lengths                                */

 queue->add_to_total_activity(count_firing);
}
//*************************************************************************

void writing_initial_seeds(mlcg32_c *random,fstream *seed_f,
                           char *seed_f_name) {

 /*  write starting simulation seeds to file so can be reused in
     next simulation (only - as overwritten for each simulation time)       */

 open_file_output(seed_f_name,seed_f);
 *seed_f << random;
 seed_f->close();
}
//*************************************************************************

void write_before_summary(ofstream *summary_f,mlcg32_c *random,double phi,
                          long int sim_run,int initial_queue,
                          int last_process_time,int initial_flag,
```

```
                        int seed_flag,int sim_no,char *in_init_f_name) {

 //  writing simulation number and output file name to file

 *summary_f << "\nSIMULATION NUMBER " << sim_no << "\n\n";

 /* describing (in summary_f_name) how system initialized,3 possibilities:
 - (a) initial conditions random
 - (b) initial conditions random (seeds reset from file)
 - (c) initial conditions given in file by user/program
              (produced by previous simulation)                          */

 //  initial_flag and seed_flag are 0 or 1

 if (!initial_flag) {
  *summary_f << "Initial conditions random";                  //  {a}
  if (seed_flag)
   *summary_f  << " (seeds reset from file)";                 //  {b}
 }
 else {
  *summary_f << "Initial conditions given by the system "     //  {c}
      << "/user, see " << in_init_f_name;
 }

 //  writing starting simulation seeds to summary file                  */

 *summary_f << "\nStarting seeds are: ";
 *summary_f << random;

 // writing starting queue length and last process time to summary file

 if (initial_flag)
   *summary_f << "\nStarting queue length = " << initial_queue
              << "  Starting last process time = " << last_process_time;

 *summary_f << "\nSimulation length in blocks = " << sim_run;

 //  writing starting phi to summary file

 if (phi)
  *summary_f << "\nStarting phi = " << fixed_and_showpoint
             << setprecision(10) << phi;
}
//*************************************************************************

void generate_statistics(queue_c *queue,ofstream *out_f,double &prob_max,
```

```
                                long int &num_samples) {

  int max_array = queue->return_max_queue_size();

  //  examines array to see how long it is

  queue->set_actual_queue_size();
  int actual_array = queue->return_actual_queue_size();

  /*  generating data to graph
      working out the probaility density function and then the
      cummulative density function (and getting log of it)                    */

  queue->setup_pdf();                    //  calculating probability distribution

  /*  returns overflow probabilty - otherwise prob_max = 0.0 (initialized)
      must be returned before queue->setup_pdf()                              */

  if (actual_array == max_array)
   prob_max = queue->return_queue(max_array - 1);

  queue->build_cdf();                    //  calculating cdf, p[q < b]
  queue->build_one_minus_cdf();          //  calculating p[q >= b]
  queue->log_queue();                    //  log10[p[q >= b]]

  //  write simulation statistics to file (queue_c inserter function)

  *out_f << queue;

  /*  finding out number of samples used to build buffer (equal total number
      of active lines come across in simulation)                              */

  num_samples = queue->return_total_activity();
}
//*************************************************************************

void write_after_summary(ofstream *summary_f,probabilities_c *prob,
                         double prob_q_ge_b_max,queue_c *queue,
                         long int count_empty_blocks,long int sum_num_samples,
                         char *out_f_name,char *sys_state_f_name,
                         int final_buffer,int last_process_time,
                         long int sim_run,stop_watch_c *watch) {

  int max_buffer = queue->return_max_queue_size() - 1;
  int actual_buffer = queue->return_actual_queue_size();
  int num_ticks = prob->return_no_ticks();
```

```
double load = prob->return_load();
double per_cent_blocks_not_empty;
long int theo_samples = long(double(sim_run)*double(num_ticks)*load);

per_cent_blocks_not_empty =
 (1.0 - (double(count_empty_blocks)/double(sim_run)))*100.0;

//  calculate simulation length

watch->diff_time();

/* writing statistics to summary file  (probabilities_c and stop_watch_c
    inserter functions)                                                  */

*summary_f << prob
           << "Time taken for simulation = " << watch
           << "\nGiven max buffer = " << max_buffer
           << "  Actual max buffer = " << (actual_buffer - 1);

if (prob_q_ge_b_max)
 *summary_f << "\nP[q >= " << max_buffer << "] = " << prob_q_ge_b_max;

*summary_f << "\nTheoretical samples = " << theo_samples
           << "  Actual samples = " << sum_num_samples
           << "\nLast state of system: " << sys_state_f_name
           << "\nLast queue length = " << final_buffer
           << "  Last process time = " << last_process_time
           << "\nBlocks not empty = " << setprecision(2)
           << per_cent_blocks_not_empty << " %"
           << "\nResults: " << out_f_name << "\n\n";
}
//*************************************************************************

void initialize_char_arrays(char *temp_out_f_name,char *temp_in_init_f_name,
                            char *temp_seed_f_name,
                            char *temp_sys_state_f_name) {

 strcpy(temp_out_f_name,"null");
 strcpy(temp_in_init_f_name,"null");
 strcpy(temp_seed_f_name,"null");
 strcpy(temp_sys_state_f_name,"null");
}
//*************************************************************************

void check_input(int sim_no,int no_lines,int no_ticks,double load,
                double phi,int initial_queue,int last_process_time,
```

```
                int initial_flag,int seed_flag,char *in_f_name,
                char *in_init_f_name,char *seed_f_name,
                char *sys_state_f_name,char *last_sys_state_f_name) {

// initial_flag and seed_flag can be only 0 or 1

check_flag(initial_flag,in_f_name);
check_flag(seed_flag,in_f_name);

// if phi = 0 => l,s,rho can't be 0

if (!phi) {
 if (!no_lines)
  general_error("number of lines = 0\n");
 if (!no_ticks)
  general_error("number of ticks = 0\n");
 if (!load)
  general_error("load = 0.0\n");
}

/* last_process_time element of [0,no_ticks]
   if s = 0 => s still to be calculated => check not relevant          */

if ((initial_flag)&(no_ticks)) {
 if ((last_process_time < 0)|(last_process_time > no_ticks)) {
  cout << error << "last process time = " << last_process_time
       << "\n\t(last process time is element of [0," << no_ticks << "] )\n";
  exit(1);
 }
}

/* if initial_flag = 1, then last_process time can not be zero if it is
   the first simulation of the batch                                   */

if (sim_no == 1) {
 if (initial_flag)
  if (!last_process_time)
   general_error("initial system state given for first simulation but"
                 "\n\ta value for the last process time was not given\n");
}
else {
 if (initial_flag) {
  if (!last_process_time) {
   int ptr0 = strcmp(in_init_f_name,last_sys_state_f_name);

   // equality => no error
```

```
   if (ptr0)
    general_error("name for initial conditions file must be the same as"
                  "\n\tthe last state of system file for the previous "
                  "simulation,\n\tsince the initial flag = 1 and the "
                  "last process time = 0\n");
  }
 }
} //  end else

/*  initial_queue can be zero but if it is non zero then a
    last_process_time must also be given                                  */

if (initial_queue < 0)
 general_error("initial queue length less than zero\n");

if (initial_queue) {
 if (!last_process_time)
  general_error("an initial queue length was given so a last "
                "\n\tprocess time must also be given\n");
}

/*  note: if null/file name is not included in the file an error
          check is not possible and an error will occur                   */

if (!initial_flag) {

 //  strcmp() returns zero if equal; equality => no error

 int ptr1 = strcmp(in_init_f_name,"null");
 if (ptr1) {
  cout << error << "name for initial conditions file not included in "
                   "file:\n\t" << in_f_name <<
                   "\n\t(check initial conditions flag, if = 0 => "
                   "write 'null' for above name)\n";
  exit(1);
 }
}

if (!seed_flag) {

 //  equality => no error

 int ptr3 = strcmp(seed_f_name,"null");
 if (ptr3) {
  cout << error << "name for seed file not included in"
```

```
                    "file:\n\t" << in_f_name <<
                    "\n\t(check reset seeds flag, if = 0 => "
                    "write 'null' for above name)\n";
   exit(1);
  }
 }

 //  equality => error

 int ptr4 = strcmp(sys_state_f_name,"null");
 if (!ptr4) {
  cout << error << "name for final system state file not included in"
       << "file:\n\t" << in_f_name << endl;
  exit(1);
 }
}
//*****************************************************************************

void check_flag(int flag,char *f_name) {

 if ((flag != 0)&&(flag != 1)) {
  cout << error << "in file: " << f_name << "\n\t(all flags must be 0 or 1)";
  exit(1);
 }
}
//*****************************************************************************

void display_screen(long int sim_run,int sim_no,stop_watch_c *watcher) {

 //  want new screen for each simulation

 sim_screen();

 //  display additional info to screen

 cout << "\nSimulation " << sim_no << "\t\tThe Simulation started at: ";
 watcher->display_start_time_to_screen();
 cout << "\n\t\t\tSimulation Run in blocks = " << sim_run << endl
      << "\n\t\t\tIn Progress ........";
}
//*****************************************************************************

void sim_screen(void) {

 clrscr();                    // clears output screen
 cout << "\n\n\n\n"
```

```
      << "\t* * * * * * * * * * * * * * * * * * * * * * * * * * * * *\n"
      << "\t*                                                         *\n"
      << "\t*       D   O   N ' T     D   I   S   T   U   R   B       *\n"
      << "\t*                                                         *\n"
      << "\t*                                                         *\n"
      << "\t*        S I M U L A T I O N   I N   P R O G R E S S      *\n"
      << "\t*                                                         *\n"
      << "\t* * * * * * * * * * * * * * * * * * * * * * * * * * * * *\n"
      << "\n\n";
}
//*****************************************************************************

void start_screen(void) {

 clrscr();                    // clears output screen
 cout << "\n\n\n\n"
      << "\t* * * * * * * * * * * * * * * * * * * * * * * * * * * * *\n"
      << "\t*                                                         *\n"
      << "\t*                 A T M  S I M U L A T O R                *\n"
      << "\t*                                                         *\n"
      << "\t*               Programmed by Tom Corcoran                *\n"
      << "\t*             (Help available in readme.doc)              *\n"
      << "\t*                                                         *\n"
      << "\t* * * * * * * * * * * * * * * * * * * * * * * * * * * * *\n"
      << "\n\n"
      << "\tPlease enter the input filename,\n"
      << "\teg. a:\\input\\names.txt: ";
}
//*****************************************************************************
```

C.4 Join program

```
//*****************************************************************************

/*  File:    join.cpp
    Author:  Tom Corcoran
    Date:    23/7/93
    Version: 1.0

    Outline:

Two files are inputed which contain simulation results. The second
file Contains the simulation results which come from a continuation
of the first simulation. The results are combined to obtain the
results Which would have been obtained if the simulation was run as
```

```
a whole.

    There are three input only files and one output file:

    Joininfo.dat:    User generated. Contains the program input parameters:

        - first_f_name
        - first_array_size
        - first_num_samples
        - second_f_name
        - second_array_size
        - second_num_samples
        - out_file_name

    The other two input files, first_f_name and second_f_name
    are generated by the simulation system and are the files to be joined.
    The combined results are written to the file out_f_name.

    Note:   eg. If the largest queue length in the first_f_name
                (or second_f_name) is 110, then the first_array_size is 111.*/

//*****************************************************************************

//  includes                            //  defines

#include <fstream.h>                    //  ofstream,ifstream
#include <conio.h>                      //  clrscr()
#include <math.h>                       //  exp()
#include <stdlib.h>                     //  exit()
#include <string.h>                     //  strcpy()
#include <iomanip.h>                    //  setiosflags()
#ifndef OPENFILE_H
 #define OPENFILE_H
 #include "openfile.h"                  //  modules for opening files
#endif
#ifndef QUEUE_H
 #define QUEUE_H
 #include "queue.h"                     //  class queue_c
#endif
#ifndef CHARSTR_H
 #define CHARSTR_H
 #include "charstr.h"                   //  class string_c
#endif
#ifndef ROUTINES_H
 #define ROUTINES_H
 #include "routines.h"                  //  modules for rounding double,
```

```
#endif                                  //  setting up and initializing arrays

//*************************************************************************

//  function prototypes

void main(void);
void read_in_array_from_file(char *in_f_name,double *this_array);
void convert_back_to_cdf(int array_size,double *this_array);
void convert_back_to_pdf(int array_size,long int num_samples,
                         double *this_array);
void join_arrays(int out_array_flag,int first_array_size,
                 int second_array_size,double *array_1,double *array_2);
void initial_screen(void);
void display_screen(void);
void display_info(char *first_f_name,char *second_f_name,char *out_f_name);

//*************************************************************************

void main(void) {

 int first_array_size;                  //  array size in first file
 int second_array_size;                 //  array size in second file
 int out_array_size;                    //  array size in output file
 int out_array_flag;                    //  indicates which array largest (1/2)
 long int first_num_samples;            //  total number of additions to array_1
 long int second_num_samples;           //  total number of additions to array_2

 //  dynamic arrays - holds contents of file 1 and file 2

 double *array_1 = NULL;
 double *array_2 = NULL;

 double *out_array = NULL;              //  points to array_1/array_2 (biggest)

 //  temporary file names

 char temp_in_f_name[32];               //  input file name
 char temp_first_f_name[32];            //  first file to be joined
 char temp_second_f_name[32];           //  second file to be joined
 char temp_out_f_name[32];              //  output file

 ifstream in_f;                         //  stream for input
 ofstream out_f;                        //  stream for output

 initial_screen();                      //  calling initial screen
```

```
//  input file name provided by user

cin >> temp_in_f_name;
string_c in_f_str(temp_in_f_name);   //  string_c class for input file

// calling screen to show work in progress

display_screen();

//  opening input file,reading parameters and closing

open_file(in_f_str,&in_f);
in_f >> temp_first_f_name >> first_array_size >> first_num_samples
     >> temp_second_f_name >> second_array_size >> second_num_samples
     >> temp_out_f_name;
in_f.close();

//  string_c calls for each file name; setting up file names

string_c first_f_str(temp_first_f_name);
string_c second_f_str(temp_second_f_name);
string_c out_f_str(temp_out_f_name);

//  allocating space for arrays

array_1 = setup_dynamic_array(first_array_size);
array_2 = setup_dynamic_array(second_array_size);

//  initializing arrays

initialize_dynamic_array(first_array_size,array_1);
initialize_dynamic_array(second_array_size,array_2);

//  reading in arrays from files

read_in_array_from_file(first_f_str,array_1);
read_in_array_from_file(second_f_str,array_2);

//  convert back to cdf

convert_back_to_cdf(first_array_size,array_1);
convert_back_to_cdf(second_array_size,array_2);

//  convert back to pdf
```

```
convert_back_to_pdf(first_array_size,first_num_samples,array_1);
convert_back_to_pdf(second_array_size,second_num_samples,array_2);

// setting size of output array and marking largest array

if (first_array_size > second_array_size) {
 out_array_size = first_array_size;
 out_array = array_1;
 out_array_flag = 1;
}
else {
 out_array_size = second_array_size;
 out_array = array_2;
 out_array_flag = 2;
}

//  joining array_1 and array_2

join_arrays(out_array_flag,first_array_size,second_array_size,
            array_1,array_2);

//  finding the total needed to normalize the new array

long int sum_samples = first_num_samples + second_num_samples;

//  creating instance of queue class

queue_c queue(out_array_size,sum_samples,out_array);

/*  generating data for graphing
    working out the probaility density function and then the
    cummulative density function (and getting log of it)                    */

queue.setup_pdf();                  //  calculating probability distribution
queue.build_cdf();                  //  calculating cdf, p[q < b]
queue.build_one_minus_cdf();        //  calculating p[q >= b]
queue.log_queue();                  //  log10[p[q >= b]]

//  output queue to file

open_file(out_f_str,&out_f);        //  opening file
out_f << &queue;                    //  outputing array (inserter function)
out_f.close();                      //  closing file

//  deleting objects
```

```
 delete array_1;
 delete array_2;

 //  write information to screen

 display_info(first_f_str,second_f_str,out_f_str);
}
//*************************************************************************

void join_arrays(int out_array_flag,int first_array_size,
                 int second_array_size,double *array_1,double *array_2) {

 //  joining arrays - adding arrays to each other

 register int j;

 if (out_array_flag == 1) {                    //  array_1 > array_2
  for (j = 1; j < second_array_size; j++)
   array_1[j] += array_2[j];
 }
 else {                                        //  array_2 > array_1
  for (j = 1; j < first_array_size; j++)
   array_2[j] += array_1[j];
 }
}
//*************************************************************************

void convert_back_to_pdf(int array_size,long int num_samples,
                         double *this_array) {

 int i = array_size - 1;
 int j;

 do {
  j = i - 1;

  /*  working out the probability value by subtracting the previous cumulative
      probability value from the current cummulative probability value      */

  this_array[i] -= this_array[j];

  /*  converting the probability back to the queue length and ensuring
      that it is a whole number                                             */

  this_array[i] = round_double(double(num_samples)*this_array[i]);
  i--;
```

```
 } while (this_array[i] != 0.0);
}
//*****************************************************************************

void convert_back_to_cdf(int array_size,double *this_array) {

 // convert back to cdf

 int array_size_minus_one = array_size - 1;
 int i = 1;

 while (i < array_size_minus_one) {

  //  the log10 value in the file was log10(1 - cdf)

  this_array[i] = (1.0 - pow(10.0,this_array[i]));
  i++;
 }

 //  the last element in a cdf is always 1

 this_array[array_size_minus_one] = 1.0;
}
//*****************************************************************************

void read_in_array_from_file(char *in_f_name,double *this_array) {

 int q_length;                          //  queue length inputed from file
 double log_value;                      //  log value corresponding to q_length
                                        //   inputed from file
 ifstream *in_f = NULL;

 //  creating new object

 in_f = new ifstream;

 //  reading in arrays from files

 open_file(in_f_name,in_f);
 *in_f >> q_length >> log_value;
 while (!in_f->eof()) {
  this_array[q_length] = log_value;
  *in_f >> q_length >> log_value;
 }
 in_f->close();
```

```
 //  deleting object

 delete in_f;
}
//*************************************************************************

void display_info(char *first_f_name,char *second_f_name,char *out_f_name) {


  cout << "\t\tSimulation output files:\n\n\t\t" << first_f_name << " and "
       << second_f_name << "\n\n\t\thave been joined and the result "
       << "is in the file:\n\n\t\t" << out_f_name << endl;
}
//*************************************************************************

void display_screen(void) {

 clrscr();                    // clears output screen
 cout << "\n\n\n\n"
      << "\t* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *\n"
      << "\t*                                                             *\n"
      << "\t*       Simulation output files are being joined....          *\n"
      << "\t*                                                             *\n"
      << "\t* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *\n"
      << "\n\n";
}
//*************************************************************************

void initial_screen(void) {

 clrscr();                    // clears output screen
 cout << "\n\n\n\n"
      << "\t* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *\n"
      << "\t*                                                             *\n"
      << "\t*                Simulation joining program                   *\n"
      << "\t*                                                             *\n"
      << "\t* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *\n"
      << "\n\n"
      << "\tPlease enter the input file name,"
      << "\n\teg. a:\\input\\demojoin.dat: ";
}
//*************************************************************************
```

C.5 Disc index

The following is a listing of the important files included in the accompanying disc:

- **readme.doc**: This file gives details on how to format the input so as to run the simulation system.
- **simatm.exe**: The executable file for running the ATM simulator.
- **join.exe**: The executable code for running the join program, which allows two or more simulations to be joined.
- **atmdata.zip**: This file is stored in the *data* directory and contains the zipped data of all the simulations described in this thesis (see readme.doc).