ACCELERATED SIMULATION OF A LEAKY BUCKET CONTROLLER

By
Marie-Claude Bonneau, B.Sc.
August 1996

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]

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

ISBN 0-612-16405-5

Canada

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

Abstract

The leaky bucket controller has been proposed to shape traffic entering an ATM network. Cells arrive at the leaky bucket controller according to a superposition of a group of heterogeneous Markov modulated sources (modeled by a multidimensional Markov process). The bucket capacity is L tokens. Every time slot the bucket leaks s tokens. If a cell arrives I tokens are added to the bucket unless it overflows. A cell producing an overflow is marked nonconforming and no tokens are added.

The proportion of nonconforming cells and the mean time between nonconforming cells are significant for evaluating the performance of the leaky bucket. Such parameters can be approximated by developing a change of measure to transform a recurrent process into a transient one. For this twisted process, the event until overflow is no longer a rare event and can be simulated efficiently. We can now give a formula for calculating the proportion of cells which are nonconforming as well as the mean time between bursts of nonconforming cells.

Remerciements

Je voudrais tout d'abord remercier ma famille pour leur encouragement tout au long de mes études.
Je voudrais remercier mon directeur de thèse, David McDonald. Ses commentaires ont été d'une aide précieuse pour la réalisation de ce travail.
Je voudrais également remercier Bryan Morris pour son support technique.
De plus, je voudrais remercier la Compagnie Newbridge et l'Université d'Ottawa pour leur soutien financier.

Contents

Abstract ii

Remerciements iii

1 Introduction 1
- 1.1 Background 1
- 1.2 The Model 3

2 The Twist 6
- 2.1 Numerical calculation of the twist 9
- 2.2 Approximation to the fluid model 10

3 Theoretical value of γ 13

4 Application 18
- 4.1 Finite Buffer Analysis 18
- 4.2 Leaky Bucket Analysis 21
- 4.3 Verification via an Infinite Buffer 23

5 Conclusions 27

A Program to find γ 28

B Program to simulate a leaky bucket 33

C Program to simulate an infinite multiplexer 56

Bibliography 75

Chapter 1

Introduction

1.1 Background

Asynchronous transfer mode or, more commonly, ATM, will play an important role in everyday life. From a phone conversation to a bank transaction, a lot of services offered are transmitted via a network of fiber optical cables. This heterogeneous mix of transmitted information has to be accurate, fast and reliable. In order to meet those requirements, ATM offers a viable solution.

The leaky bucket controller has been proposed to shape traffic entering an ATM network. We are going to analyze the leaky bucket controller for traffic entering as a superposition of a group of heterogeneous Markov-modulated sources (modeled by a multidimensional Markov process) where the cells are marked conforming or nonconforming. The time considered is discrete. Solving such a model numerically is practically impossible due to the number of states required to represent the sources. Simulating the model tends to be difficult because of the time required to obtain an accurate approximation of the numerical solution.

In order to obtain an analytic solution for the leaky bucket controller, most studies have opted for a Markov-modulated fluid model. For instance, Kosten [6] studied a model in which an infinite buffer receives cells from an infinite number of independant, mutually identical, sources. For each source, the periods of the ON and OFF states were exponentially distributed, and the transmission rate during an ON state

is uniform. Through analytical and numerical means, the distribution of the buffer content was found using the eigenvalues of the system. In another study by Anick, Mitra and Sondhi [1], where the number of sources was restrained to a finite number, an eigenfunction approach gives the distribution as well as the moments of the system. Furthermore, as $N \to \infty$, their results coincide with Kosten's result. Along with other studies of Markov-modulated fluid model, these results are then used to draw general conclusions about the discrete time model. In such a case it is best to analyze a discrete Markov-modulated model. Buffet and Duffield [3, 4] have considered a model for which an upper bound on the overflow probability was obtained. As we will show later (Section 2.2), this upper bound is asymptotically too conservative.

We propose to obtain an accurate asymptotic value the overflow probabilities for a discrete Markov-modulated model by developing a change of measure (referred to as a twist and equivalent to the importance sampling technique given in Heidelberger [5]. The article by Heidelberger gives a comprehensive review of the literature of importance sampling.) The novelty here is the explicit analytic form of the twist. The twist will thus transform a recurrent process into a transient one. In such a case, the event of buffer overflow is no longer a rare event and can be simulated. To do so, we use a technique similar to the A-cycle technique given in Heidelberger [5]. Such an analysis of the leaky bucket controller is new as is the usage of *SimS*, a *C++* object-oriented simulation library. With this, we can now give a algorithm for calculating the proportion of cells which are nonconforming.

Chapter 2 describes the twist with its parameters and compares the results obtained with the fluid model. Numerical calculation of the parameter of the twist is given in Chapter 3. Section 4.1 describes how the overflow probabilities for a finite buffer can be calculated using the twist. The twist is then applied to a leaky bucket controller (Section 4.2) as well as to an infinite buffer (Section 4.3). Finally, we draw some conclusions in Chapter 5.

1.2 The Model

We shall assume time is divided into slots defined by the link rate. We shall assume cells arrive at a leaky bucket controller according to a superposition of a group of N heterogeneous Markov modulated sources. A leaky bucket can be viewed as a buffer with a finite capacity $L + I$ where L denotes the finite capacity of a buffer and at a cell arrival, the bucket is increased by a quantity I.

The N sources consist of $M(\geq 1)$ groups of N_i stochastically identical sources, $i = 1, \ldots, M$. For each group, the N_i identical, mutually independent, bursty sources can each be ON or OFF. The changes from ON to OFF and from OFF to ON take place independently with probability a_i and d_i, respectively.

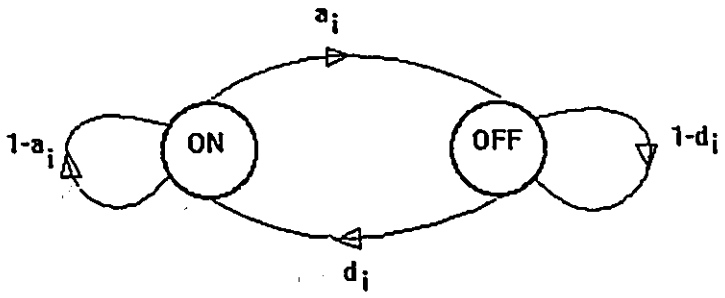

Figure 1: The model for a single source

We shall model these changes to take place instantaneously at the end of the time slot. While it is on, each source produces cells according to a Bernoulli process with rate r_i; that is cells arrive independently in each time slot with probability r_i. Multiple arrivals in a time slot are allowed but in practice this just results in jitter which we ignore in our model.

In steady state, the probability one source in group i is ON is given by

$$\frac{1/a_i}{(1/a_i + 1/d_i)} = \frac{d_i}{a_i + d_i}.$$

Hence the mean number of cells which arrive per time slot is

$$\sum_{i=1}^{M} N_i \frac{d_i}{a_i + d_i} r_i > 0.$$

We assume this mean arrival rate times I is less than the leak rate, denoted s, that is

$$I\sum_{i=1}^{M} N_i \frac{d_i}{a_i + d_i} r_i < s.$$

The leak rate is the maximum number of cells that can be removed from the bucket during a time slot.

We denote by $N_i(t)$ the number of sources which are in status ON in group i at the start of the t^{th} time slot. Following Buffet and Duffield [3], the transition from n_i to m_i sources ON has probability

$$\hat{K}_i(n_i, m_i) := \sum_{x+y=m_i} \binom{n_i}{x} a_i^{n_i-x}(1-a_i)^x \cdot \binom{N_i - n_i}{y} d_i^y (1-d_i)^{N_i-n_i-y}$$

since x of the i ON's may stay ON and y of the $N_i - i$ OFF's may turn ON. Denote the state of the sources by $N(t) \equiv (N_1(t), \ldots N_M(t))$. The transition from $\vec{n} \equiv (n_1, \ldots n_M)$ to $\vec{m} \equiv (m_1, \ldots m_M)$ has probability $\hat{K}(\vec{n}, \vec{m}) = \prod_{i=1}^{M} \hat{K}_i(n_i, m_i)$. Given $N_i(t) = n_i$, the cell arrival process of the i^{th} group at time slot t, denoted by $A_i(t)$ is a binomial random variable with parameters n_i and r_i. Hence, given $N(t) = \vec{n}$, the arrival process is given by $A(t) = \sum_i A_i(t)$.

Also

$$\begin{aligned} & E\left(e^{\gamma I A(t)} | N(t) = \vec{n}\right) \\ &= E\left(e^{\gamma I \sum_{i=1}^{M} A_i(t)} | N(t) = \vec{n}\right) \\ &= \prod_{i=1}^{M} E\left(e^{\gamma I A_i(t)} | N_i(t) = n_i\right) \\ &= \prod_{i=1}^{M} (1 - r_i + r_i e^{\gamma I})^{n_i}. \end{aligned}$$

The content $Q(t)$ of the leaky bucket evolves according to

$$Q(t+1) = (Q(t) + I \cdot A(t) - s)^+.$$

The mean time until the leaky bucket content exceeds $L + I$ is significant in performance evaluation of the leaky bucket controller. Such a parameter is difficult to

simulate due to the enormous computation time it requires because the leaky bucket is stable and rarely exceeds $L + I$. However, the mean time until the leaky bucket content exceeds $L + I$ may be approximated by a technique developed in McDonald [8] (Also see Heidelberger [5] for similar approaches to this problem). The main idea is to develop a twist or change of measure which will transform the recurrent process $(N(t), Q(t))$ into a transient process $(\mathcal{N}(t), \mathcal{Q}(t))$. The asymptotic behaviour of $(\mathcal{N}(t), \mathcal{Q}(t))$ as $L + I$ becomes large may be approximated or quickly simulated and then converted into estimates for the asymptotic behaviour of $(N(t), Q(t))$. The essential ingredient in the twist is a harmonic function like that used in Buffet and Duffield [3] and Duffield [4].

Chapter 2

The Twist

The asymptotic behaviour of the process $(N(t), Q(t))$ when L is large will first be approximated by the behaviour of $\{(N(t), Q^{\infty}(t)); t = 0, 1, \ldots\}$ where Q^{∞} evolves according to $Q^{\infty}(t+1) = Q^{\infty}(t) + I \cdot A(t) - s$. In other words we remove the boundary associated with the buffer emptying. If we define $\phi(t) = I \cdot A(t) - s$ as the increment associated with transition t then the process $(N(t), Q^{\infty}(t))$ is a Markov-additive process with a transition kernel

$$K^{\infty}((\vec{n}, l); (\vec{m}, l+k)) = \hat{K}(\vec{n}, \vec{m}) P(\phi(1) = k | N(0) = \vec{n}).$$

The generating function of this transition kernel is

$$\hat{K}_{\gamma}(\vec{n}, \vec{m}) = \hat{K}(\vec{n}, \vec{m}) E(e^{\gamma \phi(1)} | N(0) = \vec{n}, N(1) = \vec{m}).$$

The existence of eigenvalues and eigenvectors for this "Feynman-Kac" operator was studied in Ney and Nummelin [9]. There exist a $\gamma > 0$ such that $\hat{K}_{\gamma}$ has spectral radius one and there exists a positive Perron-Frobenius eigenvector $r(\vec{n})$ satisfying $r = \hat{K}_{\gamma} r$. In such a case, the eigenvector $r(\vec{n})$ is said to be a regular or harmonic function. The harmonic function $e^{\gamma l} r(\vec{n})$ may be used to twist the process $(N(t), Q^{\infty}(t))$ into the process $(\mathcal{N}(t), \mathcal{Q}^{\infty}(t))$ with transition kernel

$$\mathcal{K}^{\infty}((\vec{n}, l); (\vec{m}, l+k)) := K^{\infty}((\vec{n}, l); (\vec{m}, l+k)) \frac{r(\vec{m})}{r(\vec{n})} \frac{e^{\gamma(l+k)}}{e^{\gamma l}}.$$

The twist is thus the Markov chain harmonically transformed. Remark that paths of the twist are the same as the original chain but there has been a change of measure. Furthermore, the eigenvector of $\hat{K}_\gamma$ is in fact of the form $r(\vec{n}) = \vec{\beta}^{\vec{n}} \equiv \prod_{i=1}^{M} \beta_i^{n_i}$. To show this first remark that

$$\begin{aligned} & E(e^{\gamma\phi(1)}|N(0)=\vec{n}, N(1)=\vec{m}) \\ &= E\left(e^{\gamma I A(1)} e^{-\gamma s}|N(0)=\vec{n}, N(1)=\vec{m}\right) \\ &= e^{-\gamma s}\prod_{i=1}^{M}(1-r_i+r_i e^{\gamma I})^{n_i}. \end{aligned}$$

Also let $\alpha_i = (1 - r_i + r_i e^{\gamma I})$. Hence

$$\begin{aligned} \hat{K}_\gamma r(\vec{n}) &= \sum_{\vec{m}} \hat{K}_\gamma(\vec{n},\vec{m})\vec{\beta}^{\vec{m}} \\ &= e^{-\gamma s}\prod_{i=1}^{M}(1-r_i+r_i e^{\gamma I})^{n_i}\sum_{\vec{m}}\hat{K}(\vec{n},\vec{m})\prod_{i=1}^{M}\beta_i^{m_i} \\ &= e^{-\gamma s}\prod_{i=1}^{M}\alpha_i^{n_i}\prod_{i=1}^{M}\left(\sum_{m_i}\sum_{x+y=m_i}\binom{n_i}{x}a_i^{n_i-x}(1-a_i)^x\cdot\binom{N_i-n_i}{y}d_i^y(1-d_i)^{N_i-n_i-y}\beta_i^{m_i}\right) \\ &= e^{-\gamma s}\prod_{i=1}^{M}\alpha_i^{n_i}\prod_{i=1}^{M}\left(\sum_{x}\binom{n_i}{x}a_i^{n_i-x}(1-a_i)^x\cdot\sum_{y}\binom{N_i-n_i}{y}d_i^y(1-d_i)^{N_i-n_i-y}\beta_i^{x+y}\right) \\ &= e^{-\gamma s}\prod_{i=1}^{M}\alpha_i^{n_i}\prod_{i=1}^{M}\left((a_i+(1-a_i)\beta_i)^{n_i}(1-d_i+d_i\beta_i)^{N_i-n_i}\right) \\ &= \left(\prod_{i=1}^{M}\left(\frac{(a_i+(1-a_i)\beta_i)\alpha_i}{1-d_i+d_i\beta_i}\right)^{n_i}\right)\left(e^{-\gamma s}\prod_{i=1}^{M}(1-d_i+d_i\beta_i)^{N_i}\right) \end{aligned}$$

Hence if we want $\hat{K}_\gamma r(\vec{n}) = r(\vec{n}) = \prod_{i=1}^{M}\beta_i^{n_i}$ then we must set

$$e^{\gamma s} = \prod_{i=1}^{M}(1-d_i+d_i\beta_i)^{N_i} \tag{1}$$

$$\beta_i = \frac{(a_i+(1-a_i)\beta_i)\alpha_i}{1-d_i+d_i\beta_i} \quad , i = 1,\dots M \tag{2}$$

The twisted Markov chain can now be identified. Let $k = cI - s$ and remark that

$$\mathcal{K}^\infty((\vec{n},l);(\vec{m},l+k)) = K^\infty((\vec{n},l);(\vec{m},l+k))\frac{r(\vec{m})}{r(\vec{n})}\frac{e^{\gamma(l+k)}}{e^{\gamma l}}$$

$$= \prod_{i=1}^{M}\left[\sum_{x+y=m_i}\binom{n_i}{x}a_i^{n_i-x}(1-a_i)^x\cdot\binom{N_i-n_i}{y}d_i^y(1-d_i)^{N_i-n_i-y}\right]\times$$
$$\left[\{z_1+\ldots+z_M=c\}\prod_{i=1}^{M}\binom{n_i}{z_i}r_i^{z_i}(1-r_i)^{n_i-z_i}\right]\frac{\vec{\beta}^{\vec{m}}}{\vec{\beta}^{\vec{n}}}e^{\gamma(cI-s)}$$

by substituting the appropriate transition parameters.

Since $\vec{\beta}^{\vec{m}} \equiv \prod_{i=1}^{M}\beta_i^{m_i}$, each $\beta_i^{m_i}$ can be separated into $\beta_i^x\cdot\beta_i^y$. Similarly, $\beta_i^{n_i} = \beta_i^{n_i-x}\cdot\beta_i^x$. Doing so gives us

$$\mathcal{K}^\infty((\vec{n},l);(\vec{m},l+k)) =$$
$$\prod_{i=1}^{M}\left[\sum_{x+y=m_i}\binom{n_i}{x}\left(\frac{a_i}{\beta_i}\right)^{n_i-x}\left(\frac{(1-a_i)\beta_i}{\beta_i}\right)^x\cdot\binom{N_i-n_i}{y}(d_i\beta_i)^y(1-d_i)^{N_i-n_i-y}\right]\times$$
$$\left[\{z_1+\ldots+z_M=c\}\prod_{i=1}^{M}\binom{n_i}{z_i}(r_ie^{\gamma I})^{z_i}(1-r_i)^{n_i-z_i}\right]\cdot e^{-\gamma s}$$

This does not resemble a transition kernel. However, by normalizing $\mathcal{K}^\infty((\vec{n},l);(\vec{m},l+k))$, we obtain

$$\mathcal{K}^\infty((\vec{n},l);(\vec{m},l+k)) =$$
$$\prod_{i=1}^{M}\left[\sum_{x+y=m_i}\binom{n_i}{x}\frac{(a_i/\beta_i)^{n_i-x}(1-a_i)^x}{(1-a_i+a_i/\beta_i)^{n_i}}\cdot\binom{N_i-n_i}{y}\frac{(d_i\beta_i)^y(1-d_i)^{N_i-n_i-y}}{(1-d_i+d_i\beta_i)^{N_i-n_i}}\right]\times$$
$$\{z_1+\ldots+z_M=c\}\prod_{i=1}^{M}\binom{n_i}{z_i}\frac{(r_ie^{\gamma I})^{z_i}(1-r_i)^{n_i-z_i}}{(1-r_i+r_ie^{\gamma I})^{n_i}}$$

To show this, remark that

$$\prod_{i=1}^{M}\left(1-a_i+\frac{a_i}{\beta_i}\right)^{n_i}(a-d_i+d_i\beta_i)^{N_i-n_i}\cdot\prod_{i=1}^{M}\left(1-r_i+r_ie^{I\gamma}\right)^{n_i}e^{-\gamma s}$$
$$= e^{\gamma s}\prod_{i=1}^{M}\left(\frac{1-a_i+a_i/\beta_i}{1-d_i+d_i\beta_i}\right)^{n_i}\prod_{i=1}^{M}\alpha_i^{n_i}\cdot e^{-\gamma s}\text{ by (1)}$$
$$= \prod_{i=1}^{M}\left(\frac{1}{\beta_i}\right)^{n_i}\left(\frac{\alpha_i\left((1-a_i)\beta_i+a_i\right)}{1-d_i+d_i\beta_i}\right)^{n_i}$$
$$= \prod_{i=1}^{M}\frac{1}{\beta_i^{n_i}}\beta_i^{n_i}\text{ by (2)}$$
$$= 1$$

The twisted chain describes the same leaky bucket but the parameters of the bursty sources have changed. The changes from ON to OFF and from OFF to ON take place independently with probability $(a_i/\beta_i)\,/\,(1-a_i+a_i/\beta_i)$ and $(d_i\beta_i)\,/\,(1-d_i+d_i\beta_i)$, respectively. Also when a source is ON, it can produce cells according to a Bernoulli process with rate $\left(r_i e^{\gamma I}\right)\,/\,\left(1-r_i+r_i e^{\gamma I}\right)$, $i=1,\ldots,M$.

2.1 Numerical calculation of the twist

In the special case when there is only one group of sources ($M=1$), (1) and (2) give

$$e^{\gamma s} = (1-d+d\beta)^N \tag{3}$$

$$\beta = \frac{(a+(1-a)\beta)\alpha}{1-d+d\beta} \tag{4}$$

where $\alpha=(1-r+re^{\gamma I})$.

Solving for β in terms of γ in (3) we get

$$\beta = 1+\frac{1}{d}\left(e^{\gamma s/N}-1\right).$$

Hence

$$\left[1+\frac{1}{d}\left(e^{\gamma s/N}-1\right)\right]e^{\gamma s/N} = \alpha\left[a+(1-a)\left(1+\frac{1}{d}(e^{\gamma s/N}-1)\right)\right] \tag{5}$$

For this special case, the parameters of the twist are the following: the source will change from ON to OFF with probability

$$\frac{a\left(1+\frac{1}{d}\left(e^{\gamma s/N}-1\right)\right)^{-1}}{1-a+a\left(1+\frac{1}{d}\left(e^{\gamma s/N}-1\right)\right)^{-1}} = \frac{ad}{d+\left(e^{\gamma s/N}-1\right)(1-a)},$$

from OFF to ON with probability

$$\frac{d\left(1+\frac{1}{d}\left(e^{\gamma s/N}-1\right)\right)}{1-d+d\left(1+\frac{1}{d}\left(e^{\gamma s/N}-1\right)\right)} = 1+\frac{d-1}{e^{\gamma s/N}}.$$

During an ON period, the source will generate cells at a rate of

$$\frac{re^{\gamma I}}{1-r+re^{\gamma I}}.$$

2.2 Approximation to the fluid model

Kosten [6] and Anick, Mitra and Sondhi [1] considered a similar fluid model for which an expression for the asymptotic behaviour of the buffer content was developped. According to their model, the changes from ON to OFF and from OFF to ON take place independently with rate μ and λ respectively. c denotes the leak rate and y denotes the unit of information transmitted per unit of time. To be more precise

$$\lim_{x\to\infty} \log \Pr(\text{buffer content} > x) = -r^*$$

where $r^* = \frac{1}{y}\frac{N\lambda y/c-(\mu+\lambda)}{1-c/Ny}$ and $\frac{N\lambda y}{\mu+\lambda} < c$.

To compare the fluid model with the discrete time model, consider $1/\eta$ to be the unit of time. Define $a = \mu/\eta$, $d = \lambda/\eta$, $s = c/\eta$ and $rI = y/\eta$. Then (5) can be written as

$$\begin{aligned}&\left[1+\frac{\eta}{\lambda}\left(e^{\gamma c/N\eta}-1\right)\right]e^{\gamma c/N\eta}\\ &= (1-r+re^{\gamma y/r\eta})\left[\frac{\mu}{\eta}+\left(1-\frac{\mu}{\eta}\right)\left(1+\frac{\eta}{\lambda}(e^{\gamma c/N\eta}-1)\right)\right]\end{aligned}$$

Now do a 2-term Taylor expansion in $1/\eta$

$$\begin{aligned}&\left[1+\frac{\eta}{\lambda}\left(\frac{\gamma c}{N\eta}+\frac{\gamma^2c^2}{2N^2\eta^2}\right)\right]\left(1+\frac{\gamma c}{N\eta}\right)\\ &= \left(1+-r+r\left(1+\frac{\gamma y}{r\eta}\right)\right)\left[\frac{\mu}{\eta}+\left(1-\frac{\mu}{\eta}\right)\left(1+\frac{\eta}{\lambda}\left(\frac{\gamma c}{N\eta}+\frac{\gamma^2c^2}{2N^2\eta^2}\right)\right)\right]\end{aligned}$$

$$\left(1+\frac{\gamma c}{N\lambda}+\frac{\gamma^2c^2}{2N^2\lambda\eta}\right)\left(1+\frac{\gamma c}{N\eta}\right)=\left(1+\frac{\gamma y}{\eta}\right)\left(1+\frac{\gamma c}{N\lambda}+\frac{\gamma^2c^2}{2\lambda N^2\eta}-\frac{\gamma\mu c}{N\lambda\eta}\right)$$

$$\frac{\gamma c}{N\lambda}\left(y-\frac{c}{N}\right)=\frac{\mu c}{N\lambda}+\frac{c}{N}-y$$

$$\gamma=\frac{\mu+\lambda-yN\lambda/c}{y-c/N}$$

Thus, $\gamma = -r^*$. Some numerical values (Table 1) shows that γ converges quickly to $-r^*$.

Also, Buffet and Duffield [3] and Duffield [4] obtained an explicit upper bound of the form P[queue length $\geq x$] $\leq Ce^{-\delta x}$ where $C < 1$, that is

$$\delta = \ln\left[\frac{(1-\sigma)(a+\sigma(1-a-d))}{\sigma(1-(a+\sigma(1-a-d)))}\right] \tag{6}$$

where $\sigma = s/N$. To compare the coefficient in the exponential expression, consider $1/\eta$ to be the unit of time. Define $a = \mu/\eta$, $d = \lambda/\eta$ and $\sigma = s/N\eta$. Then (6) can be written as

$$\delta = \frac{(1-\frac{s}{N\eta})(\frac{\lambda}{\eta}+\frac{s}{N\eta}(1-\frac{\lambda}{\eta}-\frac{\mu}{\eta}))}{\frac{s}{N\eta}(1-(\frac{\lambda}{\eta}+\frac{s}{N\eta}(1-\frac{\lambda}{\eta}-\frac{\mu}{\eta})))}$$

Now, an asymptotic expansion as $\eta \to \infty$, omitting terms of η^2, shows that

$$\delta \equiv \delta(\eta) \to \delta(\infty) = \frac{\lambda + s/N}{s/N}.$$

We can see that Duffield's upper bound is, asymptotically, independent of μ. Some numerical values are given to compare the coefficient, in the exponential expression, between our value, denoted by γ, and Duffield's value, denoted by δ (Table 2). Here ρ denotes the mean arrival rate. This means Duffield's upper bound is asymptotically too conservative.

Table 1: $\gamma(\eta) \to -r^* = 0.577778$ as η increases

λ	μ	N	η	γ
0.02	0.7	10	1	0.980365
0.02	0.7	10	10	0.599550
0.02	0.7	10	100	0.579867
0.02	0.7	10	1000	0.577986
0.02	0.7	10	10000	0.577799
0.02	0.7	10	100000	0.577780
0.02	0.7	10	1000000	0.577778
0.02	0.7	10	∞	0.577778

Table 2: Comparision between coefficients in the exponential expression

					$\eta = 1$		$\eta \to \infty$	
s	λ	μ	N	ρ	γ	δ	γ	δ
1	0.0001	0.999	10	0.001	7.582	-6.371	1.109	1.001
1	0.02	0.7	10	0.278	0.980	-0.790	0.578	1.2
1	0.1	0.95	10	0.952	0.119	-0.0568	0.0556	2
1	0.02	0.183	10	0.985	0.00371	-0.00334	0.00333	1.2
1	0.1	0.95	4	0.381	2.580	-1.246	0.867	1.4
1	0.03	0.45	15	0.938	0.0424	-0.0326	0.0321	1.45
1	0.03	0.75	15	0.577	0.616	-0.424	0.354	1.45

Chapter 3

Theoretical value of γ

In order to determine numerically the parameters of the twist, we need the value γ. When $M = 1$, a value for γ can be found asymptotically ($\gamma = -r^*$). More generally, define $a_i = \mu_i/\eta$, $d_i = \lambda_i/\eta$, $s = c/\eta$ and $r_i I = y_i/\eta$. (1) and (2) can be written as

$$e^{\gamma c/\eta} = \prod_{i=1}^{M} (1 - \frac{\lambda_i}{\eta} + \frac{\lambda_i}{\eta}\beta_i)^{N_i}$$

$$\beta_i = \frac{(\frac{\mu_i}{\eta} + (1 - \frac{\mu_i}{\eta})\beta_i)(1 - r_i + r_i e^{\gamma y_i/r_i \eta})}{1 - \frac{\lambda_i}{\eta} + \frac{\lambda_i}{\eta}\beta_i}$$

An asymptotic expansion as $\eta \to \infty$, omitting terms of η^2, shows that

$$\beta_i^2 \lambda_i + \beta_i(\mu_i - \lambda_i - \gamma y_i) - \mu_i = 0 \tag{7}$$

$$\gamma = \frac{1}{c}(\sum_{k=1}^{M} N_k \lambda_k \beta_k - \sum_{k=1}^{M} N_k \lambda_k) \tag{8}$$

Combining (7) and (8) gives

$$c\lambda_i\beta_i^2 + \beta_i(c\mu_i - c\lambda_i - y_i \sum_{k=1}^{M} N_k\lambda_k\beta_k - y_i \sum_{k=1}^{M} N_k\lambda_k) - c\mu_i = 0\,, \;\; i = 1, \ldots, M$$

Due to the complexity of the system, a numerical value of γ cannot be obtained rapidly.

Instead of doing an asymptotic expansion of γ, if we consider (2) and solve for β_i, we obtain

$$d_i\beta_i^2 + (1 - d_i - \alpha_i(1 - a_i))\beta_i - a_i\alpha_i = 0$$

that is

$$\beta_i = \frac{-(1 - d_i - \alpha_i + \alpha_i a_i) + \sqrt{(1 - d_i - \alpha_i + \alpha_i a_i)^2 + 4 d_i a_i \alpha_i}}{2 d_i} \tag{9}$$

by considering only the positive root in order to obtain the largest value of β_i.

Now if we solve (1) in terms of γ, we have

$$\ln e^{\gamma s} = \ln[\prod_{i=1}^{M} (1 - d_i + d_i \beta_i)^{N_i}]$$

so

$$\gamma s = \sum_{i=1}^{M} N_i \ln(1 - d_i + d_i \beta_i). \tag{10}$$

Combining (9) and (10), we now have a function only in terms of γ,

$$f(\gamma) = \sum_{i=1}^{M} N_i \ln(1 - d_i + d_i \beta_i) - \gamma s.$$

Solving $f(\gamma) = 0$ would then give us a numerical value for γ. We used the secant method as a method for finding the root of f. Even though the secant method converges less rapidly than the Newton method, the secant method has some advantages. It requires only one evaluation of $f(\gamma_n)$, it requires less time per iteration than the Newton method, and does not require the evaluation of $f'(\gamma_n)$. This straight-line approximation method requires two initial guesses γ_0 and γ_1 and the general iteration formula is given by

$$\gamma_{n+1} = \gamma_n - f(\gamma_n) \frac{\gamma_n - \gamma_{n-1}}{f(\gamma_n) - f(\gamma_{n-1})}.$$

The iteration stops when the error between the iterate γ_n and the root γ is 10^{-9}, where the error estimate is given by $\gamma - \gamma_{n-1} \doteq \gamma_n - \gamma_{n-1}$.

The choice of the initial guesses will affect the speed of convergence. As $\gamma = 0$ is the trivial root, the initial guesses have to be different than 0 in order to obtain the non-trivial root γ. For example, if we consider a system with 3 sources ($M = 3$) and the following parameters

$a_1 = 0.5$	$d_1 = 0.02$	$N_1 = 10$	$r_1 = 1$
$a_2 = 0.6$	$d_2 = 0.04$	$N_2 = 5$	$r_2 = 1$
$a_3 = 0.7$	$d_3 = 0.02$	$N_3 = 7$	$r_3 = 1$

If one of the initial guess is 0 then the trivial root will be confirmed to be a root of $f(\gamma)$ at the third step (Table 3).

Table 3: Secant method for $f(\gamma) = 0$ with $\gamma_0 = 0$

n	γ_n	$f(\gamma_n)$	$\gamma_n - \gamma_{n-1}$
0	0.0	0.0	
1	1.0	4.500467	1.0
2	0.0	0.0	1.0
3	0.0	0.0	0.0

If one of the initial guess is close to 0 and the other one is different than 0 then the non-trivial root will be found (Table 4). The iterate γ_7 is the root γ rounded to 8 significant digits. The speed of convergence increases as the iterates become closer to γ.

Table 4: Secant method for $f(\gamma) = 0$ with $\gamma_0 \neq 0$

n	γ_n	$f(\gamma_n)$	$\gamma_n - \gamma_{n-1}$
0	0.1	3.910868×10^{-4}	
1	1.0	4.500467	0.9
2	9.992178×10^{-2}	3.809736×10^{-4}	9.000782×10^{-1}
3	9.984558×10^{-2}	3.711375×10^{-4}	7.619988×10^{-5}
4	9.697040×10^{-2}	1.184799×10^{-5}	2.875186×10^{-3}
5	9.687559×10^{-2}	3.917747×10^{-7}	9.481263×10^{-5}
6	9.687234×10^{-2}	4.408015×10^{-10}	3.242361×10^{-6}
7	9.687234×10^{-2}	1.684763×10^{-14}	3.652221×10^{-9}

The precision of the secant method can be verified graphically with *Mathematica*. For the present example, we obtained graphically (Figure 2) that the root is between 0.096 and 0.1, which agrees with our numerical value of γ.

As shown in Table 5, for various values of the initial guesses γ_0 and γ_1, the number of iterations needed to obtain γ fluctuates. For this example, the optimal case occurs when the root is between the initial guesses and the worst case occurs when both

initial guesses are greater than 1. But this is not always the case. What is optimal for one system may not be optimal for another and worst, it may give the trivial root, in which case no conclusions can be drawn.

Table 5: Secant method with γ_0 as the previous root

γ_0	γ_1	number of iterations (n)
0.258820	1.0	11
0.1	1.0	7
0.01*	0.2	9
0.5	1.0	13
0.09	0.1	5
1.0	2.0	15

* with these two initial guesses, the trivial root is found

To avoid obtaining the trivial root and to reduce the number of iterations, we will use the roots of a smaller system as γ_0 and γ_1. In the case of a one-source system ($M = 1$), 0.5 and 1.0 will be used for convenience, and for a two-source system ($M = 2$), the root of the one-source system and 1.0 will be used as γ_0 and γ_1, respectively. For any other system ($M = n$), γ_0 will be the root of the $(n-2)$-source system and γ_1 will be the root of the $(n-1)$-source system. Table 6 summarizes the results obtained for different systems, with parameters chosen such that the system is stable, that is, $\sum_{i=1}^{M} N_i \frac{d_i}{a_i+d_i} r_i < 1$

Table 6: Secant method for different systems

j	γ_0	γ_1	n	γ
1	0.5	1.0	6	0.492687
2	0.493	1.0	10	0.258820
3	0.259	0.493	10	0.009687
4	0.0969	0.259	9	0.054967
5	0.0550	0.0969	8	0.034387

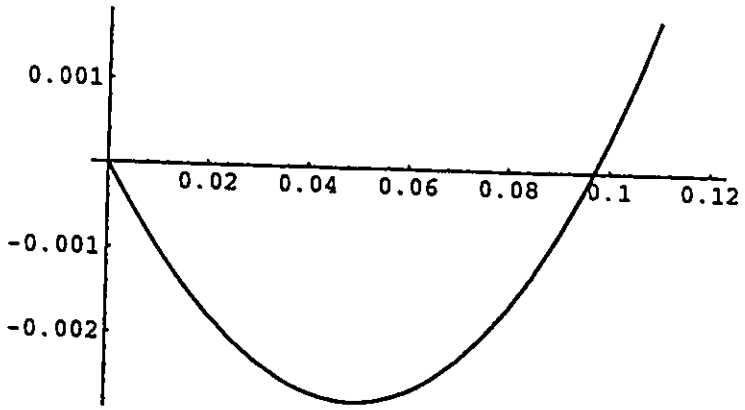

Figure 2: Graph of $f(\gamma) = 0$ between 0 and 0.1

Chapter 4

Application

4.1 Finite Buffer Analysis

A multiplexer receiving cells from multiple sources resolves contention by temporarily storing cells in a finite buffer. At the beginning of each time slot, one cell is removed from the buffer if the buffer is not empty. A cell is considered lost when the buffer exceeds length L. For such a model, we would like to know the mean number of Cells Lost per second (CL/sec). Such a parameter can be estimated by

$$\frac{\text{the number of cells lost in } T \text{ time slots}}{T \text{ time slots}} \sim CL/sec \text{ as } T \to \infty.$$

We would also like to estimate Pr(losing a cell) that is

$$\begin{aligned}
&\frac{\text{the number of cells lost in } T \text{ time slots}}{\text{the number of cells received in } T \text{ time slot}} \\
&= \left(\frac{\text{the number of cells received in } T \text{ time slots}}{T \text{ time slots}}\right)^{-1} \times \\
&\quad \left(\frac{\text{the number of cells lost in } T \text{ time slots}}{T \text{ time slots}}\right) \\
&\sim \left(\sum_{i=1}^{M} N_i \frac{d_i}{a_i + d_i} r_i\right)^{-1} \cdot CL/sec.
\end{aligned}$$

We define a cycle to be the number of time slots needed for a buffer to empty itself after at least one time slot. From ergodic theory (used in Heidelberger as the

basis for the A-cycle method [5]), we then have

$$CL/sec \sim \frac{\text{the mean number of cells lost per cycle}}{\text{the mean length of a cycle}} = \frac{\alpha}{\mu}.$$

The numerical value of these two parameters can now be determined. The mean length of a cycle (μ) is obtained by simulating the untwisted network started in steady state with an empty buffer. The average number of time slots needed to complete a cycle gives μ.

To do so, we will use the following simulation process. A nontwisted process is started where the M independent sources are in equilibrium. The process is run until a cycle is completed. Once the queue is empty, a twisted process is started where the state of each of the M independent sources is determined by the suspended nontwisted process. If the queue exceeds length L then the twisted process is changed to a nontwisted process. Again, the process is run until a cycle is completed. Every time a twisted cycle is completed, the nontwisted process is reactivated and continues for another cycle. When this cycle is completed, a twisted process is started.

To determine the mean number of cells lost per cycle (α), we need the following simulation model. Consider a Markov chain where $x_i = (s_i, q)$ denote the state at the beginning of the i^{th} time slot. Here s_i represents the state of each source and q represents the number of cells in the buffer. At the beginning of a time slot and with probability $K(x_i, x_{i+1})$, the chain will jump from state x_i to state x_{i+1} , that is, the cells received during the i^{th} time slot are stored in the buffer and at the $i+1^{th}$ time slot, one cell is removed. Let $x_0 = (s_0, 0)$ denote the initial state where the sources are in equilibrium and the queue has length 0. Let x_f denote the state where the length of the queue is close to but has not yet exceeded L. Let $y = (s_y, L + R(y))$ be the next (ficticious) state where the length of the queue exceeds L by a quantity $R(y)$. Since the buffer has a finite capacity L, a transition from x_f to y cannot occur. Instead, the size of the queue increases to L, the remaining cells are lost and then at the beginning of the next time slot, one cell is removed. The jump would be redirected to state $y^{\sim} = (s_{y^{\sim}}, L-1)$ with probability $K^{\sim}(x_f, y^{\sim}) = K(x_f, y)$ (Figure 3).

Let $V(x_0, \ldots, x_f, y^{\sim}, \ldots, x_\tau)$ denote the number of cells lost during this cycle

$(x_0, \ldots, x_f, y^{\sim}, \ldots, x_r)$ where $x_r = (s_r, 0)$. Then

$$\begin{aligned}
&\Pr(n \text{ cells are lost in one cycle}) \equiv P_n \\
&= \sum_{(x_0,\ldots,x_f,y^{\sim},\ldots,x_r)} \pi(x_0)K(x_0,x_1)K(x_1,x_2)\ldots K(x_{f-1},x_f)K^{\sim}(x_f,y^{\sim}) \times \\
&\quad K(y^{\sim},x_j)\ldots K(x_i,x_r)\chi\{V(x_0,\ldots,x_f,y^{\sim},\ldots,x_r)=n\}
\end{aligned}$$

where $x_i,\ x_j \in (x_0, \ldots, x_f, y^{\sim}, \ldots, x_r)$.

Since the event that the buffer reaches length L is a rare event, the cycles are very short and the buffer has length 0 very often. We would need to simulate the evolution of the system a large number, say N, of times to estimate P_n. The estimator of P_n is

$$\frac{1}{N}\sum_{k=1}^{N}\chi\{V(x_0(k),\ldots,x_f(k),y^{\sim}(k),\ldots,x_r(k))=n\}$$

where $(x_0(k), \ldots, x_f(k), y^{\sim}(k), \ldots, x_r(k))$ is the trajectory of the k^{th} simulation. Similarly the estimator for α is

$$\frac{1}{N}\sum_{k=1}^{N}V(x_0(k),\ldots,x_f(k),y^{\sim}(k),\ldots,x_r(k)).$$

This method is not efficient because it takes an extremely long time to simulate. Instead of using this approach, we will use a method involving the twist. Using the function r found in Chapter 2, define the harmonic function $h(x_i) = r(s_i)e^{\gamma q}$. Then P_n can be written as follows

$$\begin{aligned}
P_n &= \sum_{(x_0,\ldots,x_f,y^{\sim},\ldots,x_r)} \pi(x_0)K(x_0,x_1)\frac{h(x_1)}{h(x_0)}K(x_1,x_2)\frac{h(x_2)}{h(x_1)}\ldots K(x_{f-1},x_f)\frac{h(x_f)}{h(x_{f-1})}\times \\
&\quad K^{\sim}(x_f,y^{\sim})\frac{h(y)}{h(x_f)}\frac{h(x_0)}{h(y)}K(y^{\sim},x_j)\ldots K(x_i,x_r)\chi V(x_0,\ldots,x_f,y^{\sim},\ldots,x_r) \\
&= \sum_{(x_0,\ldots,x_f,y^{\sim},\ldots,x_r)} \pi(x_0)\mathcal{K}^{\infty}(x_0,x_1)\mathcal{K}^{\infty}(x_1,x_2)\ldots\mathcal{K}^{\infty}(x_{f-1},x_f)\mathcal{K}^{\sim}(x_f,y^{\sim}) \times \\
&\quad \frac{h(x_0)}{h(y)}K(y^{\sim},x_j)\ldots K(x_i,x_r)\chi V(x_0,\ldots,x_f,y^{\sim},\ldots,x_r)
\end{aligned}$$

since

$$\mathcal{K}^{\sim}(x_f,y^{\sim}) := K^{\sim}(x_f,y^{\sim})\frac{h(y)}{h(x_f)} = K(x_f,y)\frac{h(y)}{h(x_f)} = \mathcal{K}^{\infty}(x_f,y)$$

defines a probability transition kernel. Furthermore $h(y) = r(s_y)e^{\gamma(L+R(y))}$ and $h(x_0) = r(s_0)e^{\gamma \cdot 0} = r(s_0)$ so

$$P_n = e^{-\gamma L} \sum_{(x_0,\ldots,x_f,y^{\sim},\ldots,x_r)} \pi(x_0)\mathcal{K}^{\infty}(x_0,x_1)\mathcal{K}^{\infty}(x_1,x_2)\ldots\mathcal{K}^{\infty}(x_{f-1},x_f)\times$$
$$\mathcal{K}^{\sim}(x_f,y^{\sim})\frac{r(s_0)}{r(s_y)e^{\gamma R(y)}}K(y^{\sim},x_j)\ldots K(x_i,x_r)\chi V(x_0,\ldots,x_f,y^{\sim},\ldots,x_r).$$

We can now estimate P_n by simulation. To do so, first twist the parameters of the chain. Then notice that the factor $e^{-\gamma L}$ is known and does not influence the simulation. With a twisted process, the event that the buffer reaches length L is no longer a rare event. Every time this event occurs, $R(y)$ can be calculated by taking the difference between the length of the buffer at state y and L. Also, $r(s_0)/r(s_y) = \prod_{i=1}^{M} \beta_i^{n_{0i}-n_{yi}}$ can be calculated by monitoring the number of ON sources at the initial state and at the ficticious state, where cells are lost. Now when the queue tries to exceed length L, the parameters are untwisted. The simulation continues until the size of the buffer is 0. The number of cells lost in a cycle can then be counted.

The new estimator for P_n is therefore

$$e^{-\gamma L}\frac{1}{N}\sum_{k=1}^{N}\chi\left\{V(x_0(k),\ldots,x_f(k),y^{\sim}(k),\ldots,x_r(k))\right\}\frac{r(s_0)}{r(s_y)e^{\gamma R(y(k))}}$$

where $(x_0(k),\ldots,x_f(k),y^{\sim}(k),\ldots,x_r(k))$ is the path of the k^{th} twisted process. The new estimator for α is

$$e^{-\gamma L}\frac{1}{N}\sum_{k=1}^{N}V(x_0(k),\ldots,x_f(k),y^{\sim}(k),\ldots,x_r(k))\frac{r(s_0)}{r(s_y)e^{\gamma R(y(k))}} \equiv e^{-\gamma L}C_L.$$

This method enables us to obtain rapidly the estimator of P_n and α. It is important to note that P_n has a small value not because the event of losing n cells is rare but because the numerical value of $e^{-\gamma L}$ is small.

4.2 Leaky Bucket Analysis

As mentioned in Section 1.2, the leaky bucket controller can be viewed as a buffer with a finite capacity $L+I$. If, at a cell arrival, the content of the bucket is less than

or equal to L then the arriving cell is conforming and the content of the bucket is increased by I tokens. If, at a cell arrival, the content of the bucket is greater than L then the arriving cell is marked nonconforming. Notice in this case $y^{\sim}$ represents the state to which the jump is redirected, with a queue of maximum size $L+I-1$ (Figure 4). Every time slot, one token of content leaks out of the bucket if the bucket is not empty.

The performance of the controller is measured by the proportion of cells marked nonconforming and the mean time between bursts of nonconforming cells. Through the simulation of a twisted process, we now have the tools to approximate these parameters. The simulation of the leaky bucket is similar to the simulation of the finite buffer. A lost cell is now called a nonconforming cell.

In a specific communication simulation, the sources are described by some traffic characteristics. The standard traffic characteristics are PCR (peak cell rate), SCR (sustained cell rate), and MBS (maximum burst size).

As in Qian and McDonald [10], define the mean burst length in cells per basic time unit ($1/\mu_i$) and the mean silent period in cells per basic time unit ($1/\lambda_i$). We can obtain the parameters a_i and d_i as follows:

$$a_i = PCR_i/(LR \times MBS_i), \quad d_i = a_i \frac{PCR_i/LR}{1 - PCR_i/LR}$$

where LR represents the link rate.

We will now consider an example with two groups of sources. We assume the link rate is 620 Mb/s. The first group has 10 sources with a peak cell rate of 77.5 Mb/s, a sustained cell rate of 19.375 Mb/s and a maximum burst size of 40. The second group has 5 sources with a peak cell rate of 155.0 Mb/s, a sustained cell rate of 19.375 Mb/s and a maximum burst size of 40. The mean number of cells which arrive per time slot is 0.47 and the value of γ is 0.027038.

In Table 7, α/μ represents Pr[buffer content $> L$], σ represents the standard deviation of the buffer content and I represents the number of tokens by which the buffer content is increased.

Now, Pr[buffer contents $> L$] is estimated by $C_L e^{-\gamma L}$. In Table 7, we can see that as L increases, $e^{\gamma L} \cdot \left(C_L e^{-\gamma L}\right)$ converges (to a value close to 0.0011). There is thus

no need to simulate a process for L large. The constant multiplied by $e^{-\gamma L}$ yields a good approximation of Pr[buffer contents $> L$], as we can see for $L = 100$.

$$\Pr[\text{buffer contents } > 100] \simeq 0.0011e^{-0.027038 \cdot 100} = 7.3645 \times 10^{-5}$$

a difference of 4.1548×10^{-7} with the simulation.

4.3 Verification via an Infinite Buffer

In Sections 4.1 and 4.2, numerical results, obtained by simulation, were presented. In this section, we shall verify the algorithm by simulating a discrete Markov-modulated model for an infinite buffer capacity. We will compare the exact distribution of the buffer contents (which is given in [11]) with our simulated tail distribution of the buffer contents.

To do so, we will use the following simulation process. A nontwisted process is started where the M independent sources are in equilibrium. The process is run until a cycle is completed. Once the queue is empty, a twisted process is started where the state of each of the M independent sources is determined by the suspended nontwisted process. If the queue exceeds length L then the twisted process is changed to a nontwisted process. Again, the process is run until a cycle is completed. Every time a twisted cycle is completed, the nontwisted process is reactivated and continues for another cycle. After which, a twisted process is started.

The nontwisted process yields the average number of units of time needed to complete a cycle (μ) and the twisted process yields the number of units of time the buffer contents is greater than length L (α).

For the first example, if we consider a system with 2 sources ($M = 2$) and the following parameters

$$\begin{array}{llll} a_1 = 0.01 & d_1 = 0.03 & N_1 = 8 & r_1 = 0.05 \\ a_2 = 0.03 & d_2 = 0.01 & N_2 = 8 & r_2 = 0.15 \end{array}$$

where $\gamma = 0.231591$ can be determined by the secant method. For any given length L, we can estimate Pr[buffer contents $> L$] by α/μ, with an estimated standard

deviation of σ (Table 8). The exact distribution given by Xiong and Bruneel [11] always fall within our confidence interval except for $L < 10$. Xiong and Bruneel [11] obtained the exact distribution by an iterative algorithm, which may not be accurate near $L = 0$. It is important to note that the value of the standard deviation is large compared to α/μ. This is due to the fact that the event that the buffer reaches length L occurs approximately once every 50 cycles. Also, the cycles are not necessarily independent. The initial state for each cycle depends on the previous cycle. Our estimator for the standard deviation is based on the assumption of independent observations. We have underestimated the true standard deviation of our estimators.

For the second example, we will increase the probability of sending cells in the first source. The probabilities of exceeding L should then be larger. The parameters are now

$$\begin{array}{llll} a_1 = 0.2 & d_1 = 0.0077922 & N_1 = 8 & r_1 = 1.0 \\ a_2 = 0.03 & d_2 = 0.01 & N_2 = 8 & r_2 = 0.15 \end{array}$$

We can see in Table 9, $e^{\gamma L} \cdot (\alpha/\mu)$ converges to 0.2 and again, our results agree with Xiong and Bruneel's [11] results.

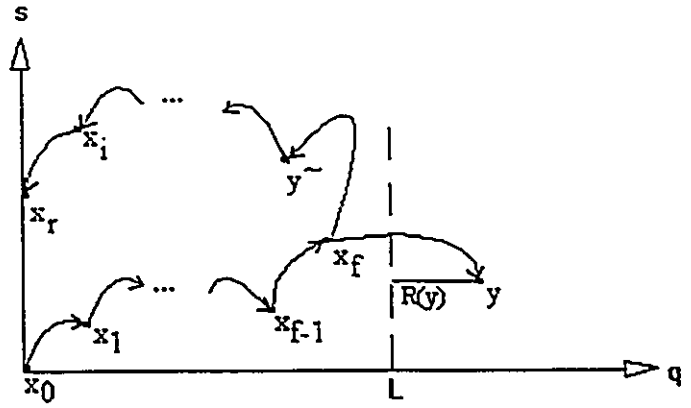

Figure 3: The simulation model

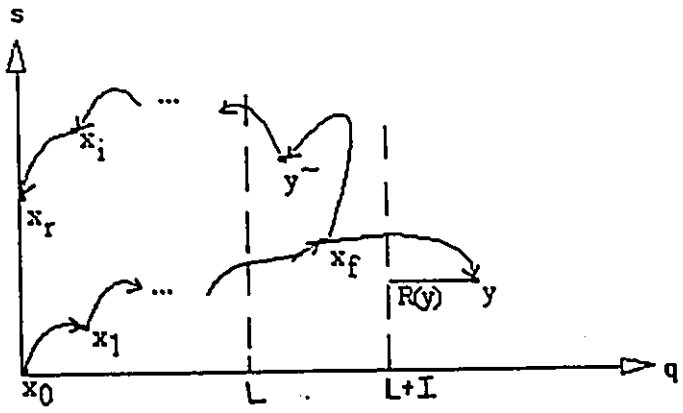

Figure 4: The leaky bucket model

Table 7: Simulation when $I = 1$

L	$e^{\gamma L} \cdot (\alpha/\mu)$	α/μ	σ
10	0.0039087	2.982753×10^{-3}	1.7649×10^{-4}
40	0.0016541	5.608938×10^{-4}	5.0988×10^{-5}
60	0.0014707	2.903930×10^{-4}	2.6490×10^{-5}
100	0.0011062	7.406116×10^{-5}	6.5839×10^{-6}

Table 8: $\gamma = 0.231591$

L	$e^{\gamma L} \cdot (\alpha/\mu)$	α/μ	σ
1	0.2046532	1.623454×10^{-1}	2.3325×10^{-3}
10	0.0955155	9.425124×10^{-3}	3.9836×10^{-4}
20	0.0709279	6.906274×10^{-4}	3.5402×10^{-5}
30	0.0670242	6.439779×10^{-5}	7.0403×10^{-6}
40	0.0511104	4.845757×10^{-6}	2.4098×10^{-7}
50	0.0523606	4.898586×10^{-7}	4.4847×10^{-8}
60	0.0455992	4.209554×10^{-8}	2.6204×10^{-9}
70	0.0517164	4.711079×10^{-9}	3.0414×10^{-10}
80	0.0425806	3.827510×10^{-10}	2.2544×10^{-11}
90	0.0482487	4.279607×10^{-11}	2.6172×10^{-12}
100	0.0443896	3.885187×10^{-12}	2.7472×10^{-13}

Table 9: $\gamma = 0.115256$

L	$e^{\gamma L} \cdot (\alpha/\mu)$	α/μ	σ
0	0.1681935	1.681935×10^{-1}	2.2429×10^{-3}
10	0.2375105	7.501229×10^{-2}	2.3548×10^{-3}
20	0.2123250	2.117875×10^{-2}	8.1999×10^{-4}
30	0.2124525	6.692842×10^{-3}	2.6301×10^{-4}
40	0.2005342	1.995201×10^{-3}	8.2967×10^{-5}
50	0.1992551	6.261196×10^{-4}	2.5088×10^{-5}
60	0.2079320	2.063568×10^{-4}	8.1302×10^{-6}

Chapter 5

Conclusions

The idea behind this thesis was to obtain an accurate value for the overflow probabilities for a leaky bucket controller which is driven by discrete Markov-modulated sources. For such a model, we have developed a change of measure and given its explicit analytic form. Using this change of measure, we may twist our recurrent process into a transient one. We can now easily simulate overflows of the leaky bucket controller and get an accurate value for the overflow probabilities.

It would be interesting, for future work, to run the simulations in parallel. The programs (Appendix B) can be easily modified. It would also be useful to use importance sampling on the starting distribution to increase the likelihood that the twisted process overloads. But even at this stage, our method considerably accelerates the simulation of the overflow probabilities for a leaky bucket controller.

Appendix A

Program to find γ

```
{\small}
-----------------------------------------------------------------------
This is the header file
-----------------------------------------------------------------------
#ifndef SecantH
#define SecantH

extern void GlobalInitialization();
extern double Definition(double gamma);
extern double Secant(double seed1, double seed2);

#endif

-----------------------------------------------------------------------
This is the program file
-----------------------------------------------------------------------
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "secant.h"
```

```
#define NumberOfTypes 2

/* global variables initialized in main */
int IncrementOfBuffer;
int N[NumberOfTypes];
double d[NumberOfTypes], a[NumberOfTypes], r[NumberOfTypes];


/* calculates the value of f(gamma) p.13 in thesis */
double Definition(double gamma)
{
  int i;
  double alpha,beta,sum=0.0;

  for(i=0;i<NumberOfTypes;i++)
    {
      alpha=1.-r[i]+r[i]*exp(gamma*IncrementOfBuffer);
      beta=(-(1.-d[i]-alpha+alpha*a[i])+
sqrt(pow(1.-d[i]-alpha+alpha*a[i], 2.0)
                        +4.*d[i]*a[i]*alpha))/(2.*d[i]);
      sum+=N[i]*log(1.-d[i]+d[i]*beta);
    }
  return(sum-gamma);
}

/* this is the general iteration */
double Secant(double seed1,double seed2)
{
  double f1,f2,gammatemp,gamma1,gamma2,error;
  int n=0;
```

```
/* initializes the values needed for the iteration */
gamma1=seed1;
gamma2=seed2;
f1=Definition(gamma1);
f2=Definition(gamma2);

/* ensures the value of the error is positive */
if (gamma2 > gamma1)
  error=gamma2-gamma1;
else
  error=gamma1-gamma2;

/* flags the iteration to stop when the error between
   the iterate and the root is 10^{-9} */
while (error > 1.e-9)
  {
    /* formula for the iteration */
    gammatemp = gamma2-((f2*(gamma2-gamma1))/(f2-f1));

    /* resets the parameters for the next iteration */
    f1=f2;
    gamma1=gamma2;
    gamma2=gammatemp;
    f2=Definition(gamma2);

    /* ensures the value of the error is positive */
    if (gamma2 > gamma1)
       error=gamma2-gamma1;
    else
```

```
        error=gamma1-gamma2;

      n++;
    } /* end of while loop */
  /* returns the root */
  return(gamma2);
} /* end of secant */


void main(int argc, char *argv[])
{
  double root, LinkRate;
  int i;

  /* reads in the parameters needed for the simulation */
  LinkRate = atof(argv[1]);

  for(i=0;i<NumberOfTypes;i++)
    {
      N[i] = atoi(argv[2+i*4]);
      r[i] = (atof(argv[4+i*4]))/(LinkRate);
      a[i]= r[i]/(atof(argv[5+i*4]));
      d[i] = a[i]*((atof(argv[3+i*4]))/(atof(argv[4+i*4])))/
             (1-((atof(argv[3+i*4]))/(atof(argv[4+i*4]))));
    }

  IncrementOfBuffer = atoi(argv[4+NumberOfTypes*4]);

  /* prints the parameters read */
  /* printf("NumberOfTypes is %d\n",NumberOfTypes); */
  for(i=0;i<NumberOfTypes;i++)
```

```
    {
      printf("N is %d, ",N[i]);
      printf("r is %f, ",r[i]);
      printf("a is %f, ",a[i]);
      printf("d is %f\n",d[i]);
    }

  /* calls the secant function with 2 initial guesses */
  root=Secant(atof(argv[2+NumberOfTypes*4]),
        atof(argv[3+NumberOfTypes*4]));

  printf("The root is %f\n",root);
}
```

Appendix B

Program to simulate a leaky bucket

```
-------------------------------------------------------------------
This is cell header file
-------------------------------------------------------------------
#ifndef CellH
#define CellH

#include <math.h>
#include <sims.h>

class cell: public sim_message
{
public:
  cell();
  ~cell();
  const char *type () const;
};

#endif
```

```
-------------------------------------------------------------------
This is cell code
-------------------------------------------------------------------
#include "cell.h"

cell::cell()
{
}
cell::~cell()
{
}

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

-------------------------------------------------------------------
This is bernoulli_entity header file
-------------------------------------------------------------------
#ifndef BernoulliEntityH
#define BernoulliEntityH

#include <math.h>
#include <sims.h>
#include "cell.h"
#include "leaky_bucket_entity.h"
```

```
class bernoulli_entity: public sim_entity
{
  double Mu,R,Lambda,Gamma,MuT,RT,LambdaT, Next_activation;
  sim_generator * random_gen;
  int NewBurst, NewBurstT, BurstLength;

  char Name[16];

public:
  int InitialOn, OverflowOn;
  double Beta;

  leaky_bucket_entity *Leaky_Ptr;
  bernoulli_entity();
  bernoulli_entity(const char *name,
                   leaky_bucket_entity *leaky_ptr,
    double scr, double pcr, double mbs,
                   double linkrate, double gamma);
  void initialize_normal();
  void initialize_twisted(double time);
  void execute();
  void twisted_cycle();
  void normal_cycle();
  void suspend_normal(double time);
  void suspend_twisted();
};

#endif
```

```
-------------------------------------------------------------------
This is bernoulli_entity code
-------------------------------------------------------------------
#include "bernoulli_entity.h"

bernoulli_entity::bernoulli_entity(const char *name,
                    leaky_bucket_entity *leaky_ptr,
                    double scr, double pcr, double mbs,
                    double linkrate, double gamma):
                    sim_entity(name)
{
  double rho,alpha;

  strcpy( Name, name );
  random_gen = new sim_generator();

  Leaky_Ptr = leaky_ptr;

  /* Calculates the untwisted parameters */
  R=pcr/linkrate;
  Mu = pcr/(linkrate*mbs);  /* a=mu */
  rho = scr/pcr;
  Lambda = Mu*rho/(1.-rho);  /* d=lambda */
  Gamma = gamma;

  /* Calculates the twisted parameters */
  alpha = 1.-R+R*exp(Gamma*IncrementOfBuffer);
  Beta = (-(1.-Lambda-alpha+alpha*Mu)+
          sqrt(pow(1.-Lambda-alpha+alpha*Mu, 2.0)
               +4.*Lambda*Mu*alpha))/(2.*Lambda);
```

```
  MuT = Mu/Beta/(1.-Mu+Mu/Beta);
  LambdaT = Lambda*Beta/(1.-Lambda+Lambda*Beta);
  RT = R*exp(Gamma*IncrementOfBuffer)/
         (1.-R+R*exp(Gamma*IncrementOfBuffer));
} /* end constructor */



void bernoulli_entity::initialize_normal()
{
  double rv_unfm;
  sim_bool rv_bern;

  /* Initializes private variables */
  NewBurst = 1;
  NewBurstT = 1;
  InitialOn = 0;
  OverflowOn = 0;

  /* Ensures the sources are activated in equilibrium */
  random_gen->draw(rv_bern,Lambda/(Lambda+Mu));
  if (rv_bern)
      activate_in(0.10);
  else
    {
      random_gen->uniform(rv_unfm, 0.0, 1.0);
      /* mean length of silence 1/lambdaTwisted */
      activate_in(log (1. - rv_unfm)*-1.*(1.0/LambdaT));
    } /* end else */
} /* end initialize_normal */
```

```
void bernoulli_entity::initialize_twisted(double time)
{
  /* Initializes private variables */
  NewBurst = 1;
  NewBurstT = 1;
  InitialOn = 0;
  OverflowOn = 0;

  activate_at(time);
} /* end initialize_twisted */

void bernoulli_entity::execute()
{
  while (TRUE)
    {
      switch (Leaky_Ptr->Event)
        {
        case 'T' :
          twisted_cycle();
          break;

        case 'N' :
          normal_cycle();
          break;

        default :
          printf(" BAD EVENT HANDLED\n");
          abort();
          break;
        } /* end switch */
    } /* end of while loop */
```

```
} /* end of execute */

void bernoulli_entity::twisted_cycle()
{
  int i;
  double rv_unfm;
  sim_bool rv_bern;
  cell *cell_ptr;
  if (NewBurstT == 1)
    {
      /* creates the burst period */
      random_gen->uniform(rv_unfm, 0.0, 1.0);
      /* mean length of a burst 1/muTwisted */
      BurstLength = (int)ceil(log (1.0 - rv_unfm)*-1.*(1./MuT));
      NewBurstT = 0;
    } /* enf if */

  /* for every unit of the burst period, a cell will be
     send with probability rTwisted */
  if (BurstLength > 0)
    {
      if (sim_clock() < (Leaky_Ptr->Time))
         InitialOn = 1;
      random_gen->draw(rv_bern, RT);
      if (rv_bern)
        {
          if (Leaky_Ptr->Queue->length() <= LengthOfBucket)
            {
              for(i=0;i<IncrementOfBuffer;i++)
                {
                  /* a cell is send to the leaky_bucket */
```

```
                cell_ptr = new cell();
                Leaky_Ptr->send_message( cell_ptr, "ME1" );
              }
            Leaky_Ptr->Conforming ++;
          }
        else
          Leaky_Ptr->Nonconforming ++;
       } /* end if rv_bern */
    BurstLength -= (int) Leaky_Ptr->TimeSlot;
    Next_activation = Leaky_Ptr->TimeSlot;
  } /* end if BurstLength */
 else
  {
    NewBurstT = 1;
    random_gen->uniform(rv_unfm, 0.0, 1.0);
    /* mean length of silence 1/lambdaTwisted */
    Next_activation = (log(1.-rv_unfm)*-1.*(1.0/LambdaT));
  } /* end else */
activate_in(Next_activation);
} /* end of twisted_cycle */

void bernoulli_entity::normal_cycle()
{
  int i;
  double rv_unfm;
  sim_bool rv_bern;
  cell *cell_ptr;

  if (NewBurst == 1)
    {
      /* creates the burst period */
```

```
    random_gen->uniform(rv_unfm, 0.0, 1.0);
    /* mean length of a burst 1/mu */
    BurstLength = (int)ceil(log(1.0 - rv_unfm)*-1.*(1./Mu));
    NewBurst = 0;
  } /* enf if */



/* for every unit of the burst period, a cell will be
   send with probability r */
if (BurstLength > 0)
  {
    if (sim_clock() < (Leaky_Ptr->Time))
      OverflowOn = 1;
    random_gen->draw(rv_bern, R);
    if (rv_bern)
      {
        if (Leaky_Ptr->Queue->length() <= LengthOfBucket)
          {
            for(i=0;i<IncrementOfBuffer;i++);;
              {
                /* a cell is send to the leaky_bucket */
                cell_ptr = new cell();
                Leaky_Ptr->send_message( cell_ptr, "ME1" );
              }
            Leaky_Ptr->Conforming ++;
          }
        else
          Leaky_Ptr->Nonconforming ++;
      } /* end if rv_bern */
    BurstLength -= (int) Leaky_Ptr->TimeSlot;
```

```
      Next_activation = Leaky_Ptr->TimeSlot;
    } /* end if BurstLength */
  else
    {
      NewBurst = 1;
      random_gen->uniform(rv_unfm, 0.0, 1.0);
      /* mean length of silence 1/lambda */
      Next_activation = (log (1. - rv_unfm)*-1.*(1.0/Lambda));
    } /* end else */
activate_in(Next_activation);
} /* end of normal_cycle */

void bernoulli_entity::suspend_normal(double time)
{
  activate_at(time + BatchLength);
}

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

-------------------------------------------------------------------
This is leaky_bucket_entity header file
-------------------------------------------------------------------
#ifndef Leaky_BucketEntityH
#define Leaky_BucketEntityH

#include <sims.h>
#include <math.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include "cell.h"
#include "program.h"

const char QueueName[] = { "ME1" };

class leaky_bucket_entity : public sim_entity
{
  int FirstOverflow, LastTime;
  char Name[40];

public:
  sim_message_queue *Queue;
  double Time, TimeSlot;
  int Nonconforming, Conforming, Jump, NumberOfTimeSlots;
  char Event;

  leaky_bucket_entity();
  leaky_bucket_entity(const char *name, double timeslot);
  void initialize_normal();
  void initialize_twisted(double time);
  void execute();
  void suspend_normal(double time);
  void suspend_twisted();
};
#endif

-------------------------------------------------------------------
This is leaky_bucket_entity code
-------------------------------------------------------------------
#include "leaky_bucket_entity.h"
```

```
leaky_bucket_entity::leaky_bucket_entity(const char *name,
double timeslot):sim_entity(name)
{
  TimeSlot = timeslot;
  create_message_queue( QueueName );
  Queue = find_queue( QueueName );

  strcpy( Name, name );
} /* end constructor */

void leaky_bucket_entity::initialize_normal()
{
  /* empties the queue */
  Queue->clear();
  Jump = 0;
  LastTime = 0;
  Event = 'N';
  FirstOverflow = 1;
  Conforming = 0;
  Nonconforming = 0;
  NumberOfTimeSlots = 0;
  Time = 0.0;
  activate_in(0.0);
} /* end intialize_normal */

void leaky_bucket_entity::initialize_twisted(double time)
{
  /* empties the queue */
  Queue->clear();
  Jump = 0;
```

```
  LastTime = 0;
  Event = 'T';
  FirstOverflow = 1;
  Conforming = 0;
  Nonconforming = 0;
  NumberOfTimeSlots = 0;
  Time = 0.0;
  activate_at(time);
} /* end initialize_twisted */

void leaky_bucket_entity::execute()
{
  cell *c;

  while (TRUE)
    {
      /* determines the number of ON bernoulli entities */
      if (NumberOfTimeSlots == 0)
        Time = sim_clock() + TimeSlot;

      /* calculates the number of unit of time needed to
         complete one cycle */
      NumberOfTimeSlots ++;

      /* if the buffer is greater than L then the leaky_bucket
         signals the bernoulli entities to untwist their
         parameters and calculates the number of times the
         buffer is greater than L*/
      if((Queue->length()>LengthOfBucket)&&(FirstOverflow==1))
        {
          Event = 'N';
```

```
            FirstOverflow = 0;
            Jump = Queue->length()-LengthOfBucket;
            Time = sim_clock() + TimeSlot;
          } /* end of if Queue->length() */


        /* if the buffer was not empty at the previous unit of
           time then one cell can be served */
        if (LastTime == 1)
          {
            c = (cell *)retrieve(NO_WAIT,IGNORE_ACTIVATES,Queue);
            delete(c);
          } /* end if */

        /* ensures a cell is removed if and only if the
           buffer is not empty */
        if (Queue->length() == 0)
          LastTime = 0;
        else
          LastTime = 1;

        /* flags the end of a cycle */
        if (Queue->length() == 0)
          sim_end_simulation();
        activate_in(TimeSlot);
      } /* end of while loop */
} /* end of execute */

void leaky_bucket_entity::suspend_normal(double time)
{
  activate_at(time + BatchLength);
```

```
} /* end of suspend_normal */

void leaky_bucket_entity::suspend_twisted()
{
  suspend();
} /* end of suspend_twisted */

-------------------------------------------------------------------
This is main header file
-------------------------------------------------------------------
#ifndef PROGH
#define PROGH

#define TRUE 1
#define FALSE 0

extern int NumberOfSources;
extern int NumberOfBatches;
extern int NumberOfTypes;
extern int LengthOfBucket;
extern int IncrementOfBuffer;
extern sim_time BatchLength;
extern sim_time Warmup;

#endif

-------------------------------------------------------------------
This is main code
-------------------------------------------------------------------
/* This program is designed to simulate multiple independent
   Bernoulli arrivals into an infinite leaky_bucket and give
```

```
   Prob[buffer > length] */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sims.h>
#include "cell.h"
#include "leaky_bucket_entity.h"
#include "bernoulli_entity.h"
#include "program.h"

int NumberOfSources;
int NumberOfBatches;
int NumberOfTypes;
int LengthOfBucket;
int IncrementOfBuffer;
sim_time BatchLength;
sim_time Warmup=0.0;

void Multi_set_parameters()
{
  NumberOfSources =
       atoi(sim_config()->get_parameter("number_of_sources"));
  LengthOfBucket =
       atoi(sim_config()->get_parameter("length_of_bucket"));
  NumberOfTypes =
       atoi(sim_config()->get_parameter("number_of_types"));
  IncrementOfBuffer =
       atoi(sim_config()->get_parameter("increment_of_buffer"));
}
```

```
class Multi_config_obj : public sim_config_obj
{
 public:
  Multi_config_obj()
    {
      add_parameter ("number_of_sources","0");
      add_parameter ("length_of_bucket","0");
      add_parameter ("number_of_types","0");
      add_parameter ("increment_of_buffer","1");
    }
};

main (int argc, char *argv[])
{
  int i, j=0, upto=0;
  double LinkRate, Gamma, Product;

  leaky_bucket_entity *Leaky_ptr_normal;
  leaky_bucket_entity *Leaky_ptr_twisted;
  char entityname[40];

  /* reads in the values of the global variables */
  sim_set_config (new Multi_config_obj());
  sim_config()->read_file (argv[1]);
  Multi_set_parameters();

  bernoulli_entity **Ber_ptr_normal=(bernoulli_entity**)malloc
    (sizeof (bernoulli_entity *) * NumberOfSources);
  bernoulli_entity **Ber_ptr_twisted=(bernoulli_entity**)malloc
    (sizeof (bernoulli_entity *) * NumberOfSources);
```

```
int *Num_p = (int *) malloc (sizeof (int) * (NumberOfTypes+1));

double *Scr_p =(double*)malloc(sizeof(double)*NumberOfTypes);
double *Pcr_p =(double*)malloc(sizeof(double)*NumberOfTypes);
double *Mbs_p =(double*)malloc(sizeof(double)*NumberOfTypes);

/* reads in the value of the parameters from the input file */
BatchLength = atof(argv[2]);
NumberOfBatches = atoi(argv[3]);
Gamma = atof(argv[4]);
LinkRate = atof(argv[5]);

Num_p[0] = 0;
for(i=0;i<NumberOfTypes;i++)
  {
    Num_p[i+1] = atoi(argv[6+i*4]);
    Scr_p[i] = atof(argv[7+i*4]);
    Pcr_p[i] = atof(argv[8+i*4]);
    Mbs_p[i] = atof(argv[9+i*4]);
  }

/* prints the parameters used in the simulation */
printf("NumberOfBatches is %d, ",NumberOfBatches);
printf("LengthOfBucket is %d, ",LengthOfBucket);
printf("Gamma is %f, ",Gamma);
printf("LinkRate is %f\n",LinkRate);
printf("Num_p[0] is %d\n",Num_p[0]);
for(i=0;i<NumberOfTypes;i++)
  {
    printf("Num_p[%d] is %d",i+1,Num_p[i+1]);
    printf(" Scr_p[%d] is %f",i,Scr_p[i]);
```

```
    printf(" Pcr_p[%d] is %f",i,Pcr_p[i]);
    printf(" Mbs_p[%d] is %f\n",i,Mbs_p[i]);
  }

/* constructs an infinite buffer which will receive cells
   every 1 unit of time */
sprintf(entityname, "%s", "LeakyN");
Leaky_ptr_normal = new leaky_bucket_entity(entityname, 1);
Leaky_ptr_normal->initialize_normal();

sprintf(entityname, "%s", "LeakyT");
Leaky_ptr_twisted = new leaky_bucket_entity(entityname, 1);

/* constructs the sources which will send cells */
while (j < NumberOfTypes)
  {
    upto+=Num_p[j+1];
    for(i=Num_p[j];i<upto;i++)
      {
        sprintf(entityname, "%s%02d", "BerN_", i);
        Ber_ptr_normal[i] = new bernoulli_entity(entityname,
                                  Leaky_ptr_normal,
                                  Scr_p[j],Pcr_p[j],Mbs_p[j],
                                  LinkRate,Gamma);
        Ber_ptr_normal[i]->initialize_normal();

        sprintf(entityname, "%s%02d", "BerT_", i);
        Ber_ptr_twisted[i] = new bernoulli_entity(entityname,
                                   Leaky_ptr_twisted,
                                   Scr_p[j],Pcr_p[j],Mbs_p[j],
                                   LinkRate,Gamma);
```

```
      } /* end of for loop */
    j++;
  } /* end of while < NumberOfTypes */

printf("\n Warmup, simulation begins \n");

j=0;
double Sum = 0.0;
double SumSquared = 0.0;
double Ber_diff = 0.0;
while (j<NumberOfBatches)
  {
    sim_run_simulation();

    Leaky_ptr_twisted->initialize_twisted
                            (Leaky_ptr_normal->next_time());
    Leaky_ptr_normal->suspend_normal
                            (Leaky_ptr_normal->next_time());

    for(i=0;i<NumberOfSources;i++)
      {
        Ber_ptr_twisted[i]->initialize_twisted
                                (Ber_ptr_normal[i]->next_time());
        Ber_ptr_normal[i]->suspend_normal
                                (Ber_ptr_normal[i]->next_time());
      } /* end for loop */

    sim_run_simulation();

    /* after a cycle, the parameters are reset to their original
       values and Product is recalculated */
```

```
    Product = 1.0;
    for(i=0;i<NumberOfSources;i++)
      {
        Ber_diff = (double)(Ber_ptr_twisted[i]->InitialOn -
                            Ber_ptr_twisted[i]->OverflowOn);
        Product *= pow( Ber_ptr_twisted[i]->Beta, Ber_diff );
        Ber_ptr_twisted[i]->suspend_twisted();
      } /* end of for loop */

    /* updates Sum and SumSquared */
    Sum +=(Product*exp(-1.0*Gamma*Leaky_ptr_twisted->Jump)*
    (Leaky_ptr_twisted->Nonconforming * 1.));
    SumSquared +=(pow(exp(-1.0*Gamma*LengthOfBucket),2.0)*
         pow(Product* xp(-1.0*Gamma*Leaky_ptr_twisted->Jump)*
    (Leaky_ptr_twisted->Nonconforming*1 ), 2.0));

    /* reinitializes the parameters used in the simulation */
    Leaky_ptr_twisted->suspend_twisted();
    j++;
  } /* end of while loop */

double alpha, deviation, mu;

alpha = exp(-1.0*Gamma*LengthOfBucket)*
              Sum/(NumberOfBatches*1.0);
deviation = sqrt(
 SumSquared/(NumberOfBatches*1.0) -
 pow(exp(-1.0*Gamma*LengthOfBucket)*
                      Sum/(NumberOfBatches*1.0),2.0))
  /sqrt(NumberOfBatches*1.0);
mu = (Leaky_ptr_normal->NumberOfTimeSlots * 1.)/
```

```
            (NumberOfBatches*1.);

  /* prints the numerical results */
  printf(" Estimate is %f, alpha is %e\n",
           Sum/(NumberOfBatches*1.0), alpha);
  printf(" Deviation of alpha is %e\n", deviation);

  printf(" mu is %f\n",mu);

  printf(" alpha/mu is %e\n",alpha/mu);
  printf(" Deviation of alpha/mu is %e\n",deviation/mu);

  printf(" Exp(gamma*L)*alpha/mu is %f\n",
           exp(Gamma*LengthOfBucket)*alpha/mu);

  printf("\n Simulation done\n");
} // end of main()
```

```
-----------------------------------------------------------------
This is the configuration file
-----------------------------------------------------------------
sim_progress = 0
sim_stack_size = 20480
sim_trace_level = 0
sim_trace_start = 0.0

number_of_sources = 15
length_of_bucket = 10
number_of_types = 2
increment_of_buffer = 1
```

Appendix C

Program to simulate an infinite multiplexer

```
-------------------------------------------------------------------
This is cell header file
-------------------------------------------------------------------
#ifndef CellH
#define CellH

#include <math.h>
#include <sims.h>

class cell: public sim_message
{
public:
  cell();
  ~cell();
  const char *type () const;
};

#endif
```

```
//-------------------------------------------------------------------
// This is cell code
//-------------------------------------------------------------------
#include "cell.h"

cell::cell()
{
}
cell::~cell()
{
}

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

//-------------------------------------------------------------------
// This is bernoulli_entity header file
//-------------------------------------------------------------------
#include "bernoulli_entity.h"

bernoulli_entity::bernoulli_entity(const char *name,
     multiplexer_entity *multi_ptr,
     double r, double mu, double lambda,
     double linkrate, double gamma):sim_entity(name)
{
  double alpha;

  strcpy( Name, name );
```

```
  random_gen = new sim_generator();

  Multi_Ptr = multi_ptr;

  /* Calculates the untwisted parameters */
  R=r;
  Mu = mu;  /* a=mu */
  Lambda = lambda;  /* d=lambda */
  Gamma = gamma;

  /* Calculates the twisted parameters */
  alpha = 1.-R+R*exp(Gamma);
  Beta = (-(1.-Lambda-alpha+alpha*Mu)+
              sqrt(pow(1.-Lambda-alpha+alpha*Mu, 2.0)
          +4.*Lambda*Mu*alpha))/(2.*Lambda);
  MuT = Mu/Beta/(1.-Mu+Mu/Beta);
  LambdaT = Lambda*Beta/(1.-Lambda+Lambda*Beta);
  RT = R*exp(Gamma)/(1.-R+R*exp(Gamma));
} /* end constructor */

void bernoulli_entity::initialize_normal()
{
  double rv_unfm;
  sim_bool rv_bern;

  /* Initializes private variables */
  NewBurst = 1;
  NewBurstT = 1;
  InitialOn = 0;
  OverflowOn = 0;
```

```
  /* Ensures the sources are activated in equilibrium */
  random_gen->draw(rv_bern,Lambda/(Lambda+Mu));
  if (rv_bern)
      activate_in(0.10);
  else
    {
      random_gen->uniform(rv_unfm, 0.0, 1.0);
      /* mean length of silence 1/lambdaTwisted */
      activate_in(log (1. - rv_unfm)*-1.*(1.0/LambdaT));
    } /* end else */
} /* end initialize_normal */

void bernoulli_entity::initialize_twisted(double time)
{
  /* Initializes private variables */
  NewBurst = 1;
  NewBurstT = 1;
  InitialOn = 0;
  OverflowOn = 0;
  activate_at(time);
} /* end initialize_twisted */

void bernoulli_entity::execute()
{
  while (TRUE)
    {
      switch (Multi_Ptr->Event)
        {
        case 'T' :
          twisted_cycle();
```

```
          break;

        case 'N' :
          normal_cycle();
          break;

        default :
          printf(" BAD EVENT HANDLED\n");
          abort();
          break;
        } /* end switch */
    } /* end of while loop */
} /* end of execute */

void bernoulli_entity::twisted_cycle()
{
  double rv_unfm;
  sim_bool rv_bern;
  cell *cell_ptr;

  if (NewBurstT == 1)
    {
      /* creates the burst period */
      random_gen->uniform(rv_unfm, 0.0, 1.0);
      /* mean length of a burst 1/muTwisted */
      BurstLength =(int)ceil(log(1.0 - rv_unfm)*-1.*(1./MuT));
      NewBurstT = 0;
    } /* enf if */
```

```
  /* for every unit of the burst period, a cell will be
     send with probability rTwisted */
  if (BurstLength > 0)
    {
      if (sim_clock() < (Multi_Ptr->Time))
         InitialOn = 1;
      random_gen->draw(rv_bern, RT);
      if (rv_bern)
        {
          /* a cell is send to the multiplexer */
          cell_ptr = new cell();
          Multi_Ptr->send_message( cell_ptr, "ME1" );
        } /* end if rv_bern */
      BurstLength -= (int) Multi_Ptr->TimeSlot;
      Next_activation = Multi_Ptr->TimeSlot;
    } /* end if BurstLength */
  else
    {
      NewBurstT = 1;
      random_gen->uniform(rv_unfm, 0.0, 1.0);
      /* mean length of silence 1/lambdaTwisted */
      Next_activation = (log(1. - rv_unfm)*-1.*(1.0/LambdaT));
    } /* end else */
activate_in(Next_activation);
} /* end of twisted_cycle */

void bernoulli_entity::normal_cycle()
{
  double rv_unfm;
  sim_bool rv_bern;
```

```
cell *cell_ptr;

if (NewBurst == 1)
  {
    /* creates the burst period */
    random_gen->uniform(rv_unfm, 0.0, 1.0);
    /* mean length of a burst 1/mu */
    BurstLength = (int) ceil (log (1.0 - rv_unfm)*-1.*(1./Mu));
    NewBurst = 0;
  } /* enf if */

/* for every unit of the burst period, a cell will be
   send with probability r */
if (BurstLength > 0)
  {
    if (sim_clock() < (Multi_Ptr->Time))
       OverflowOn = 1;
    random_gen->draw(rv_bern, R);
    if (rv_bern)
      {
        /* a cell is send to the multiplexer */
        cell_ptr = new cell();
        Multi_Ptr->send_message( cell_ptr, "ME1" );
      } /* end if rv_bern */
    BurstLength -= (int) Multi_Ptr->TimeSlot;
    Next_activation = Multi_Ptr->TimeSlot;
  } /* end if BurstLength */
else
  {
    NewBurst = 1;
    random_gen->uniform(rv_unfm, 0.0, 1.0);
```

```
      /* mean length of silence 1/lambda */
      Next_activation = (log (1. - rv_unfm)*-1.*(1.0/Lambda));
    } /* end else */
activate_in(Next_activation);
} /* end of normal_cycle */

void bernoulli_entity::suspend_normal(double time)
{
  activate_at(time + BatchLength);
}

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

-------------------------------------------------------------------
This is multiplexer_entity header file
-------------------------------------------------------------------
#ifndef MultiplexerEntityH
#define MultiplexerEntityH

#include <sims.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "cell.h"
#include "program.h"

const char QueueName[] = { "ME1" };
```

```
class multiplexer_entity : public sim_entity
{
  int FirstOverflow, LastTime;
  char Name[40];

public:
  sim_message_queue *Queue;
  double Time, TimeSlot;
  int NumberOfOverflow, Jump, NumberOfTimeSlots;
  char Event;

  multiplexer_entity();
  multiplexer_entity(const char *name, double timeslot);
  void initialize_normal();
  void initialize_twisted(double time);
  void execute();
  void suspend_normal(double time);
  void suspend_twisted();
};
#endif

-------------------------------------------------------------------
This is multiplexer_entity code
-------------------------------------------------------------------
#include "multiplexer_entity.h"

multiplexer_entity::multiplexer_entity(const char *name,
double timeslot):sim_entity(name)
{
  TimeSlot = timeslot;
```

```
    create_message_queue( QueueName );
    Queue = find_queue( QueueName );
    strcpy( Name, name );
} /* end constructor */

void multiplexer_entity::initialize_normal()
{
    /* empties the queue */
    Queue->clear();
    Jump = 0;
    LastTime = 0;
    Event = 'N';
    FirstOverflow = 1;
    NumberOfOverflow = 0;
    NumberOfTimeSlots = 0;
    Time = 0.0;
    activate_in(0.0);
} /* end intialize_normal */

void multiplexer_entity::initialize_twisted(double time)
{
    /* empties the queue */
    Queue->clear();
    Jump = 0;
    LastTime = 0;
    Event = 'T';
    FirstOverflow = 1;
    NumberOfOverflow = 0;
    NumberOfTimeSlots = 0;
    Time = 0.0;
    activate_at(time);
```

```
} /* end initialize_twisted */

void multiplexer_entity::execute()
{
  cell *c;

  while (TRUE)
    {
      /* determines the number of ON bernoulli entities */
      if (NumberOfTimeSlots == 0)
         Time = sim_clock() + TimeSlot;

      /* calculates the number of unit of time needed to
         complete one cycle */
      NumberOfTimeSlots ++;

      /* if the buffer is greater than L then the multiplexer
         signals the bernoulli entities to untwist their
         parameters and calculates the number of times
         the buffer is greater than L*/
      if (Queue->length() > LengthOfBucket)
        {
          NumberOfOverflow++;
          if (FirstOverflow == 1)
            {
              Event = 'N';
              FirstOverflow = 0;
              Jump = Queue->length()-LengthOfBucket;
              Time = sim_clock() + TimeSlot;
            } /* end of if FirstOverflow */
        } /* end of if Queue->length() */
```

```
      /* if the buffer was not empty at the previous unit of
         time then one cell can be served */
      if (LastTime == 1)
        {
          c = (cell *)retrieve(NO_WAIT,IGNORE_ACTIVATES,Queue);
          delete(c);
        } /* end if */

      /* ensures a cell is removed if and only if the
          buffer is not empty */
      if (Queue->length() == 0)
        LastTime = 0;
      else
        LastTime = 1;

      /* flags the end of a cycle */
      if (Queue->length() == 0)
        sim_end_simulation();
      activate_in(TimeSlot);
    } /* end of while loop */
} /* end of execute */

void multiplexer_entity::suspend_normal(double time)
{
  activate_at(time + BatchLength);
} /* end of suspend_normal */
```

```
void multiplexer_entity::suspend_twisted()
{
  suspend();
} /* end of suspend_twisted */

-------------------------------------------------------------------
This is main header file
-------------------------------------------------------------------
#ifndef PROGH
#define PROGH

#define TRUE 1
#define FALSE 0

extern int NumberOfSources;
extern int NumberOfBatches;
extern int NumberOfTypes;
extern int LengthOfBucket;

extern sim_time BatchLength;
extern sim_time Warmup;

#endif

-------------------------------------------------------------------
This is main code
-------------------------------------------------------------------
/* Simulation based on Xiong and Bruneel work, 1995 */
/* This program is designed to simulate multiple independent
   Bernoulli arrivals into an infinite multiplexer and give
```

```
   Prob[buffer > length] */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sims.h>
#include "cell.h"
#include "multiplexer_entity.h"
#include "bernoulli_entity.h"
#include "program.h"

int NumberOfSources;
int NumberOfBatches;
int NumberOfTypes;
int LengthOfBucket;

sim_time BatchLength;
sim_time Warmup=0.0;

void Multi_set_parameters()
{
  NumberOfSources =
     atoi(sim_config()->get_parameter("number_of_sources"));
  LengthOfBucket =
     atoi(sim_config()->get_parameter("length_of_bucket"));
  NumberOfTypes =
     atoi(sim_config()->get_parameter("number_of_types"));
}

class Multi_config_obj : public sim_config_obj
{
```

```
 public:
  Multi_config_obj()
    {
      add_parameter ("number_of_sources","0");
      add_parameter ("length_of_bucket","0");
      add_parameter ("number_of_types","0");
    }
};


main (int argc, char *argv[])
{
  int i, j=0, upto=0;
  double LinkRate, Gamma, Product;

  multiplexer_entity *Multi_ptr_normal;
  multiplexer_entity *Multi_ptr_twisted;
  char entityname[40];

  /* reads in the values of the global variables */
  sim_set_config (new Multi_config_obj());
  sim_config()->read_file (argv[1]);
  Multi_set_parameters();

  bernoulli_entity **Ber_ptr_normal=(bernoulli_entity**)malloc
    (sizeof (bernoulli_entity *) * NumberOfSources);
  bernoulli_entity **Ber_ptr_twisted=(bernoulli_entity**)malloc
    (sizeof (bernoulli_entity *) * NumberOfSources);

  int *Num_p = (int *) malloc (sizeof (int) * (NumberOfTypes+1));
```

```
double *R_p =(double*)malloc(sizeof(double)*NumberOfTypes);
double *Mu_p =(double*)malloc(sizeof(double)*NumberOfTypes);
double *Lambda_p=(double*)malloc(sizeof(double)*NumberOfTypes);


/* reads in the value of the parameters from the input file */
BatchLength = atof(argv[2]);
NumberOfBatches = atoi(argv[3]);
Gamma = atof(argv[4]);
LinkRate = atof(argv[5]);

Num_p[0] = 0;
for(i=0;i<NumberOfTypes;i++)
  {
    Num_p[i+1] = atoi(argv[6+i*4]);
    R_p[i] = atof(argv[7+i*4]);
    Mu_p[i] = atof(argv[8+i*4]);
    Lambda_p[i] = atof(argv[9+i*4]);
  }

/* prints the parameters used in the simulation */
printf("NumberOfBatches is %d, ",NumberOfBatches);
printf("LengthOfBucket is %d, ",LengthOfBucket);
printf("Gamma is %f, ",Gamma);
printf("LinkRate is %f\n",LinkRate);
printf("Num_p[0] is %d\n",Num_p[0]);
for(i=0;i<NumberOfTypes;i++)
  {
    printf("Num_p[%d] is %d",i+1,Num_p[i+1]);
    printf(" R_p[%d] is %f",i,R_p[i]);
    printf(" Mu_p[%d] is %f",i,Mu_p[i]);
```

```
    printf(" Lambda_p[%d] is %f\n",i,Lambda_p[i]);
  }



/* constructs an infinite buffer which will receive cells
   every 1 unit of time */
sprintf(entityname, "%s", "MultiN");
Multi_ptr_normal = new multiplexer_entity(entityname, 1);
Multi_ptr_normal->initialize_normal();

sprintf(entityname, "%s", "MultiT");
Multi_ptr_twisted = new multiplexer_entity(entityname, 1);

/* constructs the sources which will send cells */
while (j < NumberOfTypes)
  {
    upto+=Num_p[j+1];
    for(i=Num_p[j];i<upto;i++)
      {
        sprintf(entityname, "%s%02d", "BerN_", i);
        Ber_ptr_normal[i] = new bernoulli_entity(entityname,
                                  Multi_ptr_normal,
                                  R_p[j],Mu_p[j],Lambda_p[j],
                                  LinkRate,Gamma);
        Ber_ptr_normal[i]->initialize_normal();

        sprintf(entityname, "%s%02d", "BerT_", i);
        Ber_ptr_twisted[i] = new bernoulli_entity(entityname,
                                   Multi_ptr_twisted,
                                   R_p[j],Mu_p[j],Lambda_p[j],
```

```
                                    LinkRate,Gamma);
      } /* end of for loop */
    j++;
  } /* end of while < NumberOfTypes */

printf("\n Warmup, simulation begins \n");

j=0;
double Sum = 0.0;
double SumSquared = 0.0;
double Ber_diff = 0.0;
while (j<NumberOfBatches)
  {
    sim_run_simulation();

    Multi_ptr_twisted->initialize_twisted
                          (Multi_ptr_normal->next_time());
    Multi_ptr_normal->suspend_normal
                          (Multi_ptr_normal->next_time());

    for(i=0;i<NumberOfSources;i++)
      {
        Ber_ptr_twisted[i]->initialize_twisted
                               (Ber_ptr_normal[i]->next_time());
        Ber_ptr_normal[i]->suspend_normal
                               (Ber_ptr_normal[i]->next_time());
      } /* end for loop */

    sim_run_simulation();

    /* after a cycle, the parameters are reset to their original
```

```
        values and Product is recalculated */
      Product = 1.0;
      for(i=0;i<NumberOfSources;i++)
        {
          Ber_diff = (double)(Ber_ptr_twisted[i]->InitialOn -
                      Ber_ptr_twisted[i]->OverflowOn);
          Product *= pow( Ber_ptr_twisted[i]->Beta, Ber_diff );
          Ber_ptr_twisted[i]->suspend_twisted();
        } /* end of for loop */

      /* updates Sum and SumSquared */
      Sum +=(Product*exp(-1.0*Gamma*Multi_ptr_twisted->Jump)*
      (Multi_ptr_twisted->NumberOfOverflow * 1.));
      SumSquared +=(pow(exp(-1.0*Gamma*LengthOfBucket),2.0)*
            pow(Product*exp(-1.0*Gamma*Multi_ptr_twisted->Jump)*
    (Multi_ptr_twisted->NumberOfOverflow*1.), 2.0));

      /* reinitializes the parameters used in the simulation */
      Multi_ptr_twisted->suspend_twisted();
      j++;
    } /* end of while loop */

  /* prints the numerical results */
  printf(" Estimate is %f, alpha is %e\n",Sum/(NumberOfBatches*1.0),
            exp(-1.0*Gamma*LengthOfBucket)*Sum/(NumberOfBatches*1.0));
  printf(" Deviation is %e\n",
         sqrt(SumSquared/(NumberOfBatches*1.0) -
            pow(exp(-1.0*Gamma*LengthOfBucket)*Sum/
                      (NumberOfBatches*1.0),2.0))
         /sqrt(NumberOfBatches*1.0));
```

```
  printf("Average cycle length is %f\n",
         (Multi_ptr_normal->NumberOfTimeSlots * 1.)/
                    (NumberOfBatches*1.));

  printf("alpha/mu is %e\n",(exp(-1.0*Gamma*LengthOfBucket)*
         Sum/(NumberOfBatches*1.0))/
         ((Multi_ptr_normal->NumberOfTimeSlots * 1.)/
         (NumberOfBatches*1.)));

  printf("\n Simulation done\n");
} // end of main()
```

```
-------------------------------------------------------------------
This is the configuration file
-------------------------------------------------------------------
sim_progress = 0
sim_stack_size = 20480
sim_trace_level = 0
sim_trace_start = 0.0

number_of_sources = 16
length_of_bucket = 50
number_of_types = 2
```

Bibliography

[1] Anick, D., Mitra, D., and Sondhi, M.M. (1982), Stochastic Theory of a Data-Handling System with Multiple Sources *BSTJ61*, 1871-1894.

[2] Atkinson, K. (1993), Elementary numerical analysis (2^{nd}ed), *John Wiley & Sons, Inc*, 78-82.

[3] Buffet, E. and Duffield, N.G. (1992), Exponential Upper Bounds via Martingales for Multiplexers with Markovian Arrivals, *DIAS-STP-92-16.*

[4] Duffield, N.G. (1993), Exponential Bounds for Markovian Queues, *DIAS-STP-93-01.*

[5] Heidelberger, P. (1995). Fast Simulation of Rare Events in Queueing and Reliability Models, *ACM Transactions on Modeling and Computer Simulation, Vol. 5, No. 1*, 43-85.

[6] Kosten, L. (1974). Stochastic theory of a multi-entry buffer (I), *Delft Progress Report*, 10-18.

[7] McDonald, D.R. (1995), Asymptotics of first passage times for random walk in a quadrant, *(manuscript).*

[8] McDonald, D.R. (1995), Asymptotics of first passage times for random walk in a quadrant II, *(manuscript).*

[9] Ney and Nummelin. (1987), Markov Additive Processes I. Eigenvalue Properties and Limit Theorems, *Ann. Probab. 15*, 561-592.

[10] Qian, K. and McDonald, D.R. (1995), An Approximation Method for Complete Solutions of Markov-Modulated Fluid Models, *(manuscript)*.

[11] Xiong, Y. and Bruneel, H. (1995), A simple approach to obtain tight upper bounds for the asymptotic queueing behavior of statistical multiplexers with heterogeneous traffic, *Performance Evaluation 22*, 159-173.