INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI®

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

NOTE TO USERS

This reproduction is the best copy available

UMI

Université d'Ottawa · University of Ottawa

ACCELERATED SIMULATION OF ATM SWITCHING FABRICS

By
Gilles Lamothe, B.Sc.
November 1998

A Thesis
submitted to the School of Graduate Studies and Research
in partial fulfillment of the requirements
for the degree of
Master of Science in Mathematics[1]

© Copyright 1998
by Gilles Lamothe, B.Sc., Ottawa, Canada

[1]The M.Sc. Program is a joint program with Carleton University, administered by the Ottawa-Carleton Institute of Mathematics and Statistics

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-38756-9

Canada

Abstract

Asynchronous transfer mode (ATM) is accepted as the solution to the broadband integrated services network known as B-ISDN. The information transmitted through an ATM network is divided into fixed sized packets, called cells. These cells traverse an ATM switching fabric, wherein they are electronically switched a number of times in order to arrive at the correct destination. For design purposes, it is important to know the cell loss ratio (CLR) for the buffers in an ATM switching fabric.

We model the ATM switch, as an intree acyclical network of queues receiving N independent heterogenous Markov modulated input streams of cells. Due to the complexity of the model, simulations must be used to find the CLR for a hot spot buffer in the switching fabric. Crude Monte-Carlo simulations are not very efficient, since most services require a very small CLR, e.g. 10^{-9}. Importance sampling techniques must be used. To make cell losses less rare, we increase (exponentially twist) the arrival rate of cells directed toward the hot spot buffer. We use the twisted probabilities with a splitting importance sampling technique to obtain an efficient estimate of the CLR for this buffer.

Acknowledgements

I would like to thank my supervisors Prof. Andre Dabrowski, and Prof. David McDonald for all their help and advice. I would also like to thank Marie-Claude Bonneau at Newbridge Networks Corp. for providing us concrete applications.

I would like to thank both the National Sciences and Engineering Research Council (NSERC), and the Ottawa-Carleton Institute of Mathematics and Statistics for funding throughout my research.

Dedication

À mes parents.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Asynchronous Transfer Mode

In recent years the telecommunication industry has seen the emergence of many new services such as high-speed data transfer, video-on-demand, videophony, and many others. Asynchronous transfer mode (ATM) has been accepted by the International Telecommunications Union Telecommunication (ITU-T) as the solution to this broadband service requirement, known as the broadband integrated services digital network (B-ISDN).

Data transmitted through an ATM network is divided into 53 byte packets, called cells, see Figure 1. These cells traverse ATM switches wherein they are electronically switched to a proper connection in order to arrive at the correct destination as specified in the header of the cell. An ATM switch, see Figure 2, can have a number of ingoing connections, say n, and a number of outgoing connections, say m. A cell will enter the ATM switch through an input port, say I_i, and the switch will send it to the correct output port, say O_j, utilizing the information contained inside the header of the cell. In such a switch contention might arise, that is two or more connections might arrive simultaneously and contend for the same time slot. Such contention is remedied by queueing the cells in buffers before sending them out. Using finite queues does however create the possibility of buffer overflow and thus

Cell - "A unit of transmission in ATM. A fixed-size frame consisting of a 5-octet [bytes] header and a 48-octet [bytes] payload."
- The ATM Forum

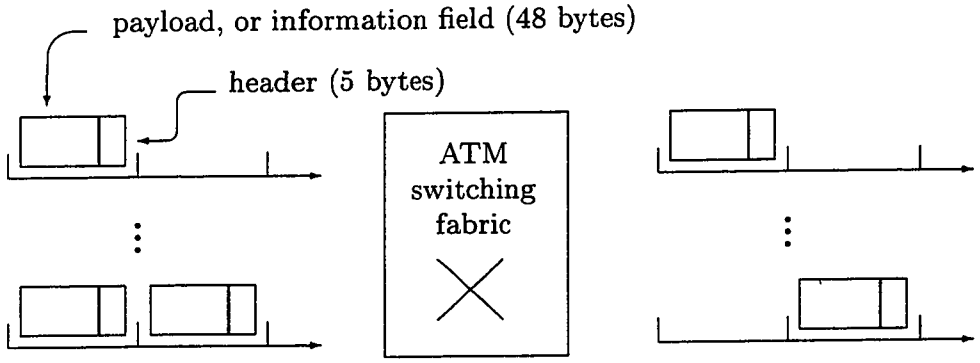

Figure 1: Fixed size packets, called cells

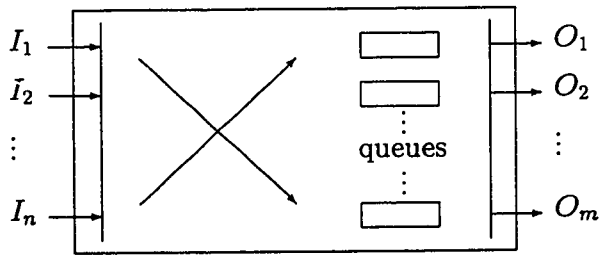

Figure 2: ATM switch with output buffering

cell (information) loss. An ideal network has semantic transparency, i.e. information reaches its correct destination with very high probability. Semantic transparency is very important in a full service network since many services are sensitive to data loss, e.g. video teleconferencing, and data transfer.

Throughout this thesis we assume that the loss of a cell can only be caused by the overflow of a buffer. A good indicator of semantic transparency is the cell loss ratio (CLR), where

$$\text{CLR} = \lim_{T\to\infty} \frac{\text{cells lost by time } T}{\text{cells offered by time } T}.$$

Bonneau (1996) accelerated the simulation of a leaky bucket controller, known also as a "generic cell rate algorithm" for an ATM buffer, in order to find the cell loss ratio. Her approach will be followed here to accelerate the simulation of an ATM switching fabric, that is a network constructed from several ATM switches interconnected together.

Our goal is to find the CLR for a buffer in an ATM switching fabric, using an importance sampling technique to accelerate simulation. For a survey on importance sampling, see Heidelberger (1995). We give a simple example of importance sampling drawn from Heidelberger (1995), to motivate the reader to understand what we are trying to accomplish.

1.2 A Motivation for Importance Sampling

Let X be an $\mathbb{R}$-valued random variable, with density p. We want to estimate the probability that X belongs to some rare event $\mathcal{A}$, that is

$$\gamma_{\mathcal{A}} := \mathrm{P}(X \in \mathcal{A}) = \mathrm{E}_p[I_{\mathcal{A}}(X)] = \int_{-\infty}^{\infty} I_{\mathcal{A}}(x)\, p(x) dx,$$

where $I_{\mathcal{A}} := I\{X \in \mathcal{A}\}$.

Let $\{X_n\}$ be a stochastic process of i.i.d. random variables distributed as X. The

frequency estimator

$$\widehat{\gamma_{\mathcal{A}}}(N) := \frac{1}{N}\sum_{n=1}^{N} I\{X_n \in A\}$$

is an unbiased estimator for $\gamma_{\mathcal{A}}$. It can be easily shown that

$$\mathrm{RE}(\widehat{\gamma_{\mathcal{A}}}(N)) \sim \sqrt{\frac{1}{N\,\gamma_{\mathcal{A}}}} \quad \text{as } \gamma_{\mathcal{A}} \to 0,$$

where RE is the relative error, i.e. the standard deviation divided by the mean. This means for small $\gamma_{\mathcal{A}}$, we must have a very large sample size N so as to obtain a "good" estimate, i.e. an estimate with a small relative error (e.g. 10%).

Suppose we can find a distribution with density p', such that $\mathcal{A}$ is a likely event. Notice the following

$$\gamma_{\mathcal{A}} = \int_{-\infty}^{\infty} I_{\mathcal{A}}(x)\,\frac{p(x)}{p'(x)}\,p'(x)dx = \mathrm{E}_{p'}[I_{\mathcal{A}}(X)\,\mathcal{L}(X)],$$

where $\mathcal{L}(x) := p(x)/p'(x)$ (we call $\mathcal{L}$ the likelihood ratio). If we generate $\{X_1, \ldots, X_n\}$ from the new distribution, then the following weighted frequency estimator

$$\widehat{\gamma_{\mathcal{A}}}(N, p') := \frac{1}{N}\sum_{n=1}^{N} \mathcal{L}(X_n)\,I\{X_n \in \mathcal{A}\}$$

is an unbiased estimator for $\gamma_{\mathcal{A}}$. There exist examples where it is possible to find a distribution p' such that

$$\mathrm{RE}(\widehat{\gamma_{\mathcal{A}}}(N, p')) < \mathrm{RE}(\widehat{\gamma_{\mathcal{A}}}(N)),$$

see Asmussen, Rubinstein, and Wang (1994). This means that the importance sampling estimator $\widehat{\gamma_{\mathcal{A}}}(N, p')$ is more efficient than the crude Monte-Carlo estimator $\widehat{\gamma_{\mathcal{A}}}(N)$.

1.3 A Guide to the Thesis

Readers with little or no exposure to stochastic processes such as regenerative processes, Markov modulated sequences, and Markov additive chains should first read

Appendix A. Appendix A serves as an introduction to these topics. Now we guide the reader through the thesis.

The purpose of Chapter 2 is two-fold. First we introduce the model we want to study, that is the ATM switching fabric with bursty traffic. Second, we justify the use of accelerated simulation in order to find the CLR for a buffer deep in an ATM switching fabric, we call this buffer of interest the hot spot buffer. We show that the problem cannot be solved analytically because of the very high dimension of the state space. Also, we argue that standard (crude Monte-Carlo) simulations are not practical because of the extremely long simulation time needed to obtain an accurate estimate, that is an estimate with a small enough relative error (e.g. 10%).

Chapter 3 can be broken into four key objectives or observations. Our first objective is to introduce the twist technique for Markov-additive (MA) chains, see McDonald (1998), and show how the twist can be used with an importance sampling technique for regenerative systems, see Heidelberger (1995), to accelerate simulations in order to find the CLR for a simple queueing model. Essentially, the twist technique consists of finding a harmonic function h for the transition kernel of a MA-chain. With the harmonic function h, we can then find the h-transform of the transition kernel, which is also a probability transition kernel. The stochastic process generated from this new probability transition kernel is also a MA-chain, which will have a positive mean increment if the original MA-chain has a negative mean increment, see McDonald (1998). This means that we can transform a stable queue into an unstable queue for which the event of losing a cell becomes likely in the twisted model. Bonneau (1996) used the twist technique in conjunction with an importance sampling technique called the "splitting" A-cycle technique, see Heidelberger (1995), to accelerate the simulation of a leaky bucket controller (or equivalently an ATM multiplexer) with bursty traffic to find the CLR. This thesis is an extension to Bonneau's thesis (1996), we show how the twist and splitting techniques can be used together to accelerate the simulation of an ATM switching fabric.

Second, we make the observation that between successive returns to an empty hot spot buffer, the workload of the hot spot buffer and the bursty sources have a Markov additive structure, with transition K^{∞}, where the workload is the additive part. We construct a MA-chain with the transition kernel K^{∞}, called the free process.

Third, we find a harmonic function h for the free process, see the harmonicity Theorem 3.1. This is the most important part of the thesis. From this harmonic function h, we get the explicit form of the probabilities governing the bursty traffic in the conjugate (twisted) model used in the importance sampling (splitting) technique. From the function h, we also get the explicit form of the likelihood ratio associated to a semi-regenerative A-cycle (called Δ-cycle here), see Chapter 4. The fourth key objective of Chapter 3 is to show that the harmonic function h can be obtained with ease using simple numerical techniques.

In Chapter 4, we construct a consistent ratio estimator for the CLR of the hot spot buffer, using the splitting technique. The likelihood ratio is given explicitly for both cases, infinite and finite buffers. We also introduce an importance sampling algorithm which reduces the number of aborted Δ-cycles.

We apply our importance sampling algorithm to two specific models, Model I and Model II, respectively in Chapter 5 and Chapter 6. Chapter 6 is particularly interesting since we show that our importance sampling algorithm applies to ATM switching fabrics with time priority queueing. It is the simplicity of the method which allows us to include ATM switching fabrics with time priority queueing protocols to the class of queueing models for which our importance sampling algorithm applies.

To finish, we prove the existence Theorem 7.1, that is we prove that the harmonic function h does exist. This is accomplished by proving the existence of constants that satisfy a certain non-linear system of equations.

1.4 New Developments

This thesis is a detailed application of Beck, Dabrowski, and McDonald (1998) to Model I and II. There are four innovations. The importance sampling splitting Δ-cycle technique from Bonneau (1996) is extended in order to find efficient estimates of the cell loss ratio for a buffer in an ATM switching fabric. The main innovation is to twist the workload directed at this buffer. Also, a technique is given to reduce the number of aborted cycles. We explicitly give the likelihood ratio, for the Δ-cycles, with a finite buffer correction. This is the most significant theoretical contribution of the thesis. We apply our importance sampling algorithm to a switching fabric with time priority queueing.

1.5 Competing Methods

In this section, we briefly discuss the differences and the similarities between this thesis and three current papers Beck, Dabrowski, and McDonald (BDM) (1998), Falkner, Devetsikiotis, Lambadaris (FDL) (1998), and L'Ecuyer and Champoux (LC) (1996). These papers discuss importance sampling for ATM networks.

LC model the bursty traffic as ON-OFF sources but only the burst lengths are random. FDL also consider ON-OFF sources, although at every time slot during a burst period the sources generate a cell with a certain probability, i.e. the sources are Bernoulli interrupted. BDM consider a more general input structure of Markov modulated sources. In this thesis, the input traffic is modelled as a special case of the Markov modulated sources from BDM. This input structure is similar to the Bernoulli interrupted sources of FDL, but we allow heterogeneous sources and we include a delay in the state space. This delay allows the reciprocal of the peak cell rate of a source to be any multiple of the time slot. This models bursty traffic with a constant bit rate during bursts less than the link rate. In this thesis a time slot is the reciprocal of the link rate of the hot spot.

BDM study a fast teller queue model, that is one queue with multiple servers. The times between services and the served batch sizes are random. In the fast teller queue model, BDM do not allow multiple types of traffic such as high/low priority. The services in LC, FDL, and this thesis are deterministic. The non-stochastic modeling of the servers is reasonable since the services inside an ATM switch are deterministic. The possibility of multiple queues per link (or server) is new in this thesis, see Chapter 6 for a high/low priority example.

As aforementioned, BDM study a one queue (infinite capacity) system with no crosstalk. In LC, FDL, and this thesis the network is an ATM switching fabric with finite buffers. There is crosstalk and the transfer rates are unrestricted. The finite buffer correction is new in this thesis.

Using an h-transform, BDM exponentially twist the length of the queue. They also remark that the workload of a hot spot queue in an ATM switching fabric could be twisted in a similar way, see [Section 5 BDM]. In this thesis we find an h-transform to exponentially twist the workload of a hot spot server. LC and FDL also calculate a twist using an algorithm proposed by Chang, Heidelberger, Juneja S., and Shahabuddin (1994) based on the theory of effective bandwidth for a single queue with Markov modulated input. Based on heuristic reasoning, LC do not twist the sources which generate cells not directed towards the hot spot, see [Section 4 LC]. We also do not twist the traffic which is not directed at the hot spot, but this follows by our h-transform. The algorithm for determining the twist of the workload given here is *much simpler* than those of LC and FDL but gives the same twist for a single queue network ($\ell^* = 1$ in LC and FDL). These "twisting" methods obtain efficient estimates of the cell loss ratio through experimentation, see [Section 4 FDL, Section 5 LC].

During a twisted A-cycle FDL and LC turn off the importance sampling when the hot spot buffer hits (or exceeds) the critical value l (i.e. the length of the hot spot buffer) for the first time, see [Section 4 FDL, Section 4 LC]. However in this thesis we turn off the importance sampling when the workload of the hot spot (not the hot

spot) reaches (or exceeds) the critical length l. The reason is to reduce the number of lost cells while using the twisted probabilities thus reducing the variance of our estimator.

Chapter 2

The Model

2.1 The ATM Switching Fabric

An ATM switching fabric is an acyclical[1] network of queues. It is actually a number of switches interconnected together in such a way that there is exactly one path for each pair of input and output ports. Figure 3 gives an illustration of a 4×4 switching fabric constructed from 2×2 switching elements. The dotted lines illustrate the path from I_4 to O_1 in the switching fabric.

We will call the buffer of interest the hot spot buffer, (upper right corner in Figure 3). We will find the cell loss ratio for this buffer. The hot spot buffer does not have to be the last buffer in a path, that is any buffer in the switching fabric can be considered as the hot spot buffer. We also define the workload, $Q[t]$, of the hot spot at time t as the number of cells in the buffers (including the source multiplexers) upstream of the hot spot at time t and which will eventually pass through the hot spot (including the cells in the hot spot buffer itself).

The transmission rate from the switch with the hot spot buffer will be denoted as LR (link rate) with units *cells/sec.* Since the cells in a switch are served at a constant rate it is natural to work in discrete time and we shall do so. The basic unit of time,

[1] Acyclical: A cell in the network cannot pass through the same queue twice.

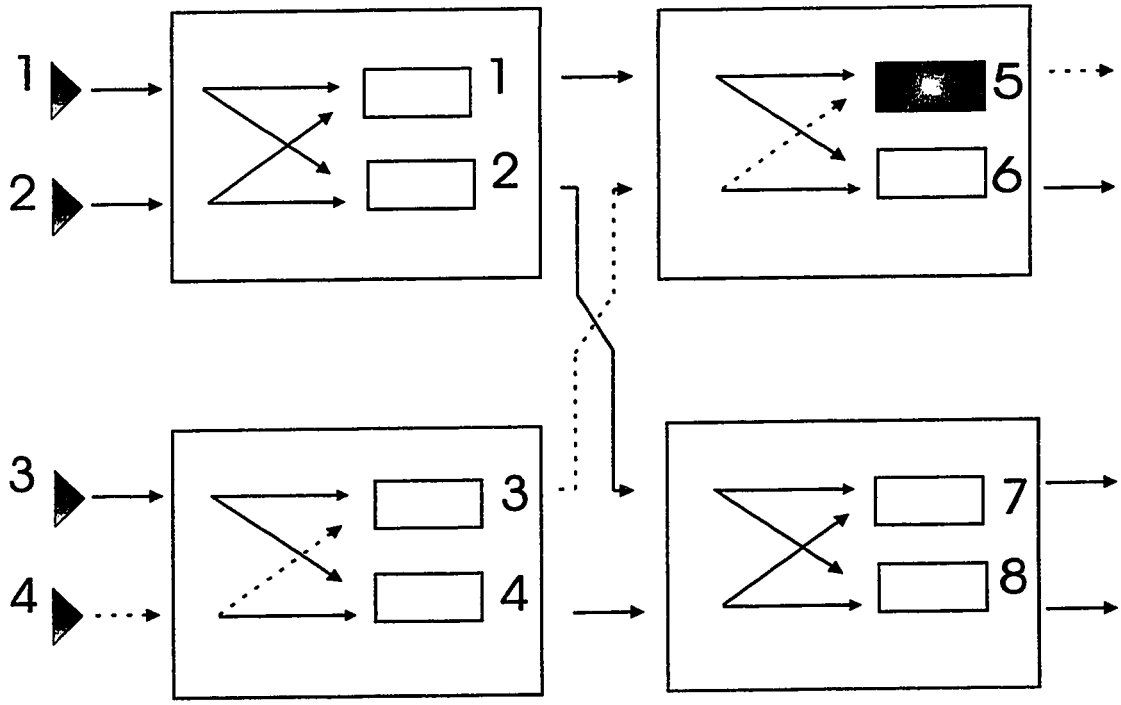

Figure 3: ATM switching fabric (4×4) buffer

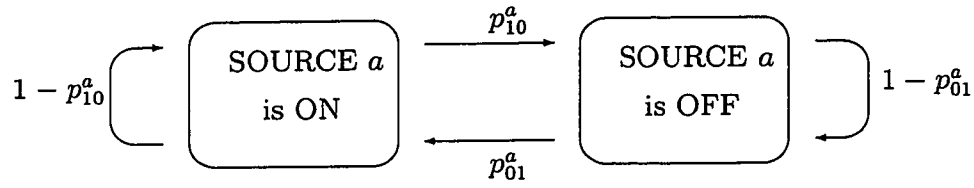

Figure 4: Transition probabilities for the underlying Markov chain

which we shall call a time slot, will be $1/LR$ seconds. Internal blocking (contention) in an ATM switching fabric can cause delay in the transfer of data unrelated to cell loss at the hot spot buffer of interest. A solution to this problem is to increase the internal link rates to the point that internal blocking is almost non-existent. Thus in practice the switches in an ATM switching fabric can have different link rates, and we make the same assumption in our model.

2.2 The Traffic

In queueing theory the input stream is typically modeled by a Poisson process, although it has become common practice to incorporate burstiness when modelling ATM traffic. Wright (1993) defines burstiness as the ratio of the peak to the mean bit rate. A bursty service has a burstiness much greater than one, that is the cells do not arrive at a constant rate. An example of a bursty service is image transfer. A user might ask for an image, which generates a burst of information, then there is a silent period until the user asks for another image. We will model this burstiness by a Markov modulated process, the Bernoulli interrupted arrival stream with deterministic inter-arrival times.

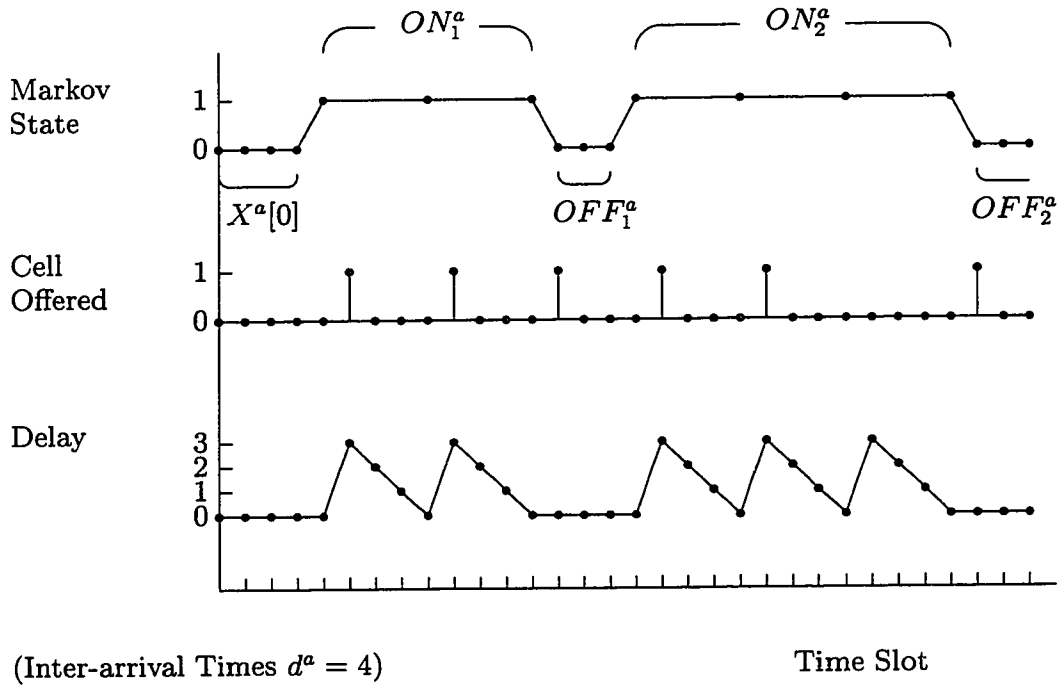

Figure 5: Arrival stream for source a

A Single Source

The source a is modulated by an ON-OFF alternating process $\{M^a[n]\}$,

$$\mathrm{K}^a := \begin{pmatrix} 1-p_{01}^a & p_{01}^a \\ p_{10}^a & 1-p_{10}^a \end{pmatrix}, \tag{1}$$

and state space $\mathcal{S}^a = \{0 \equiv \mathrm{OFF}, 1 \equiv \mathrm{ON}\}$. Appendix A gives a review of Markov modulated sequences.

While $M^a[n]$ is in state 1 (source a is ON), there is a delay d^a between cells offered to the ATM switching fabric. Every time slot the delay is reduced by one. When the delay is reduced to 0, source a will offer a cell during the next time slot with probability ρ^a, and there is a state transition according to kernel K^a to either state 0, or state 1. We define a burst period to be a block of time during which the source remains ON. Thus during a burst period there is a possible cell arrival every d^a time slots. Let $X^a[0]$ be the initial delay, i.e. the number of time slots before the first burst period, and let ON_n^a be the length of the nth burst period. The expected length of a burst period is

$$\begin{aligned} \mathrm{E}[ON_1^a] &= 1 + \sum_{x=0}^{\infty} x\, d^a\, (1-p_{10}^a)^x\, p_{10}^a \\ &= 1 + \frac{d^a\,(1-p_{10}^a)}{p_{10}^a} \\ &= \frac{d^a + (1-d^a)\,p_{10}^a}{p_{10}^a}. \end{aligned}$$

While $M^a[n]$ is in state 0 (source a is OFF), no cells are offered to the ATM switching fabric, and there a state transition according to kernel K^a to either state 0, or state 1. If the transition is to state 1, the delay is set to zero and a cell is offered with probability ρ^a. Similar to a burst period, we define an idle period to be a block of time during which the source remains OFF. Let OFF_n^a be the length of the idle period following the nth burst period. The sojourn time OFF_1^a has a geometric distribution

with parameter p_{01}^a, thus its mean is $1/p_{01}^a$. Figure 5 gives an example of an arrival stream from source a.

We construct a renewal sequence in order to find the long-run probability that the source a is ON and the delay is zero. The initial delay is $X^a[0]$. The random variables $\{X^a[n] = ON_n^a + OFF_n^a : n = 1, 2, \dots\}$ are independent and identically distributed. Taking the $X^a[n]$ as the inter-arrival times of the renewals, we have constructed an alternating renewal process. Let R_n^a be the number of time slots, that the source is ON and the delay is zero, during the nth ON-OFF cycle. The random variable R_1^a has a geometric distribution with parameter p_{10}^a, thus its mean is $1/p_{10}^a$. By the renewal reward Theorem A.3, the long-run probability that a source is ON with a delay of zero is

$$\frac{\mathrm{E}[R_1^a]}{\mathrm{E}[ON_1^a] + \mathrm{E}[OFF_1^a]} = \frac{p_{01}^a}{(d^a + (1 - d^a)\, p_{10}^a)\, p_{01}^a + p_{10}^a}. \tag{2}$$

Since a cell is offered with probability ρ^a when the source is ON and when the delay is zero, the long-run average number of cells offered to the ATM switching fabric by source a (per time slot) is

$$\frac{p_{01}^a}{(d^a + (1 - d^a)\, p_{10}^a)\, p_{01}^a + p_{10}^a}\, \rho^a. \tag{3}$$

Definition 2.1 *We say that the arrival stream generated by source a has a Bernoulli interrupted with deterministic inter-arrival times (BIDIT) distribution with parameters* $(d^a, p_{01}^a, p_{10}^a, \rho^a)$.

Remark 1: We assume the cells generated by a single source always follow the same path. If we are interested in the behavior of the hot spot for a short period of time, e.g. five to fifteen minutes, then this assumption is justified.

Total Sources

We will model the traffic into the ATM switch as a superposition of N independent heterogeneous Bernoulli interrupted arrival streams with deterministic inter-arrival times. The set $A = \{1, 2, \dots, N\}$ is the index for the N independent BIDIT sources.

Let $H \subset A$ be the index set for the sources that generate cells directed at the hot spot buffer. Partition the sources within H and $A \setminus H$ into groups of sources that generate indentically distributed arrival streams. We assume there are M groups of N_i of identically distributed sources, $N = \sum_{i=1}^{M} N_i$. Let A_h be the index set for the groups of sources which generate cells which will pass through the hot spot, the set A_n will be the index set for the other groups; thus $A_h \cup A_n$ forms a partition of the set $\{1, 2, \ldots, M\}$, see Figure 6.

All sources in group $i \in \{1, \ldots, M\}$ generate identically distributed arrival streams. Thus, the BIDIT parameters $(d^a, p_{01}^a, p_{10}^a, \rho^a)$ are constant for all sources a in group i. We denote these constants as $(d_i, \lambda_i, \mu_i, r_i)$.

Let $N_{ij}[n]$ be the number of sources from group i which are ON with delay j, where $0 \leq j \leq d_i - 1$. If $OFF_i[n]$ is the number of sources of type i which are OFF at time n then

$$OFF_i[n] = N_i - \sum_{j=0}^{d_i-1} N_{ij}[n]. \tag{4}$$

The Transition Kernel

To find the transition kernel for the Markov chain $\{(N_{ij}[n])_{j=0}^{d_i-1}\}$, we consider two cases: $d_i = 1$ and $d_i > 1$. Appendix A gives a review of Markov chains, transition kernels, etc.

Case I: $(d_i = 1)$

Suppose n_{i0} sources from group i are ON, then $N_i - n_{i0}$ sources from that group are OFF. In the next time slot the number of sources which are ON will be the sum of the number of active sources which remain ON and the number of idle sources which become ON. Therefore the transition kernel is

$$\mathrm{K}_i(n_{i0}, m_{i0}) = \sum_{x+y=m_{i0}} \mathrm{Bin}(n_{i0}, x, 1-\mu_i)\, \mathrm{Bin}(N_i - n_{i0}, y, \lambda_i),$$

where

$$\mathrm{Bin}(n,x,p) := \binom{n}{x} p^x (1-p)^{n-x}.$$

Case II: $(d_i > 1)$

Let n_{ij} be the number of sources which are currently ON with delay j and let m_{ij} be the number of sources which are ON with delay j in the next time slot. We must have $m_{ij} = n_{i,j+1}$ for $j = 1, \ldots, d_i - 2$. Of the n_{i0} sources which are ON, there are exactly m_{i,d_i-1} which remain ON. Of the $f_i = N_i - \sum_{j=0}^{d_i-1} n_{ij}$ sources which are idle, there are exactly $m_{i0} - n_{i1}$ which will become active (ON). Therefore the transition kernel is

$$\mathrm{K}_i(\vec{n}_i, \vec{m}_i) = \mathrm{Bin}(n_{i0}, m_{i,d_i-1}, 1-\mu_i)\,\mathrm{Bin}(f_i, m_{i0} - n_{i1}, \lambda_i).$$

Let $\vec{N}_i = (N_{i0}, \ldots, N_{i,d_i-1})$ and $\vec{N}[n] = (\vec{N}_1[n], \ldots, \vec{N}_M[n])$. The stochastic process $\{\vec{N}[n]\}$, which represents the state of the BIDIT sources through time, is a Markov chain with transition kernel

$$K_A[\vec{n}, \vec{m}] = \prod_{i=1}^{M} K_i[\vec{n}_i, \vec{m}_i],$$

and state space $\mathcal{S}_A = \times_{i=1}^{M} \{0, 1, \ldots, N_i\}^{d_i}$.

Theorem 2.1 (Irreducibility) *Consider the Markov chain $\{\vec{N}[n]\}$ of the BIDIT sources, with finite state space $\mathcal{S}_A$.*

(a) If $0 < \lambda_i, \mu_i < 1$ for all $i \in A$, then the Markov chain is irreducible $\{\vec{N}[n]\}$.

(b) If there exists a group i of sources which always remain ON, i.e. $(\lambda_i = 1, \mu_i = 0)$, then for any initial state $\vec{n}$ there exists a reduced state space $\mathcal{S}_A^r(\vec{n}) \subset \mathcal{S}_A$ with the following properties:

i. $\vec{n} \in \mathcal{S}_A^r(\vec{n})$,

ii. for any positive integer n, $K_A^n[\vec{n}, \vec{m}] = 0$ *for all* $\vec{m} \in \mathcal{S} \setminus \mathcal{S}_A^r(\vec{n})$,

ii. the Markov chain $\{\vec{N}[n]\}$ *with state space* $\mathcal{S}_A^r(\vec{n})$ *is irreducible.*

The proof of Theorem 2.1 is in Appendix B. For a Markov chain with a finite state space, irreducibility is a sufficient condition for the existence of a stationary distribution. Let π_A be the stationary distribution for the Markov chain $\{\vec{N}[n]\}$ of the BIDIT sources.

Generating Function for Group i

Let $A_i[n]$ be the cells offered to the hot spot during time n. The process $\{A_i[n]\}$ is a Markov modulated process with underlying Markov chain $\{\vec{N}_i[n]\}$. We can construct the Markov additive chain with Markovian part $\{\vec{N}_i[n]\}$ and increment for the additive part $\{A_i[n]\}$. The generating function for this MA-chain is

$$\begin{aligned} \widehat{\mathrm{K}}_i[\vec{n}_i, \vec{m}_i|\gamma] &= \mathrm{E}[\exp(\gamma\, A_i[1])\, I\{\vec{N}_i[1] = \vec{m}_i\}|\vec{N}_i[0] = \vec{n}_i] \\ &=: \widehat{\mathrm{K}}_\gamma^i[\vec{n}_i, \vec{m}_i] \end{aligned} \tag{5}$$

We call $\widehat{\mathrm{K}}_\gamma^i$ the generating function for the group i.

2.3 The Workload of the Hot Spot

As aforementioned the workload Q of the hot spot is the number of cells currently in the ATM switching fabric which will eventually pass through the hot spot including the cells currently in the hot spot buffer. When a cell in the hot spot is served the workload is reduced by one. We assume the hot spot to be work conserving: at each time slot if there is at least one cell waiting to be served in the hot spot buffer, one cell will be served. Let $H^-[n]$ denote the length of the hot spot buffer the instant before the service during the nth time slot. Thus the workload of the hot spot is governed by the recurrence equation

$$Q[n] = (Q[n-1] + A[n] - I\{H^-[n] > 0\})^+,$$

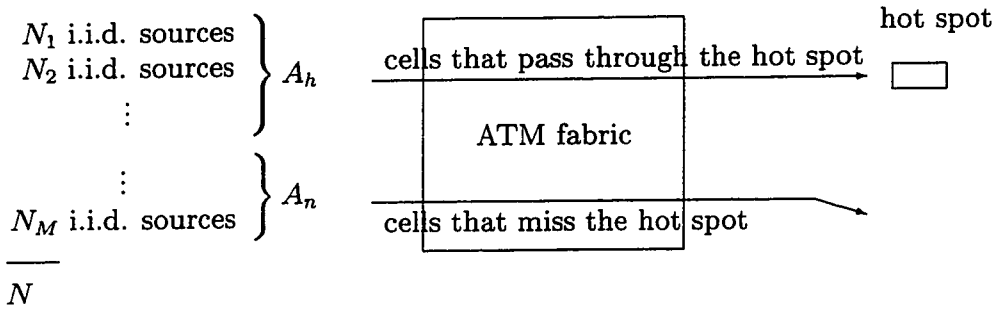

Figure 6: The N independent sources

where $x^+ := \max\{x, 0\}$ and $A[n] := \sum_{i \in A_h} A_i[n]$ is the number of cell arrivals at time n to the workload of the hot spot. We also impose the following system stability condition, the mean arrival rate directed to any buffer inside the switching fabric is strictly less than the service rate at that point. This means that the mean arrival rate directed to the hot spot buffer should be less than one, thus by (3),

$$\sum_{i \in A_h} N_i \frac{\lambda_i}{(d_i + (1 - d_i)\,\mu_i)\,\lambda_i + \mu_i} r_i < 1. \tag{6}$$

2.4 Analytical Solution

The analytical solution to the cell loss ratio is numerically intractable because of the high dimension of the state space. The purpose of this section is to illustrate this point. An example of a small ATM switching fabric is given.

We consider a 4×4 ATM switching fabric constructed from 2×2 switching elements, that is Figure 3. We assume that the transmission rates are the same for the four switching elements. The buffer B_i has a finite capacity of L_i cells. The cells are generated by N independent sources as described in Section 2.2. Since we allow more than one source per multiplexer, it is possible to have more than one arrival at the same time. This causes a delay for one of the cells, called jitter. We will assume that this delayed cell is queued inside the multiplexer. The buffer sizes for the four input multiplexers are L_i^m for $i = 1, \ldots, 4$.

We will now model the ATM switching fabric as a Markov chain. Define the following random variables

$$\vec{B}[n] := (B_1[n], \ldots, B_8[n], B_1^m[n], \ldots, B_4^m[n]),$$

representing the queue lengths of the different buffers in the switching fabric and the input buffers. Also $\vec{N}[n]$ are the states of the independent sources offering cells to the switching through the four multiplexers. We also need the destination of each source in the switching fabric, without this information we don't know for example if a cell should be switched from buffer B_1 to buffer B_5 or to buffer B_7. The random variable

$$\vec{D}_i[n] := (D_{i1}[n], \ldots, D_{i,L_i}[n])$$

will indicate the destination of the cells queued in the buffer B_i at the end of the nth time slot, where $D_{ij}[n]$ is the destination of the jth cell in the buffer, or is zero if there are less than j cells queued. Let

$$\vec{D}[n] := (\vec{D}_1[n], \ldots, \vec{D}_8[n], \vec{D}_1^m[n], \ldots, \vec{D}_4^m)$$

be the destination of the cells queued in the switching fabric. The stochastic process $\{\vec{B}[n], \vec{D}[n], \vec{N}[n]\}$ is a Markov chain with transition kernel K_F, state space S_F and stationary distribution π_F.

In order to obtain the cell loss ratio, we must first the stationary distribution π_F. To find π_F, we must solve the following linear system of equations

$$\pi_F(s) = \sum_{s' \in S_F} \pi_F(s')\, \mathrm{K}_F(s', s), \qquad \text{for all } s \in S_F.$$

The number of equations in this system is equal to the number of states in S_F. The cardinality of S_F is

$$\left(\prod_{i=1}^{4}(L_i^m + 1) \times (4+1)^{L_i^m}\right) \times \left(\prod_{i=1}^{8}(L_i + 1) \times (4+1)^{L_i}\right) \times \#\mathcal{S}_A,$$

where $\mathcal{S}_A$ is the state space for the independent sources. It is clear that the number of states increases exponentially as we increase the buffer. Actually in practice this number is extremely large. For example, suppose all buffers have a capacity of ten cells, the number of states is $(3.06 \times 10^{19}) \times \#\mathcal{S}_A$. We cannot guarantee that there exists an ordering of the states such that the matrix K_F will be a band matrix (see Burden and Faires (1993)), or another other type of sparse matrix not mentioned here. If there is an ordering of the states which converts K_F into a sparse matrix, it's construction is not obvious. Also, at this point in time the practical limit[2] for the size of a sparse matrix is approximately a billion entries, which is much too small for our CLR problem. Consequently, at this time, our problem cannot be addressed by numerical methods for large sparse matrices.

If, in addition, we allow different transmission rates, this would also increase this already large state space. To do so we would have to use a smaller time slot, e.g.

[2]Some current information on sparse matrices can be found at the following websites:

http://www.maf.wisc.edu/~ahmad/733/2.html
http://www-pub.cisc.ufl.edu/~davis/sparse/
http://math.nist.gov/MatrixMarket

one divided by the lowest common multiple of all the transmission rates, instead of one divided by the transmission rate of the hot spot buffer. We would have to associate a delay to each buffer, similar to the delay for the sources, since each buffer will have a clock cycle which is a constant multiple of the length of a time slot. At each time slot the delay will decrease by one, and when the delay is zero a cell will be served if the buffer is not empty. To incorporate this delay allows us to preserve the Markovian Property, i.e. the future depends only on the present and not the past.

To conclude, our ability to find the CLR for the hot spot buffer is contingent on our ability to find the stationary distribution π_F. However, to find π_F we must solve a system of linear equations which is generally numerically intractable, because of the high dimension of the state space S_F. Other methods such as simulation must be used.

2.5 Crude Monte-Carlo Simulation

We can use simulation techniques to estimate the CLR for the hot spot buffer. Here we discuss a cycle technique that derives from renewal theory. We will give an example to show how to apply the technique for a simple queueing model. After we will explain how to apply this technique for an ATM fabric.

Theorem A.3 tells us that for a renewal-reward process the long-run average reward is equal to the expected reward per cycle divided by the expected length of a cycle. We will utilize this fact to construct a cycle technique for our simulation. The method is known as regenerative simulation, see Chapter V.2d in Asmussen (1987), or Rubinstein (1981). Let the reward be the cell loss. The example considers a queueing model with only one queue and binomial input with parameters n and p.

Example 2.1 *Let $S[k]$ and $A[k]$ be the queue length and the arrivals respectively during the time k. The arrivals have a binomial distribution with parameters n and*

p, i.e.

$$P(A[1] = a) = \binom{n}{a} p^a (1-p)^{n-a}.$$

We will assume that the queue is work conserving that is when the queue is not empty a cell is served per time slot. We suppose also that the service is the last possible event in the time slot, i.e. the arrivals occur before the service. The queue length is governed by the equation

$$S[k] = (S[k-1] + \xi[k])^+,$$

where $\xi[k] := A[k] - 1$ *are the increments. Note, the free process associated to this queueing process is a random walk, see section A.7. The increments* $\xi[k]$ *have probability mass*

$$p(x) = \begin{cases} P(A[1] = x+1), & \text{if } x \geq -1 \\ 0, & \text{otherwise.} \end{cases}$$

We are interested in estimating the cell loss ratio for this queue. Since we know the long-run average cell arrival (np) *we only need to estimate the long-run average cell loss. A cycle will start when the queue is empty and will end a time slot before the queue empties again. The renewal reward Theorem says the long-run average cell loss is the expected cell loss per cycle divided by the expected length of a cycle. Let* $CCL[n]$ *be the cell loss associated to the nth cycle and let* $L[n]$ *be the length of that cycle. The following*

$$\frac{1}{N}\sum_{n=1}^{N} CCL[n] \qquad \text{and} \qquad \frac{1}{N}\sum_{n=1}^{N} L[n]$$

are unbiased estimators of the expected cell loss per cycle and the expected length of a cycle respectively. Thus an estimator for the long-run average cell loss is

$$\frac{\sum_{n=1}^{N} CCL[n]}{\sum_{n=1}^{N} L[n]}.$$

In Example 2.1 we see that it is very easy to estimate the long-run average cell loss for a single queue with binomial input. Assuming the simulation starts and ends with an empty queue we just have divide the total cell loss by the total time of the simulation. This very simple technique can be extended to estimate the cell loss ratio

for the hot spot buffer of an ATM switching fabric with Bernoulli interrupted input, using an A-cycle technique, see Heidelberger (1993).

Let A be a recurrent subset of the state space. A new A-cycle will start with each successive return to the set A. The renewal reward Theorem for regenerative cycles can be generalized to include A-cycles, i.e. the long-run average reward is equal to the long-run average reward per A-cycle divided by the long-run average length of an A-cycle.

In Section 2.4 we modelled the ATM switching fabric as a Markov chain. From this Markov chain we will construct a Δ-cycle (A-cycle) process using the recurrent set Δ, where Δ is the set of states with an empty hot spot buffer. A new Δ-cycle starts with each successive return to Δ. This means a cycle starts with an empty hot spot buffer and ends the time slot just before the hot spot empties again. Note the other part of the workload may be non-zero. Let $CCL_\Delta[n]$ and $L_\Delta[n]$ be the cell loss and the length of the nth Δ-cycle respectively. An estimator for the long-run average cell loss is

$$\frac{(1/N)\sum_{n=1}^{N} CCL_\Delta[n]}{(1/N)\sum_{n=1}^{N} L_\Delta[n]}.$$

If we assume that the upstream buffers to the hot spot buffer are infinite or at least large enough such that there will be no cell loss then an estimator for the cell loss ratio is

$$\widehat{CLR}_\infty(N) = \left(\frac{(1/N)\sum_{n=1}^{N} CCL_\Delta[n]}{(1/N)\sum_{n=1}^{N} L_\Delta[n]}\right) \Big/ \left(\sum_{i\in A_h} N_i \frac{\lambda_i}{(d_i + (1-d_i)\,\mu_i)\,\lambda_i + \mu_i} r_i\right),$$

where $\sum_{i\in A_h} N_i\, r_i\, \lambda_i/[(d_i + (1-d_i)\,\mu_i)\lambda_i + \mu_i]$ is the long-run average cells offered to the hot spot buffer, see Section 2.2. Realistically the buffers upstream to the hot spot buffer are finite which means the long-run average offered traffic to the workload of the hot spot is larger than the long-run average offered traffic to the hot spot buffer. A cell can be lost in the fabric before it reaches the hot spot. When the buffers in the fabric are finite the long-run average offered traffic must also be estimated. Let $COT_\Delta[n]$ be the offered traffic to the hot spot buffer during the nth Δ-cycle. An

estimator for the long-run average offered traffic to the hot spot is

$$\frac{(1/N)\sum_{n=1}^{N} COT_{\Delta}[n]}{(1/N)\sum_{n=1}^{N} L_{\Delta}[n]}.$$

Hence after simplification we get the following estimator for the cell loss ratio for the hot spot buffer

$$\widehat{CLR}(N) = \frac{(1/N)\sum_{n=1}^{N} CCL_{\Delta}[n]}{(1/N)\sum_{n=1}^{N} COT_{\Delta}[n]}.$$

We will call $\widehat{CLR}(N)$ the crude Monte-Carlo estimator. To summerize: start the simulation with the hot spot buffer empty and stop the simulation right before the hot spot buffer empties for the Nth time where N is some positive integer. The crude Monte-Carlo estimate is the average cell loss per Δ-cycle divided by the average offered cells to the hot spot per Δ-cycle.

The computational effort (number of cells generated during the simulation) of the crude Monte-Carlo method does become great when the CLR is extremely small. For example, with a CLR of 10^{-8}, we would need to generate on average at least 10^8 cells to get a least one cell loss. On a Model 355 RS6000 (IBM), we can generate approximately 500 cells per (computational) second. This means to generate 10^8 cells, it would take approximately 2.3 (computational) days. We also want the estimate to be accurate. In other words, we want an estimate with a small relative error (standard deviation divided by the mean), e.g. 10%. Through experimentation, see Chapter 5 and Chapter 6, it seems for a relative error of 10%, the number of cells needed during a simulation is at least 100 times the reciprocal of the CLR. Thus, for a CLR of 10^{-8}, we would need at least 10^{10} cells, which translates to more than 7 (computational) months. Most services require a cell loss ratio in the range of 10^{-7} to 10^{-11} (a cell loss is a rare event). Therefore it is not practical to use crude Monte-Carlo simulation techniques to estimate the CLR. We must use importance sampling techniques.

Chapter 3

The Twist

Our goal in this chapter is to find a random process satisfying three criteria;

- it should be a simple transform of the original ATM fabric model, **(C1)**
- hot spot buffer overflows should be likely events, **(C2)**

and

- there should be a straightforward manner of computing the cell loss ratio for the hot spot buffer in the original ATM fabric from that of the new process. **(C3)**

We find what we seek in the large deviation theory for Markov additive chains (MA-chains), (see Appendix A for the definition of MA-chains).

Let $\{(S^{\infty}[k], M[k]) : k = 0, 1, 2, \dots\}$ be the free MA-chain associated to a MA-chain with boundary, $\{S[k], M[k]\}$, such that $S^{\infty}[k]$ is the additive part and $M[k]$ is the Markovian part, see Example A.5. McDonald (1998) studies the asymptotics of the additive part $S[k]$. To do so he utilizes a conjugate process to the free MA-chain known as the twisted process. We will now proceed to define the twist for the case where the increments $\xi[k]$ take values in $\mathbb{Z}$. Let K^{∞} be the transition kernel of the

free MA-chain. Now define $\mathcal{K}^\infty$ with the following exponential change of measure

$$\mathcal{K}^\infty[(s,\vec{n}),(s',\vec{m})] = \exp\{\theta\,(s'-s)\}\,\frac{r(\vec{m})}{r(\vec{n})}\,\mathrm{K}^\infty[(s,\vec{n}),(s',\vec{m})],$$

and where r is a positive function. If we can find a θ and postulate a suitable r such that

$$\sum_{(s',\vec{m})\in\mathbb{Z}\times S} \mathcal{K}^\infty[(s,\vec{n}),(s',\vec{m})] = 1,$$

then $\mathcal{K}^\infty$ is a transition kernel, called the twisted kernel (or the h-transform of K), and it generates an associated MA-chain called the twisted process. The following example finds the twist for a random walk on $\mathbb{Z}$ (called the associated random walk by Feller (1966)).

Example 3.1 (Random Walk) *Let $\{S[k] : k = 0,1,\dots\}$ be a random walk on $\mathbb{Z}$ as defined in Appendix A.7, and assume the mean increment $\mu = E[X[1]]$ is negative. So the generating function $\mathcal{M}(t)$ for this random walk is strictly convex. Assume that there exists $\theta > 0$ such that*

$$\mathcal{M}(\theta) = \sum_{x\in\mathbb{Z}} \exp(\theta\,x)\,p(x) = 1.$$

The generating function $\mathcal{M}(t)$ is necessarily defined for all t from 0 to θ. Actually θ is the unique value greater than zero such that $\mathcal{M}(\theta) = 1$: this follows from the strict convexity of $\mathcal{M}$, $\mathcal{M}(0) = \mathcal{M}(\theta) = 1$ and $\mathcal{M}'(0) = \mu < 0$. The transition kernel for the random walk is $K(s,s') = p(s'-s)$. The twisted kernel is

$$\mathcal{K}(s,s') = \exp\{\theta\,(s'-s)\}\,K(s,s'),$$

and let $\{\mathcal{S}[k]\}$ be the associated random walk. The mean increment $\tilde{\mu}$ of this new process is positive, since $\tilde{\mu} = \mathcal{M}'(\theta) > 0$ by the strict convexity of the generating function, see Figure 7.

In the preceding example we have constructed a twisted process from a random walk on $\mathbb{Z}$. This twisted process is also a random walk on $\mathbb{Z}$ but has a positive mean

increment unlike the negative mean increment for the original random walk. Hence the name twist: the mean increment is "twisted" such that the process drifts in an opposite direction. Clearly the twisted process satisfies the second criterion **(C2)** since the queue overflows for the associated random walk with boundary are likely events (the mean increment is positive). In Example 3.2, we show how the twisted process satisfies the second criterion **(C1)** and the third criterion **(C3)**, respectively.

Example 3.2 (Example 2.1 continued) *We will use the associated random walk to transform the queueing model of Example 2.1. Let θ be the unique real positive number such that the generating function is* 1. *We have*

$$\begin{aligned} 1 = \mathcal{M}(\theta) &= \sum_{x\in\mathbb{Z}} \exp(\theta x)\, p(x) \\ &= \sum_{x=-1}^{n-1} e^{\theta x} \binom{n}{x+1} p^{x+1} (1-p)^{n-x-1} \\ &= e^{-\theta} \sum_{x=-1}^{n-1} \binom{n}{x+1} (e^{\theta} p)^{x+1} (1-p)^{n-x-1} \\ &= e^{-\theta} (e^{\theta} p + 1 - p)^{n}. \end{aligned} \tag{7}$$

The twisted kernel is

$$\begin{aligned} \mathcal{K}(s,s') &= \exp\{\theta\,(s'-s)\}\, K(s,s') \\ &= \exp\{\theta\,(s'-s)\}\, p(s'-s) \\ &= \exp\{\theta\,(s'-s)\} \binom{n}{s'-s+1} p^{s'-s+1} (1-p)^{n-(s'-s+1)} \\ &= \binom{n}{s'-s+1} \left(\frac{e^{\theta} p}{e^{\theta} p + 1 - p}\right)^{s'-s+1} \left(\frac{1-p}{e^{\theta} p + 1 - p}\right)^{n-(s'-s+1)} \qquad \textit{by (7).} \end{aligned}$$

So the arrivals for the twisted model have a binomial distribution with parameters n and $(e^{\theta} p)/(e^{\theta} p + 1 - p)$. Therefore the chain generated by the twisted kernel describes the same queueing model as in Example 2.1 but with different arrival rates, hence it satisfies criterion **(C1)**.

We will use the twist to find the cell loss ratio of the queueing model introduced in Example 2.1. It is a single queue with binomial input. We need to find the long-run average number of cells lost per time slot. We have shown that it is the expected number of cells lost per cycle divided by the expected length of a cycle,

$$\frac{E[CCL]}{E[L]}.$$

We find the denominator using the crude Monte-Carlo simulations from Example 2.1. To find the numerator we will use a twisted cycle technique which is analogous to importance sampling, (c.f. Heidelberger (1993)).

As aforementioned a cycle starts with the queue empty and ends the time slot before it empties again. Let T_0 be the set of all cycle trajectories. For a certain cycle trajectory let y be the length of the queue which exceeds the finite capacity of the queue l (by $R(y)$) for the first time. As in Bonneau (1996), we call this state y a "fictitious" state. Assuming the length of the queue before the overflow was s_f we redirect the transition from s_f to $y^{\sim} = l$ with probability $K^{\sim}(s_f, y^{\sim}) = K(s_f, y)$. For $\omega \in T_0$ let $V(\omega)$ be the number of cells lost during the cycle ω. The expected number of cells lost per cycle is

$$\sum_{\{0,s_1,\ldots,s_f,y^{\sim},\ldots,s_r\}\in T_0} V(0, s_1, \ldots, s_f, y^{\sim}, \ldots, s_r)\, \pi(0) \times K(0, s_1) \cdots K(s_{f-1}, s_f)\, K^{\sim}(s_f, y^{\sim}) \cdots K(x_{r-1}, x_r).$$

Using the twisted kernel $\mathcal{K}(s, s') = \exp(\theta\,(s' - s))\, K(s, s')$ the expected number of cells lost per cycle is also

$$\sum_{\{0,s_1,\ldots,s_f,y^{\sim},\ldots,s_r\}\in T_0} V(0, s_1, \ldots, s_f, y^{\sim}, \ldots, s_r)\, \pi(0)\, \exp(\theta\,(0 - s_1))\, \mathcal{K}(0, s_1) \cdots \exp(\theta\,(s_{f-1} - s_f))\, \mathcal{K}(s_{f-1}, s_f)\, \exp(\theta\,(s_f - l - R(y)))\, \mathcal{K}^{\sim}(s_f, y^{\sim}) \times K(l, s_{f+2}) \cdots K(x_{r-1}, x_r).$$

Simplification gives us

$$e^{-\theta l} \sum_{\{0,s_1,\ldots,s_f,y^{\sim},\ldots,s_r\}\in T_0} V(0, s_1, \ldots, s_f, y^{\sim}, \ldots, s_r)\, e^{-\theta\, R(y)}\, \pi(0)\, \mathcal{K}(0, s_1) \cdots \mathcal{K}(s_{f-1}, s_f)\, \mathcal{K}^{\sim}(s_f, y^{\sim})\, K(l, s_{f+2}) \cdots K(x_{r-1}, x_r). \tag{8}$$

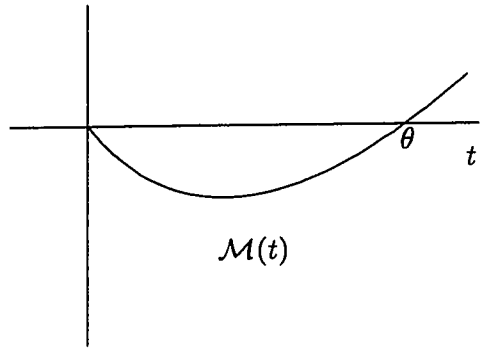

Figure 7: Generating Function of the Random Walk on $\mathbb{Z}$ (Example 3.1)

A twisted cycle is a cycle where the twisted probabilities are utilized until the queue exceeds the capacity l and the original probabilities are utilized for the remainder of the cycle. Let R_n be the value by which the queue exceeds l for the first time during the nth twisted cycle and $CCL[n]$ is the cell loss during the same cycle. By (8), we see that

$$e^{-\theta l}\frac{1}{N}\sum_{n=1}^{N} CCL[n] \times e^{-\theta R_n}$$

is an unbiased estimator for $E[CCL]$.

Let us proceed to define our ATM switching fabric as a Markov additive chain.

3.1 Markov Additive Chain

In Section 2.4, we model the arrivals and services of the ATM switching fabric as a Markov chain. The set Δ contains all states with an empty hot spot buffer. Define T_0^Δ as the first time the hot spot buffer is empty and T_n^Δ as the nth return to Δ for $n = 1, 2, \ldots$. Consider a pair $(T_i^\Delta, T_{i+1}^\Delta)$ such that $T_{i+1}^\Delta - T_i^\Delta > 1$. The workload Q follows the following recurrence equation

$$Q[n] = Q[n-1] + A[n] - 1,$$

for $n = T_i^\Delta + 1, \ldots, T_{i+1}^\Delta$. Actually the stochastic process $\{(Q[n], \vec{N}[n]) : n = T_i^\Delta, \ldots, T_{i+1}^\Delta\}$ is a Markov additive chain indexed on a finite set, where $Q[n]$ is the workload of the hot spot and $\vec{N}[n]$ are the states of the sources, see Section 2.2. Let K be the transition kernel for this MA-chain, we have

$$\begin{aligned} & \mathrm{K}[(q, \vec{n}), (q + c - 1, \vec{m})] \\ = \; & \mathrm{P}(\vec{N}[T_i^\Delta + 1] = \vec{m}, Q[T_i^\Delta + 1] - Q[T_i^\Delta] = c - 1 | \vec{N}[T_i^\Delta] = \vec{n}). \end{aligned}$$

Define the free process $\{W^\infty[n]\}$ as

$$W^\infty[n] := \begin{cases} Q[n] & \text{if } n = 0 \\ W^\infty[n-1] + A[n] - 1 & \text{if } n = 1, 2, \ldots \end{cases}$$

The stochastic process $\{(W^\infty[n], \vec{N}[n]) : n = 0, 1, \ldots\}$ is a Markov additive chain with transition kernel K^∞.

The Transition Kernel K^∞

The random variables $\{\xi[n]\}$ are the increments for the MA-chain $\{W^\infty[n], \vec{N}[n]\}$, i.e. $\xi[n] = A[n] - 1$ where $A[n]$ is the number of cells offered to the workload of the hot spot during the time slot n. Consider any one of these cells. This cell must be offered by a source in some group $i \in A_h$, which was ON with delay zero at the end of time slot $n - 1$. Let $\vec{n} \in \mathcal{S}_A$ be the state of the sources at the end of the time slot $n - 1$. During the time slot n the maximum number of cells offered to the hot spot from the sources in group $i \in A_h$ is n_{i0}, since this is the number of sources that are ON with

delay zero at the end of the time slot $n-1$. Define the set of possible combinations for c arrivals to the workload of the hot spot as

$$S_c[\vec{n}] := \left\{ (z_i)_{i\in A_h} : 0 \le z_i \le n_{i0} \ \forall i \in A_h, \sum_{i\in A_h} z_i = c \right\}.$$

If $k = c-1$, then

$$\begin{aligned} \mathrm{P}(\xi[1] = k | \vec{N}[0] = \vec{n}) &= \sum_{\vec{z}\in S_c[\vec{n}]} \prod_{i\in A_h} \binom{n_{i0}}{z_i} r_i^{z_i} (1-r_i)^{n_{i0}-z_i} \\ &= \sum_{\vec{z}\in S_c[\vec{n}]} \prod_{i\in A_h} \mathrm{Bin}(n_{i0}, z_i, r_i). \end{aligned}$$

Take $\vec{n}, \vec{m} \in \mathcal{S}_A$ and $q, q' \in \mathbb{Z}$ where $q' = q + c - 1$, the transition kernel for the free MA-chain is

$$\begin{aligned} \mathrm{K}^\infty[(q,\vec{n}),(q',\vec{m})] &= \mathrm{P}(M[1] = \vec{m}, \xi[1] = k | M[0] = \vec{n}) \\ &= \mathrm{P}(\xi[1] = k | M[0] = \vec{n}) \ \times \prod_{i=1}^{M} \mathrm{K}_i[\vec{n}, \vec{m}]. \end{aligned}$$

3.2 Conjugate Process

Our goal is to overload the hot spot buffer. We will do so by overloading the workload of the hot spot using the twisted probabilities. Notice that away from Δ the stochastic processes $\{(W^\infty[n], \vec{N}[n])\}$ and $\{(Q[n], \vec{N}[n])\}$ have the same underlying probability structure, i.e. the transition probabilities are the same. Thus we can use the twisted probabilities to generate twisted Δ-cycles similar to the twisted cycles generated in Example 3.2.

We will construct the conjugate (twisted) MA-chain $\{\mathcal{W}^\infty[n], \vec{\mathcal{N}}[n]\}$ to the MA-chain $\{Q^\infty[n], \vec{N}[n]\}$ by finding a positive harmonic function h for the kernel K^∞. By definition this means h satisfies the averaging property

$$h(q,\vec{n}) = \sum_{(q',\vec{m})} \mathrm{K}^\infty[(q,\vec{n}),(q',\vec{m})]\, h(q',\vec{m}). \tag{9}$$

The harmonic function will be of the form $h(q,\vec{n}) = \exp(\theta\, q)\, r(\vec{n})$, where θ is a positive real number and r is some positive function defined on $\mathcal{S}_A$. The operator

$$\begin{aligned} \mathcal{K}^{\infty}[(q,\vec{n}),(q',\vec{m})] &:= \exp\{\theta\,(q'-q)\}\,\frac{r(\vec{m})}{r(\vec{n})}\,\mathrm{K}^{\infty}[(q,\vec{n}),(q',\vec{m})] \\ &= \mathrm{K}^{\infty}[(q,\vec{n}),(q',\vec{m})]\,\frac{h(q',\vec{m})}{h(q,\vec{n})} \end{aligned} \tag{10}$$

is a probability transition kernel by the averaging property (9). We call $\mathcal{K}^{\infty}$ the h-transform of K^{∞}, or the twisted kernel.

We assume that there exists positive real numbers θ and β_i for $i \in A_h$ and real numbers α_i for $i \in A_h$ that satisfy the following system of equations:

$$e^{\theta} = \prod_{i\in A_h} (1-\lambda_i+\lambda_i\,\beta_i)^{N_i}, \tag{11}$$

$$e^{-\alpha_i} = ((1-\mu_i)\,\exp\{\alpha_i\,(d_i-1)\} + \mu_i\,\beta_i^{-1})\,(e^{\theta}\,r_i + 1 - r_i) \quad \forall i \in A_h \tag{12}$$

$$e^{-\alpha_i} = 1-\lambda_i+\lambda_i\,\beta_i \qquad \forall i \in A_h \tag{13}$$

Define the positive functions r_i (for all $i \in \{1,\ldots,M\}$) as

$$r_i(\vec{n}_i) := \begin{cases} \beta_i^{N_i-f_i}\,\exp\{\alpha_i \sum_{j=1}^{d_i-1} j\, n_{ij}\}, & \text{if } i \in A_h \text{ and } d_i > 1 \\ \beta_i^{n_{i0}}, & \text{if } i \in A_h \text{ and } d_i = 1 \\ 1, & \text{if } i \in A_n \end{cases}$$

where $f_i = N_i - \sum_{j=0}^{d_i-1} n_{ij}$. Take the positive function r to be the product of the r_i, that is $r(\vec{n}) := \prod_{i=1}^{M} r_i(\vec{n}_i)$, where $\vec{n} = (\vec{n}_1, \vec{n}_2, \ldots, \vec{n}_M)$.

Now we present Lemma 3.1, which we use to prove that h is a harmonic function for the transition kernel K^{∞}.

Lemma 3.1 *Let $N_{ij}[n]$ be the number of BIDIT sources in group i, which are ON at the end of the nth time slot with an associated delay j. The following equations hold:*

if $d_i > 1$ *then*

$$\sum_{j=1}^{d_i-1} j\,(N_{ij}[n+1] - N_{ij}[n]) = (d_i - 1)\,N_{i,d_i-1}[n+1] - \sum_{j=1}^{d_i-1} N_{ij}[n], \tag{14}$$

$$OFF_i[n] - OFF_i[n+1] = N_{i0}[n+1] - N_{i1}[n] - (N_{i0}[n] - N_{i,d_i-1}[n+1])\,, \tag{15}$$

and, if $d_i = 1$ *then*

$$OFF_i[n] - OFF_i[n+1] = N_{i0}[n+1] - N_{i0}[n]. \tag{16}$$

Proof: The deterministic nature of the inter-arrival times gives us the following relation

$$N_{ij}[n+1] = N_{i,j+1}[n] \qquad \text{for } j = 1, \ldots, d_i - 2, \tag{17}$$

since a possible arrival which is delayed by $j+1$ time slots will be delayed by j time slots in the next time slot.

The equation (14) follows from

$$\begin{aligned}
&\sum_{j=1}^{d_i-1} j\,(N_{ij}[n+1] - N_{ij}[n]) \\
&= (d_i - 1)\,N_{i,d_i-1}[n+1] + \sum_{j=1}^{d_i-2} j\,N_{i,j+1}[n] - \sum_{j=1}^{d_i-1} j\,N_{ij}[n] \quad \text{by (17)} \\
&= (d_i - 1)\,N_{i,d_i-1}[n+1] + \sum_{j=2}^{d_i-1} (j-1)\,N_{ij}[n] - \sum_{j=1}^{d_i-1} j\,N_{ij}[n] \\
&= (d_i - 1)\,N_{i,d_i-1}[n+1] - \sum_{j=1}^{d_i-1} N_{ij}[n].
\end{aligned}$$

The equation (15) follows from

$$\begin{aligned}
&OFF_i[n] - OFF_i[n+1] \\
&= \sum_{j=0}^{d_i-1} N_{ij}[n+1] - \sum_{j=0}^{d_i-1} N_{ij}[n] \quad \text{by (4)} \\
&= N_{i0}[n+1] + N_{i,d_i-1}[n+1] - N_{i0}[n] - N_{i,1}[n] \quad \text{by (17)} \\
&= N_{i0}[n+1] - N_{i1}[n] - (N_{i0}[n] - N_{i,d_i-1}[n+1])\,.
\end{aligned}$$

For all $i \in A_h \cup A_n$ such that $d_i = 1$, we have

$$OFF_i[n] - OFF_i[n+1] = N_{i0}[n+1] - N_{i0}[n],$$

since $OFF_i[n] = N_i - N_{i0}[n]$ for all n from (4).

□

Assuming that $\vec{m}_i$ is a possible state reached in a one-step transition from $\vec{n}_i$ and $d_i > 1$ then we have the following relations from (14) and (15):

$$\sum_{j=1}^{d_i-1} j\,(m_{ij} - n_{ij}) = (d_i - 1)\,m_{i,d_i-1} - \sum_{j=1}^{d_i-1} n_{ij} \qquad \forall i \in A_h \tag{18}$$

$$f_i - f_i' = m_{i0} - n_{i1} - (n_{i0} - m_{i,d_i-1}) \qquad \forall i \in A_h. \tag{19}$$

If $d_i = 1$ then by (16)

$$f_i - f_i' = m_{i0} - n_{i0}. \tag{20}$$

Let $A_{h1} \subset A_h$ be the index set for the groups of sources offering cells to the workload of the hot spot with inter-arrival times equal to one, i.e. $d_i = 1$ for all $i \in A_{h1}$. Take $i \in A_h \setminus A_{h1}$, the ratio $r_i(\vec{m}_i)/r_i(\vec{n}_i)$ is equal to

$$\begin{aligned}
&\beta_i^{(f_i - f_i')} \exp\left\{\alpha_i \sum_{j=1}^{d_i-1} j\,(m_{ij} - n_{ij})\right\} \\
&= (\beta_i^{-1})^{(n_{i0} - m_{i,d_i-1})}\, \beta_i^{(m_{i0} - n_{i1})} \exp\left\{\alpha_i \sum_{j=1}^{d_i-1} j\,(m_{ij} - n_{ij})\right\} \quad \text{by (19)} \\
&= (\beta_i^{-1})^{(n_{i0} - m_{i,d_i-1})}\, \beta_i^{(m_{i0} - n_{i1})}\, (e^{\alpha_i (d_i - 1)})^{m_{d_i-1}} \exp\left\{-\alpha_i \sum_{j=1}^{d_i-1} n_{ij}\right\} \quad \text{by (18).}
\end{aligned}$$

Since $f_i = N_i - \sum_{j=0}^{d_i-1} n_{ij}$ we get

$$\exp\left\{-\alpha_i \sum_{j=1}^{d_i-1} n_{ij}\right\} = \exp\left\{-\alpha_i\,(N_i - f_i - n_{i0})\right\}$$

which using (12) is equal to

$$\left[((1-\mu_i)\,\exp\{\alpha_i\,(d_i-1)\}+\mu_i\,\beta_i^{-1})(e^{\theta}\,r_i+1-r_i)\right]^{-n_{i0}}\,(1-\lambda_i+\lambda_i\,\beta_i)^{N_i-f_i}.$$

Hence the ratio $r_i(\vec{m}_i)/r_i(\vec{n}_i)$ (for $i \in A_h \setminus A_{h1}$) is equal to

$$\frac{(\beta_i^{-1})^{(n_{i0}-m_{i,d_i-1})}\,(e^{\alpha_i\,(d_i-1)})^{m_{d_i-1}}}{((1-\mu_i)\,e^{\alpha_i\,(d_i-1)}+\mu_i\,\beta_i^{-1})^{n_{i0}}}\,\frac{\beta_i^{(m_{i0}-n_{i1})}}{(1-\lambda_i+\lambda_i\,\beta_i)^{f_i}}\;\times \\ (e^{\theta}\,r_i+1-r_i)^{-n_{i0}}\,(1-\lambda_i+\lambda_i\,\beta_i)^{N_i}. \tag{21}$$

Take $i \in A_{h1}$, by (20) the ratio $r_i(\vec{m}_i)/r_i(\vec{n}_i)$ is equal to

$$\beta_i^{f_i-f_i'}=\beta_i^{m_{i0}-n_{i0}}.$$

Substituting $e^{-\alpha_i}$ from (12) into (13), with $d_i=1$, gives

$$1=1^{n_{i0}}=\left(\frac{1-\lambda_i+\lambda_i\,\beta_i}{(1-\mu_i+\mu_i\,\beta_i^{-1})\,(e^{\theta}\,r_i+1-r_i)}\right)^{n_{i0}}.$$

Hence the ratio $r_i(\vec{m}_i)/r_i(\vec{n}_i)$ (for $i \in A_{h1}$) is equal to

$$\beta_i^{m_{i0}-n_{i0}}\times\left(\frac{1-\lambda_i+\lambda_i\,\beta_i}{(1-\mu_i+\mu_i\,\beta_i^{-1})\,(e^{\theta}\,r_i+1-r_i)}\right)^{n_{i0}} \\ =\beta_i^{m_{i0}}\left(\frac{\beta_i^{-1}}{1-\mu_i+\mu_i\,\beta_i^{-1}}\right)^{n_{i0}}\,(e^{\theta}\,r_i+1-r_i)^{-n_{i0}}\,\frac{(1-\lambda_i+\lambda_i\,\beta_i)^{N_i}}{(1-\lambda_i+\lambda_i\,\beta_i)^{N_i-n_{i0}}}. \tag{22}$$

Taking $q'=q+c-1$, by (11) we get

$$\begin{aligned}\frac{h(q',\vec{m})}{h(q,\vec{n})} &= \exp\{\theta(q'-q)\}\times\prod_{i=1}^{M}\frac{r_i(\vec{m}_i)}{r_i(\vec{n}_i)} \\ &= \exp\{\theta\,(c-1)\}\times\prod_{i\in A_b}\frac{r_i(\vec{m}_i)}{r_i(\vec{n}_i)} \qquad \text{since } r_i\equiv 1\ \forall i\in A_n.\end{aligned}$$

Thus the ratio $h(q', \vec{m})/h(q, \vec{n})$ is equal to

$$\exp\{\theta\, c\} \times \prod_{i \in A_h} (e^{\theta}\, r_i + 1 - r_i)^{-n_{i0}} \times$$
$$\prod_{i \in A_h \setminus A_{h1}} \left[\frac{(\beta_i^{-1})^{(n_{i0} - m_{i,d_i-1})}\, (e^{\alpha_i\,(d_i-1)})^{m_{d_i-1}}}{((1-\mu_i)\, e^{\alpha_i\,(d_i-1)} + \mu_i\, \beta_i^{-1})^{n_{i0}}} \cdot \frac{\beta_i^{(m_{i0}-n_{i1})}}{(1 - \lambda_i + \lambda_i\, \beta_i)^{f_i}} \right] \times$$
$$\prod_{i \in A_{h1}} \left[\beta_i^{m_{i0}} \left(\frac{\beta_i^{-1}}{1 - \mu_i + \mu_i\, \beta_i^{-1}} \right)^{n_{i0}} \cdot (1 - \lambda_i + \lambda_i\, \beta_i)^{n_{i0} - N_i} \right] \times$$
$$\prod_{i \in A_n \setminus A_{n1}} 1 \times$$
$$\prod_{i \in A_{n1}} 1.$$

Note, if $\sum_{i \in A_h} z_i = c$ and if $x + y = m_{i0}$ for all $i \in A_{h1}$, then the ratio $h(q', \vec{m})/h(q, \vec{n})$ becomes

$$\prod_{i \in A_h} e^{\theta\, z_i}\, (e^{\theta}\, r_i + 1 - r_i)^{-n_{i0}} \times$$
$$\prod_{i \in A_h \setminus A_{h1}} \left[\frac{(\beta_i^{-1})^{(n_{i0} - m_{i,d_i-1})}\, (e^{\alpha_i\,(d_i-1)})^{n'_{d_i-1}}}{((1-\mu_i)\, e^{\alpha_i\,(d_i-1)} + \mu_i\, \beta_i^{-1})^{n_{i0}}} \cdot \frac{\beta_i^{(m_{i0}-n_{i1})}}{(1 - \lambda_i + \lambda_i\, \beta_i)^{f_i}} \right] \times$$
$$\prod_{i \in A_{h1}} \left[\frac{(\beta_i^{-1})^{n_{i0}-x}}{(1 - \mu_i + \mu_i\, \beta_i^{-1})^{n_{i0}}} \cdot \frac{\beta_i^{y}}{(1 - \lambda_i + \lambda_i\, \beta_i)^{N_i - n_{i0}}} \right] \times$$
$$\prod_{i \in A_n \setminus A_{n1}} 1 \times$$
$$\prod_{i \in A_{n1}} 1.$$

Now the transition kernel for the free MA-chain is

$$
\begin{aligned}
&\mathrm{K}^{\infty}[(q,\vec{n}),(q',\vec{m})] \\
&= \mathrm{P}(\xi[1]=k|M[0]=\vec{n}) \times \prod_{i\in A_h} \mathrm{K}_i(\vec{n}_i,\vec{m}_i) \\
&= \left[\sum_{\vec{z}\in S_c[\vec{n}]} \prod_{i\in A_h} \mathrm{Bin}(n_{i0},z_i,r_i)\right] \times \\
&\prod_{i\in A_h\setminus A_{h1}} [\mathrm{Bin}(n_{i0},m_{i,d_i-1},1-\mu_i)\cdot \mathrm{Bin}(f_i,m_{i0}-n_{i1},\lambda_i)] \times \\
&\prod_{i\in A_{h1}} \left[\sum_{x+y=m_{i0}} \mathrm{Bin}(n_{i0},x,1-\mu_i)\cdot \mathrm{Bin}(N_i-n_{i0},y,\lambda_i)\right] \\
&\prod_{i\in A_n\setminus A_{n1}} [\mathrm{Bin}(n_{i0},m_{i,d_i-1},1-\mu_i)\cdot \mathrm{Bin}(f_i,m_{i0}-n_{i1},\lambda_i)] \times \\
&\prod_{i\in A_{n1}} \left[\sum_{x+y=m_{i0}} \mathrm{Bin}(n_{i0},x,1-\mu_i)\cdot \mathrm{Bin}(N_i-n_{i0},y,\lambda_i)\right],
\end{aligned}
$$

where $A_{n1}\subset A_n$ is the index set for the groups of sources which do not offer cells to the workload of the hot spot with inter-arrival times equal to one, i.e. $d_i=1$ for all $i\in A_{n1}$. For readability, let

$$
(\lambda\,\beta)_i := \frac{\lambda_i\,\beta_i}{1-\lambda_i+\lambda_i\,\beta_i} \quad \text{and} \quad (\mu\,\beta^{-1})_i := \frac{\mu_i\,\beta_i^{-1}}{(1-\mu_i)\,e^{\alpha_i\,(d_i-1)}+\mu_i\,\beta_i^{-1}}.
$$

Thus the operator $\mathcal{K}^{\infty}$ as defined by (10), that is

$$
\mathcal{K}^{\infty}[(q,\vec{n}),(q',\vec{m})] = \mathrm{K}^{\infty}[(q,\vec{n}),(q',\vec{m})]\,\frac{h(q',\vec{m})}{h(q,\vec{n})},
$$

is equal to

$$\left[\sum_{\vec{x}\in S_c[\vec{n}]}\prod_{i\in A_h}\mathrm{Bin}\left(n_{i0},z_i,\frac{e^\theta r_i}{1-r_i+e^\theta r_i}\right)\right]\times$$
$$\prod_{i\in A_h\setminus A_{h1}}\left[\mathrm{Bin}\left(n_{i0},m_{i,d_i-1},1-(\mu\beta^{-1})_i\right)\cdot\mathrm{Bin}\left(f_i,m_{i0}-n_{i1},(\lambda\beta)_i\right)\right]\times$$
$$\prod_{i\in A_{h1}}\left[\sum_{x+y=m_{i0}}\mathrm{Bin}\left(n_{i0},x,1-(\mu\beta^{-1})_i\right)\cdot\mathrm{Bin}\left(N_i-n_{i0},y,(\lambda\beta)_i\right)\right]\times$$
$$\prod_{i\in A_n\setminus A_{n1}}\left[\mathrm{Bin}(n_{i0},m_{i,d_i-1},1-\mu_i)\cdot\mathrm{Bin}(f_i,m_{i0}-n_{i1},\lambda_i)\right]\times$$
$$\prod_{i\in A_{n1}}\left[\sum_{x+y=m_{i0}}\mathrm{Bin}(n_{i0},x,1-\mu_i)\cdot\mathrm{Bin}(N_i-n_{i0},y,\lambda_i)\right].$$

Compare the operator $\mathcal{K}^\infty$ with the transition kernel K^∞. The only difference between the two operators are the traffic parameters μ_i, λ_i, and r_i (for K^∞ and $i\in A_h$) which have been changed to $(\mu\beta)_i=(\mu_i\beta_i^{-1})/((1-\mu_i)\,e^{\alpha_i(d_i-1)}+\mu_i\beta_i^{-1})$, $(\lambda\beta)_i=(\lambda_i\beta_i)/(1-\lambda_i+\lambda_i\beta_i)$ and $(e^\theta r_i)/(e^\theta r_i+1-r_i)$, respectively for $\mathcal{K}^\infty$. Notice that all of the new parameters are between zero and one, hence by observation

$$\sum_{(q',\vec{m})}\mathrm{K}^\infty[(q,\vec{n}),(q',\vec{m})]\,\frac{h(q',\vec{m})}{h(q,\vec{n})}=1.$$

Theorem 3.1 (Harmonicity) *Suppose there exists positive real numbers θ and β_i for all $i\in A_h$ and real numbers α_i for $i\in A_h$, that satisfy the following system of equations,*

$$\begin{aligned}
e^\theta &= \prod_{i\in A_h}(1-\lambda_i+\lambda_i\beta_i)^{N_i},\\
e^{-\alpha_i} &= ((1-\mu_i)\,\exp\{\alpha_i(d_i-1)\}+\mu_i\beta_i^{-1})\,(e^\theta r_i+1-r_i)\quad\forall i\in A_h\\
e^{-\alpha_i} &= 1-\lambda_i+\lambda_i\beta_i\qquad\forall i\in A_h.
\end{aligned}$$

The positive function $h:\mathbb{Z}\times\mathcal{S}_A\to\mathbb{R}^+$, defined as $(q,\vec{n})\mapsto\exp(\theta q)\,r(\vec{n})$ (where r is the function defined on page (33)), is a harmonic function for the transition kernel

K^∞. Moreover, the operator $\mathcal{K}^\infty$ defined as

$$\mathcal{K}^\infty[(q,\vec{n}),(q',\vec{m})] := K^\infty[(q,\vec{n}),(q',\vec{m})]\,\frac{h(q',\vec{m})}{h(q,\vec{n})} \tag{23}$$

is a probability transition kernel.

We call $\mathcal{K}^\infty$ the twisted kernel, or the h-transform of K^∞ and we call the process it generates the twisted MA-chain. Let φ_A be the stationary distribution for the Markovian part of the twisted MA-chain. This new chain describes the same ATM switching fabric as the original chain but with different traffic parameters, thus criterion **(C1)** is satisfied. Table 1 indicates the changes for the BIDIT parameters under the twist. The deterministic inter-arrival time between possible arrivals during a burst period, d_i, doesn't change under the twist. The other BIDIT parameters, (λ_i, μ_i, r_i), also remain the same for sources that do not generate cells directed to the hot spot. Although if the sources in group i generate cells directed to the hot spot, then the parameters (λ_i, μ_i, r_i) change under the twist, see Table 1.

Sources in group i generate cells directed to the hot spot		
Description	Original	Twisted
deterministic inter-arrival time during burst periods	d_i	d_i
transition probability for the modulating Markov chain (OFF to ON)	λ_i	$(\lambda_i \beta_i)/(1 - \lambda_i + \lambda_i \beta_i)$
transition probability for the modulating Markov chain (ON to OFF)	μ_i	$(\mu_i \beta_i^{-1})/((1 - \mu_i)\, e^{\alpha_i (d_i - 1)} + \mu_i \beta_i^{-1})$
probability of offering a cell when ON with a delay of zero	r_i	$(e^{\theta} r_i)/(e^{\theta} r_i + 1 - r_i)$

Sources in group i generate cells **not** directed to the hot spot		
Description	Original	Twisted
deterministic inter-arrival time during burst periods	d_i	d_i
transition probability for the modulating Markov chain (OFF to ON)	λ_i	λ_i
transition probability for the modulating Markov chain (ON to OFF)	μ_i	μ_i
probability of offering a cell when ON with a delay of zero	r_i	r_i

Table 1: The original and twisted BIDIT parameters

3.3 Positive Drift

In Section 3.2, we found a simple transform of our ATM switching fabric thus satisfying criterion (**C1**). The second criterion (**C2**) demands that hot spot buffer overflows be likely under the twist. Under the assumption that the hot spot buffer grows to infinity with the workload of the hot spot, the second criterion is satisfied if the mean increment of the twisted MA-chain $\{\mathcal{W}^\infty, \vec{\mathcal{N}}\}$ is positive.

In Example 3.1, we notice that the associated (twisted) random walk has a positive mean increment when the original random walk has a negative mean increment. Similarly this result holds for the mean increment of the twisted MA-chain, which is

$$\tilde{d}_1 := \sum_{\vec{n}\in\mathcal{S}_A} \varphi_A(\vec{n}) \mathrm{E}_{(0,\vec{n})}[\xi[1]],$$

where φ_A is the stationary distribution for the Markovian part of the twisted MA-chain.

Define the function

$$\Psi(\gamma,\xi) := \mathrm{E}_{(0,\vec{n})}[\exp(\gamma\, W^\infty[\tau_{\vec{n}}] - \xi\, \tau_{\vec{n}})],$$

with domain of convergence

$$\mathcal{D} := \{(\gamma,\xi) \in \mathbb{R}^2 : \Psi(\gamma,\xi) < \infty\}.$$

The increment for the additive part W^∞ is bounded by the number of possible cells offered to the workload of the hot spot during a time slot which is finite. Thus, $\widehat{K}_\gamma(\vec{n},\vec{m}) < \infty$ for all $\gamma \in \mathbb{R}$, $\vec{n},\vec{m} \in \mathcal{S}_A$ and the sets

$$D(\vec{n},\vec{m}) := \{\gamma \in \mathbb{R} : \widehat{K}_\gamma(\vec{n},\vec{m}) < \infty\} = \mathbb{R} \tag{24}$$

are open for all $\vec{n},\vec{m} \in \mathcal{S}_A$. By Theorem A.4, the set $\mathcal{D}$ is open.

Since the set $\mathcal{D}$ is open, and from (6), the mean increment is negative, that is

$$\sum_{\vec{n}\in\mathcal{S}_A} \pi_A(\vec{n}) \mathrm{E}_{(0,\vec{n})}[\xi[1]] = \sum_{i\in A_h} N_i \frac{\lambda_i}{(d_i + (1-d_i)\,\mu_i)\,\lambda_i + \mu_i} r_i - 1 < 0,$$

by the Drift Lemma 2.3 in McDonald (1998), the mean increment for the twisted MA-chain is positive ($\tilde{d}_1 > 0$). Actually, we show directly that $\tilde{d}_1 > 0$ at the end of Chapter 7.

3.4 Finding the Twist *via* Numerical Methods

To find the twisted probabilities we need to obtain positive real numbers θ, β_i for all $i \in A_h$ and real numbers α_i for all $i \in A_h$ that satisfy the following system of equations

$$e^{\theta} = \prod_{i \in A_h} (1 - \lambda_i + \lambda_i \beta_i)^{N_i}, \tag{25}$$

$$e^{-\alpha_i} = ((1 - \mu_i) \exp\{\alpha_i (d_i - 1)\} + \mu_i \beta_i^{-1}) (e^{\theta} r_i + 1 - r_i) \quad \forall i \in A_h \tag{26}$$

$$e^{-\alpha_i} = 1 - \lambda_i + \lambda_i \beta_i \qquad \forall i \in A_h. \tag{27}$$

In this section, we show that this non-linear system of equations can be reduced to finding the roots of a finite number of univariate equations, which can easily be solved using numerical methods.

In Section 7.2, we define $\alpha_i(\cdot)$ and $\beta_i(\cdot)$ (for all $i \in A_h$) as functions of γ. Also we show that the positive real number θ is the unique positive root of the strictly convex function

$$\Lambda(\gamma) := -\gamma - \sum_{i \in A_h} N_i \alpha_i(\gamma),$$

and that θ, $\alpha_i(\theta)$ and $\beta_i(\theta)$ (for all $i \in A_h$) satisfy the equations (25-27).

If we can calculate $\alpha_i(\gamma)$ for any positive γ, then we can use the Bisection Algorithm 3.1 to obtain θ numerically.

Algorithm 3.1 (Bisection)

B1 Find two positive real numbers x_0 *and* x_0' *such that* $\Lambda(x_0) < 0 < \Lambda(x_0')$ *and* $0 < x_0 < x_0'$, *see Figure 8.*

B2 Pick a new pair of positive numbers (x_{n+1}, x_{n+1}') *using*

$$(x_{n+1}, x_{n+1}') = \begin{cases} ((x_n + x_n')/2, x_n'), & \text{if } \Lambda((x_n + x_n')/2) < 0 \\ (x_n, (x_n + x_n')/2), & \text{if } \Lambda((x_n + x_n')/2) > 0. \end{cases}$$

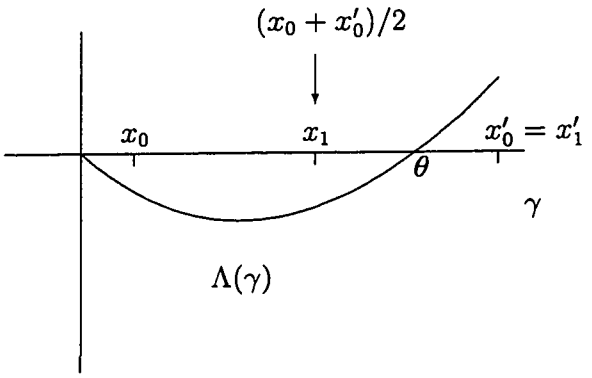

Figure 8: Bisection Method

B3 If $|(x_{n+1} + x'_{n+1})/2 - (x_n + x'_n)/2|/|(x_{n+1} + x'_{n+1})/2| < \epsilon$, *where* $\epsilon > 0$ *is some tolerance level, then STOP and* $\theta = (x_{n+1} + x'_{n+1})/2$, *else return to B2.*

Fix $i \in A_h$. We can obtain $\alpha_i(\gamma)$ explicitly or implicitly using numerical methods for any positive γ.

Case 1: ($\lambda_i = 1, \mu_i = 0$, i.e. the sources in group i are always ON)
Using (43), we have

$$\alpha_i(\gamma) = -\frac{1}{d_i} \log(e^\gamma r_i + 1 - r_i). \tag{28}$$

Case 2: ($0 < \lambda_i, \mu_i < 1$, and $d_i = 1$)
Using (45), we have

$$\alpha_i(\gamma) := -\log\left\{\frac{(1-\lambda_i) + (1-\mu_i)\,(e^\gamma r_i + 1 - r_i) + \sqrt{D_i(\gamma)}}{2}\right\}, \tag{29}$$

where

$$\begin{aligned} D_i(\gamma) := {} & [-(1 \dot{-} \lambda_i) - (1-\mu_i)\,(e^\gamma r_i + 1 - r_i)]^2 \\ & - 4\,[-\mu_i \lambda_i\,(e^\gamma r_i + 1 - r_i) + (1-\mu_i)\,(1-\lambda_i)\,(e^\gamma r_i + 1 - r_i)]. \end{aligned}$$

Case 3: ($0 < \lambda_i, \mu_i < 1$, and $d_i > 1$)
Consider the function Ξ_i, defined as follows

$$\begin{aligned}\Xi_i(\alpha_i, \gamma) := &(1-\lambda_i)\, e^{\alpha_i} + (1-\mu_i)\, e^{\alpha_i d_i}\, (e^{\gamma} r_i + 1 - r_i) \\ &- (1-\mu_i)\,(1-\lambda_i)\, e^{\alpha_i (d_i+1)}\, (e^{\gamma} r_i + 1 - r_i) \\ &+ \mu_i \lambda_i e^{2\alpha_i}\, (e^{\gamma} r_i + 1 - r_i).\end{aligned}$$

For all $\gamma > 0$, we define $\alpha_i(\gamma)$ implicitly as the unique negative root of the function

$$\varphi_i^{\gamma}(\alpha_i) := \Xi_i(\alpha_i, \gamma) - 1, \tag{30}$$

see (48). Note, the function φ_i^{γ} is continuous, so we can easily find its unique negative root, i.e. $\alpha_i(\gamma)$, using numerical methods.

It remains to show that we can find $\beta_i(\theta)$ for all $i \in A_h$. Using (27), we have

$$\beta_i(\theta) = \frac{\exp(-\alpha_i(\theta)) + \lambda_i - 1}{\lambda_i}. \tag{31}$$

In Example 3.3, we find the twist numerically for a model with two groups of BIDIT sources. The sources in the two groups will have a deterministic inter-arrival time equal to one and greater than one, respectively, i.e. $d_1 = 1$ and $d_2 > 1$.

Example 3.3 *Consider a model with two groups of BIDIT sources. The group* 1 *has two sources with BIDIT parameters equal to* $(d_1, \lambda_1, \mu_1, r_1) = (1, 0.001, 0.0001, 0.25)$ *and the group* 2 *has two sources with BIDIT parameters equal to* $(d_2, \lambda_2, \mu_2, r_2) = (5, 0.004, 0.0001, 0.25)$.

For group 1, *the function* $\alpha_1(\cdot)$ *is known explicitly, see (29). For group* 2, *for any positive* γ, $\alpha_2(\gamma)$ *is the unique negative root of the function* φ_i^{γ} *defined by (30).*

The constant θ *is the unique positive root of the function*

$$\Lambda(\gamma) = -\gamma - 2\,\alpha_1(\gamma) - 2\,\alpha_2(\gamma).$$

We do not know Λ *explicitly, although for any positive* γ *we can find* $\Lambda(\gamma)$ *numerically. Figure 9 displays a plot of* $\Lambda(\gamma)$ *calculated at the points* $\gamma = n/20$ *for* $n = 1, \ldots, 32$. *From this graph, we see that* $1.4 < \theta < 1.6$.

Using the mathematical package Maple and the Bisection Algorithm 3.1, with floating points of 15 digits, a tolerance level of 10^{-10}, *and initial points* $x_0 = 1.4$ *and* $x_0' = 1.6$, *the algorithm stops after* 32 *iterations. We obtain* $\theta = 1.4942241373$, $\alpha_1(\theta) = -0.62260847243$, *and* $\alpha_2(\theta) = -0.1245035963$. *Using (31), we get* $\beta_1(\theta) = 864.7833340879$, *and* $\beta_2(\theta) = 34.1465233790$.

Using the twisted probabilities from Table 1, the sources from group 1 *and group* 2 *have respectively been transformed to BIDIT sources with parameters*

$$(1, 0.46, 0.12 \times 10^{-6}, 0.60) \text{ and } (5, 0.12, 0.96 \times 10^{-5}, 0.60).$$

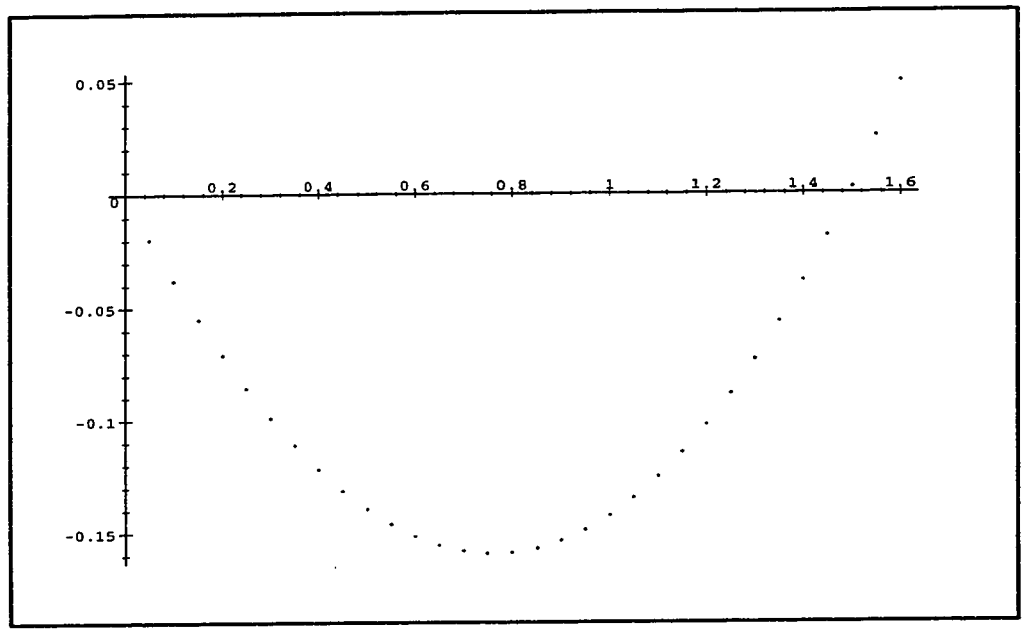

Figure 9: Plot: $\Lambda(\gamma)$ versus γ, for Example 3.3

Chapter 4

Importance Sampling

Heidelberger (1995) explains how importance sampling could be adapted to accelerate the simulation of queueing models such as ATM, described as the "splitting" A-cycle method. Our importance sampling algorithm is similar to this A-cycle method, which was also described and applied by Bonneau (1996) to accelerate the simulation of an ATM multiplexer. We will generalize this method in the sense that we will extend it to our switching fabric model. Our method is referred to as the "splitting" Δ-cycle method. The topology of an ATM switching fabric also allows us to introduce a "super-cycle" (Δ_0-cycle) which reduces the number of aborted cycles, i.e. cycles wherein the length of the hot spot does not reach its finite capacity l. Essentially, in this chapter we show the twisted process can be used in a straightforward manner to compute the cell loss ratio in the original ATM fabric, thus satisfying the criterion **(C3)**.

The splitting Δ-cycle method exploits the fact that the twisted process has positive drift, i.e. the workload of the hot spot tends to infinity. Essentially we use the twisted process to overload the hot spot. A new Δ-cycle, as in Section 2.5, starts at each successive return to Δ; Δ is the set of all states with an empty hot spot buffer. A splitting Δ-cycle consists of two Δ-cycles (the twisted Δ-cycle and the original Δ-cycle), both starting from the same state in Δ. The twisted Δ-cycle uses the twisted probabilities (from Table 1) until the workload of the hot spot reaches

(or exceeds) the critical value l, where l is the capacity of the hot spot buffer. Notice that unlike the single queue Example 3.2, we "untwist" the sources (i.e. return to the original probabilities) when the workload reaches the critical length l and not when the hot spot buffer itself reaches the length l. During an original Δ-cycle, as the name suggests, the original transition probabilities are used. We use the original Δ-cycle for purposes of stationarity. Forcing a twisted Δ-cycle to start in the same state as the original Δ-cycle associated to it ensures that the twisted Δ-cycle starts in steady state (for the original untwisted queueing model). As we will see this is essential.

The twisted process $\{\mathcal{W}^\infty, \vec{\mathcal{N}}\}$ is the conjugate process to the free MA-chain $\{W^\infty, \vec{N}\}$, which has the same underlying probability structure as the Markov additive structures $\{Q, \vec{N}\}$ (Q is the workload of the hot spot buffer) embedded between return times to Δ, as described in Section 3.1. This justifies the use of the twisted probabilities as described in the preceding paragraph, since a Δ-cycle is a possible trajectory for the ATM fabric between successive returns to an empty hot spot buffer.

Recall the CLR for the hot spot buffer is the long-run average number of cells lost (per time slot) by the hot spot buffer divided by the long-run average number of cells offered (per time slot) to the hot spot buffer. The numerator is equal the long-run average number of cells lost per Δ-cycle divided the length of the long-run average length of a Δ-cycle. We can estimate it using a Δ-cycle simulation technique.

4.1 Infinite Upstream Buffers

Following the order established in Section 2.5, we first consider infinite upstream buffers to the hot spot buffer. Only the hot spot buffer can overflow. Let Ω_Δ be the sample space for the Δ-cycle trajectories. The elements of Ω_Δ can take two forms. If the workload never reaches the critical value l during the Δ-cycle $\omega \in \Omega_\Delta$ then ω is of the form $\{w_0, w_1, \ldots, w_r\}$, where $w_k = (q_k, m_k)$ (q_k is the workload and m_k is the state of the sources). If the workload reaches (or exceeds) the critical value l during the Δ-cycle $\omega \in \Omega_\Delta$ then ω is of the form $\{w_0, w_1, \ldots, w_f, c_{f+1}, w_{f+1}, c_{f+2}, w_{f+2}, \ldots, c_r, w_r\}$,

where c_k represents the number of cells lost in the hot spot associated to the one-step transition w_{k-1} to w_k and f is the smallest integer such that $q_f \geq l$. As in Example 3.2, we redirect $(q_k + c_k, m_k)$ to the state $w_k = (q_k, m_k)$.

Remark: We use the symbol $\oplus$ as a binary operation defined as $(q_k, m_k) \oplus c_k = (q_k + c_k, m_k)$.

Consider a Δ-cycle generated with the original source parameters it has probability mass function

$$\nu(\omega) := \pi(w_0) \prod_{k=1}^{r} \mathrm{K}(w_{k-1}, w_k),$$

if ω is of the first form, or

$$\nu(\omega) := \pi(w_0) \prod_{k=1}^{f} \mathrm{K}(w_{k-1}, w_k \oplus c_k) \prod_{k=f+1}^{r} \mathrm{K}(w_{k-1}, w_k \oplus c_k),$$

(where $c_0 = \ldots = c_{f-1} = 0$) if ω is of the second form. Consider a Δ-cycle generated as a twisted Δ-cycle then it has probability mass function

$$\nu'(\omega) := \pi(w_0) \prod_{k=1}^{r} \mathcal{K}(w_{k-1}, w_k),$$

if ω is of the first form, or

$$\nu'(\omega) := \pi(w_0) \left(\prod_{k=1}^{f} \mathcal{K}^{\infty}(w_{k-1}, w_k \oplus c_k) \right) \left(\prod_{k=f+1}^{r} \mathrm{K}^{\infty}(w_{k-1}, w_k \oplus c_k) \right),$$

if ω is of the second form. Notice that the source parameters are "untwisted" when the workload reaches (or exceeds) the critical length l.

If ω is of the first form then

$$\begin{aligned} \nu(\omega) &= \pi(w_0) \prod_{k=1}^{r} \mathrm{K}(w_{k-1}, w_k) \\ &= \pi(w_0) \prod_{k=1}^{r} \mathcal{K}(w_{k-1}, w_k) \frac{h(w_{k-1})}{h(w_k)}. \end{aligned}$$

Simplication gives

$$\nu(\omega) = \frac{h(w_0)}{h(w_r)}\,\pi(w_0)\prod_{k=1}^{r}\mathcal{K}(w_{k-1},w_k) = \frac{h(w_0)}{h(w_r)}\,\nu'(\omega).$$

Similarly, if ω is of the second form then

$$\begin{aligned}\nu(\omega) &= \pi(w_0)\left(\prod_{k=1}^{f}\mathrm{K}(w_{k-1},w_k\oplus c_k)\right)\left(\prod_{k=f+1}^{r}\mathrm{K}(w_{k-1},w_k\oplus c_k)\right)\\ &= \pi(w_0)\left(\prod_{k=1}^{f}\mathcal{K}(w_{k-1},w_k\oplus c_k)\,\frac{h(w_{k-1})}{h(w_k\oplus c_k)}\right)\left(\prod_{k=f+1}^{r}\mathrm{K}(w_{k-1},w_k\oplus c_k)\right).\end{aligned}$$

Since $c_0 = \ldots = c_{f-1} = 0$, cancellation gives

$$\begin{aligned}\nu(\omega) &= \frac{h(w_0)}{h(w_f\oplus c_f)}\,\pi(w_0)\left(\prod_{k=1}^{f}\mathcal{K}^{\infty}(w_{k-1},w_k\oplus c_k)\right)\left(\prod_{k=f+1}^{r}\mathrm{K}^{\infty}(w_{k-1},w_k\oplus c_k)\right)\\ &= \frac{h(w_0)}{h(w_f\oplus c_f)}\,\nu'(\omega).\end{aligned}$$

We define the function $\mathcal{L}$ on Ω_Δ as

$$\mathcal{L}(\omega) = \begin{cases} h(w_r)/h(w_0), & \text{if } \omega \text{ is of the first form}\\ h(w_f\oplus c_f)/h(w_0), & \text{if } \omega \text{ is of the second form.}\end{cases}$$

We call the function $\mathcal{L}$ the likelihood ratio since $\nu(\omega) = \mathcal{L}(\omega)\,\nu'(\omega)$ for all $\omega\in\Omega_\Delta$.

Let $V(\omega)$ be the number of cells lost in the hot spot buffer in Δ-cycle ω. The expected number of cells lost per Δ-cycle is equal to

$$\mathrm{E}_\nu[V] = \sum_{w\in\Omega_\Delta} V(\omega)\,\nu(\omega) = \sum_{w\in\Omega_\Delta} V(\omega)\,\mathcal{L}(\omega)\,\nu'(\omega) = \mathrm{E}_{\nu'}[\mathcal{L}\,V] \tag{32}$$

If $CCL_\Delta[n]$ is the number of cells lost during the nth twisted Δ-cycle and $\mathcal{L}_n$ is the likelihood ratio associated to that same Δ-cycle then by (32)

$$\frac{1}{N}\sum_{n=1}^{N} CCL_\Delta[n]\cdot\mathcal{L}_n$$

is an unbiased estimator for the expected number of cells lost per Δ-cycle, since each twisted Δ-cycle starts in steady state.

If $L_\Delta[n]$ is the length of the n-th original Δ-cycle then

$$\frac{1}{N}\sum_{n=1}^{N} L_\Delta[n]$$

is an estimator for the expected length of a Δ-cycle. Hence the long-run average number of cells lost (per time slot) can be estimated by

$$\left(\frac{1}{N}\sum_{n=1}^{N} CCL_\Delta[n]\cdot\mathcal{L}_n\right) \Big/ \left(\frac{1}{N}\sum_{n=1}^{N} L_\Delta[n]\right).$$

Since the upstream buffers to the hot spot buffer are infinite the long-run average number of cells offered to the hot spot buffer is equal to the number of cells offered to the workload of the hot spot buffer which is

$$\sum_{i\in A_h} N_i \frac{\lambda_i}{(d_i+(1-d_i)\,\mu_i)\,\lambda_i+\mu_i}\,r_i.$$

Proposition 4.1 *A consistent estimator for the cell loss ratio of the hot spot buffer is*

$$\left(\frac{(1/N)\sum_{n=1}^{N} CCL_\Delta[n]\cdot\mathcal{L}_n}{(1/N)\sum_{n=1}^{N} L_\Delta[n]}\right) \Big/ \left(\sum_{i\in A_h} N_i \frac{\lambda_i}{(d_i+(1-d_i)\,\mu_i)\,\lambda_i+\mu_i}\,r_i\right)$$

when the upstream buffers to the hot spot buffer are infinite.

4.2 Finite Upstream Buffers

As remarked in Section 2.5, in reality the buffers in an ATM switching fabric can only hold a finite number of cells. But we ignore this fact when constructing the twist. We assume that the increment for the workload (at each time slot) is the number of cells offered to the workload minus a service, i.e. A[n]-1. From this we should subtract the number of cells lost in the buffer upstream to the hot spot buffer which would

have eventually passed through the hot spot buffer. Since we do not account for these losses the likelihood ratio $\mathcal{L}$ as defined for infinite buffers is not correct, that is in general $\nu(\omega)$ is not equal to $\mathcal{L}(\omega)\,\nu(\omega)$. We show that we can multiply the likelihood ratio by an extra factor to compensate for these cell losses.

Let Ω_Δ be the sample space for the Δ-cycle trajectories. The elements of Ω_Δ can take two forms. If the workload never reaches the critical value l during the Δ-cycle $\omega \in \Omega_\Delta$ then ω is of the form $\{w_0, c_1, w_1, \ldots, c_r, w_r\}$, where $w_k = (q_k, m_k)$ (q_k is the workload and m_k is the state of the sources) and c_k represents the number of cells lost in the workload of the hot spot associated to the one-step transition w_{k-1} to w_k. If the workload reaches (or exceeds) the critical value l during the Δ-cycle $\omega \in \Omega_\Delta$ then ω is of the form $\{w_0, c_1, w_1, \ldots, w_f, c_{f+1}, w_{f+1}, c_{f+2}, w_{f+2}, \ldots, c_r, w_r\}$, where f is the smallest integer such that $q_f \geq l$. Note that the first form is simply a special case of the second, so we will only consider Δ-cycles of the second form.

Consider a Δ-cycle generated with the original source parameters, it has probability mass function

$$\nu(\omega) := \pi(w_0) \prod_{k=1}^{f} \mathrm{K}(w_{k-1}, w_k \oplus c_k) \prod_{k=f+1}^{r} \mathrm{K}(w_{k-1}, w_k \oplus c_k).$$

If a Δ-cycle is generated as a twisted Δ-cycle then it has probability mass function is

$$\nu'(\omega) := \pi(w_0) \prod_{k=1}^{f} \mathcal{K}(w_{k-1}, w_k \oplus c_k) \prod_{k=f+1}^{r} \mathrm{K}(w_{k-1}, w_k \oplus c_k),$$

As in the preceding section, we have

$$\begin{aligned}
\nu(\omega) &= \pi(w_0) \prod_{k=1}^{f} \mathrm{K}(w_{k-1}, w_k \oplus c_k) \prod_{k=f+1}^{r} \mathrm{K}(w_{k-1}, w_k \oplus c_k) \\
&= \pi(w_0) \prod_{k=1}^{f} \mathcal{K}(w_{k-1}, w_k \oplus c_k) \frac{h(w_{k-1})}{h(w_k \oplus c_k)} \prod_{k=f+1}^{r} \mathrm{K}(w_{k-1}, w_k \oplus c_k).
\end{aligned}$$

Since $h(q,m) = \exp(\theta\, q)\, r(m)$, we have

$$h(w_k \oplus c_k) = \exp(\theta\,(q_k + c_k))\, r(m_k) = \exp(\theta\, c_k)\, h(w_k).$$

Hence $\nu(\omega)$ is equal to

$$\pi(w_0) \prod_{k=1}^{f} \mathcal{K}(w_{k-1}, w_k \oplus c_k)\, \exp(-\theta\, c_k) \frac{h(w_{k-1})}{h(w_k)} \prod_{k=f+1}^{r} \mathrm{K}(w_{k-1}, w_k \oplus c_k).$$

Simplification gives

$$\begin{aligned}\nu(\omega) &= \exp\left\{-\theta \sum_{k=1}^{f} c_k\right\} \frac{h(w_0)}{h(w_f)}\, \pi(w_0) \prod_{k=1}^{f} \mathcal{K}(w_{k-1}, w_k \oplus c_k) \prod_{k=f+1}^{r} \mathrm{K}(w_{k-1}, w_k \oplus c_k) \\ &= \exp\left\{-\theta \sum_{k=1}^{f} c_k\right\} \frac{h(w_0)}{h(w_f)}\, \nu'(\omega).\end{aligned}$$

We define the partial likelihood ratio $\mathcal{L}$ on Ω_Δ as

$$\mathcal{L}(\omega) = \frac{h(w_0)}{h(w_f)},$$

and the extra factor $\mathcal{C}$ on Ω_Δ as

$$\mathcal{C}(\omega) = \exp\left\{-\theta \sum_{k=1}^{f} c_k\right\},$$

hence $\nu(\omega) = \mathcal{C}(\omega)\, \mathcal{L}(\omega)\, \nu'(\omega)$ for all $\omega \in \Omega_\Delta$.

Thus, similar to (32), for infinite upstream buffers, the following is true when we have finite upstream buffers to the hot spot buffer:

$$\mathrm{E}_\nu[V] = \mathrm{E}_{\nu'}[\mathcal{C}\, \mathcal{L}\, V], \tag{33}$$

where $V(\omega)$ is the number of cells lost in the hot spot buffer during the Δ-cycle ω. If $CCL_\Delta[n]$ is the number of cells lost during the nth twisted Δ-cycle and $\mathcal{L}_n$ and

$\mathcal{C}_n$ are the partial likelihood ratio and the extra factor respectively associated to that same Δ-cycle then by (33) and since each twisted Δ-cycle starts in steady state,

$$\frac{1}{N}\sum_{n=1}^{N} CCL_{\Delta}[n]\cdot\mathcal{L}_n\cdot\mathcal{C}_n$$

is an unbiased estimator for the expected number of cells lost per Δ-cycle. If $L_{\Delta}[n]$ is the length of the n-th original Δ-cycle then

$$\frac{1}{N}\sum_{n=1}^{N} L_{\Delta}[n]$$

is an estimator for the expected length of a Δ-cycle. Hence the long-run average number of cells lost (per time slot) can be estimated by

$$\left(\frac{1}{N}\sum_{n=1}^{N} CCL_{\Delta}[n]\cdot\mathcal{L}_n\cdot\mathcal{C}_n\right)\Big/\left(\frac{1}{N}\sum_{n=1}^{N} L_{\Delta}[n]\right).$$

Since the upstream buffers to the hot spot are finite, the long-run average number of cells offered to the workload is not necessarily the same as the long-run average number of cells offered to the hot spot. If $COT_{\Delta}[n]$ is the number of cells offered to the hot spot buffer at the nth original Δ-cycle then

$$\frac{(1/N)\sum_{n=1}^{N} COT_{\Delta}[n]}{(1/N)\sum_{n=1}^{N} L_{\Delta}[n]}$$

is a consistent estimator for the long-run offered traffic to the hot spot (per time slot).

Proposition 4.2 *A consistent estimator for the cell loss ratio of the hot spot buffer is*

$$\widehat{CLR}(N) := \frac{(1/N)\sum_{n=1}^{N} CCL_{\Delta}[n]\cdot\mathcal{L}_n\cdot\mathcal{C}_n}{(1/N)\sum_{n=1}^{N} COT_{\Delta}[n]},$$

when the upstream buffers to the hot spot buffer are finite.

Proof: The estimator $\widehat{CLR}(N)$ is equal to

$$\left(\frac{(1/N)\sum_{n=1}^{N} CCL_{\Delta}[n]\cdot\mathcal{L}_n\cdot\mathcal{C}_n}{(1/N)\sum_{n=1}^{N} L_{\Delta}[n]}\right)\Big/\left(\frac{(1/N)\sum_{n=1}^{N} COT_{\Delta}[n]}{(1/N)\sum_{n=1}^{N} L_{\Delta}[n]}\right),$$

which is a consistent estimator for cell loss ratio of the hot spot buffer.

□

To summarize, we present our importance sampling algorithm.

Algorithm 4.1 (Splitting Δ-Cycle)

IS1 *Start the simulation in some state in Δ so the hot spot buffer is empty.*

IS2 *Run the untwisted process until the network returns to Δ.*

IS3 *Run the twisted process, while counting the cells lost from the workload of the hot spot.*

IS4 *If either the workload of the hot spot reaches (or exceeds) the critical length l, or if the hot spot buffer empties then stop the twisted process. If the latter occurs there is no cell loss ($CCL_\Delta[n] = 0$), and skip to IS7.*

IS5 *Calculate the extra factor $\mathcal{C}_n$ and the partial likelihood ratio $\mathcal{L}_n$.*

IS6 *Run the untwisted process until the hot spot buffer is empty (i.e. we return to Δ), while counting the number of lost cells, $CCL[n]$.*

IS7 *Return the fabric to the state it was before the last twisted run. Now run the untwisted process until we return to Δ.*

IS8 *Repeat IS3 to IS7 for n from 1 to N, that is for N Δ-cycles. The average number of cells lost per Δ-cycle is equal to*

$$(1/N) \sum_{n=1}^{N} CCL_\Delta[n]\, \mathcal{L}_n \mathcal{C}_n.$$

4.3 The Δ_0-Cycles

An aborted twisted Δ-cycle is a trajectory which is generated using the twisted probabilities but for which the workload never reaches the critical value l, i.e. the capacity of the hot spot. During such a twisted Δ-cycle the hot spot cannot overflow, so there is no cell loss. With *a priori* knowledge that a twisted Δ-cycle would "abort" to Δ

before reaching the critical value, we could just skip that twisted Δ-cycle, i.e. not simulate it and record the number of losses for that twisted Δ-cycle as zero. We claim that it is possible to identify some (not all) of the aborted twisted Δ-cycles from the initial state of the ATM switching fabric. Actually we can only detect some of the twisted Δ-cycles which are of length one, i.e. the hot spot buffer remains empty and yet this can be a significant fraction.

The offered cells to the hot spot buffer are cells carried by the buffers upstream to the hot spot buffer. We want to identify the possible arrivals to the hot spot buffer from the upstream buffers. This task of identification must be carried out with forethought to the different clock cycles of the buffers in the switching fabric. Until now we have not thoroughly discussed the convention used for internal switching. We assume switches in the ATM switching fabric with the same transmission rate serve at the very same instant. This means that the hot spot buffer cannot serve a cell that arrives at the end of the time slot. However it is possible to have a cell arrival at the hot spot before the end of the time slot, since we allow for different internal transmission speeds. Such a cell can be served by the hot spot buffer at the end of that same time slot if it is the only cell waiting. For example, for one queue feeding the hot spot buffer, where the upstream buffer is running at twice the speed, a cell could be offered to the hot spot buffer at the mid-point of the time slot. At the end of that time slot, the cell is served if it is the only cell waiting in the hot spot.

Assuming that a cell offered to the workload of the hot spot by a source cannot reach the hot spot buffer during that same time slot, then it is possible to know the length of the hot spot for the next time slot. This information is contained in the upstream buffers to the hot spot. If a cell will be offered to the hot spot at the end of the next time slot then the hot spot will necessarily not be empty, since that cell cannot be served. However, if there are no such possible arrivals and the number of cells offered to the hot spot before the end of the next time slot is less than two, then the hot spot buffer will still be empty. If there is only one arrival, it will be served and the hot spot buffer remains empty.

As aforementioned we skip the detected aborted twisted Δ-cycle, i.e. we count the Δ-cycle and associate to it a loss of zero cells. In practice a large number of these aborted twisted Δ-cycles could be clumped together, this is the motivation for the "super" Δ_0-cycle. A Δ_0-cycle constists of multiple original Δ-cycles but only one twisted Δ-cycle. The twisted Δ-cycle is associated to the last original Δ-cycle in the Δ_0-cycle. The last original Δ-cycle in a Δ_0-cycle is the only one in the bundle which is not detected as a sure aborted cycle. We must simulate an associated twisted Δ-cycle to this last original Δ-cycle since we do know if it will abort or not. We modified the importance sampling Algorithm 4.1 to include Δ_0-cycles.

Algorithm 4.2 (with Δ_0-cycles)

IS1 *Start the simulation at a state in Δ so the hot spot buffer is empty.*

IS2 *Run the untwisted process.*

IS3 *While the hot spot buffer remains empty at each time slot increment the number of Δ-cycles by one and associate to it a cell loss of zero ($CCL_\Delta[n] = 0$). These Δ-cycles are aborted cycle of length one. If either N Δ-cycles are completed, or if the hot spot buffer has the possibility of being non-empty at the end of the next time slot then stop the process. If the former occurs the simulation is completed and skip to step IX.*

IS4 *Run the twisted process, while counting the cells lost from the workload of the hot spot.*

IS5 *If either the workload of the hot spot reaches (or exceeds) the critical length l, or if the hot spot buffer empties then stop the twisted process. If the latter occurs there is no cell loss ($CCL_\Delta[n] = 0$), and skip to IS8.*

IS6 *Calculate the extra factor C_n and the likelihood ratio $\mathcal{L}_n$.*

IS7 *Run the untwisted process until the hot spot buffer is empty (i.e. we return to Δ), while counting the number of lost cells, $CCL_\Delta[n]$.*

IS8 *Return the fabric to the state it was before the last twisted run and repeat steps IS2 to IS7 until N Δ-cycles are completed.*

IS9 *The average number of cells lost per Δ-cycle is equal to*

$$(1/N) \sum_{n=1}^{N} CCL[n]\, \mathcal{L}_n\, \mathcal{C}_n.$$

Chapter 5

Application

In this chapter, we find the CLR of the hot spot buffer for a specific model. In this model the internal switches have greater transmission speeds than the transmission rate for the switch containing the hot spot buffer (i.e. the link rate). The upstream buffers to the hot spot buffer are assumed infinite.

5.1 Model I

Figure 10 illustrates the network for Model I. The transmission rate from the buffers A and B to the hot spot buffer H is 1.67E6 cells per second. The transmission rate from the source multiplexers to the buffers A and B is 353208 cells per second. The output link rate is 353208 cells per second. The buffer H is the hot spot. Note a time slot is $1/\mathrm{LR} = 1/353208$ seconds.

We have four independent and identically distributed bursty sources (i.e one group $M = 1$). The four sources are characterized by the following traffic parameters: a peak cell rate (PCR) of 353208 cells per second, a sustained cell rate (SCR) 27370 cells per second, and an average burst size (ABS) of 210 cells. Let AIP denote the average idle period. At each time slot during a burst period a source will offer a cell to the workload of the hot spot (i.e. $d_1 = r_1 = 1$). The average burst size and the average idle period are the parameters for the geometric sojourn times for the ON

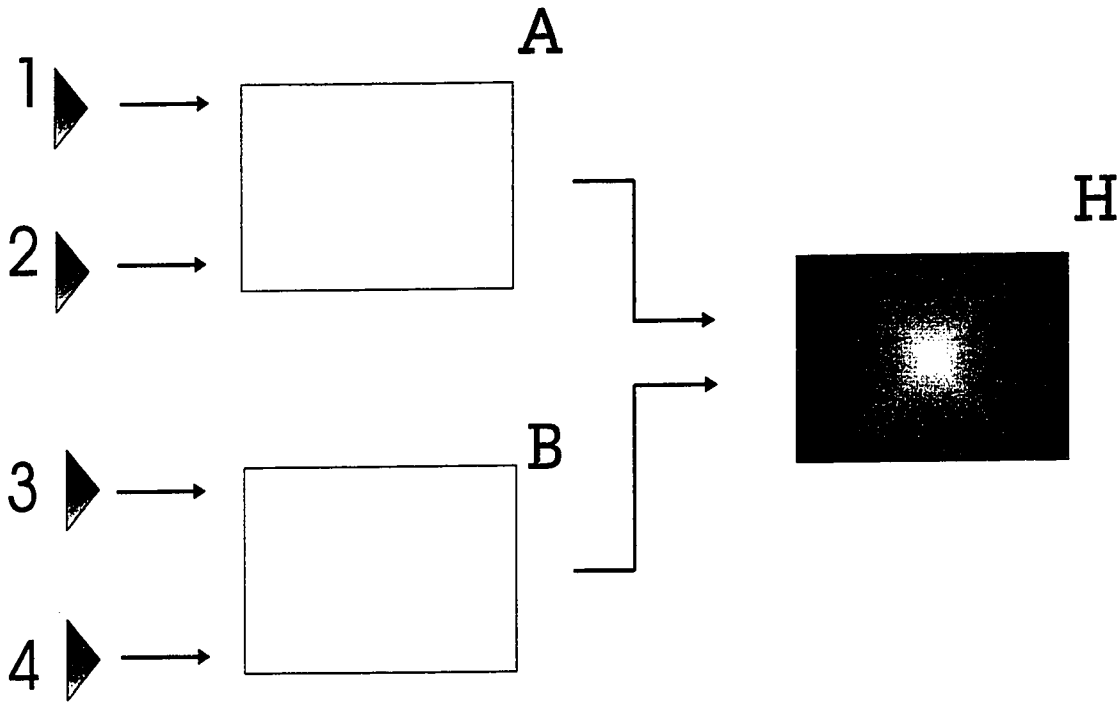

Figure 10: Model I

and OFF period respectively. Hence the transition probabilities from ON to OFF and OFF to ON are

$$\mu_1 = \frac{1}{\text{ABS}} \qquad \text{and} \qquad \lambda_1 = \frac{1}{\text{AIP}},$$

respectively. The sustained cell rate is the mean rate at which a source will offer a cell. During a burst period a source offers cells at the peak cell rate and during the idle period no cells are offered, thus

$$\text{SCR} = \frac{\text{ABS}}{\text{ABS} + \text{AIP}} \text{PCR}.$$

With simple algebraic manipulations, we can show that

$$\lambda_1 = \frac{1}{\text{AIP}} = \frac{\text{SCR}}{\text{ABS}\,(\text{PCR} - \text{SCR})}.$$

The numerical values for the transition probabilities are approximately $\mu_1 = 0.00476$ and $\lambda_1 = 0.0004$. The long-run average number of cells offered to the workload of the hot spot is

$$4\,\frac{\lambda_1}{\lambda_1 + \mu_1} = 0.3099590043,$$

which is less than one. The mean increment for the free process is negative.

To find the twisted probabilities, we need to find the unique positive root θ of the function

$$\Lambda(\gamma) = -\gamma - 4\,\alpha_1(\gamma),$$

where $\alpha_1(\cdot)$ is explicitly defined by (29) since $d_1 = 1$. From Figure 11, we know that $0.0045 < \theta < 0.005$. Using the Bisection Algorithm 3.1, we get $\theta = 0.004761525$, which means $\alpha_1(\theta) = -0.001190381255$. Using (31), we have $\beta_1(\theta) = 3.9777677464$.

The twisted transition probabilities are

$$\frac{\lambda_1\,\beta_1}{1 - \lambda_1 + \lambda_1\,\beta_1} = 1.58919 \times 10^{-3} \qquad \text{from ON to OFF}$$

and

$$\frac{\mu_1\,\beta_1^{-1}}{1 - \mu_1 + \mu_1\,\beta_1^{-1}} = 1.20141 \times 10^{-3} \qquad \text{from OFF to ON.}$$

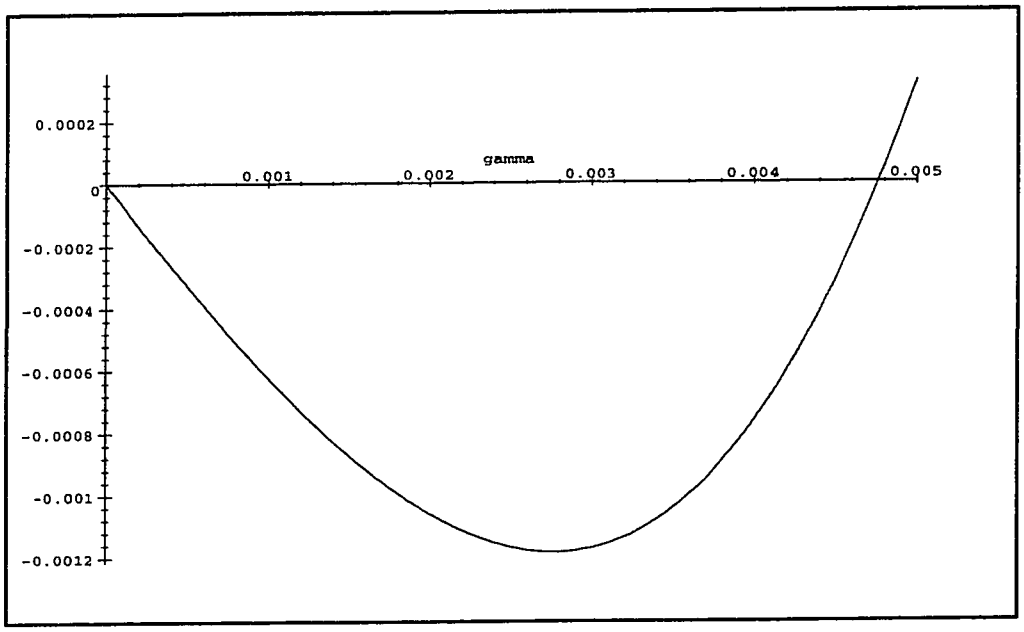

Figure 11: Plot: $\Lambda(\gamma)$ versus γ, for Model I

The long-run average number of cells offered to the workload of the hot spot buffer (per time slot) is 2.28, which is greater than one. The twisted free process has a positive drift, i.e. the workload will tend to infinity.

5.2 Simulation Results

Some simulation results are displayed in Table 2. The relative error is estimated by simulating independent batches. The batch size for all simulations is ten. We calculate the relative error as the sample standard deviation divided by the sample mean.

The computational effort of the simulation is measured by the average number of cells generated. Notice that all these simulations have a computational effort in the order of 10^6. With this computational effort, the twisted estimates, for the CLR, are

Buffer Size	Type of Estimate	$\widehat{CLR} \pm$ Relative Error	Average Number of Cells Generated
300	TE	$1.80 \times 10^{-2} \pm 10.14\%$	1.80×10^{6}
300	CME	$1.87 \times 10^{-2} \pm 9.13\%$	1.40×10^{6}
1200	TE	$1.87 \times 10^{-4} \pm 16.4\%$	2.68×10^{6}
1200	CME	$1.78 \times 10^{-4} \pm 59.77\%$	2.70×10^{6}
2000	TE	$4.19 \times 10^{-6} \pm 5.26\%$	4.62×10^{6}
2000	CME	$0.00 \pm ??$	4.71×10^{6}
2000	CME	$5.2 \times 10^{-6} \pm 205.32\%$	1.42×10^{7}
3000	TE	$3.67 \times 10^{-8} \pm 6.43\%$	5.95×10^{6}

TE - Twisted Estimate
CME - Crude Monte-Carlo Estimate

Table 2: Simulation results for Model I

very good in the sense that relative errors for the twisted estimates range between 5.26% to 16.4%.

The crude Monte-Carlo estimates are not very good for small cell loss ratios. Even at an order of 10^{-4} with a computational effort of about 2 million cells the crude Monte-Carlo estimate has a relative error of 59.77%. At an order of 10^{-6} and a computational effort of 4.71 million cells, no cells were lost at all during the crude Monte-Carlo simulations, see buffer size 2000. With a computational effort of 14.2 million cells, we were able to obtain a crude Monte-Carlo estimate for a buffer size of 2000, although the estimate has a relative error of 205.32%, which is very poor.

For a cell loss ratio in the order of 10^{-8} the twisted estimate has a relative error of 6.43%, which is very good, considering the computational effort is approximately 6 million. It is apparent, through experimentation, that our importance sampling technique can be considered as a variance reduction tool for this model.

Chapter 6

Priority Queueing

In this chapter, we assume a time priority mechanism is used by the switches in the ATM switching fabric. The cells are buffered into two different queues waiting to be served, see Figure 12. The cells are separated by priority, high and low. As long as there is a high priority cell waiting at each time slot a high priority cell is served. If there are no high priority cells to be served at the end of a time slot and there is a low priority cell waiting then a low priority cell is served. We assume that a source can only generate cells of one type, either low priority, or high priority, but not both. It is possible to use the twist that we found in Chapter 3.2 to find the CLR for a buffer in this priority queueing ATM network.

First we consider the hot spot buffer to be one of the high priority buffers. A source which generates cells of low priority will belong to a group a sources which generate independent and identically distributed streams of cells that aren't directed at the hot spot buffer. Similarly a source, which generates high priority cells, is grouped with sources that generate independent and identically distributed streams of cells. The cells generated by a particular high priority source are either always directed to the hot spot buffer, or are never directed to the hot spot buffer. The workload of the hot spot buffer is the number of high priority cells which are either directed to the hot spot buffer but are waiting in an upstream buffer, or are waiting in the hot spot buffer. The validity of our Δ-cycle method, which uses twisted traffic parameters for

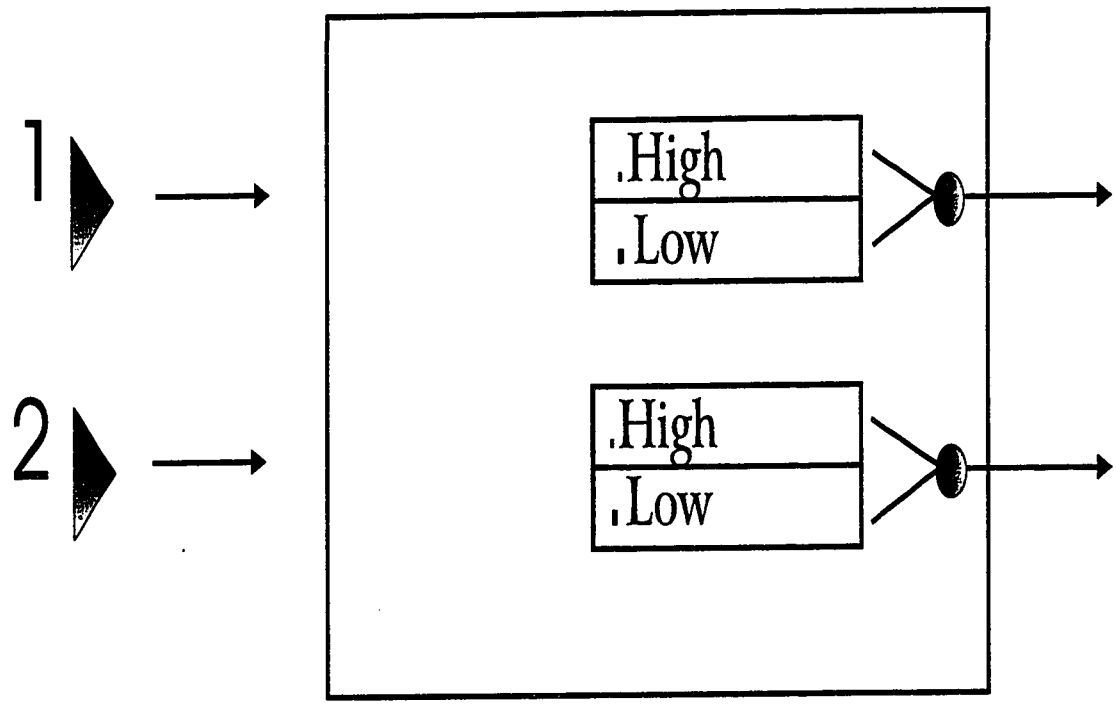

Figure 12: A 2 × 2 switching element with time priority queueing

the sources that offer cells to the workload of the hot spot, depends upon the use of the free process $\{W^{\infty}, M\}$ to adequately describe the transitions of the workload between return times to Δ. The set Δ contains all states which have an empty hot spot buffer. Between return times to Δ the increment for the workload of the hot spot is $A[n] - 1$, that is $Q[n] = Q[n-1] + A[n] - 1$, where $A[n]$ are the cells offered to the workload, since there is always a cell in the hot spot waiting to be served. Remember, the variables $W^{\infty}[n]$ are generated by the following recurrence equation

$$W^{\infty}[n] = W^{\infty}[n-1] + A[n] - 1,$$

with initial value $W^{\infty}[0] = Q[0]$. Clearly $\{Q, M\}$ and $\{W^{\infty}, M\}$ have the same transition probabilities between return times to Δ. Therefore, we can use our Δ-cycle method.

Now take the hot spot buffer to be one of the low priority buffers. For this case the workload of the hot spot buffer is not only the number of cells in the switching fabric which are directed to the hot spot buffer and those waiting in the hot spot buffer itself. We must also add the number of cells which are waiting in the high priority buffer associated to the hot spot, and the number of cells which are directed to this same high priority buffer but are waiting in upstream buffers. We must account for these high priority cells, since for every cell waiting in the high priority buffer associated to the hot spot buffer we must add a time slot to the total time before a low priority cell in the hot spot buffer can be served. Between return times to Δ, a cell from the workload is served per time slot. Note, these served cells are not necessarily cells which were waiting in the hot spot, it could be a cell which was waiting in the high priority buffer associated to the low priority hot spot buffer. The increment for the workload is $A[n] - 1$ between return times to Δ. Hence, our Δ-cycle method is valid by a similar argument used for the high priority hot spot buffer. Although, for the low priority hot spot buffer case, some sources which generate high priority cells will have twisted parameters. Model II gives an example of a network with time priority queueing.

6.1 Model II

The network for Model II is a 4×4 switching fabric, with time priority queueing. The transmission rate from the buffers A to the buffers B is 1.67E6 cells per second. The transmission rate from the source multiplexers to the buffers A and B is 353208 cells per second. The output link rate is 353208 cells per second. The hot spot buffer is the low priority buffer. The high priority buffers have a finite capacity of 500 cells and the low priority buffers have a finite capacity of 1000 cells, except for the hot spot buffer. Similar to Model I, a time slot is $1/\mathrm{LR} = 1/353208$ seconds.

We have two groups of two independent and identically distributed sources (i.e two groups $M = 2$). All cells are directed to the hot spot buffer or the high priority buffer associated to the hot spot buffer. The two sources in group 1 generate bursty streams of cells (of low priority), which are characterized by the following traffic parameters: a peak cell rate (PCR) of 353208 cells per second, a sustained cell rate (SCR) 27370 cells per second, and an average burst size (ABS) of 210 cells. Note, the sources in group are indentical to the sources from Model I. Hence, the transition probabilities from ON to OFF and OFF to ON are

$$\mu_1 = \frac{1}{\mathrm{ABS}} \qquad \text{and} \qquad \lambda_1 = \frac{\mathrm{SCR}}{\mathrm{ABS}\,(\mathrm{PCR} - \mathrm{SCR})}.$$

The numerical values for the transition probabilities are approximately $\mu_1 = 0.00476$ and $\lambda_1 = 0.0004$. Note, $d_1 = r_1 = 1$, see Model I.

The two sources in group 2 generate non-bursty streams of cells characterized by the traffic parameter: a sustained cell rate (SCR) of 16560 cells per second. These sources always remain ON. They generate streams modelled by a Bernoulli process with parameter $SCR/LR = r_2$, which is approximately equal to $r_2 = 0.04688$. At each time slot a source in this group will offer a cell with probability r_2, so we have $d_2 = 1$.

At the end of a time slot, the workload is the number of cells in the fabric directed to the hot spot buffer and the high priority buffer associated to the hot spot, including

the cells waiting in the hot spot and the high priority buffer associated to the hot spot. The long-run average number of cells offered to the workload of the hot spot is

$$2\frac{\lambda_1}{\lambda_1+\mu_1}+2r_2=.2487,$$

which is less than one. The mean increment for the free process is negative.

To find the twisted probabilities we need to find the unique positive root of

$$\Lambda(\gamma)=-\gamma-2\,\alpha_1(\gamma)-2\,\alpha_2(\gamma),$$

where $\alpha_1(\cdot)$, and $\alpha_2(\cdot)$ are given by (29) and (28), respectively. From Figure 13, we know that $0.007<\theta<0.008$. Using the bisection Algorithm 3.1, we get $\theta=0.0078416801$ which means $\alpha_1(\theta)=-3.6903087096\times10^{-4}$ and $\alpha_2(\theta)=-3.5518092096\times10^{-3}$. By (31), we have $\beta_1(\theta)=9.8954382967$ and $\beta_2(\theta)=1.0003690987$.

For group 1, the twisted probabilities are

$$\frac{\lambda_1\beta_1}{1-\lambda_1+\lambda_1\beta_1}=1.58919\times10^{-3}\qquad\text{from ON to OFF}$$

and

$$\frac{\mu_1\beta_1^{-1}}{1-\mu_1+\mu_1\beta_1^{-1}}=1.20141\times10^{-3}\qquad\text{from OFF to ON.}$$

For group 2, the Bernoulli parameter r_2 has been transformed to

$$\frac{e^{\theta}r_2}{e^{\theta}r_2+1-r_2}=0.047236.$$

The long-run average number of cells offered to the workload of the hot spot buffer (per time slot) is 1.87, which is greater than one. The twisted free process has a positive drift, i.e. the workload will tend to infinity.

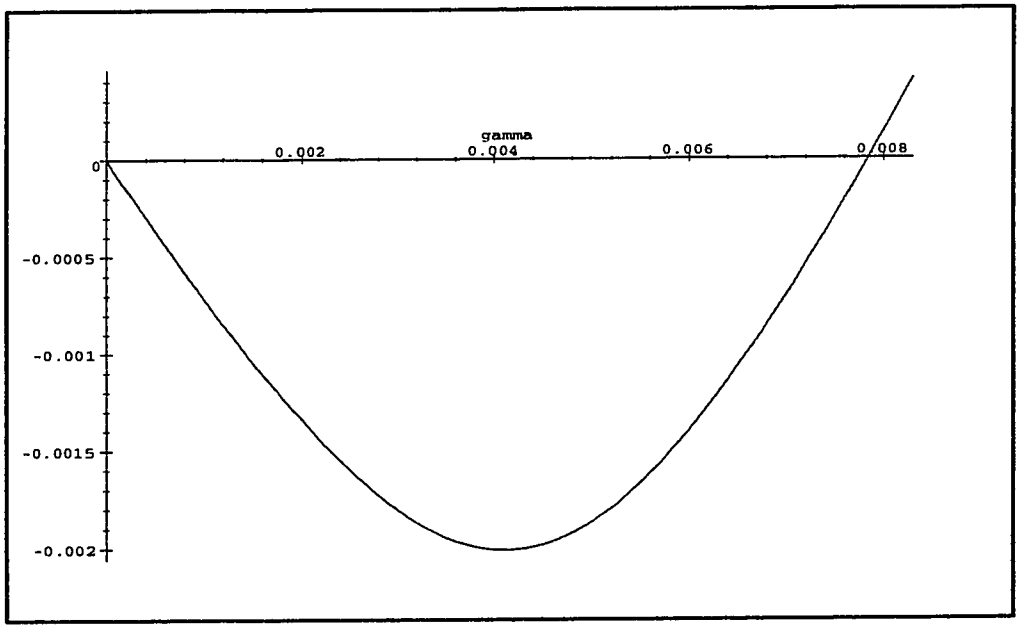

Figure 13: Plot: $\Lambda(\gamma)$ versus γ, for Model II

Buffer Size	Type of Estimate	$\widehat{CLR} \pm$ Relative Error	Average Number of Cells Generated
100	TE	$2.72 \times 10^{-2} \pm 6.89\%$	1.55×10^{6}
100	CME	$2.78 \times 10^{-2} \pm 8.2\%$	1.33×10^{6}
200	TE	$1.24 \times 10^{-2} \pm 5.19\%$	1.81×10^{6}
200	CME	$1.28 \times 10^{-2} \pm 14.9\%$	1.33×10^{6}
300	TE	$5.61 \times 10^{-3} \pm 5.37\%$	2.49×10^{6}
300	CME	$5.76 \times 10^{-3} \pm 22.7\%$	1.33×10^{6}
800	TE	$1.1 \times 10^{-4} \pm 4.20\%$	4.39×10^{6}
1300	TE	$2.19 \times 10^{-6} \pm 6.43\%$	6.06×10^{6}
1500	TE	$4.55 \times 10^{-7} \pm 4.23\%$	6.12×10^{6}
1800	TE	$4.32 \times 10^{-8} \pm 5.93\%$	7.15×10^{6}

TE - Twisted Estimate
CME - Crude Monte-Carlo Estimate

Table 3: Simulation results for Model II

6.2 Simulation Results

The simulation results are in Table 3. The computational effort (the average number of cells generated during the simulation) is always in the order of 10^6. The batch size for each estimate is ten.

For small buffer sizes, 100 to 300, both the twisted and crude estimates are comparable. The problem with the crude Monte-Carlo techniques is that for small cell loss ratios, we need to generate an extremely large number of cells during the simulations to get a good estimate. Here, we show that with a computational effort in the order of a million cells, the twisted simulations gave good estimates, i.e. small relative errors. Actually, for a CLR of 4.32×10^{-8}, with a computational effort of 7.15×10^{6}, the relative error for the twisted estimate is 5.93%. This is an extremely efficient estimate, since we only needed about 7 million cells to get a good estimate for a CLR in the order of 10^{-8}.

Chapter 7

Existence

Beck, Dabrowski and McDonald (1998) (BDM) give the asymptotics for a fast teller queue using the twisted process of the free MA-chain associated with the fast teller queue. In this thesis we show that the free MA-chain has an associated twisted chain which can be used to significantly accelerate the simulation of buffer overflows. As opposed to BDM the Markovian part is simplified, i.e. it is just the number of sources with delay j. In this case we have a construction of the twisted process contingent upon a solution to the system of equations. Here we establish the existence of the twisted process within this simplified setting by showing such a solution exists.

7.1 Existence of the Twist

In this section we give a sufficient condition such that we can find positive numbers θ and β_i for all $i \in A_h$, and negative real numbers α_i for all $i \in A_h$ such that the equation (11-13) are satisfied. We state the theorem.

Theorem 7.1 (Existence) *If $\sum_{i \in A_h} N_i/d_i > 1$ then there exist positive real numbers θ and β_i for all $i \in A_h$, and negative real numbers α_i for all $i \in A_h$ that satisfy*

the following system of equations

$$e^{\theta} = \prod_{i \in A_h} (1 - \lambda_i + \lambda_i \beta_i)^{N_i}, \tag{34}$$

$$e^{-\alpha_i} = ((1 - \mu_i) \exp\{\alpha_i (d_i - 1)\} + \mu_i \beta_i^{-1}) (e^{\theta} r_i + 1 - r_i) \quad \forall i \in A_h \tag{35}$$

$$e^{-\alpha_i} = 1 - \lambda_i + \lambda_i \beta_i \qquad \forall i \in A_h. \tag{36}$$

Moreover, the h-transform (23) of K^{∞} exists.

Remark: Note, $\sum_{i \in A_h} N_i/d_i$ is the long-run average number of cells offered to the workload of the hot spot (per timeslot) if each source always ON, and each source in group i is offering a cell at every d_i time slot. If this sum is greater than one, then there exists a configuration of the sources such that the average increment of the free process $\{W^{\infty}[n]\}$ is positive. If this is the case, the theorem tells us that the twist will find one of these configurations.

First, we will establish that we can find differentiable functions $\alpha_i(\cdot)$ and $\beta_i(\cdot)$ on $(-\delta, \infty)$ (for some positive δ), such that $\alpha_i(\gamma) < 0$ and $\beta_i(\gamma) > 1$, for all $\gamma > 0$ and,

$$e^{-\alpha_i(\gamma)} = ((1 - \mu_i) e^{(\alpha_i(\gamma)(d_i - 1))} + \mu_i \beta_i(\gamma)^{-1}) (e^{\gamma} r_i + 1 - r_i) \quad \forall i \in A_h \tag{37}$$

$$e^{-\alpha_i(\gamma)} = 1 - \lambda_i + \lambda_i \beta_i(\gamma) \qquad \forall i \in A_h. \tag{38}$$

Define,

$$\Lambda(\gamma) := -\gamma - \sum_{i \in A_h} N_i \alpha_i(\gamma).$$

Second, we will show that there exists a positive real number θ such that $\Lambda(\theta) = 0$. This means,

$$0 = \Lambda(\theta) = -\theta - \sum_{i \in A_h} N_i \alpha_i(\theta) = -\theta + \sum_{i \in A_h} \log(1 - \lambda_i + \lambda_i \beta_i(\theta))^{N_i} \quad \text{by (38)},$$

which implies

$$e^{\theta} = \prod_{i \in A_h} (1 - \lambda_i + \lambda_i \beta_i(\theta))^{N_i}. \tag{39}$$

Notice that (37) and (38) with $\gamma = \theta$, and (39), gives us the equations (34-36).

7.2 The Functions $\alpha_i(\cdot)$ and $\beta_i(\cdot)$

Fix $i \in A_h$, we are interested in the solutions of the system

$$\begin{aligned} e^{-\alpha_i} &= ((1-\mu_i)\, e^{(\alpha_i (d_i-1))} + \mu_i\, \beta_i^{-1})\, (e^{\gamma} r_i + 1 - r_i) \\ e^{-\alpha_i} &= 1 - \lambda_i + \lambda_i\, \beta_i. \end{aligned}$$

Multiply both sides of the first equation by $e^{\alpha_i}\, \beta_i$, and solve for β_i in the second equation. We get

$$\begin{aligned} \beta_i &= ((1-\mu_i)\, e^{\alpha_i\, d_i}\, \beta_i + \mu_i\, e^{\alpha_i})\, (e^{\gamma} r_i + 1 - r_i) \\ \beta_i &= (e^{-\alpha_i} - 1 + \lambda_i)/\lambda_i. \end{aligned}$$

Substitute β_i from the second equation into the first equation. The first equation becomes

$$(e^{-\alpha_i} - 1 + \lambda_i)/\lambda_i = ((1-\mu_i)\, e^{\alpha_i\, d_i}\, ((e^{-\alpha_i} - 1 + \lambda_i)/\lambda_i) + \mu_i\, e^{\alpha_i})\, (e^{\gamma} r_i + 1 - r_i).$$

Multiply both sides of this equation by λ_i, add $1 - \lambda_i$, and then multiply e^{α_i}. We get

$$\begin{aligned} 1 &= (1-\lambda_i)\, e^{\alpha_i} + [(1-\mu_i)\, e^{\alpha_i\, d_i}\, (1 - (1-\lambda_i)\, e^{\alpha_i}) + \mu_i\, \lambda_i\, e^{2\,\alpha_i}]\, (e^{\gamma} r_i + 1 - r_i) \\ &= (1-\lambda_i)\, e^{\alpha_i} + (1-\mu_i)\, e^{\alpha_i\, d_i}\, (e^{\gamma} r_i + 1 - r_i) \\ &\quad - (1-\mu_i)\, (1-\lambda_i)\, e^{\alpha_i\, (d_i+1)}\, (e^{\gamma} r_i + 1 - r_i) \\ &\quad + \mu_i\, \lambda_i\, e^{2\,\alpha_i}\, (e^{\gamma} r_i + 1 - r_i) =: \Xi_i(\alpha_i, \gamma). \end{aligned} \tag{40}$$

So, we are interested in the solutions of the following system

$$\begin{aligned} \Xi_i(\alpha_i, \gamma) &= 1 \\ \beta_i &= (e^{-\alpha_i} - 1 + \lambda_i)/\lambda_i. \end{aligned}$$

Suppose, that for some positive δ, $\alpha_i(\cdot)$ can be defined implicitly by

$$\Xi_i(\alpha_i(\gamma), \gamma) = 1,$$

as a differentiable function of γ on $(-\delta, \infty)$ such that $\alpha_i(\gamma) < 0$ for all $\gamma > 0$. Then,

$$\beta_i(\gamma) := (e^{-\alpha_i(\gamma)} - 1 + \lambda_i)/\lambda_i, \tag{41}$$

is a differentiable function on $(-\delta, \infty)$. Also, for all $\gamma > 0$

$$\begin{aligned}\beta_i(\gamma) &= (e^{-\alpha_i(\gamma)} - 1 + \lambda_i)/\lambda_i \\ &> (1 - 1 + \lambda_i)/\lambda_i \quad \text{since } \alpha_i(\gamma) < 0 \ (\forall \gamma > 0) \\ &= 1.\end{aligned} \tag{42}$$

It remains to prove the existence of the differentiable function $\alpha_i(\cdot)$, see Lemma 7.1.

Lemma 7.1 *For some $\delta > 0$, there exists a differentiable function $\alpha_i : (-\delta, \infty) \to \mathbb{R}$, such that $\Xi_i(\alpha_i(\gamma), \gamma) = 1$. Moreover, for all $\gamma > 0$, $\alpha_i(\gamma) < 0$, and $\alpha_i(0) = 0$.*

Proof: We consider the three cases ($\lambda_i = 1, \mu_i = 0$) that is the sources always remain ON, ($0 < \lambda_i, \mu_i < 1$ and $d_i = 1$), and ($0 < \lambda_i, \mu_i < 1$ and $d_i > 1$).

Case I, ($\lambda_i = 1, \mu_i = 0$)**:** For this case we can find $\alpha_i(\cdot)$ explicitly. By (40), we have

$$\Xi(\alpha_i, \gamma) = e^{\alpha_i d_i} (e^{\gamma} r_i + 1 - r_i).$$

Since we want $\Xi(\alpha_i(\gamma), \gamma) = 1$, the function $\alpha_i(\cdot)$ must be defined as

$$\alpha_i(\gamma) := -\frac{1}{d_i} \log(e^{\gamma} r_i + 1 - r_i). \tag{43}$$

The function $\alpha_i(\cdot)$ as defined by (43) is differentiable on $\mathbb{R}$. Also, for any positive γ, $e^{\gamma} r_i + 1 - r_i$ is greater than one, thus $\alpha_i(\gamma) < 0$. And $\alpha_i(0) = -(1/d_i) \log(1) = 0$.

Case II, ($0 < \lambda_i, \mu_i < 1$ **and** $d_i = 1$)**:** For this case, we can also find $\alpha_i(\cdot)$ explicitly. By (40), we have

$$\begin{aligned}\Xi(\alpha_i, \gamma) &= (1 - \lambda_i)\, e^{\alpha_i} + (1 - \mu_i)\, e^{\alpha_i} (e^{\gamma} r_i + 1 - r_i) \\ &\quad - (1 - \mu_i)(1 - \lambda_i)\, e^{2\alpha_i} (e^{\gamma} r_i + 1 - r_i) + \mu_i \lambda_i\, e^{2\alpha_i} (e^{\gamma} r_i + 1 - r_i) \\ &= [(1 - \lambda_i) + (1 - \mu_i)(e^{\gamma} r_i + 1 - r_i)]\, e^{\alpha_i} \\ &\quad + [\mu_i \lambda_i (e^{\gamma} r_i + 1 - r_i) - (1 - \mu_i)(1 - \lambda_i)(e^{\gamma} r_i + 1 - r_i)]\, e^{2\alpha_i}.\end{aligned}$$

To solve $\Xi(\alpha_i, \gamma) = 1$, we will consider $e^{-2\alpha_i}(1 - \Xi(\alpha_i, \gamma)) = 0$ as a quadratic equation in $e^{-\alpha_i}$. The discriminant for this quadratic equation is

$$\begin{aligned} D_i(\gamma) &:= [-(1-\lambda_i) - (1-\mu_i)(e^{\gamma} r_i + 1 - r_i)]^2 \\ &\quad - 4[-\mu_i \lambda_i (e^{\gamma} r_i + 1 - r_i) + (1-\mu_i)(1-\lambda_i)(e^{\gamma} r_i + 1 - r_i)] \\ &= [(1-\lambda_i) - (1-\mu_i)(e^{\gamma} r_i + 1 - r_i)]^2 + 4\mu_i \lambda_i (e^{\gamma} r_i + 1 - r_i) \geq 0. \end{aligned} \tag{44}$$

Thus, both roots of this quadratic equation are real. We will define $\exp\{-\alpha_i(\gamma)\}$ as the root with the positive sign, that is

$$\exp\{-\alpha_i(\gamma)\} := \frac{(1-\lambda_i) + (1-\mu_i)(e^{\gamma} r_i + 1 - r_i) + \sqrt{D_i(\gamma)}}{2},$$

where $D_i(\gamma)$ is defined by (44). This means,

$$\alpha_i(\gamma) := -\log\left\{\frac{(1-\lambda_i) + (1-\mu_i)(e^{\gamma} r_i + 1 - r_i) + \sqrt{D_i(\gamma)}}{2}\right\}. \tag{45}$$

The function $\alpha_i(\cdot)$ as defined by (45) is differentiable on $\mathbb{R}$. Now, we show that for all $\gamma > 0$, $\alpha_i(\gamma) < 0$. It is sufficient to show that

$$\frac{(1-\lambda_i) + (1-\mu_i)(e^{\gamma} r_i + 1 - r_i) + \sqrt{D_i(\gamma)}}{2} > 1.$$

Since $D_i(\gamma) \geq 0$ for all γ, the inequality is equivalent to

$$D_i(\gamma) > \{2 - [(1-\lambda_i) + (1-\mu_i)(e^{\gamma} r_i + 1 - r_i)]\}^2.$$

Add to both sides $-[(1-\lambda_i) + (1-\mu_i)(e^{\gamma} r_i + 1 - r_i)]^2$. We get

$$\begin{aligned} &- 4[-\mu_i \lambda_i (e^{\gamma} r_i + 1 - r_i) + (1-\mu_i)(1-\lambda_i)(e^{\gamma} r_i + 1 - r_i)] \\ &> 4 - 4[(1-\lambda_i) + (1-\mu_i)(e^{\gamma} r_i + 1 - r_i)]. \end{aligned}$$

Divide both sides by 4, we get

$$(\mu_i + \lambda_i - 1)(e^{\gamma} r_i + 1 - r_i) > \lambda_i - (1-\mu_i)(e^{\gamma} r_i - 1 + r_i).$$

Add $-\lambda_i - (1-\mu_i)(e^{\gamma} r_i - 1 + r_i)$ to both sides, and simplify, we get

$$\lambda_i ((e^{\gamma} r_i + 1 - r_i) - 1) > 0.$$

Thus, $\alpha_i(\gamma) < 0$ if and only if $e^{\gamma} r_i + 1 - r_i > 1$. Therefore $\alpha_i(\gamma) < 0$ for all $\gamma > 0$. Also, it is easy to show that $\alpha_i(0) = 0$.

Case III, $(0 < \lambda_i, \mu_i < 1$ **and** $d_i > 1)$**:** The function Ξ_i at the origin is one, that is $\Xi_i(0,0) = 1$. By the following four observations, for all $\gamma > 0$, there exists a $x_i < 0$ such that $\Xi_i(x_i, \gamma) = 1$.

1. $\Xi_i(\alpha_i, \gamma)$ is a differentiable (continuous) function in α_i,
2. $\lim_{\alpha_i \to -\infty} \Xi_i(\alpha_i, \gamma) = 0$,
3. $\lim_{\alpha_i \to \infty} \Xi_i(\alpha_i, \gamma) = -\infty$,
4. $\Xi_i(0, \gamma) = 1 - \lambda_i + \lambda_i (e^{\gamma} r_i + 1 - r_i) > 1$ for all $\gamma > 0$.

For fixed $\gamma_0 > 0$, consider $\Xi_i(\alpha_i, \gamma_0)$ as a function of α_i. The observations 1 to 4 also imply the existence of a stationary point x_0, i.e. the derivative $\Xi_i(\alpha_i, \gamma_0)$ (with respect to α_i) at $\alpha_i = x_0$ is zero. By Lemma B.1, $\Xi_i(\alpha_i, \gamma)$ as a function of α_i only has one stationary point, let $x_0(\gamma_0)$ be this unique stationary point. Therefore, for all $\gamma > 0$, there exists a unique negative $x_i = x_i(\gamma)$ such that $\Xi_i(x_i(\gamma), \gamma) = 1$. Notice that $x_i(\gamma)$ must be smaller than the stationary point $x_0(\gamma)$.

We will now show that $x_i(\gamma)$ is differentiable on $(0, \infty)$. Fix $\gamma_0 > 0$, take $x_i = x_i(\gamma_0)$, we know that

$$\Xi_i(x_i, \gamma_0) = 1 \text{ and } \frac{\partial}{\partial \alpha_i} \Xi(\alpha_i, \gamma)\Big|_{(\alpha_i, \gamma) = (x_i, \gamma_0)} \neq 0,$$

since $x_i(\gamma_0) < x_0(\gamma_0)$, where $x_0(\gamma_0)$ is the unique stationary point of $\Xi_i(\alpha_i, \gamma_0)$. By the implicit function theorem, for some $\epsilon > 0$, there exists a unique differentiable function g on $N_\epsilon(\gamma_0) := (\gamma_0 - \epsilon, \gamma_0 + \epsilon)$ such that

$$g(\gamma_0) = x_i < 0 \text{ and } \Xi_i(g(\gamma), \gamma) = 1 \text{ for all } \gamma \in N_\epsilon(\gamma_0).$$

Since g is differentiable and negative at γ_0, it must be negative on some neighbourhood of γ_0. From the uniqueness of the negative root, we must have $g(\gamma) = x_i(\gamma)$ on

some neighbourhood of γ_0. This implies that $x_i(\gamma)$ is differentiable on some neighbourhood of γ_0. Since γ_0 was arbitrary, $x_i(\gamma)$ is differentiable on $(0, \infty)$.

We will now extend $x_i(\gamma)$ such that it is differentiable on $(-\delta, \infty)$ for some positive δ. We know that $\Xi_i(0,0) = 1$. To apply the implicit function theorem we need $\partial/\partial\alpha_i \Xi_i(\alpha_i, \gamma) \neq 0$ at $(0,0)$. The first partial derivative of Ξ_i (defined by (40)) with respect to the abscissa is

$$\begin{aligned}\partial/\partial\alpha_i\, \Xi_i(\alpha_i, \gamma) = {} & (1-\lambda_i)\, e^{\alpha_i} + d_i\,(1-\mu_i)\, e^{\alpha_i d_i}\,(e^{\gamma} r_i + 1 - r_i) \\ & - (d_i+1)\,(1-\mu_i)\,(1-\lambda_i)\, e^{\alpha_i (d_i+1)}\,(e^{\gamma} r_i + 1 - r_i) \\ & + 2\,\mu_i\,\lambda_i\, e^{2\alpha_i}\,(e^{\gamma} r_i + 1 - r_i).\end{aligned}$$

At the origin $(\alpha_i, \gamma) = (0,0)$ this partial deritive is equal to $(d_i + (1-d_i)\,\mu_i)\,\lambda_i + \mu_i$, which by (2) for any source a in group i is equal to

$$\frac{1}{p_{01}^a}\,\frac{\mathrm{E}[ON_1^a + OFF_1^a]}{\mathrm{E}[R_1^a]} > 1,$$

where $ON_1^a + OFF_1^a$ is the length of the first ON-OFF cycle and R_1^a counts the number of times the source is ON with delay zero during that cycle. By the implicit function theorem, for some $\delta > 0$, there exists a unique differentiable function f on $N_\delta(0) := (-\delta, \delta)$ such that

$$f(0) = 0 \text{ and } \Xi_i(f(\gamma), \gamma) = 1 \text{ for all } \gamma \in N_\delta(0).$$

We find the derivative of f at zero implicitly. Taking the derivative with respect to γ on the left-hand side of $\Xi_i(f(\gamma), \gamma) = 1$ gives

$$\begin{aligned}& (1-\lambda_i)\, e^{f(\gamma)}\, f'(\gamma) + (1-\mu_i)\, e^{d_i f(\gamma)}\, d_i\, f'(\gamma)\,(e^{\gamma} r_i + 1 - r_i) \\ & + (1-\mu_i)\, e^{d_i f(\gamma)}\, e^{\gamma} r_i \\ & - (1-\lambda_i)\,(1-\mu_i)\, e^{(d_i+1) f}\,(d_i+1)\, f'(\gamma)\,(e^{\gamma} r_i + 1 - r_i) \\ & - (1-\lambda_i)\,(1-\mu_i)\, e^{(d_i+1) f(\gamma)}\, e^{\gamma} r_i \\ & + \mu_i\,\lambda_i\, e^{2 f(\gamma)}\, 2\, f'(\gamma)\,(e^{\gamma} r_i + 1 - r_i) + \mu_i\,\lambda_i\, e^{2 f(\gamma)}\, e^{\gamma} r_i.\end{aligned} \tag{46}$$

Since $f(0) = 0$, at $\gamma = 0$ the derivative of the left-hand side becomes

$$\begin{aligned}
&(1-\lambda_i)\, f'(0) + (1-\mu_i)\, d_i\, f'(0) \\
&+ (1-\mu_i)\, r_i - (1-\lambda_i)\,(1-\mu_i)\,(d_i+1)\, f'(0) \\
&- (1-\lambda_i)\,(1-\mu_i)\, r_i \\
&+ \mu_i\, \lambda_i\, 2\ f'(0) + \mu_i\, \lambda_i\, r_i.
\end{aligned}$$

Of course, the derivative on the right-hand side of $\Xi_i(f(\gamma), \gamma) = 1$ is equal to zero. Solving for $f'(0)$, we get

$$\begin{aligned}
f'(0) &= -\frac{(1-\mu_i)\, r_i - (1-\lambda_i)\,(1-\mu_i)\, r_i + \mu_i\, \lambda_i\, r_i}{(1-\lambda_i) + (1-\mu_i)\, d_i - (1-\lambda_i)\,(1-\mu_i)\,(d_i+1) + 2\,\mu_i\,\lambda_i} \\
&= -\frac{\lambda_i}{(d_i + (1-d_i)\,\mu_i)\,\lambda_i + \mu_i}\, r_i < 0.
\end{aligned} \tag{47}$$

Since $f'(0) < 0$ there exists a $\kappa > 0$ such that $f(\gamma) < 0$ for all $\gamma \in (0, \kappa)$. By the uniqueness of the negative root, $f(\gamma) = x_i(\gamma)$ for all $\gamma \in (0, \kappa)$.

Define $\alpha_i(\cdot)$ as f on $(-\delta, 0]$ and $x_i(\cdot)$ on $(0, \infty)$. Therefore $\alpha_i(\cdot)$ is a differentiable function on $(-\delta, \infty)$ such that

$$\alpha_i(\gamma) := \begin{cases} 0, & \text{if } \gamma = 0, \\ x_i(\gamma), & \text{if } \gamma > 0, \end{cases} \tag{48}$$

where $x_i(\gamma)$ is the unique negative real number such that $\Xi_i(x_i(\gamma), \gamma) = 1$.

□

Remark: The derivative of $\alpha_i(\cdot)$ at zero is given by (47).

7.3 The Function $\Lambda(\cdot)$

Here we give the properties of the function

$$\Lambda(\gamma) := -\gamma - \sum_{i \in A_h} \alpha_i(\gamma),$$

where $\alpha_i(\cdot)$ are as defined in the previous section. First, we show that $\exp\{N_i\,\alpha_i(\gamma)\}$ is the eigenvalue of a generating function, and use this fact to show that the function $\alpha_i(\cdot)$ is convex. The proof of Lemma 7.2 is in Appendix B.

Let $\vec{n}_i$ to be the current states of the sources in group i. Define the function r_γ^i as

$$r_\gamma^i(\vec{n}_i) := \begin{cases} \beta_i(\gamma)^{N_i - f_i}\,\exp\{\alpha_i(\gamma)\,\sum_{j=1}^{d_i-1} j\,n_{ij}\}, & \text{if } i \in A_h \text{ and } d_i > 1 \\ \beta_i(\gamma)^{n_{i0}}, & \text{if } i \in A_h \text{ and } d_i = 1 \\ 1, & \text{if } i \in A_n. \end{cases}, \qquad (49)$$

where $f_i = N_i - \sum_{i=0}^{d_i-1} n_{ij}$. The matrix $\widehat{\mathrm{K}}_\gamma^i$ is the generating function for the group i, see (5).

Lemma 7.2 *The function r_γ^i is a right eigenvector associated to the eigenvalue $\exp(-N_i\,\alpha_i(\gamma))$ for the matrix $\widehat{K}_\gamma^i$, that is*

$$\sum_{\vec{m}_i} \widehat{K}_\gamma^i(\vec{n}_i, \vec{m}_i)\, r_\gamma^i(\vec{m}_i) = \exp(-N_i\,\alpha_i(\gamma))\, r_\gamma^i(\vec{m}_i),$$

where r_γ^i is the vector defined by (49).

Corollary 7.1 *The functions $-\alpha_i(\cdot)$ as defined in Section 7.2 are convex functions for all $i \in A_h$. Moreover, the function $\Lambda(\gamma) := -\gamma - \sum_{i\in A_h} N_i\,\alpha_i(\gamma)$ is also convex.*

Proof: The generating function for the source i, that is $\widehat{\mathrm{K}}_\gamma^i$, has a positive eigenvalue $\exp(-N_i\,\alpha_i(\gamma))$ with positive eigenvector r_γ^i, that is $r_\gamma^i(\vec{n}_i) > 0$ for all $\vec{n}_i$. Define $\varepsilon := \max\{r_\gamma^i(\vec{n}_i)\}$ and $\nu := \min\{r_\gamma^i(\vec{n}_i)\}$. Denote, $(\widehat{\mathrm{K}}_\gamma^i)^n$ as the nth power of the matrix $\widehat{\mathrm{K}}_\gamma^i$. For all $\vec{n}_i$ and $\vec{m}_i$, we have

$$\frac{1}{\varepsilon}\,(\widehat{\mathrm{K}}_\gamma^i)^n(\vec{n}_i, \vec{m}_i)\, r_\gamma^i(\vec{m}_i) \le (\widehat{\mathrm{K}}_\gamma^i)^n(\vec{n}_i, \vec{m}_i) \le \frac{1}{\nu}\,(\widehat{\mathrm{K}}_\gamma^i)^n(\vec{n}_i, \vec{m}_i)\, r_\gamma^i(\vec{m}_i).$$

Sum over $\vec{m}_i$, take the log and multiply by $1/n$, we get

$$\begin{aligned}
&\frac{1}{n}\log\left\{\sum_{\vec{m}_i}(\widehat{\mathrm{K}}_\gamma^i)^n(\vec{n}_i,\vec{m}_i)\,r_\gamma^i(\vec{m}_i)\right\}-\frac{1}{n}\log\varepsilon\\
\le&\frac{1}{n}\log\left\{\sum_{\vec{m}_i}(\widehat{\mathrm{K}}_\gamma^i)^n(\vec{n}_i,\vec{m}_i)\right\}\\
\le&\frac{1}{n}\log\left\{\sum_{\vec{m}_i}(\widehat{\mathrm{K}}_\gamma^i)^n(\vec{n}_i,\vec{m}_i)\,r_\gamma^i(\vec{m}_i)\right\}-\frac{1}{n}\log\nu.
\end{aligned}$$

Hence by the squeeze rule,

$$\begin{aligned}
&\lim_{n\to\infty}\frac{1}{n}\log\left\{\sum_{\vec{m}_i}(\widehat{\mathrm{K}}_\gamma^i)^n(\vec{n}_i,\vec{m}_i)\right\}\\
&=\lim_{n\to\infty}\frac{1}{n}\log\left\{\sum_{\vec{m}_i}(\widehat{\mathrm{K}}_\gamma^i)^n(\vec{n}_i,\vec{m}_i)\,r_\gamma^i(\vec{m}_i)\right\}\\
&=\log\frac{1}{n}\log\left[(\exp(-N_i\,\alpha_i(\gamma)))^n\,r_\gamma^i(\vec{n}_i)\right]=-N_i\,\alpha_i(\gamma).
\end{aligned}\tag{50}$$

Also,

$$\begin{aligned}
&\frac{1}{n}\log\mathrm{E}_{\vec{n}_i}[\exp(\gamma\sum_{k=1}^{n}A_i[k])]\\
&=\frac{1}{n}\log\sum_{\vec{n}_i(1),\ldots,\vec{n}_i(n-1),\vec{m}_i}\widehat{\mathrm{K}}_\gamma^i(\vec{n}_i,\vec{n}_i(1))\cdots\widehat{\mathrm{K}}_\gamma^i(\vec{n}_i(n-1),\vec{m}_i)\quad\text{by independence}\\
&=\frac{1}{n}\log\left\{\sum_{\vec{m}_i}(\widehat{\mathrm{K}}_\gamma^i)^n(\vec{n}_i,\vec{m}_i)\right\}.
\end{aligned}\tag{51}$$

Since the log of a Laplace transform is convex, and the limit of convex functions is also convex, by (50) and (51), the function $-N_i\,\alpha_i(\gamma)$ is convex. Since the sum of convex functions is convex, therefore $\Lambda(\gamma)$ is also convex.

□

Now we show that the derivative of $\Lambda(\cdot)$ at zero is negative.

Lemma 7.3 (Derivative at zero) *Let $\alpha_i(\gamma)$ be the function defined in Section 7.2. The derivative of $\alpha_i(\cdot)$ at zero is*

$$\frac{d}{d\gamma}\alpha_i(\gamma)\Big|_{\gamma=0} = -\frac{\lambda_i}{(d_i+(1-d_i)\,\mu_i)\,\lambda_i+\mu_i}\,r_i. \tag{52}$$

Consequently, the derivative of $\Lambda(\gamma) := -\gamma - \sum_{i\in A_h} N_i\,\alpha_i(\gamma)$ at zero is negative, i.e. $\Lambda'(0) < 0$.

Proof: The function $\alpha_i(\cdot)$ is defined implicitly by $\Xi(\alpha_i(\gamma),\gamma) = 1$ and $\alpha_i(0) = 0$. By (47), the derivative of $\alpha_i(\cdot)$ at zero is

$$\begin{aligned}\alpha_i'(0) &= -\frac{(1-\mu_i)\,r_i - (1-\lambda_i)\,(1-\mu_i)\,r_i + \mu_i\,\lambda_i\,r_i}{(1-\lambda_i)+(1-\mu_i)\,d_i - (1-\lambda_i)\,(1-\mu_i)\,(d_i+1) + 2\,\mu_i\,\lambda_i}\\ &= -\frac{\lambda_i}{(d_i+(1-d_i)\,\mu_i)\,\lambda_i+\mu_i}\,r_i.\end{aligned}$$

Thus, the derivative of Λ at zero is

$$\Lambda'(0) = -1 + \sum_{i\in A_h} N_i\,\frac{\lambda_i}{(d_i+(1-d_i)\,\mu_i)\,\lambda_i+\mu_i}\,r_i,$$

which is negative by (6).

□

Now we establish some limiting properties.

Lemma 7.4 *There exists a real number C_i, such that*

$$C_i + \frac{\gamma}{d_i} \le -\alpha_i(\gamma).$$

Consequently,

$$\lim_{\gamma\to\infty} -\alpha_i(\gamma) = \infty \text{ and } \lim_{\gamma\to\infty}\beta_i(\gamma) = \infty.$$

Also,

$$\text{if } \sum_{i\in A_h}\frac{N_i}{d_i} > 1 \text{ then } \lim_{\gamma\to\infty}\Lambda(\gamma) = \infty.$$

Proof: Consider (37), that is

$$e^{-\alpha_i(\gamma)} = ((1-\mu_i)\, e^{(\alpha_i(\gamma)\,(d_i-1))} + \mu_i\, \beta_i(\gamma)^{-1})\,(e^{\gamma}\, r_i + 1 - r_i).$$

Clearly,

$$e^{-\alpha_i(\gamma)} \geq ((1-\mu_i)\, e^{\alpha_i(\gamma)\,(d_i-1)})\, e^{\gamma}\, r_i.$$

Multiply both sides by $e^{-\alpha_i(\gamma)\,(d_i-1)}$ and take the log. We get

$$-\alpha_i(\gamma) \geq \frac{1}{d_i} \log((1-\mu_i)\, r_i) + \frac{\gamma}{d_i},$$

take $C_i = \frac{1}{d_i} \log((1-\mu_i)\, r_i)$.
This implies that $\lim_{\gamma\to\infty} -\alpha_i(\gamma) = \infty$. And by (41), we get $\lim_{\gamma\to\infty} \beta_i(\gamma) = \infty$.
Notice,

$$\begin{aligned}\Lambda(\gamma) = -\gamma - \sum_{i\in A_h} N_i\, \alpha_i(\gamma) &\geq -\gamma + \sum_{i\in A_h} N_i \frac{\gamma}{d_i} + \sum_{i\in A_h} N_i\, C_i \\ &= \gamma\,(\sum_{i\in A_h} \frac{N_i}{d_i} - 1) + \sum_{i\in A_h} N_i\, C_i\end{aligned}$$

Therefore, if $\sum_{i\in A_h} N_i/d_i > 1$ then $\lim_{\gamma\to\infty} \Lambda(\gamma) = \infty$.

□

Lemma 7.5 *If* $\lim_{\gamma\to\infty} \Lambda(\gamma) = \infty$ *then there exists a positive real number* θ *such that* $\Lambda(\theta) = 0$.

Proof: The function Λ is convex, and $\Lambda(0) = 0$, $\Lambda'(0) < 0$. Clearly, if $\Lambda(\gamma) \to \infty$ as $\gamma \to \infty$, then there exists a positive real number θ such that $\Lambda(\theta) = 0$.

□

We now give the proof of Theorem 7.1.

Proof of Theorem 7.1: As remarked on page (74), if there exists a positive real number θ such that $\Lambda(\theta) = 0$, then the constants that satisfy the non-linear system in the statement of the theorem exist. By Lemma 7.4, if $\sum_{i\in A_h} N_i/d_i > 1$ then $\lim_{\gamma\to\infty} \Lambda(\gamma) = \infty$. By Lemma 7.5, this is sufficient for the existence of a positive root for the function Λ.

□

Now we have the tools to show directly that $\tilde{d}_1 > 0$, that is the mean increment for the free process is positive. Equivalently, this means the twisted long-run average number of cells offered (per time slot) to the workload of the hot spot buffer is greater than one. Define,

$$\lambda_i^\gamma := \frac{\lambda_i \beta_i(\gamma)}{1 - \lambda_i + \lambda_i \beta_i(\gamma)}, \tag{53}$$

$$\mu_i^\gamma := \frac{\mu_i (\beta_i(\gamma))^{-1}}{(1-\mu_i) \exp\{\alpha_i(\gamma)(d_i - 1)\} + \mu_i (\beta_i(\gamma))^{-1}}, \text{ and} \tag{54}$$

$$r_i^\gamma := \frac{e^\gamma r_i}{e^\gamma r_i + 1 - r_i}. \tag{55}$$

Let θ be the positive real number such that $\Lambda(\theta) = 0$. For any source in group i, the twisted probabilities of going from ON to OFF, and from OFF to ON are respectively μ_i^θ and λ_i^θ. The twisted probability of offering a cell when ON and the delay is zero is r_i^θ. Note, the long-run average number of cells offered (per time slot) to the workload by the twisted BIDIT sources is

$$\sum_{i \in A_h} N_i \frac{\lambda_i^\theta}{(d_i + (1-d_i)\mu_i^\theta)\lambda_i^\theta + \mu_i^\theta} r_i^\theta.$$

Consider the function $\alpha_i(\cdot)$, which is defined explicitly by $\Xi_i(\alpha_i(\gamma), \gamma) = 1$, see (40) for the definition of Ξ_i. Take the derivative, with respect to γ, on both sides of $\Xi_i(\alpha_i(\gamma), \gamma) = 1$, and evaluate at $\gamma = \theta$. As (46), we get

$$\begin{aligned}
&(1-\lambda_i)\, e^{\alpha_i(\theta)}\, \alpha_i'(\theta) + (1-\mu_i)\, e^{d_i\, \alpha_i(\theta)}\, d_i\, \alpha_i'(\theta)\, (e^\theta r_i + 1 - r_i) \\
&+ (1-\mu_i)\, e^{d_i\, \alpha_i(\theta)}\, e^\theta r_i \\
&- (1-\lambda_i)(1-\mu_i)\, e^{(d_i+1)\alpha_i}\, (d_i+1)\, \alpha_i'(\theta)\, (e^\theta r_i + 1 - r_i) \\
&- (1-\lambda_i)(1-\mu_i)\, e^{(d_i+1)\alpha_i(\theta)}\, e^\theta r_i \\
&+ \mu_i \lambda_i\, e^{2\alpha_i(\theta)}\, 2\, \alpha_i'(\theta)\, (e^\theta r_i + 1 - r_i) + \mu_i \lambda_i\, e^{2\alpha_i(\theta)}\, e^\theta r_i = 0.
\end{aligned} \tag{56}$$

Using (35) and (36), that is

$$\begin{aligned} e^{-\alpha_i(\theta)} &= ((1-\mu_i)\exp\{\alpha_i(\theta)(d_i-1)\} + \mu_i(\beta_i(\theta))^{-1})(e^{\theta} r_i + 1 - r_i) \\ e^{-\alpha_i(\theta)} &= 1 - \lambda_i + \lambda_i \beta_i(\theta), \end{aligned}$$

we can substitute $e^{\alpha_i(\theta)}$ into (56). We get

$$\begin{aligned} &(1-\lambda_i^{\theta})\,\alpha_i'(\theta) + (1-\mu_i^{\theta})\,d_i\,\alpha_i'(\theta) \\ &+ (1-\mu_i^{\theta})\,r_i^{\theta} - (1-\lambda_i^{\theta})(1-\mu_i^{\theta})\,e^{(d_i+1)\alpha_i}\,(d_i+1)\,\alpha_i'(\theta) \\ &- (1-\lambda_i^{\theta})(1-\mu_i^{\theta})\,r_i^{\theta} + 2\,\mu_i^{\theta}\lambda_i^{\theta}\,\alpha_i'(\theta) + \mu_i^{\theta}\lambda_i^{\theta}\,r_i^{\theta} = 0. \end{aligned}$$

Solving for $\alpha_i'(\theta)$, we obtain (after simplification)

$$\alpha_i'(\theta) = -\frac{\lambda_i^{\theta}}{(d_i + (1-d_i)\,\mu_i^{\theta})\,\lambda_i^{\theta} + \mu_i^{\theta}}\,r_i^{\theta}.$$

Thus, the derivative of $\Lambda(\gamma) := -\gamma - \sum_{i\in A_h} N_i\,\alpha_i(\gamma)$ at $\gamma = \theta$ is equal to

$$\Lambda'(\theta) = -1 + \sum_{i\in A_h} N_i \frac{\lambda_i^{\theta}}{(d_i + (1-d_i)\,\mu_i^{\theta})\,\lambda_i^{\theta} + \mu_i^{\theta}}\,r_i^{\theta} = \tilde{d}_1.$$

Since, Λ is a convex function such that $\Lambda(0) = 0$ and $\Lambda'(0) < 0$, then $\Lambda'(\theta) > 0$. Therefore, $\tilde{d}_1 > 0$.

Chapter 8

Conclusions and Future Directions

We explicitly give a change of measure, which transforms our queueing model from a stable to an unstable system. Using both processes (original and twisted) with a splitting importance sampling technique, we were able to find efficient estimates of the CLR for a buffer in an ATM switching fabric with Markov modulated input for two specific models. Through experimentation, we justified the use of our importance sampling algorithm as a variance reduction method.

In the future, the asymptotics for a buffer in an ATM switching fabric with BIDIT sources should be completed. It should be verified that our importance sampling algorithm is asymptotically optimal. One could also try to generalize the method in order to include different queueing networks, e.g. semantic priority queueing, feedback networks.

Appendix A

Stochastic Processes

The purpose of this appendix is two-fold. First we give a quick review of elementary stochastic processes, i.e. sequences of random variables, such as Markov chains and renewal processes. Second we give an introduction to Markov modulated chains, and Markov additive chains.

A.1 Markov Chains

Let S be a finite or countable set (called the state space) and let $\{M[k] : k = 0, 1, 2, \dots\}$ be a sequence of random variables whose ranges are contained in S. The sequence is a Markov Chain if

$$\mathrm{P}(M[n+1] = j | M[0] = i_0, \dots, M = i_n) = \mathrm{P}(M[n+1] = j | M[n] = i_n).$$

This means the distribution of any future random variable $M[n+1]$ is independent of the past $\{M[0], \dots, M[n-1]\}$ and depends only on the present M. If the one-step transition probabilities are independent of time that is for n=0,1,2, ...

$$\mathrm{P}(M[n+1] = j | M[n] = i) = \mathrm{P}(M[1] = j | M[0] = i),$$

then we say that the sequence is a homogenous Markov Chain. The transition kernel K of a homogenous Markov Chain is a non-negative function defined on $S \times S$ as

$$\mathrm{K}(i, j) = \mathrm{P}(M[1] = j | M[0] = i) =: k_{i,j},$$

the one-step transition probabilities.

Example A.1 (Two State Markov Chain) *Let us assume that the incoming stream of cells into our switching fabric is determined by N independent sources which can either be ON or OFF. Let a represent one of these sources and let the probabilities of going from ON to OFF and OFF to ON be p_{10}^a and p_{01}^a respectively, see figure 4. We can model the alternating ON-OFF process $\{M^{[a]}[k] : k = 1, 2, \dots\}$ of source a with state space $\mathcal{S}^{[a]} = \{0 \equiv OFF, 1 \equiv ON\}$ as a two state Markov Chain. The transition kernel $K^a = (k_{i,j}^a)$ of the chain is*

$$K^a = \begin{pmatrix} 1 - p_{01}^a & p_{01}^a \\ p_{10}^a & 1 - p_{10}^a \end{pmatrix}.$$

A.2 Non-Negative Matrices

In applied probability, we often encounter non-negative square matrices, such as the transition kernel for a Markov chain with a finite state space. The Perron-Frobenius Theorem A.1 describes spectral properties of these matrices.

Let $A = (a_{ij})$ be a non-negative $p \times p$ matrix. We say that the matrix A is irreducible if for any pair $(i, j) \in \{1, \dots, p\} \times \{1, \dots, p\}$ there exists a non-negative integer $n = n(i, j)$ such that $A_{ij}^n > 0$. Let $\mathrm{sp}(A) \subset \mathbb{C}$ be the set of eigenvalues for the matrix A. The set $\mathrm{sp}(A)$ is finite. The spectral radius λ_0 of A is defined as

$$\lambda_0 = \max\{|\lambda| : \lambda \in \mathrm{sp}(A)\}.$$

Theorem A.1 (Perron-Frobenius) *Let A be an irreducible matrix with non-negative real entries. The spectral radius λ_0 of A, is strictly positive, and is also an eigenvalue for the matrix A. Furthermore, there exists strictly positive left and right eigenvectors corresponding to the eigenvalue λ_0 which are unique up to a constant.*

Proof: See Section 4.4 in Graham (1987).

□

The spectral radius λ_0 of an irreducible matrix is called the Perron-Frobenius eigenvalue. The Proposition A.1 gives the explicit form of the Perron-Frobenius eigenvalue for a 2×2 irreducible non-negative matrix.

Proposition A.1 *Let A be a 2×2 irreducible matrix with non-negative real entries. The sprectrum of A is real, i.e. $sp(A) \subset \mathbb{R}$, and the Perron-Frobenius eigenvalue λ_0 of the matrix A is equal to*

$$\lambda_0 = \frac{1}{2}\left(tr(A) + \sqrt{(tr(A))^2 - 4\,\det(A)}\right),$$

where $tr(A)$ and $\det(A)$ are respectively the trace and the determinant of the matrix A.

Proof: A 2×2 matrix can have at most two distinct eigenvalues, and complex eigenvalues come in conjugate pairs. Assume the contrary that one of the eigenvalues is non-real (complex). The other eigenvalue must also be non-real, but the Perron-Frobenius Theorem A.1 says that there exists a positive real eigenvalue, which leads to a contradiction. Therefore the spectrum is real.

Take $\lambda \in \mathrm{sp}(A)$, where $A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$.
The eigenvalue λ must satisfy

$$\begin{aligned} 0 = \det(A - \lambda I_2) &= (a_{11} - \lambda)(a_{22} - \lambda) - a_{12}\,a_{21} \\ &= a_{11}\,a_{22} - a_{12}\,a_{21} - \lambda\,(a_{11} + a_{22}) + \lambda^2 = \det(A) - \lambda\,\mathrm{tr}(A) + \lambda^2. \end{aligned}$$

This implies

$$\lambda = \frac{1}{2}\left(\mathrm{tr}(A) + \sqrt{(\mathrm{tr}(A))^2 - 4\,\det(A)}\right), \text{ or } \lambda = \frac{1}{2}\left(\mathrm{tr}(A) - \sqrt{(\mathrm{tr}(A))^2 - 4\,\det(A)}\right).$$

The entries of A are non-negative, so the trace of A is also non-negative. Therefore the spectral radius (Perron-Frobenius eigenvalue) is

$$\lambda_0 = \frac{1}{2}\left(\mathrm{tr}(A) + \sqrt{(\mathrm{tr}(A))^2 - 4\,\det(A)}\right).$$

□

A.3 Stationary Distribution

Let $\{M[k] : k = 1, 2, \dots\}$ be a Markov chain with transition kernel K and state space S. Let π be a probability mass function defined on S. If π satisfies

$$\pi(s) = \sum_{s' \in S} \pi(s')\, \mathrm{K}(s', s),$$

then we call π the stationary distribution of the Markov chain $\{M[k]\}$.

Remark 1: If S is finite and the transition kernel K is irreducible, then by the Perron-Frobenius Theorem A.1 the Markov chain $\{M[k]\}$ has a stationary distribution π and it is unique.

Remark 2: Under certain conditions, $\lim_{n\to\infty} \mathrm{K}^n(s', s) = \pi(s)$ for all $s, s' \in S$, see McDonald (1994). Hence, $\pi(s)$ is the long-run probability of being in state s.

A.4 Harmonic Functions

Let G be a region in the plane. If the function $u : G \to \mathbb{R}$ satisfies the Laplace partial differential equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0,$$

then we say that the function is harmonic. There exists the following characterization for continuous harmonic functions, see Conway (1978). If the function $u : G \to \mathbb{R}$ is continuous then it is a harmonic function if and only if for any closed disc $\bar{B}(a; r)$ about a of radius r in G we have the following averaging property

$$u(a) = \frac{1}{2\pi} \int_0^{2\pi} u(a + r\, e^{i\theta})\, d\theta.$$

There exists an analog to this for Markov chains. Let $\{M[k]\}$ be a Markov chain

with countable state space S and transition kernel K. A positive function with domain S is called a harmonic function for the transition kernel K if it satisfies the following averaging property

$$h(m) = \sum_{m' \in S} \mathrm{K}(m, m')h(m').$$

If there exists such a function h we can then h-transform the transition kernel K to construct a new transition kernel $\mathcal{K}$. To do this let $\mathcal{K}(m, m') = \mathrm{K}(m, m')\, h(m')/h(m)$. Clearly the new kernel is a probability transition function since $\mathcal{K} \geq 0$ and

$$\sum_{m' \in S} \mathcal{K}(m, m') = 1.$$

So we have just constructed a new Markov chain $\{\mathcal{M}[k]\}$ with transition kernel $\mathcal{K}$ from the original Markov chain $\{M[k]\}$. We call this new chain the conjugate process. It is evident that $h \equiv 1$ is always a harmonic function for the kernel K, and if we apply the h-transform utilizing this function the conjugate process is simply the original chain. The h-transform is a very important tool for the twist technique as developed by McDonald (1998).

Remark: If h is a harmonic function for the transition kernel K, then $\{h(M)\}$ is a martingale. This follows from

$$h(m) = \sum_{m' \in S} \mathrm{K}(m, m')h(m') = \mathrm{E}[h(M[1])|M[0] = m] = \mathrm{E}[h(M[n+1])|M[n] = m].$$

A.5 Renewal Theory

Let $\{T[k] : k = 0, 1, \dots\}$ be a sequence of increasing random variables with range $\mathbb{N}_0 = 0, 1, 2, \dots$ where $0 = T[0] < T[1]$. The increments (inter-arrivals) are defined to be $X[n] = T[n] - T[n-1]$. We say that the process $\{T[k] : k = 0, 1, 2, \dots\}$ is a renewal process if the increments $\{T[k] - T[k-1] : k = 1, 2, \dots\}$ are independent strictly positive random variables. We also say that the renewal process is homogeneous if the inter-arrivals $X[n]$ are independent identically distributed with mean μ for $n = 2, 3, \dots$

Example A.2 (Markov Chain) *A recurrent homogeneous Markov chain* $\{M[k]\}$ *has a homogeneous renewal process embedded in it. Suppose that the initial state is* i*, define the renewals as* $T[0] = 0$ *and for* $k = 1, 2, \ldots,$

$$T[k] = \min\{n : n > T[k-1], M[n] = i\}.$$

That is the renewals are the return times to state i*. By the Markovian property the inter-arrivals* $\{X[k]\}$ *are independent and indentically distributed.*

Let $N[t] = \sup\{n : T[n] \leq t\}$. The variable $N[t]$ represents the number of renewals up to time t thus we call it the renewal counting process. The next result shows the rate at which $N[t]$ goes to infinity when t goes to infinity.

Theorem A.2 *If* $N[t]$ *is the counting process of a homogeneous renewal process, then*

$$P\left(\lim_{t\to\infty} \frac{N[t]}{t} = \frac{1}{\mu}\right) = 1,$$

where μ *is the mean inter-arrival time.*

proof: see proposition 6.2.2 in McDonald (1994)

□

We call $1/\mu$ the rate of the renewal process. Now we are ready to define the renewal reward process which leads to a powerful result for cycles which are defined from renewals, such cycles are called regenerative cycles. Let $\{T[k]\}$ be a homogeneous renewal process, with inter-arrivals $\{X[k]\}$. To each $X[n]$ associate a random variable R_n. If the variables $(X[k], R_k)$ are independent and identically distributed with mean $(\mu, \mathrm{E}[R])$ for $k = 2, 3, \ldots,$ then we call $\{R_k : k = 1, 2, \ldots\}$ the renewal reward process associated to $\{T[k] : k = 0, 1, 2 \ldots\}$. If we have a completed cycle every time a renewal occurs, the average reward for completed cycles by time t is

$$\frac{1}{t}\sum_{n=1}^{N[t]} R_n.$$

Theorem A.3 *Let $\{R_k\}$ be a renewal reward process where the expected reward per cycle is $E[R]$ and μ is the mean of the inter-arrivals. If $E[R]$ and μ are both finite then with probability one*

$$\frac{1}{t}\sum_{n=1}^{N[t]} R_n \to \frac{E[R]}{\mu}.$$

proof: The sequence $\{R_k\}_{k=2}^{\infty}$ is independent and identically distributed and E[R] is finite thus by the strong law of large numbers

$$\frac{1}{N[t]}\sum_{n=1}^{N[t]} R_n \to \mathrm{E}[R].$$

So by Theorem A.2

$$\frac{1}{t}\sum_{n=1}^{N[t]} R_n = \frac{N[t]}{t}\frac{\sum_{n=1}^{N[t]} R_n}{N[t]} \to \frac{\mathrm{E}[R]}{\mu}.$$

□

The theorem tells us that the long run average reward is equal to the expected reward for a cycle divided by the expected length of a cycle.

Example A.3 (Alternating Process) *Let $\{M^{[a]}[k]\}$ be the two-state Markov chain from example A.1 that describes the ON-OFF transitions of the source a. This Markov Chain can be described as a renewal process. Let us say that the process is initially ON. It will remain ON for a certain time N_1 and then it will be OFF for a certain time F_1. So if we denote N_n and F_n to be the sojourn time of the nth ON and nth OFF period respectively, then $\{N_n + F_n\}$ is a sequence of independent and indentically distributed random variables. Let the inter-arrivals of the renewal be $X_n = N_n + F_n$, that is a cycle consist of an ON and OFF period. We have just constructed an alternating renewal process. The last theorem tells us that if the reward is the number of times the chain remains ON, then the long-run proportion of time that the source remains ON is*

$$\frac{E[N_1]}{E[N_1] + E[F_n]}.$$

Now the sojourn time N_1 has a geometric distribution with parameter p_{10}^a, that is $P(N_1 = k) = (1 - p_{10}^a)^{k-1} p_{10}^a$. Therefore the mean of N_1 is $1/p_{10}^a$. Similarly the mean

of F_1 is $1/p^a_{01}$. Therefore the long-run probability that the source a is ON is

$$\frac{1/p^a_{10}}{1/p^a_{10} + 1/p^a_{01}} = \frac{p^a_{01}}{p^a_{01} + p^a_{10}}.$$

A.6 Markov Modulated Sequence

Let $\{A[n] : n = 1, 2, \dots\}$ be a sequence of discrete valued random variables. Let $\{M[n] : n = 0, 1, 2, \dots\}$ be a Markov chain with transition kernel K_A and state space $\mathcal{S}$. If we sample the random variable $B[n]$ according to the value of $(M[n-1], M[n])$ then $\{B[n]\}$ is a Markov modulated sequence, and we say that $\{B[n]\}$ is modulated by the Markov chain $\{M[n]\}$.

Define the probability mass functions G_{ij} for all $i, j \in \mathcal{S}$ as

$$\begin{aligned} G_{ij}(x) &:= \mathrm{P}(A[1] = x, M[1] = j | M[0] = i) \\ &= \mathrm{K}_A(i,j)\, \mathrm{P}(A[1] = x | M[0] = i, M[1] = j). \end{aligned} \tag{57}$$

The matrix $G = (G_{ij})$ is called the semi-Markov kernel.

Example A.4 (Bernoulli Interrupted) *Let $\{M^{[a]}[k]\}$ be the two-state Markov chain from Example A.1 that describes the ON-OFF transitions of the source a. Assume that the transition occur instantaneously at the end of the time slots. If the source is ON at the end of time slot n then during time slot $n+1$ a cell is offered by source a with probability ρ^a. If the source is OFF at the end of the time slot then no cells are offered.*

Let $B^a[n]$ be the number of cells offered by source a during the nth time slot. The process $\{B^a[n]\}$ is a Markov modulated sequence with semi-Markov kernel G^a. Note, the random variable is only sampled according to the value of $M^a[n-1]$, thus

$$P(B^a[1] = x | M^a[0] = i, M^a[1] = j) = P(B^a[1] = x | M^a[0] = i).$$

Hence the semi-Markov kernel G^a is equal to

$$\begin{pmatrix} (1-p_{01}^a)\,P(B^a[1]\in\cdot|M^a[0]=0) & p_{01}^a\,P(B^a[1]\in\cdot|M^a[0]=0) \\ p_{10}^a\,P(B^a[1]\in\cdot|M^a[0]=0) & (1-p_{10}^a)\,P(B^a[1]\in\cdot|M^a[0]=0) \end{pmatrix}.$$

We call the source a a Bernoulli interrupted source; during the ON periods source a generates cells according to a Bernoulli process.

A.7 Markov Additive Chains

Random Walk

Let $\{X[k] : k = 1, 2, \dots\}$ be a sequence of independent and indentically distributed random variables with range in $\mathbb{Z}$. If $S[n] = S[0] + X[1] + \dots + X[n]$ for $n = 1, 2, \dots$ then the sequence $\{S[k] : k = 0, 1, \dots\}$ is called a random walk ON $\mathbb{Z}$. Let p be the probability mass function for the increments that is $p(x) = \mathrm{P}(X[1] = x)$. The generating function of the random walk is

$$\mathcal{M}(t) = \mathrm{E}[\exp(t\,X[1])] = \sum_{x\in\mathbb{Z}} \exp(t\,x)\,p(x).$$

If the mean increment $\mu = \mathrm{E}[X[1]]$ is non-zero then $\mathcal{M}(t)$ is a strictly convex function. This follows from the strict convexity of $\exp(t\,x)$ when $x \neq 0$ and $\mathrm{P}(X[1] \neq 0) > 0$.

There exists a generalization of a random walk on $\mathbb{Z}^d$ where the increments are modulated by some Markov chain.

Markov Additive Chain

Let $\{\xi[k] : k = 1, 2, \dots\}$ be a Markov modulated sequence of $\mathbb{Z}^d$ valued random variables, with semi-Markov kernel G, see (57), and underlying Markov chain $\{M[k] : k = 0, 1, 2, \dots\}$. We assume the Markov chain $\{M[k]\}$ is irreducible with transition kernel K_A and finite state space $S = 1, 2, \dots, N$. Let $S[k] = S[0] + \sum_{i=1}^{k} \xi[k]$. Thus

$\{S[k]\}$ is a random walk with Markov modulated increments. We call the sequence $\{(S[k], M[k]) : k = 0, 1, 2, \ldots\}$ a Markov additive chain (MA-chain) and K defined as

$$\mathrm{K}[(s, m), (s', m')] := G_{m,m'}(s' - s).$$

The operator K is called the MA transition kernel.

We define the generating function of the MA transition kernel as

$$\widehat{\mathrm{K}}(m, m'|\alpha) = \mathrm{E}[\exp(\xi[1] \cdot \alpha)\, I\{M[1] = m'\}|M[0] = m],$$

where $\cdot$ is the dot product on $\mathbb{R}^d$ and $\alpha \in \mathbb{R}^d$. Ney and Nummelin (1987) studied the existence and regularity properties of the eigenvalues and eigenfunctions of this "Feynman-Kac" operator. Define the domain of convergence for the elements of the generating function as

$$D(m, m') := \left\{\alpha \in \mathbb{R}^d : \widehat{\mathrm{K}}(m, m'|\alpha) < \infty\right\}.$$

Take $\alpha \in \cap_{m,m'} D(m, m')$ then the matrix $\widehat{\mathrm{K}}_\alpha$ whose ij th element is $\widehat{\mathrm{K}}(i, j|\alpha)$ has non-negative real entries. Thus this matrix has a Perron-Frobenius eigenvalue $e^{\Lambda(\alpha)}$ with associated positive right and left eigenvectors.

Define the function Ψ on $\mathbb{R}^{d+1}$ as

$$\Psi(\alpha, \xi) := \mathrm{E}_{(\vec{0},m)}[\exp(\alpha \cdot S[\tau_m] - \xi\, \tau_m)], \tag{58}$$

where τ_m is the first return time to the state m. Define the following two sets $\mathcal{D}$ and $\mathcal{D}_1$ as follows

$$\mathcal{D}_1 := \left\{\alpha \in \mathbb{R}^d : \exists \Lambda \in \mathbb{R} \text{ such that } \Psi(\alpha, \Lambda) < \infty\right\},$$

and

$$\mathcal{D} := \left\{(\alpha, \xi) \in \mathbb{R}^{d+1} : \Psi(\alpha, \xi) < \infty\right\}.$$

Lemma A.1 *The function Ψ as defined in (58) satisfies*

$$\Psi(\alpha,\xi)=\sum_{n=1}^{\infty}e^{-\xi n}\sum_{\vec{w}\in W_m(n)}\prod_{i=1}^{n}\widehat{K}(w_{i-1},w_i|\alpha), \tag{59}$$

where

$$W_m(n):=\{(w_0,w_1,\ldots,w_n)\in S^n : w_0=w_n=m, w_i\neq m \text{ for } i=1,\ldots,n-1\}.$$

.

proof: For $z\in\mathbb{Z}^d$ define the set

$$\Upsilon_z(n):=\left\{(c_1,\ldots,c_n)\in\times_{i=1}^{n}\mathbb{Z}^d : c_1+\ldots+c_n=z\right\}.$$

By the conditional independence property for Markov modulated sequences,

$$\mathrm{P}_m(S[n]=z,\tau_m=n)=\sum_{\vec{w}\in W_m(n)}\sum_{\vec{c}\in\Upsilon_z(n)}\prod_{i=1}^{n}P(\xi[i]=c_1,M[i]=w_i|M[i-1]=w_{i-1}).$$

If we multiply both side by $\exp(\alpha\cdot z)$ and sum over z then

$$\sum_{z\in\mathbb{Z}^d}\exp(\alpha\cdot z)\,\mathrm{P}_m(S[n]=z,\tau_m=n)=\sum_{\vec{w}\in W_m(n)}\prod_{i=1}^{n}\widehat{\mathrm{K}}(w_{i-1},w_i|\alpha).$$

Through conditioning and by the definition of Ψ,

$$\begin{aligned}\Psi(\alpha,\xi)&=\sum_{n=1}^{\infty}e^{-\xi n}\sum_{z\in\mathbb{Z}^d}\exp(\alpha\cdot z)\,\mathrm{P}_m(S[n]=z,\tau_m=n)\\&=\sum_{n=1}^{\infty}e^{-\xi n}\sum_{\vec{w}\in W_m(n)}\prod_{i=1}^{n}\widehat{\mathrm{K}}(w_{i-1},w_i|\alpha).\end{aligned}$$

□

Theorem A.4 *If the state space S of the Markovian part $\{M[k]\}$ is finite then*

$$\mathcal{D}_1=\cap_{m,m'}D(m,m').$$

Moreover, if the sets $D(m,m')$ are open for all $m,m'\in S$, then

1. $\mathcal{D}_1$ *is open,*

2. $\mathcal{D}$ *is open, and*

3. $\Psi(\alpha, \Lambda(\alpha)) = 1$ *for all* $\alpha \in \mathcal{D}_1$,
   where $\Lambda(\alpha)$ *is the log Perron-Frobenius eigenvalue of* $\widehat{K}_\alpha$.

proof: The inclusion $\mathcal{D}_1 \subset \cap_{m,m'} D(m,m')$ is an immediate consequence of Lemma A.1. If $\alpha \in D(m,m')$ for all $m, m' \in S$, then

$$\kappa(\alpha) := \max_{(m,m') \in S^2} | \widehat{\mathrm{K}}_\alpha(m,m) |$$

is finite, since $\#S = N < \infty$. The cardinality of $W_m(n)$ is $(\#S)^{n-1} = N^n/N$. Take ξ such that $\kappa(\alpha) N < e^\xi$. By Lemma A.1,

$$\Psi(\alpha, \xi) \leq \sum_{n=1}^{\infty} e^{-\xi n} (\kappa(\alpha) N)^n / N < \infty.$$

Thus $\alpha \in \mathcal{D}_1$, which implies $\cap_{m,m'} D(m,m') \subset \mathcal{D}_1$. Therefore, $\mathcal{D}_1 = \cap_{m,m'} D(m,m')$.

Assume that the sets $D(m,m')$ are open for all $m, m' \in S$. We will now proceed to prove statements 1,2, and 3.

1. Trivial.

2. Follows from Lemma 4.3 in Ney and Nummelin (1987).

3. See Ney and Nummelin (1987).

□

In the next example Markov additive chains with boundary are introduced. They are very useful to model queues with Markov modulated input.

Example A.5 (Boundary) *Let $\{M[k] : k = 0, 1, \dots\}$ be the Markov chain which modulates the independent sources. Let $\{\xi[k] : k = 1, 2, \dots\}$ be the sequence of increments with range $\mathbb{Z}$. For example the increment at time k could be the arrivals during time k minus a service. Since a queue can not have a negative number of customers we will define the additive part (the number of customers in the queue) in the following way*

$$S[k] = (S[k-1] + \xi[k])^+,$$

where $x^+ := max\{x, 0\}$. The following sequence $\{(S[k], M[k])\}$ is a Markov additive chain with boundary. It has an associated free chain $\{(S^\infty[k], M[k])\}$, where the additive part is unrestricted (it can be negative), i.e.

$$S^\infty[k] = S^\infty[k-1] + \xi[k] = S^\infty[0] + \sum_{i=1}^{k} \xi[i].$$

The free chain is simply a Markov additive chain.

Appendix B

Proofs and Technical Results

Lemma B.1 *Consider the function* $f : \mathbb{R} \to \mathbb{R}$ *defined as*

$$x \mapsto a_1 \exp(b_1 x) + a_2 \exp(b_2 x) + a_3 \exp(b_3 x) - a_4 \exp(b_4 x),$$

where $a_1, \ldots, a_4, b_1, \ldots, b_4 \in (0, \infty)$ *and* $b_4 > b_1, b_2, b_3$. *The function* f *has at most one stationary point.*

Proof: The derivative of f is

$$a_1 b_1 \exp(b_1 x) + a_2 b_2 \exp(b_2 x) + a_3 b_3 \exp(b_3 x) - a_4 b_4 \exp(b_4 x).$$

By setting the derivative equal to zero we get

$$\frac{1}{a_4 b_4} \left(a_1 b_1 e^{(b_1 - b_4) x} + a_2 b_2 e^{(b_2 - b_4) x} + a_3 b_3 e^{(b_3 - b_4) x} \right) = 1. \tag{60}$$

The function on the left hand side of (60) is a strictly decreasing function, since $b_4 > b_1, b_2, b_3$. This means the left hand side of (60) can equal one for at most one x. Therefore, the function f has at most one stationary point.

□

Proof of Theorem 2.1(a): The stochastic process $\{\vec{N}[n]\}$ is a Markov chain with

transition kernel K_A and state space $\mathcal{S}_A$, see Section 2.2. We denote $K_A^n[\vec{n}, \vec{m}]$ as the transition probability from $\vec{n}$ to $\vec{m}$ in n time slots. We want to show that for all $(\vec{n}, \vec{m}) \in \mathcal{S}_A \times \mathcal{S}_A$ there exists a positive integer T such that $K_A^T[\vec{n}, \vec{m}] > 0$, i.e. the chain $\{\vec{N}[n]\}$ is irreducible.

Let $\delta_d := \max\{d_1, d_2, \ldots, d_M\}$. Suppose that at time zero the BIDIT sources are in state $\vec{n}$. Consider the trajectory from time slot zero to time slot δ_d, where the sources become idle (OFF) as soon as possible and remain idle until the time slot δ_d. For example, at time zero there are n_{ij} sources in the group i which are ON with an associated delay j. At time slot j these n_{ij} sources will have an associated delay 0, we then assume that at the time slot $j+1$ they will become idle and will remain idle until time slot δ_d. Denote $P_{0,\delta_d}(\vec{n}, \vec{m})$ as the probability associated to this trajectory. We have

$$K_A^{\delta_d}[\vec{n}, \vec{0}] \geq P_{0,\delta_d}(\vec{n}, \vec{0}) > 0.$$

At time δ_d, the BIDIT sources are in state $\vec{0}$, i.e. the sources are all off. Our goal is to find a trajectory, with positive probability, such that at the time slot $2\,\delta_d$ the sources are in state $\vec{m}$.

Consider the sources in group i. These sources will remain idle until the time slot $2\,\delta_d - (d_i - 1)$. Assume that m_{i1} sources from group i become active (ON) at the time slot $2\,\delta_d - (d_i - 1)$ and remains active. This means at the time slot $2\,\delta_d$, from group i, there will be m_{i1} ON sources with an associated delay 1. Similarly, of the remaining idle sources from group i at the time slot $2\,\delta_d - (d_i - 2)$ we assume m_{i2} become active and remain active. Thus, at the time slot $2\,\delta_d$, from group i, there will be m_{i2} ON sources with an associated delay 2.

To be more general, at the time slot $2\,\delta_d - (d_i - j)$ we assume m_{ij} become active and remain active. Hence, at the time slot $2\,\delta_d$, from group i, there will be m_{ij} ON sources with an associated delay j. Denote $P_{\delta_d, 2\,\delta_d}(\vec{0}, \vec{m})$ as the probability associated

to this possible trajectory from the time δ_d to the time slot $2\,\delta_d$. We have

$$\mathrm{K}_A^{\delta_d}[\vec{0},\vec{m}] \geq P_{0,\delta_d}(\vec{0},\vec{m}) > 0.$$

Therefore, by letting $T = 2\,\delta_d$,

$$\begin{aligned}\mathrm{K}_A^{T}[\vec{n},\vec{m}] &\geq \mathrm{K}_A^{\delta_d}[\vec{n},\vec{0}]\,\mathrm{K}_A^{\delta_d}[\vec{0},\vec{m}] \\ &\geq P_{0,\delta_d}(\vec{n},\vec{0})\,P_{\delta_d,2\,\delta_d}(\vec{0},\vec{m}) > 0,\end{aligned}$$

and the Markov chain $\{\vec{N}[n]\}$ is irreducible. Actually, we can similarly show that for any $n \geq 2\,\delta_d$ and any two states $\vec{n}$ and $\vec{m}$, the n-step transition probability from $\vec{n}$ to $\vec{m}$ is strictly greater than zero, i.e. $\mathrm{K}_A^n[\vec{n},\vec{m}] > 0$.

□

Proof of Theorem 2.1(b): Let NB and B be the index sets for the groups of non-bursty sources and the groups of bursty sources, respectively. The states of the sources of the non-bursty sources are deterministic, and periodic. Consider the group $i \in NB$ of non-bursty sources. At every d_i time slots the sources in group i return to their initial state. Hence, if n_0 is the lowest common multiple of $\{d_i : i \in NB\}$, then after n_0 time slots the non-bursty sources have returned to their initial state.

Let $\vec{s}_n = (\vec{n}_i : i \in NB)$ be the initial state of the non-bursty sources. From the initial state we know with probability one the next state of the non-bursty sources, and from this state we know the next state of the non-bursty source, and so on until we return to the initial state $\vec{s}_n$ after n_0 time slots. Any state that was not hit in the first n_0 time slots will never be hit; all other states are recurrent and are in the same communication class as $\vec{s}$. Let $C[\vec{s}_n]$ be the communication class of the initial state $\vec{s}_n$.

From Theorem 2.1(a), the states of the bursty sources are all in one communication class, let $C(B)$ be the set of the bursty sources. Actually, if n is large enough, for all $\vec{s}_b = (\vec{n}_i : i \in B)$ and $\vec{t}_b = (\vec{m}_i : i \in B)$, the n-step transition probability from

$\vec{s}_b$ to $\vec{t}_b$ is greater that zero.

Let $\mathcal{S}_A^r = C(B) \times C[\vec{s}_b] \subset \mathcal{S}_A$. Take $\vec{n} \in S_A^r$ as the initial state and $\vec{m} \in \mathcal{S}_A^r$ as some other state. There exists a positive integer $1 \leq t \leq n_0$ such that the non-bursty sources are in state $\vec{t}_n = (\vec{m}_i : i \in NB)$ after t time slots. From the periodicity of the non-bursty sources, the hitting times for the state $\vec{t}_n$ are $\{t + i\, n_0 : i = 1, 2, \dots\}$. Take $k \in \{t + i\, n_0 : i = 1, 2, \dots\}$ large enough such that the k-step transition from $\vec{s}_b = (n_i : i \in B)$ to $\vec{t}_b = (m_i : i \in B)$ is greater than zero. Hence, from the independence of the sources, for all $\vec{n}, \vec{m} \in \mathcal{S}_A^r$ there exists a positive integer k such that $\mathrm{K}_A^k[\vec{n}, \vec{m}] > 0$. Therefore the transition kernel K_A defined on $\mathcal{S}_A^r$ is irreducible.

□

Proof of Theorem 7.2:

Case I ($i \in A_n$): The sources in group i generate cells that do not pass through the hot spot. Thus, the total cell offered to the workload of the hot spot by the sources in group i during time slot n is always zero, i.e. $A_i[n] = 0$ for all n. We have

$$\begin{aligned}
\widehat{\mathrm{K}}_\gamma^i(\vec{n}_i, \vec{m}_i) &= \mathrm{E}[\exp(\gamma\, A_i[1])\, I\{\vec{N}_i = \vec{m}_i\} | \vec{N}_i = \vec{n}_i] \\
&= \mathrm{E}[I\{\vec{N}_i = \vec{m}_i\} | \vec{N}_i = \vec{n}_i] \\
&= \mathrm{K}_i[\vec{n}_i, \vec{m}_i].
\end{aligned}$$

Hence $\widehat{\mathrm{K}}_\gamma^i$ is a stochastic matrix, with eigenvalue $1 = \exp(-N_i\, \alpha_i(\gamma))$ (for all $i \in A_n$, for all γ, define $\alpha_i(\gamma) := 0$) and right eigenvector $(1, \dots, 1)^t = r_\gamma^i$.

Case II ($i \in A_h$ **and** $d_i = 1$): There exists $\alpha_i = \alpha_i(\gamma)$, and $\beta_i = \beta_i(\gamma)$ that satisfy the equations (37-38). Similar to the implication of (22) from (12) and (13), it is easy to show using (37-38) that the ratio $r_\gamma^i(m_{i0})/r_\gamma^i(n_{i0})$ is equal to

$$\beta_i^{m_{i0}} \left(\frac{\beta_i^{-1}}{1 - \mu_i + \mu_i\, \beta_i^{-1}} \right)^{n_{i0}} (e^\gamma\, r_i + 1 - r_i)^{-n_{i0}}\, \frac{(1 - \lambda_i + \lambda_i\, \beta_i)^{N_i}}{(1 - \lambda_i + \lambda_i\, \beta_i)^{N_i - n_{i0}}}. \qquad (61)$$

The generating function for group i is equal to

$$\begin{aligned}
&\widehat{\mathrm{K}}_\gamma^i(n_{i0}, m_{i0}) \\
&= \mathrm{E}[\exp(\gamma\, A_i[1])\, I\{\vec{N}_i = \vec{m}_i\} | \vec{N}_i = \vec{n}_i] \\
&= \mathrm{K}_i(\vec{n}_i, \vec{m}_i) \times \sum_{z=0}^{n_{i0}} e^{\gamma z} \mathrm{Bin}(n_{i0}, z, r_i) \\
&= \left\{ \sum_{x+y=m_{i0}} \mathrm{Bin}(n_{i0}, x, 1-\mu_i)\, \mathrm{Bin}(N_i - n_{i0}, y, \lambda_i) \right\} \times \left\{ \sum_{z=0}^{n_{i0}} e^{\gamma z} \mathrm{Bin}(n_{i0}, z, r_i) \right\}.
\end{aligned}$$

Using (61), $\widehat{\mathrm{K}}_\gamma^i(n_{i0}, m_{i0})\, r_\gamma^i(m_{i0}) / r_\gamma^i(n_{i0})$ is equal to

$$\begin{aligned}
&(1 - \lambda_i + \lambda_i\, \beta)^{N_i} \times \left\{ \sum_{z=0}^{n_{i0}} \mathrm{Bin}\left(n_{i0}, z, \frac{e^\gamma\, r_i}{e^\gamma\, r_i + 1 - r_i} \right) \right\} \times \\
&\left\{ \sum_{x+y=m_{i0}} \mathrm{Bin}\left(n_{i0}, x, \frac{1-\mu_i}{1 - \mu_i + \mu_i\, \beta_i^{-1}} \right) \mathrm{Bin}\left(N_i - n_{i0}, y, \frac{\lambda_i\, \beta_i}{\lambda_i\, \beta_i + 1 - \lambda_i} \right) \right\}.
\end{aligned}$$

Hence,

$$\sum_{m_{i0}} \widehat{\mathrm{K}}_\gamma^i(n_{i0}, m_{i0})\, r_\gamma^i(m_{i0}) / r_\gamma^i(n_{i0}) = (1 - \lambda_i + \lambda_i\, \beta)^{N_i} = \exp(-N_i\, \alpha_i) \quad \text{by (38)}.$$

Case III ($i \in A_h$ **and** $d_i > 1$): There exists $\alpha_i = \alpha_i(\gamma)$, and $\beta_i = \beta_i(\gamma)$ that satisfy the equations (37-38). Similar to the implication of (21) from (12) and (13), it is easy to show using (37-38) that the ratio $r_\gamma^i(\vec{m}_i) / r_\gamma^i(\vec{n}_i)$ is equal to

$$\begin{aligned}
&\frac{(\beta_i^{-1})^{(n_{i0} - m_{i,d_i-1})}\, (e^{\alpha_i\,(d_i-1)})^{m_{i,d_i-1}}}{((1-\mu_i)\, e^{\alpha_i\,(d_i-1)} + \mu_i\, \beta_i^{-1})^{n_{i0}}} \frac{\beta_i^{(m_{i0} - n_{i1})}}{(1 - \lambda_i + \lambda_i\, \beta_i)^{f_i}} \times \\
&(e^\gamma\, r_i + 1 - r_i)^{-n_{i0}}\, (1 - \lambda_i + \lambda_i\, \beta_i)^{N_i}.
\end{aligned} \tag{62}$$

The generating function for group i is equal to

$$\begin{aligned}
&\widehat{\mathrm{K}}^i_\gamma(n_{i0}, m_{i0}) \\
&= \mathrm{E}[\exp(\gamma\, A_i[1])\, I\{\vec{N}_i = \vec{m}_i\} | \vec{N}_i = \vec{n}_i] \\
&= \mathrm{K}_i(\vec{n}_i, \vec{m}_i) \times \sum_{z=0}^{n_{i0}} e^{\gamma z} \mathrm{Bin}(n_{i0}, z, r_i) \\
&= \mathrm{Bin}(n_{i0}, m_{i,d_i-1}, 1-\mu_i)\, \mathrm{Bin}(f_i, m_{i0} - n_{i1}, \lambda_i) \times \left\{ \sum_{z=0}^{n_{i0}} e^{\gamma z} \mathrm{Bin}(n_{i0}, z, r_i) \right\}.
\end{aligned}$$

Using (62), $\widehat{\mathrm{K}}^i_\gamma(\vec{n}_i, \vec{m}_i)\, r^i_\gamma(\vec{m}_i)/r^i_\gamma(\vec{n}_i)$ is equal to

$$\begin{aligned}
&(1-\lambda_i+\lambda_i\,\beta)^{N_i} \times \left\{ \sum_{z=0}^{n_{i0}} \mathrm{Bin}\left(n_{i0}, z, \frac{e^\gamma\, r_i}{e^\gamma\, r_i + 1 - r_i}\right) \right\} \times \\
&\mathrm{Bin}\left(n_{i0}, m_{i,d_i-1}, \frac{1-\mu_i}{1-\mu_i+\mu_i\,\beta_i^{-1}}\right)\, \mathrm{Bin}\left(f_i, m_{i0} - n_{i1}, \frac{\lambda_i\,\beta_i}{\lambda_i\,\beta_i + 1 - \lambda_i}\right).
\end{aligned}$$

Hence,

$$\sum_{m_{i0}} \widehat{\mathrm{K}}^i_\gamma(n_{i0}, m_{i0})\, r^i_\gamma(m_{i0})/r^i_\gamma(n_{i0}) = (1-\lambda_i+\lambda_i\,\beta)^{N_i} = \exp(-N_i\,\alpha_i) \quad \text{by (38)}.$$

□

Appendix C

C++ Programs Using SimS

C.1 Source Code for Model I

The Cell Header File

```
#ifndef CellH
#define CellH

#include <sims.h>
#include "prog.h"

class cell: public sim_message
{
public:
    cell(Destination dest);
    ~cell();
    const char *type() const;
    Destination get_dest();
private:
    Destination WhereTo;
};

#endif
```

Cell Module

```
#include "cell.h"

cell::cell(Destination dest)
{
     WhereTo=dest;
}
cell::~cell() {}

const char *cell::type() const
{ return ("cell"); }

Destination cell::get_dest()
{ return (WhereTo); }
```

The Buffer Header File

```
#ifndef Buffer_EntityH
#define Buffer_EntityH

#include <iostream.h>  //for debugging
#include <stdlib.h>
#include <sims.h>
#include <math.h>
#include <string.h>  //for strcmp
#include "cell.h"
#include "prog.h"
class buffer_entity: public sim_entity
{
public:
     sim_message_queue *Queue[7];
     buffer_entity::buffer_entity(
          const char *name, int hcapacity,
          double faststep);
     ~buffer_entity();
     void suspend_normal(sim_time time);
     void initialize_twisted(sim_time time,
          int state, double next);
     void reinitialize_normal();
     void initialize_normal();
     void suspend_twisted();
     void execute();
     char* GetName();
     void AdjustTime();
     static sim_bool twist;
     sim_bool IsHotSpot();
     sim_bool wait,intermediate,last;
     void SetWorkload(int workload);
     static void Clear();
     void SusPend();
     void Activate(sim_time when);
     int QLength,AType,NumberOfTimeSlots,
     int CellsToHotspot;
     sim_time steps;
     sim_bool Empty;
     static sim_bool WarmUp;
     static double NumOfDCycles;
     static sim_bool Continue;
     static int hotNumber;
     static int ExtraFactor;
     static int LostCellsCount;
     static sim_bool untwistFlag;
     static sim_bool cycleFlag;
     static sim_bool Acycle[2];
     static int CycleCount;
     static sim_bool lastCycle;
     static int xbool[2];
     static sim_bool HalfTime;
     static int ServeBin;
     static sim_time Time;
     char Qname[7][20];
     int POINTER, HotSpotCapacity,
          Workload;
     int CarriedCells, OfferedCells;
     int LostCells, ServeState,
          TimeSlotsCount;
     int NextBuffer[7],BufferState[7];
     double FastStep, NextStep;
     sim_bool EndCycle,Twist,FirstHit;
private:
     char Name[40];
     sim_generator *pRandom;
     sim_bool first,Outlet,TBuffer,Inlet;
     sim_bool EmptyBuffer,HotSpot;
     double transmissionRate;
     double floatCells;
     int Order;
     sim_time NextTime;
};
#endif
```

Buffer Module

```
#include "buffer.h"

buffer_entity::buffer_entity(
         const char *name, int hcapacity,
         double faststep):sim_entity(name)
{
     strcpy(Name,name);

     HotSpotCapacity=hcapacity;

     char queuename[6];
     int i;
     for (i=0;i<=6;i++)
       {
        sprintf(queuename,"%s%d","Q",i);
        create_message_queue(queuename);
        Queue[i]=find_queue(queuename);
       }

     /*

  (source
   multiplexers)
        3  -->    (internal buffer)
                   | --> 1  -->
        4  -->             \
                            |--> 0 (hot spot)
        5  -->             /
                   | --> 2  -->
        6  -->    (internal buffer)

       */

     NextBuffer[3]=1;
     NextBuffer[4]=1;
     NextBuffer[5]=2;
     NextBuffer[6]=2;
     NextBuffer[1]=0;
     NextBuffer[2]=0;

     // names of next buffers
     strcpy(Qname[3],"Q1");
     strcpy(Qname[4],"Q1");
     strcpy(Qname[5],"Q2");
     strcpy(Qname[6],"Q2");
     strcpy(Qname[1],"Q0");
     strcpy(Qname[2],"Q0");


     // states 1-fast (just internal)
     //         0-slow
     //               (all but internal)
     BufferState[0]=0;
     BufferState[1]=1;
     BufferState[2]=1;
     BufferState[3]=0;
     BufferState[4]=0;
     BufferState[5]=0;
     BufferState[6]=0;

     ServeState=2;
     Twist=FALSE;
     FastStep=faststep;

     pRandom = new sim_generator;


     if (SeedIndex!=0)
      {
         for (i=1;i<=SeedIndex;i++)
            {
             pRandom->set_next_seed();
            }
      }



}


buffer_entity::~buffer_entity() {}


void buffer_entity::execute()
{

     cell *pCell;
     double NextActivation;

     while (TRUE)
    {
```

```
    if (Queue[0]->length()==0)
                    EndCycle=TRUE;
    if ((Twist)&&(Workload>=
       HotSpotCapacity)) FirstHit=TRUE;
    TimeSlotsCount++;
    if ((FirstHit)||(EndCycle==
                TRUE)) sim_end_simulation();
  }

  switch(ServeState)
  {
  case 2: ServeState=1;
          NextActivation=FastStep;
          break;
  case 1: if (((ceil(sim_clock())-
              sim_clock())-FastStep)==0)
  {
          ServeState=2;
          NextActivation=FastStep;
  }
          if (((ceil(sim_clock())-
              sim_clock())-FastStep)>0)
  {
          ServeState=1;
          NextActivation=FastStep;
  }
          if (((ceil(sim_clock())
              -sim_clock())-FastStep)<0)
  {
          ServeState=0;
          NextActivation=ceil(sim_clock())
                                -sim_clock();
          NextStep=FastStep-(ceil(sim_clock())
                                -sim_clock());
  }
          break;
  case 0: ServeState=1;
          NextActivation=NextStep;
          break;
  }

  activate_in(NextActivation);

} // end while
```

```
for (POINTER=0;POINTER<6;POINTER++)
  {
  if ((ServeState==2)||
      (BufferState[POINTER]
            ==ServeState))
  {
    if (Queue[POINTER]->length()>0)
      {
        pCalls=(call *) retrieve(
          NO_WAIT,IGNORE_ACTIVATES,
          Queue[POINTER]);

        if (POINTER==0)
          { delete(pCall);
            Workload--;
            CarriedCalls++;
          }
        else {
          if (NextBuffer[POINTER]==0)
              {
              OfferedCalls++;
              if (Queue[0]->length()>=
                        HotSpotCapacity)
                  {
                  delete(pCall); //lost call
                  LostCalls++;
                  Workload--;
                  if (Twist) ExtraFactor++;
                  }
    }
      else {
            send_message(pCall,
                  Qname[POINTER]);
          }
    }
        else send_message(pCall,
                  Qname[POINTER]);

          }
        } //end else if POINTER

      } //end if ServeState
  } //end for

  if (!(ServeState==1))
    {
```

```
}

void buffer_entity::
        reinitialize_normal()
{
    ExtraFactor=0;
    Twist=FALSE;
    TimeSlotsCount=0;
    LostCalls=0;
    Twist=FALSE;
    FirstHit=FALSE;
    activate_at(sim_clock()+NextTime);
    EndCycle=FALSE;
    CarriedCalls=0;
    OfferedCalls=0;
}

void buffer_entity::initialize_normal()
{
    ExtraFactor=0;
    Workload=0;
    Twist=FALSE;
    TimeSlotsCount=0;
    LostCalls=0;
    CarriedCalls=0;
    OfferedCalls=0;
    Twist=FALSE;
    FirstHit=FALSE;
    activate_at(0.0);
}

void buffer_entity::initialize_twisted(
    sim_time time,int state,double next)
{
    ExtraFactor=0;
    OfferedCalls=0;
    CarriedCalls=0;
    NextStep=next;
    ServeState=state;
    TimeSlotsCount=0;
    Time=0.0;
    LostCalls=0;
    Twist=TRUE;
    FirstHit=FALSE;
    EndCycle=FALSE;
    activate_at(time);
}

void buffer_entity::suspend_twisted()
{ suspend(); }

void buffer_entity::suspend_normal(
                    sim_time time)
{
    NextTime=time;
    suspend();
}

char* buffer_entity::GetName()
{
    return(Name);
}

sim_bool buffer_entity::IsHotSpot()
{
    return(HotSpot);
}
```

The Source Header File

```
#ifndef SourceEntityH
#define SourceEntityH
#include <math.h>
#include <sims.h>
#include "buffer.h"
#include "cell.h"
#include "prog.h"

class source_entity: public sim_entity
{
public:
    source_entity(const char *name,
                  double SCR,
                  double PCR, double ABS,
                  Destination dtn);
    source_entity::source_entity(
          const char *name,
          double SCR, double PCR,
          double ABS,
          Destination dtn, double beta,
          const char *qName,
          buffer_entity *Bptr);
    ~source_entity();
    void execute();
    void normalCycle();
    void twistedCycle();
    void initialize_twisted
      (double time,int length);
    void initialize_normal();
    void suspend_normal
              (sim_time time);
    void suspend_twisted();
    void reinitialize_normal();
    void untwist();
    void twist();
    void MakeBufferPtr
          (buffer_entity *Buffer_ptr);
    sim_bool IsHotSpot();
    void SetStateT(int state,sim_time when);
    void SetState(int state, sim_time when);
    void untwist(sim_time when);
    int GetState();
    int initState();
    void SusPend();
    void Activate(sim_time when);
    int OnOrOff;
    double EigenVec[2];
    int IsOnOff();
    double Mu,Lambda;
    int CreatedCellsCount,BurstLength;
    int InitialOn,OverflowOn;
    char QueueName[8];

private:
    char Name[20];
    buffer_entity *pBuffer;
    double R,NextActivation,MuT,RT,LambdaT;
    sim_generator *pRandom;
    Destination dest;
    sim_bool HotSpot,NewBurst,NewBurstT;
    sim_time NextTime;
};

#endif
```

Source Module

```
#include "source.h"

source_entity::source_entity(const char *name,
           double SCR, double PCR, double ABS,
           Destination dtn):sim_entity(name)
{
     strcpy(Name,name);
     pRandom = new sim_generator();

     int i;
     if (SeedIndex!=0)
        {
           for (i=1;i<=SeedIndex;i++)
           {
               pRandom->set_next_seed();
           }
        }

     R=1.0;
     Mu = 1.0/(double) ABS;
     Lambda = Mu*(SCR/(PCR-SCR));
     HotSpot = FALSE;
     dest = dtn;
     RT=R;
     MuT=Mu;
     LambdaT=Lambda;

}

source_entity::source_entity(const char *name,
           double SCR, double PCR, double ABS,
           Destination dtn, double beta,
           const char *qName,
           buffer_entity *Bptr):sim_entity(name)
{
     strcpy(Name,name);
     pRandom = new sim_generator();

     int i;
     if (SeedIndex!=0)
       {
          for (i=1;i<=SeedIndex;i++)
             {
               pRandom->set_next_seed();
             }
       }

     R=1.0;
     Mu = 1.0/ABS;
     Lambda = Mu*(SCR/(PCR-SCR));
     HotSpot = TRUE;
     dest = dtn;

     CreatedCellsCount=0;

     strcpy(QueueName,qName);
     pBuffer=Bptr;


     RT=1.0;
     MuT=Mu/(Mu+(1-Mu)*(beta));
     LambdaT=(Lambda*beta)/(1-Lambda+Lambda*beta);

}

source_entity::~source_entity() {}

void source_entity::execute()
{
     while(TRUE)
     {

        if (pBuffer->Twist) twistedCycle();
                            else normalCycle();
     }
}


void source_entity::normalCycle()
{
     double rv_unfm;
     cell *pCell;

     if (NewBurst)
       {
          pRandom->uniform(rv_unfm,0.0,1.0);
          BurstLength= (int) ceil(log(1.0-rv_unfm)
                                        *-1.0*(1.0/Mu));
          NewBurst=FALSE;
       }
```

```
            // here we assume that prob to send = 1

            pCell = new cell(dest);
            pBuffer->send_message(pCell,QueueName);
            if (HotSpot)  { pBuffer->Workload++;
                            CreatedCellsCount++;
                          }
            BurstLength--;
            NextActivation = 1.0;
          }
        else {
            NewBurstT=TRUE;

            //calculate silent period
            pRandom->uniform(rv_unfm,0.0,1.0);
            NextActivation=cell(log((1.0-rv_unfm)
                                    *-1.0*(1.0/LambdaT));
          }

        activate_in(NextActivation);

      }

void source_entity::initialize_twisted(double time,
                                       int length)
    {

        OverflowOn=0;

        BurstLength=0;

        if ((length=0)||(time-sim_clock()>1.0))
            {
              InitialOn=0;
              NewBurstT=FALSE;
            }
        else {
              InitialOn=1;
              NewBurstT=TRUE;
            }
        suspend();
        activate_in(0.1);
      }

void source_entity::initialize_normal()
    {
```

```
    if (BurstLength>0)
        {

        // here we assume that prob to send = 1

        pCell = new cell(dest);
        pBuffer->send_message(pCell,QueueName);
        if (HotSpot)  { pBuffer->Workload++;
                        CreatedCellsCount++;
                      }
        BurstLength--;
        NextActivation = 1.0;
      }
    else {
        NewBurstT=TRUE;

        //calculate silent period
        pRandom->uniform(rv_unfm,0.0,1.0);
        NextActivation=cell(log(1.0-rv_unfm)
                            *-1.0*(1.0/LambdaT));
      }

    activate_in(NextActivation);

  }

void source_entity::twistedCycle()
  {

    double rv_unfm;
    cell *pCell;

    if (NewBurstT)
        {
          pRandom->uniform(rv_unfm,0.0,1.0);
          BurstLength= (int) cell(log(1.0-rv_unfm)
                                  *-1.0*(1.0/MuT));
          NewBurstT=FALSE;
        }

    if (BurstLength>0)
        {
```

```
    NewBurst=FALSE;
        BurstLength=0;
    }
    suspend();
    activate_in(0.1);
}

void source_entity::MakeBufferPtr
                (buffer_entity *Buffer_ptr)
{
    pBuffer=Buffer_ptr;
}

sim_bool source_entity::IsHotSpot()
{
    return (HotSpot);
}

int source_entity::IsOnOff()
{
    if ((((OnOrOff==1)&&(BurstLength>0))||
        ((OnOrOff==0)&&
            (next_time()-sim_clock()<1.0)))  return(1);
                                    else return(0);
}
```

```
double rv_unfm;
sim_bool rv_bern;

BurstLength=0;
NewBurst=TRUE;
InitialOn=0;
OverflowOn=0;
pRandom->draw(rv_bern,Lambda/(Lambda+Mu));
if (rv_bern)
    activate_at(0.1);
else {
    // calculate the silence period
    pRandom->uniform(rv_unfm,0.0,1.0);
    activate_at(0.1+ceil(log(1.0-rv_unfm)
                    *-1.0*(1.0/Lambda)));
    }
}

void source_entity::reinitialize_normal()
{
  activate_in(NextTime);
}

void source_entity::suspend_normal(sim_time time)
{
    NextTime=time;
    suspend();
}

void source_entity::suspend_twisted()
{ suspend(); }

void source_entity::untwist()
{

    if ((BurstLength>0)||((NewBurst)
                    &&(next_time()<1.0)))
        {
            OverflowOn=1;
            BurstLength=0;
            NewBurst=TRUE;
        }
    else {  OverflowOn=0;
```

The Main Header File

```
#ifndef PROGH
#define PROGH

extern int LengthOfBuffer;
extern int NumberOfDest;
extern int NumberOfBuffers;
extern int NumberOfSources;
extern int NumOfWarmup;
extern double NumOfCycles;
extern int SeedIndex;
extern unsigned int *SendTo,*ArriveAt;
extern double *Jitter;
extern double Gamma,Beta;

typedef int Destination;  // for readability

#endif
```

Main Module

```
#include <fstream.h>
#include <stdlib.h>    // for atof, atoi
#include <sims.h>
#include <stdio.h>     // for sprintf
#include <string.h>    // for strcmp
#include <new.h>
#include "prog.h"
#include "source.h"
#include "buffer.h"

int LengthOfBuffer,NumberOfDest,NumberOfBuffers,SeedIndex;
int NumOfWarmup,NumberOfSources;
unsigned int *SendTo,*ArriveAt;
double *Jitter,NumOfCycles;
double Gamma,Beta;

double buffer_entity::NumOfDCycles=0.0;
sim_bool buffer_entity::twist=FALSE;
int buffer_entity::LostCellsCount=0;
sim_bool buffer_entity::untwistFlag=FALSE;
sim_bool buffer_entity::cycleFlag=FALSE;
int buffer_entity::ExtraFactor=0;
int buffer_entity::CycleCount=0;
sim_bool buffer_entity::WarmUp=TRUE;
sim_time buffer_entity::Time=0.0;

void getParameters(char *fileName, double *get, char *Mode);
void getParameters(char *fileName, int (*NextBuffer)[2], char *Mode);
void getParameters(char *fileName, sim_bool  *get, char *Mode);
void getParameters(char *fileName, int *get, char *Mode);

void Multi_set_parameters()
{
     LengthOfBuffer=atoi(sim_config()->
             get_parameter("length_of_buffer"));
     NumberOfDest=atoi(sim_config()->
             get_parameter("number_of_dest"));
     NumberOfBuffers=atoi(sim_config()->
             get_parameter("number_of_buffers"));
     NumberOfSources=atoi(sim_config()->
             get_parameter("number_of_sources"));
     NumOfWarmup=atoi(sim_config()->
             get_parameter("number_of_warmup"));
     NumOfCycles=atof(sim_config()->
             get_parameter("number_of_cycles"));
     Gamma=atof(sim_config()->get_parameter("gamma_"));
     Beta=atof(sim_config()->get_parameter("beta_"));
     SeedIndex=atoi(sim_config()->
             get_parameter("seed_index"));
}

class Multi_config_obj:public sim_config_obj
{
public:
     Multi_config_obj()
     {
       add_parameter("length_of_buffer","0");
       add_parameter("number_of_dest","0");
       add_parameter("number_of_buffers","0");
       add_parameter("number_of_sources","0");
       add_parameter("number_of_warmup","0");
       add_parameter("number_of_cycles","0");
       add_parameter("gamma_","0.0");
       add_parameter("beta_","0.0");
       add_parameter("seed_index","0");
     }
};

int main(int argc, char *argv[])
{
```

```
char ConfigFileName[80],InputFileName[80];

if (argc<2)
   { cout<<endl<<"the configuration file name :";
     cin>>ConfigFileName;
   }
else strcpy(ConfigFileName,argv[1]);

if (argc<3)
   { cout<<endl<<"the input file name :";
     cin>>InputFileName;
   }
else strcpy(InputFileName,argv[2]);

sim_set_config(new Multi_config_obj());
sim_config()->read_file(ConfigFileName);
Multi_set_parameters();
buffer_entity::NumOfDCycles=NumOfCycles;

//read in parameters as PCR, etc.....
double *transrate = new double[NumberOfBuffers];
double *scr = new double[NumberOfSources];
double *pcr = new double[NumberOfSources];
double *abs = new double[NumberOfSources];
sim_bool *Source_Hot = new, sim_bool[NumberOfSources];
int *Source_Dest = new int[NumberOfSources];
int *Qlength = new int[NumberOfBuffers];

int i;
// initialize the source hotspot
for (i=0;i<NumberOfSources-1;i++)
   {
     Source_Hot[i]=FALSE;
   }

getParameters(InputFileName,scr,"SCR");
getParameters(InputFileName,pcr,"PCR");
getParameters(InputFileName,abs,"ABS");
getParameters(InputFileName,transrate,"TR");
for (i=0;i<NumberOfBuffers;i++) transrate[i]=transrate[i]/pcr[0];
getParameters(InputFileName,Source_Hot,"S_HOT");
getParameters(InputFileName,Source_Dest,"S_DEST");
getParameters(InputFileName,Qlength,"QLENGTH");

source_entity *source_ptr[NumberOfSources];
source_entity *Tsource_ptr[NumberOfSources];
buffer_entity *buffer_ptr;
buffer_entity *Tbuffer_ptr;
char entityname1[15],entityname[15];

// construct the switching fabric
buffer_ptr = new buffer_entity("fabric",LengthOfBuffer,.2115017964);
Tbuffer_ptr = new buffer_entity("Tfabric",LengthOfBuffer,.2115017964);

char queuename[8];

//construct the source entities
for (i=0;i<NumberOfSources-1;i++)
   {
     sprintf(entityname,"%d","Source",i);
     sprintf(entityname1,"%d","TSource",i);
     sprintf(queuename,"%d","Q",i+3);

     if (Source_Hot[i])
        {
          source_ptr[i] = new source_entity(entityname,
                                            scr[i],pcr[i],abs[i],
                                            Source_Dest[i],
                                            Beta,queuename,buffer_ptr);
          Tsource_ptr[i] = new source_entity(entityname1,
                                             scr[i],pcr[i],abs[i],
                                             Source_Dest[i],
                                             Beta,queuename,Tbuffer_ptr);
        }
     else {
          source_ptr[i] = new source_entity(entityname,
                                            scr[i],pcr[i],abs[i],
                                            Source_Dest[i]);
          Tsource_ptr[i] = new source_entity(entityname1,
                                             scr[i],pcr[i],abs[i],
                                             Source_Dest[i]);
          }
   }

// remove parameter arrays from the free store
delete [] scr;
delete [] pcr;
delete [] abs;
delete [] transrate;
delete [] Source_Hot;
delete [] Source_Dest;
```

```
delete [] Qlength;


//warmup
cout<<"Start of WarmUp"<<endl;

// initialize the Buffers and the sources
buffer_ptr->initialize_normal();

for (i=0;i<NumberOfSources;i++)
    {
      source_ptr[i]->initialize_normal();
    }

for (i=1;i<=NumOfWarmup;i++)
    {
      sim_run_simulation();
      buffer_ptr->EndCycle=FALSE;
    }

cout<<"End of WarmUp"<<endl;

// reinitialize the number of cells offered to hotspot
for (i=0;i<NumberOfSources;i++)
   {
     source_ptr[i]->CreatedCellsCount=0;
     Tsource_ptr[i]->CreatedCellsCount=0;
   }

int Length,k,j;
cell *pCell1,*pCell2;
double factor;
int initHotNum;
double TotalT=0.0,TotalTime=0.0,TotalTimeT=0.0;
int Total=0;
double Ber_diff;
sim_bool hit=FALSE;
int hotNumWhenHit=0;
int TotalCarried=0,TotalOffered=0;

cout<<"Start of Simulation"<<endl;
for (i=1;i<=(int) NumOfCycles;i++)
   {

   for (j=0;j<=6;j++)
     {
```

```
   // make copy of the contents of the buffers to the twisted
   // buffers
   Tbuffer_ptr->Queue[j]->clear();
   Length=buffer_ptr->Queue[j]->length();
   if (Length>=1)
     {
      for (k=1;k<=Length;k++)
         {
           sprintf(queuename,"%s%d","Q",j);
           pCell1=(cell*) buffer_ptr->retrieve(NO_WAIT,
                IGNORE_ACTIVATES,buffer_ptr->Queue[j]);
           pCell2= new cell(pCell1->get_dest());

           Tbuffer_ptr->send_message(pCell1,
                                        queuename);
           buffer_ptr->send_message(pCell2,
                                        queuename);
         }
       }
   }

Tbuffer_ptr->initialize_twisted(buffer_ptr->next_time(),
              buffer_ptr->ServeState,buffer_ptr->NextStep);
buffer_ptr->suspend_normal(buffer_ptr->next_time()-sim_clock());


for (j=0;j<NumberOfSources;j++)
    {
     Tsource_ptr[j]->initialize_twisted(source_ptr[j]->
          next_time(),source_ptr[j]->BurstLength);
     source_ptr[j]->suspend_normal(source_ptr[j]->
          next_time()-sim_clock());
     }
// initHotNum is the workload at the
// beginning of the twisted cycle
initHotNum=buffer_ptr->Workload;
Tbuffer_ptr->Workload=buffer_ptr->Workload;

sim_run_simulation();

if (Tbuffer_ptr->FirstHit)
   {
    for (j=0;j<NumberOfSources;j++)
        {
          Tsource_ptr[j]->untwist();
        }
Tbuffer_ptr->FirstHit=FALSE;
```

```
  Tbuffer_ptr->Twist=FALSE;
  hit=TRUE;
  hotNumWhenHit=Tbuffer_ptr->Workload;
  sim_run_simulation();
      } else { hit=FALSE; }

  // after a cycle, the parameters are
  // reset to their original values
  // and the factor (i.e. likelihood ratio)
  // is calculated
  Ber_diff=0.0;
  factor=1.0;
  for (j=0;j<NumberOfSources;j++)
     {
      Ber_diff = (double)
            (Tsource_ptr[j]->InitialOn-
             Tsource_ptr[j]->OverflowOn);
      factor *= pow(3.9777677845,Ber_diff);
      Tsource_ptr[j]->suspend_twisted();
      source_ptr[j]->reinitialize_normal();
     }
   if (!hit) hotNumWhenHit=Tbuffer_ptr->Workload;
   factor *= exp(Gamma
        *(initHotNum-hotNumWhenHit-
         Tbuffer_ptr->ExtraFactor));


   TotalT+=factor*(double)Tbuffer_ptr->LostCells;
   TotalTimeT+=Tbuffer_ptr->TimeSlotsCount;
   Tbuffer_ptr->suspend_twisted();
   buffer_ptr->reinitialize_normal();

   sim_run_simulation();

   TotalCarried+=buffer_ptr->CarriedCells;
   TotalOffered+=buffer_ptr->OfferedCells;
   Total+=buffer_ptr->LostCells;
   TotalTime+=buffer_ptr->TimeSlotsCount;
  }


cout<<"End of Simulation"<<endl<<endl;

cout<<"carried cells to the hot spot (Original): "
     <<TotalCarried<<endl;
cout<<"offered cells to the hot spot (Original): "
     <<TotalOffered<<endl;
cout<<"lost cells (Original): "<<Total<<endl;
cout<<"total simulation time (Original): "
     <<TotalTime<<endl;
cout<<"avg. length of a cycle (Original): ";
cout<<(double) TotalTime/ (double) NumOfCycles
     <<endl;
cout<<"avg. cell loss per cycle: "
     <<(double) Total/ (double) NumOfCycles<<endl;
cout<<"avg. cell loss per time slot: "
     <<(double) Total/ (double) TotalTime<<endl;
cout<<endl;
cout<<"lost cells (Twisted): "<<TotalT<<endl;
cout<<"avg. length of a cycle (Twisted): "
     <<TotalTimeT/(double)NumOfCycles<<endl;
cout<<"avg. cell loss per cycle: "
     <<(double) TotalT/ (double) NumOfCycles
     <<endl;
cout<<"avg. cell loss per time slot: "
     <<(double) TotalT/ TotalTime<<endl;
cout<<endl<<endl;

double expectedOffered=0;

// long run avg. offered number
// of cells per time slot is

for (j=0;j<NumberOfSources;j++)
    {
    expectedOffered+=source_ptr[j]->Lambda/
         (source_ptr[j]->Lambda+source_ptr[j]->Mu);
    }
cout<<"long-run average number of cells offered per time slot: ";
cout<<expectedOffered<<endl;

cout<<endl;
cout<<"CLR (Original): "<<((double) Total/TotalTime)/
          expectedOffered<<endl;
cout<<"CLR (Twisted): "<<((double) TotalT/TotalTime)/
          expectedOffered<<endl;
cout<<endl<<endl;

int sent1=0,sent2=0;

for (i=0;i<NumberOfSources;i++)
    {
     sent1+=source_ptr[i]->CreatedCellsCount;
     sent2+=Tsource_ptr[i]->CreatedCellsCount;
```

```
}

    cout<<"Total Load (Original): "<<sent1<<endl;
    cout<<"Total Load (Twisted): "<<sent2<<endl;


    system("ps -e |grep prog");

} //end main

void getParameters(char *fileName, double *get, char *Mode)
{
enum MODE{Start,CheckMode,Get};
enum STATE{ON,BETWEEN};

ifstream fin(fileName);  //open for reading

int i(0),Count(0);
char ch,buffer[80];
MODE mode(Start);
STATE state(BETWEEN);

while (fin.get(ch))
   {
   switch(mode)
     {
     case Start: if (ch==':') mode=CheckMode;
                 break;

     case CheckMode: if (ch==':')
                       { buffer[i]='\0';
                         i=0;
                         if (!strcmp(buffer,Mode))
                           { mode=Get;
                             state=BETWEEN; }
                         else mode=Start;
                       }
                     else buffer[i++]=ch;
                     break;

     case Get:  if (state==ON)
                  { if (ch!='\'')  { buffer[i++]=ch; }
                    else { buffer[i]='\0';
                           i=0;
                           get[Count++]=
                                atof(buffer);
                           state=BETWEEN;
                         }
                  }
                else { if (ch=='\'')  { state=ON; }
                       else { if (ch==':')
                                mode=
                                  CheckMode;
                            }
                     break;
                }
            }
          }
        fin.close();
        }

void getParameters(char *fileName,  int (*NextBuffer)[2], char *Mode)
{
   enum MODE{Start,CheckMode,NEXTBUFFER};
   enum STATE{ON,BETWEEN,NEXT};

   ifstream fin(fileName);  //open for reading

   int from(0),to(0),i(0);
   char ch,buffer[80];
   MODE mode(Start);
   STATE state(BETWEEN);

   while (fin.get(ch))
   {
      switch(mode)
      {
      case Start: if (ch==':') mode=CheckMode;
                  break;
      case CheckMode: if (ch==':')
                        { buffer[i]='\0';
                          i=0;
                          if (!strcmp(buffer,Mode))
                            { mode=NEXTBUFFER;
                              state=BETWEEN; }
                          else mode=Start;
                        }
                      else buffer[i++]=ch;
                      break;
      case NEXTBUFFER: switch(state)
                       {
                       case ON: switch(ch)
```

```
    {
      case '>': state=NEXT;
                buffer[i]='\0';
                i=0;
                from=atoi(buffer);
                break;
      case '\'': state=BETWEEN;
                buffer[i]='\0';
                i=0;
                NextBuffer[from][to]=
                    atoi(buffer);
                break;
      default: buffer[i++]=ch;
               break;
      }
      break;
    case NEXT: switch(ch)
               {
                 case '>': state=ON;
                           buffer[i]='\0';
                           i=0;
                           to=atoi(buffer);
                           break;
                 default: buffer[i++]=ch;
                          break;
               }
               break;
    case BETWEEN: switch(ch)
                  {
                    case '\'': state=ON;
                               break;
                    case ':': mode=CheckMode;
                              break;
                  }
                  break;
  }
  fin.close();
}

void getParameters(char *fileName, sim_bool  *get, char *Mode)
{
  enum MODE{Start,CheckMode,Get};
  enum STATE{ON,BETWEEN};
  ifstream fin(fileName);  //open for reading

  int i(0);
  char ch,buffer[80];
  MODE mode(Start);
  STATE state(BETWEEN);

  while (fin.get(ch))
  {
    switch(mode)
    {
    case Start: if (ch==':') mode=CheckMode;
                break;

    case CheckMode: if (ch=='.')
                    { buffer[i]='\0';
                      i=0;
                      if (!strcmp(buffer,Mode))
                          { mode=Get;
                            state=BETWEEN; }
                      else mode=Start;
                    }
                    else buffer[i++]=ch;
                    break;
    case Get:  if (state==ON)
                 { if (ch!='\'')
                     { buffer[i++]=ch; }
                   else { buffer[i]='\0';
                          i=0;
                          get[atoi(buffer)]=TRUE;
                          state=BETWEEN;
                        }
                 }
               else { if (ch=='\'') state=ON;
                      else if (ch==':') { mode=CheckMode; }
                    }
               break;
  }
  }
  fin.close();
}

void getParameters(char *fileName, int *get, char *Mode)
{
  enum MODE{Start,CheckMode,Get};
  enum STATE{ON,BETWEEN};
```

```
ifstream fin(fileName);  //open for reading

int i(0),Count(0);
char ch,buffer[80];
MODE mode(Start);
STATE state(BETWEEN);

while (fin.get(ch))
{
    switch(mode)
    {
    case Start: if (ch==':') mode=CheckMode;
                break;

    case CheckMode: if (ch==':')
                    { buffer[i]='\0';
                      i=0;
                      if (!strcmp(buffer,Mode))
                        { mode=Get;
                          state=BETWEEN; }
                      else mode=Start;
                    }
                    else buffer[i++]=ch;
                    break;

    case Get:  if (state==ON)
                 { if (ch!='\'')  { buffer[i++]=ch; }
                   else { buffer[i]='\0';
                          i=0;
                          get[Count++]=atoi(buffer);
                          state=BETWEEN;
                        }
                 }
               else { if (ch=='\'')  { state=ON; }
                      else { if (ch==':')  mode=CheckMode; }
                    }
               break;
    }
}
fin.close();
}
```

C.2 Source Code for Model II

The Cell Header File

```
#ifndef CellH
#define CellH

#include <sims.h>
#include "prog.h"

class cell: public sim_message
{
public:
    cell(Destination dest, int priority);
    ~cell();
    const char *type() const;
    Destination get_dest();
    int Priority;
private:
    Destination WhereTo;
};

#endif
```

Cell Module

```
#include "cell.h"

cell::cell(Destination dest,
      int priority)
{    WhereTo=dest;
     Priority=priority; }

cell::~cell() {}

const char *cell::type() const
{ return ("cell"); }

Destination cell::get_dest()
{ return (WhereTo); }
```

The Buffer Header File

```
#ifndef Buffer_EntityH
#define Buffer_EntityH

#include <stdlib.h>
#include <sims.h>
#include <math.h>
#include <string.h>  //for strcmp
#include "cell.h"
#include "prog.h"

class buffer_entity: public sim_entity
{
public:
     sim_message_queue *Queue[2];
     buffer_entity(const char *name,
        sim_bool hSp, sim_bool InLet,
        int qlength0, int qlength1,
        sim_bool tbuffer,
        sim_bool Intermediate,
        double TrRate);
     ~buffer_entity();
     void suspend_normal
           (sim_time time);
     void initialize_twisted
           (sim_time time);
     void reinitialize_normal();
     void initialize_normal();
     void suspend_twisted();
     void execute();
     char* GetName();
     buffer_entity *B1, *B2,
               *NextBuffer[4];
     char NextQueue[8][20];
     static sim_bool twist;
     sim_bool IsHotSpot();
     sim_bool wait,intermediate;
     static void Clear();
     int QLength[2],NumberOfTimeSlots,
                        CellsToHotspot;
     sim_time steps;
     sim_bool Empty[2];
     static sim_bool WarmUp;
     static double NumOfDCycles;
     static sim_bool Continue;
     static int hotNumber;
     static int ExtraFactor;
     static int LostCellsCount;
     static sim_bool untwistFlag;
     static sim_bool cycleFlag;
     static int TimeSlotsCount;
     static int CycleCount;
     static sim_bool lastCycle;
     static sim_bool HalfTime;
     static int ServeBin;
     static sim_time Time;
private:
     char Name[40];
     sim_generator *pRandom;
     sim_bool TBuffer,Inlet;
     sim_bool EmptyBuffer,
              HotSpot,FirstHit;
     double transmissionRate;
     double floatCells;
     int Order;
     sim_time NextTime;
};
#endif
```

Buffer Module

```
#include "buffer.h"

buffer_entity::buffer_entity(const char *name,
            sim_bool hSp,
            sim_bool InLet,int qlength0,
            int qlength1, sim_bool tbuffer,
            sim_bool Intermediate,
            double TrRate):sim_entity(name)
{
     strcpy(Name,name);
     HotSpot=hSp;
     Inlet=InLet;
     QLength[0]=qlength0;
     QLength[1]=qlength1;
     TBuffer=tbuffer;
     intermediate=Intermediate;
     steps=1.0/TrRate;

     EmptyBuffer=TRUE;

     transmissionRate=TrRate;

     pRandom = new sim_generator;

     int i;
     if (SeedIndex!=0)
     {
        for (i=1;i<=SeedIndex;i++)
           {
            pRandom->set_next_seed();
           }
     }

     //create high priority queue
     char Qname[20];
     sprintf(Qname,"%s%d",Name,0);
     create_message_queue(Qname);
     Queue[0]=find_queue(Qname);
     Queue[0]->clear();
     Empty[0]=TRUE;

     //create low priority queue
     sprintf(Qname,"%s%d",Name,1);
     create_message_queue(Qname);
     Queue[1]=find_queue(Qname);
     Queue[1]->clear();
     Empty[1]=TRUE;
}

buffer_entity::~buffer_entity() {}

void buffer_entity::execute()
{
     int pointer;
     cell *pCell;
     buffer_entity *pBufferTo;

     while (TRUE)
    {

    // algorithm to keep proper ordering
    if (Order==0)
      {   if (B1!=NULL) B1->wait=TRUE;
                  if (B2!=NULL) B2->wait=TRUE;
                  Order=1;
                  activate_in(0.0);
      }
    if (Order==1)
      {
          Order=2;
          if (wait) { activate_in(0.0); }
      }
    if (Order==2)
      { Order=0;
        wait=FALSE;
        // now execute entities are in proper order

       //point to high priority (0) if not empty
            if (!Empty[0]) pointer=0;
                     else pointer=1;
            if (Inlet) pointer=0;

      if (HotSpot)
         {
          NumberOfTimeSlots++;

          if (!(Empty[0]&&Empty[1]))
 {
             pCell=(cell *) retrieve(
                      NO_WAIT,
                      IGNORE_ACTIVATES,
                      Queue[pointer]);
```

```
    delete(pCell);
    if (HotSpot) hotNumber--;
    }
    // check to see if the queue is empty
  Empty[0]=(Queue[0]->length()==0);
  Empty[i]=(Queue[i]->length()==0);

    // check for end of cycle
    if (Queue[1]->length()<=1)
      {
        sim_end_simulation();
        activate_in(1.0);
      }
    else {
          if ((hotNumber>Qlength[i])
                        &&(FirstHit))
            {
              twist=FALSE;
              Time=sim_clock()+1.0;
              FirstHit=FALSE;
              untwistFlag=TRUE;
              sim_end_simulation();
    }
            activate_in(1.0);
          }
      }
    else {
        if (!(Empty[0]&&Empty[i]))
          {
            pCell=(cell *)
              retrieve(NO_WAIT,
              IGNORE_ACTIVATES,
              Queue[pointer]);

    pBufferTo=NextBuffer[pCell->get_dest()];

    if ((pBufferTo->IsHotSpot())
          &&(pointer==1)) CellsToHotspot++;
    if ((pBufferTo->Queue[pCell->Priority]->length()
          >=pBufferTo->Qlength[pCell->Priority])&&
          (pBufferTo->Qlength[pCell->Priority]!=0))
  { if (pCell->get_dest()==0)
              {
                hotNumber--;
                if (twist) ExtraFactor++;
                if ((pBufferTo->IsHotSpot())
                        &&(pointer==1))
                    LostCellsCount++;
                }
                delete(pCell);
            }
            else {
                pBufferTo->send_message(pCell,
                NextQueue[pCell->get_dest()+4+pCell->Priority]);
            }
            // check to see if the queue is empty
            Empty[0]=(Queue[0]->length()==0);
            Empty[i]=(Queue[i]->length()==0);

            activate_in(steps);
          }
        }
      }
    }

  void buffer_entity::reinitialize_normal()
  {
    untwistFlag=FALSE;
    twist=FALSE;
    ExtraFactor=0;
    NumberOfTimeSlots=0;
    Time=0.0;
    LostCellsCount=0;
    twist=FALSE;
    FirstHit=FALSE;
    activate_at(sim_clock()+NextTime);
  }

  void buffer_entity::initialize_normal()
  {
    untwistFlag=FALSE;
    twist=FALSE;
    ExtraFactor=0;
    NumberOfTimeSlots=0;
    LostCellsCount=0;
    twist=FALSE;
    FirstHit=FALSE;
    activate_at(0.0);
  }

  void buffer_entity::initialize_twisted
                          (sim_time time)
  {
```

```
    untwistFlag=FALSE;
    ExtraFactor=0;
    NumberOfTimeSlots=0;
    Time=0.0;
    LostCallsCount=0;
    twist=TRUE;
    FirstHit=TRUE;
    if ((HotSpot)&&(sim_clock()==0.0))
                        activate_at(time+1.0);
      else activate_at(time);
    }

    void buffer_entity::suspend_twisted()
    { suspend(); }

    void buffer_entity::suspend_normal
                        (sim_time time)
    {
        NextTime=time;
        suspend();
    }

    void buffer_entity::Clear()
    {
        ExtraFactor=0;
        TimeSlotsCount=0;
        LostCallsCount=0;
    }

    char* buffer_entity::GetName()
    {
        return(Name);
    }

    sim_bool buffer_entity::IsHotSpot()
    {
        return(HotSpot);
    }
```

The Source Header File

```
#ifndef SourceEntityH
#define SourceEntityH

#include <math.h>
#include <sims.h>
#include "buffer.h"
#include "cell.h"
#include "prog.h"

class source_entity: public sim_entity
{
public:
     source_entity(const char *name,
          double SCR, double PCR,
          double ABS, Destination dtn,
          int priority);
     source_entity(const char *name,
          double SCR, double PCR,
          double ABS, Destination dtn,
          int priority,double Alpha,
          double Gamma);
     ~source_entity();
     void execute();
     void normalCycle();
     void twistedCycle();
     void GenCellHigh(int priority,
               sim_bool twisted);
     void initialize_twisted
            (double time,
              int length);
     void initialize_normal();
     void suspend_normal(
               sim_time time);
     void suspend_twisted();
     void reinitialize_normal();
     void untwist();
     void twist();
     void MakeBufferPtr(buffer_entity
          *Buffer_ptr);
     sim_bool IsHotSpot();
     void SetStateT(int state,
               sim_time when);
     void SetState(int state,
               sim_time when);
     void untwist(sim_time when);
     int OnOrOff;
     double EigenVec[2];
     int IsOnOff();
     double Mu,Lambda;
     int CreatedCellsCount;
     int BurstLength,Priority;
     int InitialOn,OverflowOn;
private:
     char Name[20];
     buffer_entity *pBuffer;
     double R,NextActivation,
               MuT,RT,LambdaT;
     sim_generator *pRandom;
     Destination dest;
     sim_bool HotSpot,
               NewBurst,NewBurstT;
     sim_time NextTime;
};
#endif
```

Source Module

```
#include "source.h"

source_entity::source_entity(const char *name,
           double SCR, double PCR, double ABS,
           Destination dtn,
           int priority):sim_entity(name)
{
     strcpy(Name,name);
     pRandom = new sim_generator();

     int i;
     if (SeedIndex!=0)
        {
          for (i=1;i<=SeedIndex;i++)
          {
              pRandom->set_next_seed();
          }
        }

     if (priority==1)
      {
        R=1.0;
        Mu = 1.0/(double) ABS;
        Lambda = Mu*(SCR/(PCR-SCR));
       }
 else {
        R=PCR/(double)353208.0;
        Mu=0.0;
        Lambda =1.0;
       }
     HotSpot = FALSE;
     dest = dtn;
     RT=R;
     MuT=Mu;
     LambdaT=Lambda;
     Priority=priority;

}

source_entity::source_entity(
         const char *name,
         double SCR, double PCR,
         double ABS, Destination dtn,
         int priority, double Alpha,
         double Gamma):sim_entity(name)
{
     strcpy(Name,name);
     pRandom = new sim_generator();

     int i;
     if (SeedIndex!=0)
       {
        for (i=1;i<=SeedIndex;i++)
          {
          pRandom->set_next_seed();
          }
       }

     if (priority==1)
       {
         R=1.0;
         Mu = 1.0/ABS;
         Lambda = Mu*(SCR/(PCR-SCR));
         RT=1.0;
         MuT=Mu/(Mu+(1-Mu)*(9.895438431));
         LambdaT=(Lambda*9.85438431)/
               (1-Lambda+Lambda*9.85438431);
       }
 else {
          R=PCR/(double)353208;
          Mu=0.0;
          Lambda =1.0;
          MuT=0.0;
          LambdaT=1.0;
          RT=(1.000369099+R-1)/1.000369099;
        }
      HotSpot = TRUE;
      dest = dtn;

    CreatedCellsCount=0;

    Priority=priority;

cout<<Name<<"  "<<Priority<<endl;

}

source_entity::~source_entity() {}

void source_entity::execute()
{
     while(TRUE)
```

```
{
    if (buffer_entity::twist)
        { if (Priority==1) twistedCycle();
          else GenCallHigh(Priority,TRUE);
        }
    else { if (Priority==1) normalCycle();
           else GenCallHigh(Priority,FALSE);
         }
    }
}

void source_entity::normalCycle()
{
    double rv_unfm;
    call *pCall;
    char WhereTo[20];

    if (NewBurst)
        {
        pRandom->uniform(rv_unfm,0.0,1.0);
        BurstLength= (int)
            ceil(log(1.0-rv_unfm)
              *-1.0*(1.0/Mu));
        NewBurst=FALSE;
        }

    if (BurstLength>0)
        {
        // here we assume that prob to send = 1
        pCall = new call(dest,Priority);
        sprintf(WhereTo,"%s%d",
                pBuffer->GetName(),0);
        pBuffer->send_message(pCall,WhereTo);
        if (HotSpot)
            { buffer_entity::hotNumber++;
              CreatedCallsCount++;
            }
        BurstLength--;
        NextActivation = 1.0;
        }
    else {
        NewBurst=TRUE;

        //calculate silent period
        pRandom->uniform(rv_unfm,0.0,1.0);
        NextActivation=ceil(log(1.0-rv_unfm)
              *-1.0*(1.0/Lambda));
        }

    activate_in(NextActivation);
}

void source_entity::GenCallHigh
    (int priority,sim_bool twisted)
{
    sim_bool rv_born;
    call *pCall;
    char WhereTo[20];

    if (twisted) pRandom->draw(rv_born,RT);
                 else  pRandom->draw(rv_born,R);

    if (rv_born)
        {
        pCall = new call(dest,Priority);
        sprintf(WhereTo,"%s%d",
                pBuffer->GetName(),0);
        pBuffer->send_message(pCall,WhereTo);
        if (HotSpot)
            { buffer_entity::hotNumber++;
              CreatedCallsCount++;
            }
        }
    activate_in(1.0);
}

void source_entity::twistedCycle()
{
    double rv_unfm;
    call *pCall;
    char WhereTo[20];

    if (NewBurstT)
        {
        pRandom->uniform(rv_unfm,0.0,1.0);
        BurstLength= (int) ceil(log(1.0-rv_unfm)
              *-1.0*(1.0/MuT));
        NewBurstT=FALSE;
        }
```

```
    }

    suspend();
    activate_in(0.1);
}


void source_entity::initialize_normal()
{
    double rv_unfm;
    sim_bool rv_bern;

    if (Priority==0) activate_at(0.1);
    else
      {

    BurstLength=0;
    NewBurstT=TRUE;
    InitialOn=0;
    OverflowOn=0;
    pRandom->draw(rv_bern,
        Lambda/(Lambda+Mu));
    if (rv_bern)
        activate_at(0.1);
    else {
        // calculate the silence period
        pRandom->uniform(rv_unfm,0.0,1.0);
        activate_at(0.1+call(log(1.0-rv_unfm)
            *-1.0*(1.0/Lambda)));
        }
    }
}


void source_entity::reinitialize_normal()
{
    activate_in(NextTime);
}


void source_entity::suspend_normal
                        (sim_time time)
{
    NextTime=time;
    suspend();
}
```

```
    if (BurstLength>0)
      {
      // here we assume that prob to send = 1

      pCell = new cell(dest,Priority);

      sprintf(WhereTo,"%s%d",pBuffer->GetName(),0);

      pBuffer->send_message(pCell,WhereTo);
      if (HotSpot)  { buffer_entity::hotNumber++;
                      CreatedCellsCount++;
                    }
          BurstLength--;
          NextActivation = 1.0;
          }
        else {
            NewBurstT=TRUE;

            //calculate silent period
          pRandom->uniform(rv_unfm,0.0,1.0);
          NextActivation=call(log(1.0-rv_unfm)
                *-1.0*(1.0/LambdaT));
          }

      activate_in(NextActivation);
    }


void source_entity::initialize_twisted
            (double time,int length)
{
    if (Priority=1)
      {
    OverflowOn=0;

    BurstLength=0;
    if ((length=0)&&(time=sim_clock()>1.0))
      {
          InitialOn=0;
          NewBurstT=FALSE;
      }
    else {
          InitialOn=1;
          NewBurstT=TRUE;
        }
```

```
void source_entity::suspend_twisted()
{ suspend(); }

void source_entity::untwist()
{
    if (Priority==1)
        {
    if ((BurstLength>0)||((NewBurstT)
                && (next_time()<1.0)))
        {
            OverflowOn=1;
            BurstLength=0;
            NewBurst=TRUE;
        }
    else {  NewBurst=FALSE;
            BurstLength=0;
        }
        }
    suspend();
    activate_in(0.1);
}

void source_entity::untwist(sim_time when)
{
    SetState(IsOnOff(),when);
}

void source_entity::SetState(int state,
                             sim_time when)
{   double rv_unfm,silence;
    OnOrOff=0;;
    pRandom->uniform(rv_unfm,0.0,1.0);
    BurstLength = (int) ceil(log(1.0-rv_unfm)
        *-1.0*(1.0/Mu));
    if (state==1)
        {
        if (sim_clock()==0.0)
            activate_at(1.0-when);
        else activate_at(ceil(sim_clock())-when);
        }
    else { pRandom->uniform(rv_unfm,0.0,1.0);
           silence=ceil(log(1.0-rv_unfm)
                *-1.0*(1.0/Lambda));

           if ((sim_clock()==0.0)&&(silence==0.0))
                activate_at(1.0-when);
           else activate_at(ceil(sim_clock())-
                            when+silence);
        }
}

void source_entity::SetStateT(int state,
                              sim_time when)
{   double rv_unfm;
    OnOrOff=0;
    pRandom->uniform(rv_unfm,0.0,1.0);
    BurstLength = (int) ceil(log(1.0-rv_unfm)
                        *-1.0*(1.0/MuT));
    if (state==1)
        {
        activate_at(ceil(sim_clock())-when);
        }
    else { pRandom->uniform(rv_unfm,0.0,1.0);
           activate_at(ceil(sim_clock())-when+
               ceil(log(1.0-rv_unfm)*-1.0*(1.0/LambdaT)));
        }
}

void source_entity::MakeBufferPtr(
                buffer_entity *Buffer_ptr)
{
    pBuffer=Buffer_ptr;
}

sim_bool source_entity::IsHotSpot()
{
    return (HotSpot);
}

int source_entity::IsOnOff()
{
    if ((((OnOrOff==1)&&(BurstLength>0))||
          ((OnOrOff==0)&&
          (next_time()-sim_clock()<1.0)))  return(1);
                                    else return(0);
}
```

The Main Header File

```
#ifndef PROGH
#define PROGH

extern int LengthOfBuffer;
extern int NumberOfDest;
extern int NumberOfBuffers;
extern int NumberOfSources;
extern int NumOfWarmup;
extern double NumOfCycles;
extern int SeedIndex;
extern double Gamma;

typedef int Destination;  // for readability

#endif
```

Main Module

```
#include <fstream.h>
#include <stdlib.h>    // for atof, atoi
#include <sims.h>
#include <stdio.h>    // for sprintf
#include <string.h>   // for strcmp
#include "prog.h"
#include "source.h"
#include "buffer.h"


int LengthOfBuffer,NumberOfDest,
NumberOfBuffers,SeedIndex;
int NumOfWarmup,NumberOfSources;
double NumOfCycles;
double Gamma;

double buffer_entity::NumOfDCycles=0.0;
int buffer_entity::hotNumber=0;
sim_bool buffer_entity::twist=FALSE;
int buffer_entity::LostCellsCount=0;
sim_bool buffer_entity::untwistFlag=FALSE;
int buffer_entity::TimeSlotsCount=0;
sim_bool buffer_entity::cycleFlag=FALSE;
int buffer_entity::ExtraFactor=0;
int buffer_entity::CycleCount=0;
sim_bool buffer_entity::lastCycle=FALSE;
sim_bool buffer_entity::HalfTime=FALSE;
int buffer_entity::ServeBin=0;
sim_bool buffer_entity::Continue=TRUE;
sim_bool buffer_entity::WarmUp=TRUE;
sim_time buffer_entity::Time=0.0;

void getParameters(char *fileName,
       double *get, char *Mode);
void getParameters(char *fileName,
       int (*NextBuffer)[2], char *Mode);
void getParameters(char *fileName,
      sim_bool  *get, char *Mode);
void getParameters(char *fileName,
      int *get, char *Mode);

void Multi_set_parameters()
{
     LengthOfBuffer=atoi(sim_config()->
           get_parameter("length_of_buffer"));
     NumberOfDest=atoi(sim_config()->
           get_parameter("number_of_dest"));
     NumberOfBuffers=atoi(sim_config()->
           get_parameter("number_of_buffers"));
     NumberOfSources=atoi(sim_config()->
           get_parameter("number_of_sources"));
     NumOfWarmup=atoi(sim_config()->
           get_parameter("number_of_warmup"));
     NumOfCycles=atof(sim_config()->
           get_parameter("number_of_cycles"));
     Gamma=atof(sim_config()->
           get_parameter("gamma_"));
     SeedIndex=atoi(sim_config()->
           get_parameter("seed_index"));
}

class Multi_config_obj:public sim_config_obj
{
public:
     Multi_config_obj()
     {
       add_parameter("length_of_buffer","0");
       add_parameter("number_of_dest","0");
       add_parameter("number_of_buffers","0");
       add_parameter("number_of_sources","0");
       add_parameter("number_of_warmup","0");
       add_parameter("number_of_cycles","0");
       add_parameter("gamma_","0.0");
       add_parameter("seed_index","0");
```

```
    }
};

int main(int argc, char *argv[])
{

cout<<buffer_entity::LostCellsCount<<endl;

    char ConfigFileName[80],InputFileName[80];

    if (argc<2)
      { cout<<endl
             <<"the configuration file name :";
        cin>>ConfigFileName;
      }
 else { strcpy(ConfigFileName,argv[1]); }
    if (argc<3)
      { cout<<endl
             <<"the input file name :";
        cin>>InputFileName;
      }
 else { strcpy(InputFileName,argv[2]); }

    sim_set_config(new Multi_config_obj());
    sim_config()->read_file(ConfigFileName);
    Multi_set_parameters();

    buffer_entity::NumOfDCycles=NumOfCycles;

    //read in parameters as PCR, etc.....
    double *transrate = new
                        double[NumberOfBuffers];
    double *scr = new double[NumberOfSources];
    double *pcr = new double[NumberOfSources];
    double *abs = new double[NumberOfSources];
    int *dFlags = new int[NumberOfBuffers*2];
    double *Alpha = new double[NumberOfSources];
    int (*NextBuffer)[2];
    NextBuffer = new int[NumberOfBuffers][2];
    sim_bool *HotSpot = new sim_bool
                        [NumberOfBuffers];
    sim_bool *Source_Hot = new sim_bool
                        [NumberOfSources];
    int *Source_Dest = new int[NumberOfSources];
    sim_bool *outlet = new sim_bool[NumberOfBuffers];
    sim_bool *inlet = new sim_bool[NumberOfBuffers];
    int *Qlength = new int[2*NumberOfBuffers];
    sim_bool *inter = new sim_bool[NumberOfBuffers];
    int *sourceType = new int[NumberOfSources];

    int i;
    // initialize HotSpot
    for (i=0;i<=NumberOfBuffers-1;i++)
    {
        HotSpot[i]=FALSE;
        outlet[i]=FALSE;
        inter[i]=FALSE;
    }

    // initialize the source hotspot
    for (i=0;i<=NumberOfSources-1;i++)
    {
        Source_Hot[i]=FALSE;
    }

    getParameters(InputFileName,scr,"SCR");
    getParameters(InputFileName,pcr,"PCR");
    getParameters(InputFileName,abs,"ABS");
    getParameters(InputFileName,transrate,"TR");
    for (i=0;i<NumberOfBuffers;i++) transrate[i]=
                        transrate[i]/
                        transrate[NumberOfBuffers-1];
    getParameters(InputFileName,Alpha,"ALPHA");
    getParameters(InputFileName,NextBuffer,
                        "NEXTBUFFER");
    getParameters(InputFileName,HotSpot,"HOTSPOT");
    getParameters(InputFileName,Source_Hot,"S_HOT");
    getParameters(InputFileName,dFlags,"DFLAGS");
    getParameters(InputFileName,Source_Dest,"S_DEST");
    getParameters(InputFileName,outlet,"OUTLET");
    getParameters(InputFileName,inlet,"INLET");
    getParameters(InputFileName,Qlength,"QLENGTH");
    getParameters(InputFileName,inter,"INTER");
    getParameters(InputFileName,sourceType,"STYPE");

    source_entity *source_ptr[NumberOfSources];
    source_entity *Tsource_ptr[NumberOfSources];
    buffer_entity *buffer_ptr[NumberOfBuffers+1];
    buffer_entity *Tbuffer_ptr[NumberOfBuffers+1];
    char entityname1[15],entityname[15];

    //construct the source entities
    for (i=0;i<=NumberOfSources-1;i++)
    {
```

```
  sprintf(entityname,"%s%d","Source",i);
  sprintf(entityname1,"%s%d","TSource",i);
  if (Source_Hot[i])
    {
     source_ptr[i] = new source_entity(
             entityname,
             scr[i],pcr[i],abs[i],
             Source_Dest[i],sourceType[i],
             Alpha[0],Gamma);
     Tsource_ptr[i] = new source_entity(
             entityname1,
             scr[i],pcr[i],abs[i],
             Source_Dest[i],sourceType[i],
             Alpha[0],Gamma);
     }
else {
     source_ptr[i] = new source_entity(
             entityname,
             scr[i],pcr[i],abs[i],
             Source_Dest[i],sourceType[i]);
     Tsource_ptr[i] = new source_entity(
             entityname1,
             scr[i],pcr[i],abs[i],
             Source_Dest[i],sourceType[i]);
     }

 }


// construct the buffer entities
for (i=0;i<=NumberOfBuffers-1;i++)
{
  sprintf(entityname,"%s%d","Buffer",i);

  buffer_ptr[i] = new buffer_entity(entityname,
          HotSpot[i],inlet[i],Qlength[i],
          Qlength[i+NumberOfBuffers],
          FALSE,inter[i],transrate[i]);
 }

 // construct the Tbuffer entities
 for (i=0;i<=NumberOfBuffers-1;i++)
 {
   sprintf(entityname,"%s%d","TBuffer",i);

   Tbuffer_ptr[i] = new buffer_entity(entityname,
           HotSpot[i],inlet[i],Qlength[i],
           Qlength[i+NumberOfBuffers],
           TRUE,inter[i],transrate[i]);
}

//last buffer_ptr is NULL
buffer_ptr[NumberOfBuffers]=0;
Tbuffer_ptr[NumberOfBuffers]=0;

for (i=0;i<NumberOfSources;i++)
{
     source_ptr[i]->MakeBufferPtr(
                        buffer_ptr[i%4]);
     Tsource_ptr[i]->MakeBufferPtr(
                         Tbuffer_ptr[i%4]);
}
int j,test;

char Qname0[20],Qname1[20],
      TQname0[20],TQname1[20];
for (i=0;i<=NumberOfBuffers-1;i++)
{
  Tbuffer_ptr[i]->B1=Tbuffer_ptr[NextBuffer[i][0]];
  buffer_ptr[i]->B1=buffer_ptr[NextBuffer[i][0]];
  Tbuffer_ptr[i]->B2=Tbuffer_ptr[NextBuffer[i][1]];
  buffer_ptr[i]->B2=buffer_ptr[NextBuffer[i][1]];

  test=1;
  for (j=0;j<=3;j++)
  {
   if (dFlags[i+NumberOfBuffers]&test)
       {
       if (testAdFlags[i])
         {
         buffer_ptr[i]->NextBuffer[j]=
               buffer_ptr[NextBuffer[i][1]];
         sprintf(Qname0,"%s0",buffer_ptr[i]->
                   NextBuffer[j]->GetName());
         sprintf(Qname1,"%s1",buffer_ptr[i]->
                   NextBuffer[j]->GetName());
         buffer_ptr[i]->NextQueue[j]=Qname0;
         buffer_ptr[i]->NextQueue[j+4]=Qname1;
         Tbuffer_ptr[i]->NextBuffer[j]=
                   Tbuffer_ptr[NextBuffer[i][1]];
         sprintf(TQname0,"%s0",Tbuffer_ptr[i]->
                   NextBuffer[j]->GetName());
         sprintf(TQname1,"%s1",Tbuffer_ptr[i]->
                   NextBuffer[j]->GetName());
```

```
    Tbuffer_ptr[i]->NextQueue[j]=TQname0;
    Tbuffer_ptr[i]->NextQueue[j+4]=TQname1;
      }
    else {
      buffer_ptr[i]->NextBuffer[j]=
          buffer_ptr[NextBuffer[i][0]];
      sprintf(Qname0,"%s0",buffer_ptr[i]->
          NextBuffer[j]->GetName());
      sprintf(Qname1,"%s1",buffer_ptr[i]->
          NextBuffer[j]->GetName());
      buffer_ptr[i]->NextQueue[j]=Qname0;
      buffer_ptr[i]->NextQueue[j+4]=Qname1;
      Tbuffer_ptr[i]->NextBuffer[j]=
          Tbuffer_ptr[NextBuffer[i][0]];
      sprintf(TQname0,"%s0",Tbuffer_ptr[i]->
          NextBuffer[j]->GetName());
      sprintf(TQname1,"%s1",Tbuffer_ptr[i]->
          NextBuffer[j]->GetName());
      Tbuffer_ptr[i]->NextQueue[j]=TQname0;
      Tbuffer_ptr[i]->NextQueue[j+4]=TQname1;
      }

    }
  else {
    buffer_ptr[i]->NextBuffer[j]=
        buffer_ptr[NumberOfBuffers];
    Tbuffer_ptr[i]->NextBuffer[j]=
        Tbuffer_ptr[NumberOfBuffers];
    }
  test<<i;   // left bitshift
  }
}

// remove parameter
// arrays from the free store
delete [] scr;
delete [] pcr;
delete [] abs;
delete [] transrate;
delete [] dFlags;
delete [] NextBuffer;
delete [] HotSpot;
delete [] Source_Hot;
delete [] Source_Dest;
delete [] outlet;
delete [] Qlength;
delete [] inlet;

// initialize the Buffers
for (j=0;j<NumberOfBuffers;j++)
{
        buffer_ptr[j]->initialize_normal();
}

for (i=0;i<NumberOfSources;i++)
    {
        source_ptr[i]->initialize_normal();
    }

int Length,k;
call *pCall1,*pCall2;
int *nowOnOff = new int[NumberOfSources];
double factor;
int initHotNum;
double TotalT=0.0,TotalTime=0.0,
    TotalTimeT=0.0;
int Total=0;

for (j=0;j<NumberOfSources;j++)
{
  nowOnOff[j]=source_ptr[j]->isOnOff();
}

cout<<endl<<"start simulation"<<endl;

buffer_entity::CycleCount=0;
i=0;  // use for Delta_zero count
buffer_entity::WarmUp=FALSE;
double Bar_diff;
sim_bool hit=FALSE;
int hotNumWhenHit=0,queueType;
char Name[20];

for (i=1;i<=NumOfCycles;i++)
{

buffer_entity::twist=FALSE;
buffer_entity::LostCallsCount=0;

sim_run_simulation();

Total+=buffer_entity::LostCallsCount;
```

```
                (buffer_ptr[j]->next_time());
            buffer_ptr[j]->suspend_normal
                (buffer_ptr[j]->next_time()
                -sim_clock());
        }
    }
    for (j=0;j<NumberOfSources;j++)
    {
        Tsource_ptr[j]->
            initialize_twisted(source_ptr[j]->
            next_time(),source_ptr[j]->
            BurstLength);
        source_ptr[j]->suspend_normal(
            source_ptr[j]->next_time()
            -sim_clock());
    }

    sim_run_simulation();

    if (buffer_entity::untwistFlag)
        {
        for (j=0;j<NumberOfSources;j++)
    { Tsource_ptr[j]->untwist(); }
        hit=TRUE;
        hotNumWhenHit=buffer_entity::hotNumber;
        sim_run_simulation();
    } else { hit=FALSE; }

    // after a cycle, the parameters
    // are reset to their original values
    //  and the factor is calculated
    Ber_diff=0.0;
    factor=1.0;
    for (j=0;j<NumberOfSources;j++)
        {
        if (Tsource_ptr[j]->Priority==1)
    {
            Ber_diff = (double) (Tsource_ptr[j]->
                InitialOn-Tsource_ptr[j]->
                OverflowOn);
            factor *= pow(9.895438431,Ber_diff);
    }
    Tsource_ptr[j]->suspend_twisted();
    source_ptr[j]->reinitialize_normal();
    }
initHotNum=buffer_entity::hotNumber;

for (j=0;j<NumberOfBuffers;j++)
    {

    // make copy of the contents
    // of the buffers
    for (queueType=0;queueType<=1;
                queueType++)
        {
Tbuffer_ptr[j]->Queue[queueType]->
            clear();
        Length=buffer_ptr[j]->
            Queue[queueType]->length();
        if (Length>=1)
    {
        for (k=1;k<=Length;k++)
        {
            pCell1=(cell*) buffer_ptr[j]->
                retrieve(NO_WAIT,
                    IGNORE_ACTIVATES,
                    buffer_ptr[j]->Queue
                        [queueType]);
            pCell2= new cell(pCell1->
                get_dest(),pCell1->Priority);
            sprintf(Name,"%s%d",Tbuffer_ptr[j]->
                GetName(),queueType);
            Tbuffer_ptr[j]->send_message(pCell1,Name);
            sprintf(Name,"%s%d",buffer_ptr[j]->
                GetName(),queueType);
            buffer_ptr[j]->send_message(pCell2,Name);
        }
    }
    }

    for (queueType=0;queueType<=1;queueType++)
    {
        Tbuffer_ptr[j]->
            Empty[queueType]=buffer_ptr[j]->
                        Empty[queueType];
    }
    if (buffer_ptr[j]->IsHotSpot())
        TotalTime+=buffer_ptr[j]->
        NumberOfTimeSlots;
    Tbuffer_ptr[j]->initialize_twisted
```

```
    }
cout<<"Total Load Normal: "<<sent1
    <<"  Total Load Twisted: "
    <<sent2<<endl;

sent1=0; sent2=0;

for (i=0;i<NumberOfBuffers;i++)
    {
    sent1+=buffer_ptr[i]->CallsToHotspot;
    sent2+=Tbuffer_ptr[i]->CallsToHotspot;
    }

cout<<"Total Calls Received (normal): "<<sent1<<endl;
cout<<"Total Calls Received (twisted): "<<sent2<<endl;
cout<<endl<<"CLR: (twisted) "<<(double)TotalT/(double)sent1;
cout<<"    (normal) "<<(double) Total/(double)sent1<<endl;
}

void getParameters(char *fileName, double *get, char *Mode)
{
enum MODE{Start,CheckMode,Get};
enum STATE{ON,BETWEEN};

ifstream fin(fileName);  //open for reading

int i(0),Count(0);
char ch,buffer[80];
MODE mode(Start);
STATE state(BETWEEN);

while (fin.get(ch))
{
switch(mode)
    {
    case Start: if (ch==':') mode=CheckMode;
                break;

    case CheckMode: if (ch=='-')
                        { buffer[i]='\0';
                          i=0;
                          if (!strcmp(buffer,Mode))
                              { mode=Get;
                                state=BETWEEN; }
                          else mode=Start;
                        }
                    else buffer[i++]=ch;
```

```
if (!hit) hotNumWhenHit=
    buffer_entity::hotNumber;

factor = exp(Gamma*(initHotNum-hotNumWhenHit));

TotalT+=factor*(double)buffer_entity::
            LostCallsCount*exp(-Gamma*(double)
            buffer_entity::ExtraFactor);

for (j=0;j<NumberOfBuffers;j++)
    {
    if (Tbuffer_ptr[j]->IsHotSpot())
        TotalTimeT+=Tbuffer_ptr[j]->
            NumberOfTimeSlots;
    Tbuffer_ptr[j]->suspend_twisted();
    buffer_ptr[j]->reinitialize_normal();
    }

buffer_entity::hotNumber=initHotNum;
}

cout<<endl<<"Number of Delta-Cycles: "
    <<NumOfCycles<<endl;
cout<<endl<<"End Of Simulation:"
    <<endl<<endl;
cout<<"Lost Calls Normal: "
    <<Total<<endl;
cout<<"Lost Calls Twisted: "
    <<TotalT<<endl;
cout<<"Avg Length of TCycles "
    <<(TotalTime?/(double) NumOfCycles
    <<endl;
cout<<"TotalTime "<<TotalTime<<endl;
cout<<"Avg Length of Cycles: "
    <<(TotalTime/(double) NumOfCycles
    <<endl;
cout<<"Twisted Estimate: "
    <<(TotalT/TotalTime
    <<"  Normal Estimate: "
    <<Total/TotalTime<<endl<<endl;

int sent1=0,sent2=0;

for (i=0;i<NumberOfSources;i++)
    {
    sent1+=source_ptr[i]->CreatedCallsCount;
    sent2+=Tsource_ptr[i]->CreatedCallsCount;
```

```
                    break;
                }
            case Get:  if (state==ON)
                    { if (ch!='\'') { buffer[i++]=ch; }
                        else { buffer[i]='\0';
                               i=0;
                               get[Count++]=atoi(buffer);
                               state=BETWEEN;
                             }
                    }
                    else { if (ch=='\'')  { state=ON; }
                           else { if (ch==':')  mode=CheckMode; }
                    }
                    break;
        }
    }
    fin.close();
}

void getParameters(char *fileName,  int (*NextBuffer)[2], char *Mode)
{
    enum MODE{Start,CheckMode,NEXTBUFFER};
    enum STATE{ON,BETWEEN,NEXT};

    ifstream fin(fileName);  //open for reading

    int from(0),to(0),i(0);
    char ch,buffer[80];
    MODE mode(Start);
    STATE state(BETWEEN);

    while (fin.get(ch))
    {
        switch(mode)
        {
        case Start: if (ch==':') mode=CheckMode;
                    break;
        case CheckMode: if (ch=':')
                        { buffer[i]='\0';
                          i=0;
                          if (!strcmp(buffer,Mode))
                                { mode=NEXTBUFFER;
                                  state=BETWEEN; }
                        else mode=Start;
                        }
                        else buffer[i++]=ch;
                        break;
case NEXTBUFFER: switch(state)
                 {
                 case ON: switch(ch)
                          {
                          case '>': state=NEXT;
                                    buffer[i]='\0';
                                    i=0;
                                    from=atoi(buffer);
                                    break;
                          case '\'': state=BETWEEN;
                                     buffer[i]='\0';
                                     i=0;
                                     NextBuffer[from][to]=
                                         atoi(buffer);
                                     break;
                          default: buffer[i++]=ch;
                                   break;
                          }
                          break;
                 case NEXT: switch(ch)
                            {
                            case '>': state=ON;
                                      buffer[i]='\0';
                                      i=0;
                                      to=atoi(buffer);
                                      break;
                            default: buffer[i++]=ch;
                                     break;
                            }
                            break;
                 case BETWEEN: switch(ch)
                               {
                               case '\'': state=ON;
                                          break;
                               case ':': mode=CheckMode;
                                         break;
                               }
                               break;
                 }
    }
    fin.close();
}

void getParameters(char *fileName, sim_bool  *get, char *Mode)
{
```

```
enum MODE{Start,CheckMode,Get};
enum STATE{ON,BETWEEN};

ifstream fin(fileName);  //open for reading

int i(0);
char ch,buffer[80];
MODE mode(Start);
STATE state(BETWEEN);

while (fin.get(ch))
{
    switch(mode)
    {
    case Start: if (ch==':') mode=CheckMode;
                break;

    case CheckMode: if (ch==':')
                        { buffer[i]='\0';
                          i=0;
                          if (!strcmp(buffer,Mode))
                              { mode=Get;
                                state=BETWEEN; }
                          else mode=Start;
                        }
                    else buffer[i++]=ch;
                    break;
    case Get:  if (state==ON)
                   { if (ch!='\'')
                         { buffer[i++]=ch; }
                     else { buffer[i]='\0';
                            i=0;
                            get[atoi(buffer)]=TRUE;
                            state=BETWEEN;
                          }
                   }
               else { if (ch=='\'') state=ON;
                      else if (ch==':') { mode=CheckMode; }
                    }
               break;
    }
}
fin.close();
}

void getParameters(char *fileName, int *get, char *Mode)
{

enum MODE{Start,CheckMode,Get};
enum STATE{ON,BETWEEN};

ifstream fin(fileName);  //open for reading

int i(0),Count(0);
char ch,buffer[80];
MODE mode(Start);
STATE state(BETWEEN);

while (fin.get(ch))
{
    switch(mode)
    {
    case Start: if (ch==':') mode=CheckMode;
                break;

    case CheckMode: if (ch==':')
                        { buffer[i]='\0';
                          i=0;
                          if (!strcmp(buffer,Mode))
                              { mode=Get;
                                state=BETWEEN; }
                          else mode=Start;
                        }
                    else buffer[i++]=ch;
                    break;
    case Get:  if (state==ON)
                   { if (ch!='\'')  { buffer[i++]=ch; }
                     else { buffer[i]='\0';
                            i=0;
                            get[Count++]=atoi(buffer);
                            state=BETWEEN;
                          }
                   }
               else { if (ch=='\'')  { state=ON; }
                      else { if (ch==':')  mode=CheckMode; }
                    }
               break;
    }
}
fin.close();
}
```

Bibliography

[1] Asmussen, S., (1987) *Applied Probability and Queues.* John Wiley & Sons, Chichester.

[2] Asmussen S., Reuven R.Y., and Wang C-L. (1994) *Regenerative Rare Events Simulation via Likelihood Ratios*, J. Appl. Prob., **31**, 797-815.

[3] Beck B., Dabrowski A.R., and McDonald D.R. (1998) *A Unified Approach to Fast Teller Queues and ATM*, to appear in Advances in Applied Probability.

[4] Bonneau, M-C., (1996) *Accelerated Simulation of a Leaky Bucket Controller*, M.Sc. thesis, University of Ottawa.

[5] Burden, R. L., and Faires, J.D. (1993) *Numerical Analysis.* PWS Publishing Company, Boston.

[6] Chang C.-S., Heidelberger P., Juneja S., and Shahabuddin P., (1994). *Effective Bandwidth and Fast Simulation of ATM intree Networks*, Performance Evaluation, **20**, 45-65.

[7] Conway, J.B., (1978) *Functions of One Complex Variable I.* Springer-Verlag, New York.

[8] Falkner M., Devetsikiotis M., and Lambadaris L. (1998) *Issues in Fast Simulation of Networks of Queues by Use of Effective and Decoupling Bandwidths*, to appear in ACM Transactions on Modelling and Computer Simulation.

[9] Feller W., (1966) *An Introduction to Probability Theory and Its Applications*, Volume II. John Wiley & Sons, New York.

[10] Graham, A., (1987) *Nonnegative Matrices and Applicable Topics in Linear Algebra.* John Wiley & Sons, New York.

[11] Heidelberger P. (1995) *Fast Simulation of Rare Events in Queueing and Reliability Models*, ACM Transactions on Modeling and Computer Simulation, **5**, 43-85.

[12] L'Ecuyer, P. and Champoux, Y. (1996) *Importance Sampling for Large ATM-Type Queueing Networks* Proceedings of the 1996 Winter Simulation Conference, IEEE Press, 309-316

[13] McDonald, D. (1994) *Elements of Applied Probability for Engineering, Mathematics and Systems Science.* Quality Advice, Ottawa.

[14] McDonald, D.R. (1998) *Asymptotics of First Passage Times for Random Walk in an Orthant*, to appear in Annals of Applied Probability.

[15] Ney P., and Nummelin E., (1987) *Markov Additive Processes I. Eigenvalue Properties and Limit Theorems*, Annals of Probability, **15**, 561-592.

[16] Rubinstein, R.Y. (1981) *Simulation and the Monte Carlo Method.* Wiley, New York.

[17] Wright, D., (1993) *Broadband: Business Services, Technologies, and Strategic Impact.* Artech House, Boston.

Glossary of Symbols

Index