Université d'Ottawa · University of Ottawa

PERMISSION DE REPRODUIRE ET DE DISTRIBUER LA THÈSE

PERMISSION TO REPRODUCE AND DISTRIBUTE THE THESIS

NOM DE L'AUTEUR / NAME OF AUTHOR:	Hong YAN
ADRESSE POSTALE / MAILING ADDRESS:	1102 - 52 Bayswater Avenue Ottawa ON K1Y 4K3
GRADE / DEGREE: M. Sc. (Systems Science)	**ANNÉE D'OBTENTION / YEAR GRANTED** 2003
TITRE DE LA THÈSE / TITLE OF THESIS:	Performance Evaluation of Access Mechanism Corresponding to Stochastic Inputs

L'auteur permet, par la présente, la consultation et le prêt de cette thèse en conformité avec les règlements établis par le bibliothécaire en chef de l'Université d'Ottawa. L'auteur autorise aussi l'Université d'Ottawa, ses successeurs et cessionnaires, à reproduire cet exemplaire par photograhie ou photocopie pour fins de prêt ou de vente au prix coûtant aux bibliothèques ou aux chercheurs qui en feront la demande.

Les droits de publication par tout autre moyen et pour vente au public demeureront la propriété de l'auteur de la thèse sous réserve des règlements de l'Université d'Ottawa en matière de publication de thèses.

The author hereby permits the consultation and the lending of this thesis pursuant to the regulations established by the Chief Librarian of the University of Ottawa. The author also authorizes the University of Ottawa, its successors and assignees, to make reproductions of this copy by photographic means or by photocopying and to lend or sell such reproductions at cost to libraries and to scholars requesting them.

The right to publish the thesis by other means and to sell it to the public is reserved to the author, subject to the regulations of the University of Ottawa governing the publication of theses.

N.B. LE MASCULIN COMPREND ÉGALEMENT LE FÉMININ

Apr. 9. 2003

DATE

(AUTEUR) SIGNATURE (AUTHOR)

Université d'Ottawa · University of Ottawa

Université d'Ottawa · University of Ottawa

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

YAN, Hong

AUTEUR DE LA THÈSE - AUTHOR OF THESIS

M.Sc. (Systems Science)

GRADE - DEGREE

Faculty of Graduate and Postdoctoral Studies

FACULTÉ, ÉCOLE, DÉPARTEMENT - FACULTY, SCHOOL, DEPARTMENT

TITRE DE LA THÈSE - TITLE OF THE THESIS

Performance Evaluation of Access Control Mechanism Corresponding to Stochastic Inputs

Nasir Uddin Ahmed

DIRECTEUR DE LA THÈSE - THESIS SUPERVISOR

EXAMINATEURS DE LA THÈSE - THESIS EXAMINERS

T. Yeap

O. Yang

J.-M. De Koninck, Ph.D.

LE DOYEN DE LA FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

SIGNATURE

DEAN OF THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

Performance Evaluation of Access Control Mechanism Corresponding to Stochastic Inputs

A thesis submitted to the Faculty of Graduate and Postdoctoral Studies
in partial fulfillment of the degree of
Master of Science

Hong Yan
Systems Science,
University of Ottawa,
Ottawa, Ontario, Canada K1N 6N5

January 2003

©Hong Yan University of Ottawa, 2003

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.

0-612-79387-7

Canada

ACKNOWLEDGEMENTS

I would like to express my appreciation to my supervisors, Dr. NasirUddin Ahmed and Dr. Luis Orozco-Barbosa, for their support, understanding and guidance during my completion of the thesis.

I would also like to thank my colleagues Dr. Xinhou Hua, Ms. Bo Li, Mr. Cheng Li and Mr. Qun Wang for their valuable discussions and suggestions.

Dedication

In memory of my father. To my mother and my husband Jiangyong, for their love, encouragements and support.

Abstract

In this thesis, we construct traffic models, which exhibit both short-range dependence (SRD) and long-range dependence (LRD) using Poisson process and doubly stochastic Poisson (or Cox) process (DSPP) driven by fractional Brownian motion (FBM). We also develop a novel dynamic system model for the token bucket (TB) control algorithm used in computer networks. In this model, token buckets police incoming traffic, and one multiplexor serving all the token pools multiplexes conforming traffic using round robin scheme. The state of the system is formally defined and control strategies are also proposed. We study several issues related to performance corresponding to different stochastic inputs using the proposed model located at the edge of the backbone network. The numerical results demonstrate that this system can be adapted to any kind of stochastic traffic. The results can be served as a tool for the designers of such controllers to set up different system parameters.

Contents

List of Figures

List of Tables

Acronyms

BM	Brownian Motion
FBM	Fractional Brownian Motion
FGN	Fractional Gaussian Noise
FSN	Fractal Shot Noise
IAT	Inter-arrival Time
Imix	Internet mix
LRD	Long Range Dependence
MSS	Maximum Segment Size
MTU	Maximum Transmission Unit
MPEG	Moving Picture Experts Group
QoS	Quality of Service
SRD	Short Range Dependence
TB	Token Bucket
VBR	Variable Bit Rate
HPP	Homogeneous Poisson Process
NHPP	Nonhomogeneous Poisson Process
DSPP	Doubly Stochastic Poisson Process

Symbols

$A_i(t_k)$	Multiplexor true allocation to i-th user at time t_k
C	Network link rate
$e(t_k)$	End point of accepting the last conforming traffic by multiplexor at $[t_k, t_{k+1})$
e_i	Arbitration factor to the i-th user by the network provider
$g_i(t_k)$	Sum of all the conforming traffic up to ith user at time t_k of the time slot $[t_k, t_{k+1})$
$G_i(t_k)$	Conforming traffic of the i-th user at time t_k
h	Length of time interval (slot)$[t_k, t_{k+1})$
I_k	time interval (slot)$[t_k, t_{k+1})$
J	Total cost
J_M	Average value of total cost by using M samples
K	Total number of time interval (slot)
$l(t_k)$	Largest number of users served up to time t_k
$L(t_k)$	Losses at multiplexor at time t_k
M	The number of sample paths used in Monte Carlo method
M_o	The number of samples such that the average value converges to the expected value
$N_c(r)$	Number of times in the state of congestion during $[t_0, t_r)$
$q(t_k)$	Multiplexor buffer state at time t_k
Q	Multiplexor Buffer capacity
$R(t_k)$	Non-conforming traffic at time t_k
T	TB capacity

$u_i(t_k)$	Number of tokens generated by i-th TB at for the interval $[t_k, t_{k+1})$
$V_i(t_k)$	Incoming packet size of i-th user at time t_k
$\alpha(t_k)$	Weight given to multiplexor losses at t_k
$\beta_i(t_k)$	Weight given to ith TB losses at t_k
$\gamma(t_k)$	Weight given to waiting cost at t_k
$\rho(t_k)$	TB state at t_k
τ_c	First time in congestion
$\chi(t_k)$	Available space in the multiplexor buffer at time t_k after service during $[t_k, t_{k+1})$
$\mathcal{T}(t_k)$	Throughput during $[t_k, t_{k+1})$
$\Theta_i(t_k)$	Permissible multiplexor space allocation to i-th user at time t_k
μ	Time average of traffic intensity
μ_p	Traffic peak rate
$\bigwedge$	Minimum value of two values
$\bigvee$	Maximum value of two values
$\bigoplus$	Module

Chapter 1

Introduction

1.1 Motivation

Network traffic is highly diverse at present. Every type of traffic has different requirements in terms of bandwidth, delay, jitter, loss and network availability. Applications such as electronic commerce, health-care applications, digital publishing, data mining and voice over IP (VoIP) integrate large amounts of data. High volumes of data are located on several sites interconnected through networks. This situation calls for the requirements of Quality of Service (QoS). There has been a major effort for providing QoS to the current network in recent years and this would include services offering a variety of performance guarantees. The token bucket (or leaky bucket) algorithm has been proposed and used in many control areas to improve performance integrated packet communication networks, such as admission control, access control, flow control and congestion control [1][2][3][4][5][6][7][8][9].

In their paper, Yin and Hluchyj presented a discrete-time system model based on the "buffered leaky bucket" for fixed size cells transmission over the network [3]. Tang and Tai investigated the derivation of optimal token bucket parameters and analyzed the

relationships among observed traffic patterns, token bucket parameters, queue sizes, and queuing delays [4]. The contribution of Vamvakos and Anantharam was to control the long range dependence traffic by using the leaky bucket mechanism [6]. Reference [7] showed that a token bucket as a shaper could control the MPEG video transmission rate over the guaranteed service. In reference [8], a new dynamic model for the token bucket control mechanism, in which a multiplexor is added at an access node, was presented. But there has been limitations which have to be improved and completed. In particular, the following aspects are valuable to be addressed:

1. As we know, the real network traffic has various types and most of them are stochastic. The key to performance evaluation for token bucket control system is the traffic modelling, which is able to simulate various types of traffic. There is no such mechanism in most previous studies. The authors of papers [8][9] proposed the concept of traffic model but they just demonstrated it with a given deterministic traffic, MPEG video trace, as its input traffic.

2. So far, most of previous works only studied one of the aspects of traffic management and performance or presented static models. The work [8][9] presented a dynamic model, but there are limitations: it is based on the assumption that the number of bytes in one packet can be dropped partially by the system.

3. Most of previous works have not considered how to allocate the resources fairly at the access node, such as the trade-off between the user demand and the network provider control. In addition, feedback delay is not taken into account in the previous works.

Therefore, we believe that it is very valuable to develop a qualitative approach for traffic modelling, an advanced multiplexing algorithm for the system model, an improved feedback control strategy, and further investigations of the system performance using the dynamics of the access control mechanism located at the edge of the backbone network.

1.2 Objective

In this thesis, we focus on the following points:

1. We develop traffic models which can capture characteristics of network traffic, both short range dependence (SRD) and long range dependence (LRD) respectively.

2. We also improve the system model mentioned in the work [8][9] by changing the algorithm of packet scheduling to make the system model conform with practical constraints so that the effectiveness of the system model is enhanced.

3. We evaluate the system performance using an objective functional which is a measure of packet losses and service delay by feeding stochastic inputs, which are generated by the traffic models mentioned above and the actual Bellcore traffic (BCpAug89). We also compute different system measures in terms of network utilization, throughput, packet losses and system congestion.

1.3 Contribution

Our main contributions can be summarized as follows:

1. We develop various traffic models. These models can capture the short range dependence (SRD) and the long range dependence (LRD) characteristics of real network traffic. Specially, doubly stochastic Poisson process driven by the fractional Brownian motion (FBM) is a new idea to successfully simulate the self-similar traffic.

2. We have constructed a dynamic system model of an access control mechanism to be used at the edge of network. A token bucket algorithm and a round robin packet scheduling algorithm have been used to build up the dynamic system model. In particular, the round robin packet scheduling policy as a multiplexing algorithm is a good idea to avoid dropping the partial of packets. Therefore, it is more practical

and effective than that in previous work [8][9]. The system is formally formulated. Several control strategies are also proposed. Especially, a feedback control algorithm is considered to eliminate losses at the multiplexor and to avoid hogging of system resources.

3. The system performance is evaluated in detail by feeding different stochastic traffic. We study packet losses, waiting delay, utilization and system congestion. The expected number of times in the state of congestion and the mean first time to congestion are good performance measures to evaluate the system congestion.

4. The Monte Carlo Method has been used to compute the expected values of functions of random processes. The results obtained by using Monte Carlo technique seem to be fairly useful.

1.4 Thesis Organization

The thesis is organized as follows. Chapter 2 introduces the basic concepts. Chapter 3 describes system model and control strategy. The complete system model consists of the traffic model, the token bucket model and the multiplexer model. The traffic models include Poisson process and doubly stochastic Poisson (or Cox) process, where intensity function follows fractional Brownian motion. The state of system is defined in detail. We propose control strategies including an open loop control law and a feedback control law. We discuss the feedback control law in detail. Chapter 4 defines objective function and network performance measures in terms of QoS to the network provider. Chapter 5 presents assumptions and the basic data used in simulation experiments. We also analyze the system performance by feeding different inputs, which are generated by the proposed traffic models and part of the Bellcore Ethernet traffic trace (BC-pAug89)[10], and verify the system model's effectiveness to different types of sources. Finally a conclusion is given in Chapter 6.

Chapter 2

Basic Concepts

This chapter introduces some basic concepts which are used in the thesis. Section 1 presents the concept of quality of service. Section 2 introduces proprieties of network traffic and traffic modelling. In Section 3, we examine the principles and components of traffic management. Here we introduce traffic policing and shaping, congestion control and round robin packet scheduling. The token bucket algorithm in traffic management is described in Section 4. Finally, we introduce the Monte Carlo method which is used in simulation experiments.

2.1 Quality of Service

The term Quality of Service (QoS) refers to the quality of network services. Its objective as described in [11] is to deliver better network services from end to end in order to satisfy customer network applications.

QoS can be characterized by its parameters, bandwidth, latency, jitter and reliability. Different applications have different QoS parameters. Therefore, QoS mechanisms can allocate the bandwidth resources, avoid network congestion, control latency and jitter, and improve the data loss characteristics of the network.

There are three main levels of QoS: best effort service, differentiated service and guaranteed service. Best effort service is the first level of QoS, which only provides a no reliable network service. Internet is best effort service. Differentiated service is an intermediate QoS level, in which some traffic is treated better than the rest. The guaranteed service is the highest level of QoS. In the guaranteed service framework, end-to-end reservation of resources is made for a specific traffic type.

2.2 Traffic Characteristics and Modelling

Network traffic is very complex and it is difficult to characterize. After years of study, people find some characteristics such as self-similarity in traffic patterns, short range dependence (SRD) and long range dependence (LRD) in complex temporal correlation. Based on different traffic characteristics, traffic modelling is classified into two categories: SRD and LRD.

2.2.1 Self-similarity

A self-similar phenomenon displays structural similarities across a wide range of timescales. Self-similarity is the property associated with fractals, which exhibit self-similarity at multiple scales [12]. Network packet traffic has been observed to have a self-similar (or fractal) characteristic, which means the traffic has similar statistical properties at a range of different timescales: milliseconds, seconds, minutes, hours and days. The degree of self-similarity is measured by the Hurst parameter H. When the Hurst parameter H takes values from 0.5 to 1, the traffic is self-similar. $H = 0.50$ means that the time series is independent and $H = 1.0$ means fully dependent. When the Hurst parameter is closer to 1.0, the observed traffic has a high degree of self-similarity.

2.2.2 Short Range Dependence and Long Range Dependence

Network traffic has a complex temporal correlation, which can be characterized by SRD and LRD. SRD traffic has such propriety with potential human interaction. Therefore, SRD traffic is non-bursty, the autocorrelation function falls off much faster with time and usually has exponential probability distribution. The typical examples of SRD traffic are telephone traffic, voice-over IP, telnet connection arrivals, ftp session arrivals and web session arrivals. Compared to SRD traffic, the LRD traffic is more bursty. It is statistically self-similar over a wide range of time scales. Its autocorrelation function decays very slowly and falls into heavy-tailed distribution and infinite variance. It has been shown that web request traffic, LAN data traffic and WAN data traffic possess LRD [12][14][15][16][17].

2.2.3 Traffic Modelling

Based on network traffic characteristics, traffic models fall into two categories: SRD and LRD [13][14]. In the references [13][14], comparison of stochastic processes exhibiting SRD and LRD can be found. The simplest SRD traffic model is the Poisson process model [15], which is very suitable for modelling the traffic related to human activity, such as telephone traffic and session arrivals. These have been shown to be consistent with a homogeneous Poisson process [18][19]. Traffic exhibiting LRD can be well modelled by using fractional Brownian motion (FBM) [13][15][20]. FBM has been shown to have a good statistical fit with data measurements in Ethernet LAN segments, which has self-similarity (fractals), heavy-tailed probability distribution and burstiness [15][17].

2.3 Traffic Management

Traffic management is a critical problem in the development of networks. The objective of traffic management at the edge of the network is to efficiently allocate network re-

sources including buffers and bandwidth, deliver QoS guarantees for various applications and provide overall optimization of network resources. Therefore, traffic management is concerned with problems such as network bandwidth allocation, congestion minimization, reduction of delay. Traffic management consists of such essential components including access control, queuing and scheduling, congestion control.

2.3.1 Access Control (Policing and Shaping)

QoS requires traffic management functions at the ingress of the network: traffic policing and shaping. Traffic policing can control the maximum traffic rate at the edge of the network [21][22]. Policing function is shown in Figure 2.1. It is achieved by dropping excess traffic through a token bucket when the traffic rate reaches the configured maximum rate. Traffic shaping is about regulating the average rate of data transmission [21][22]. Traffic shaping is to force incoming traffic to conform to a certain specified behavior as shown in Figure 2.2. It changes the traffic characteristics by delaying excess packets in a buffer or a leaky bucket and re-scheduling these packets later. The difference of traffic policing and traffic shaping is mentioned in [22]. Traffic policing can transmit bursts, while traffic shaping can put excess packets in a buffer, then propagate these excess packets later over increments of time. Traffic shaping makes the packet output rate smooth. Both policing and shaping mechanisms use the traffic descriptor such as the token bucket to ensure service.

2.3.2 Congestion Control

Congestion is the most significant problem in IP networks [23]. Congestion is a state of network resource in which the traffic on the resource exceeds its output capacity over an interval of time [23]. It means that congestion in a network or a network node occurs when the arriving packets exceed the output capacity. Congestion typically results in buffer overflow and subsequent packet loss. Finally, congestion causes the degradation of

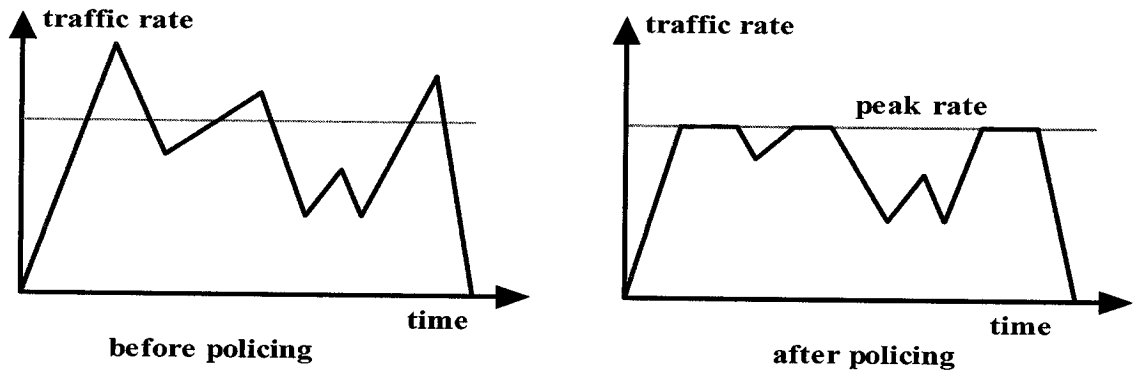


Figure 2.1: Policing Function [22]

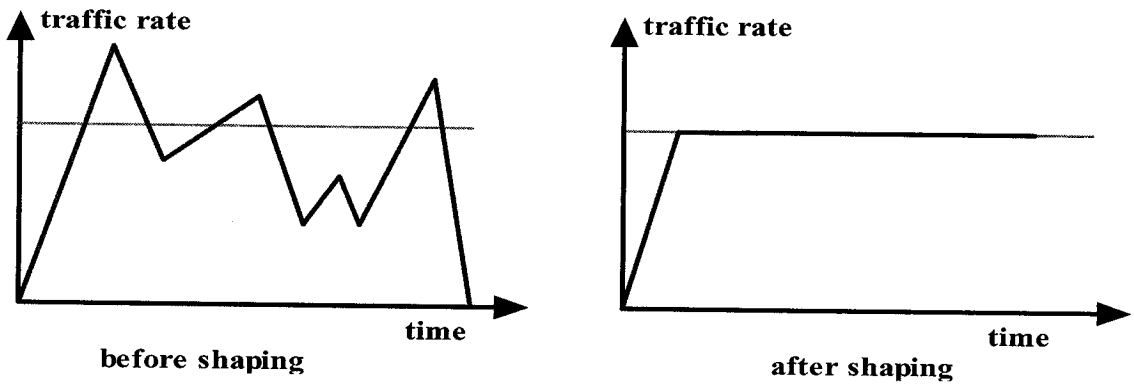


Figure 2.2: Shaping Function [22]

QoS and the network performance will be degraded.

There are three kinds of congestion based on time scale. They are long period time congestion (weeks to months), medium period time congestion (minutes to days) and short time period congestion (Pico seconds to minutes). Long period time congestion is usually solved by updating hardware. As remedial measures for medium period time congestion, re-configuring logical topology or adjusting some router parameters are applied. Regulating the traffic rate can control short time period congestion.

Congestion control is an approach to congestion management. It is to make sure that the network can deliver. There are two types of congestion control techniques. One is congestion recovery mechanisms, which attempt to remedy an already congested resource. The other is congestion avoidance mechanisms, which monitor network traffic loads and try to avoid congestion at the network bottleneck point and to keep a network always in the non-congestion state [24]. In this thesis, we focus on congestion avoidance in our system (multiplexor) during the short time period.

2.3.3 Round Robin Scheduling

Round robin scheduling is one of the oldest, simplest, fairest and most widely used scheduling algorithms, designed especially for time-sharing systems. It is an arrangement of choosing all elements in a group equally in some rational order, usually from the beginning to the end of a list and then starting again and so on. The principle of round robin is to serve the active queues one after the other. It can also be easy to implement.

Round robin is also used to schedule packets in order to share the buffer or the bandwidth fairly. In the thesis, round robin packet scheduling determines the order in which packets conformed by token buckets are transmitted into the buffer of the multiplexor.

2.4 Token Bucket Terminology

Token bucket algorithm has been developed and employed in various control areas in computer networks, such as access control and congestion control.

A token bucket is a typical traffic descriptor, a set of parameters that characterize or regulate traffic sources [21][25]. A token bucket is characterized by two parameters. One is a token generation rate, which creates and places tokens into the bucket and the other is its capacity (bucket size), which can be expressed in terms of tokens. The standard token bucket terminology is: tokens are generated into the bucket at one rate. When the bucket is full of tokens, newly arriving tokens are rejected. When a packet arrives, if the number of tokens in the pool is greater than or equal to the number of bytes in the arriving packet, the packet is accepted and may exit the system immediately. On the other hand, if there are not enough tokens in the bucket, the packet may be dropped (rejected), marked in a particular way or buffered and wait for fresh supply of tokens in the bucket [21][25][26].

A token bucket can work as a policer and a shaper. As a policer, shown in Figure 2.3, the token bucket typically drops packets. If the number of tokens in the token pool is less than the number of bytes in the arriving packet, the packet is dropped immediately. The token bucket as a shaper is shown in Figure 2.4. As a shaper, it buffers the excess packets and will not release until sufficient number of tokens are put into the bucket. When the buffer is full, the arriving packet can not be accepted by the buffer and will be discarded.

Note that a token bucket is not a physical container of packets, but a numerical counter of tokens to schedule the release time of packets. Over a relatively long period of time, the established token rate is regarded as the average data rate, and for short periods of time, the traffic in terms of bytes over fixed time interval will never exceed the token bucket's capacity plus tokens that are placed in the bucket during the same time interval [21][26].

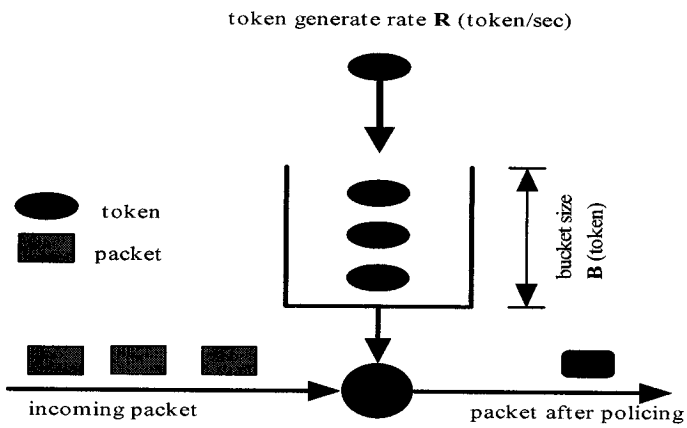


Figure 2.3: Token Bucket Type I

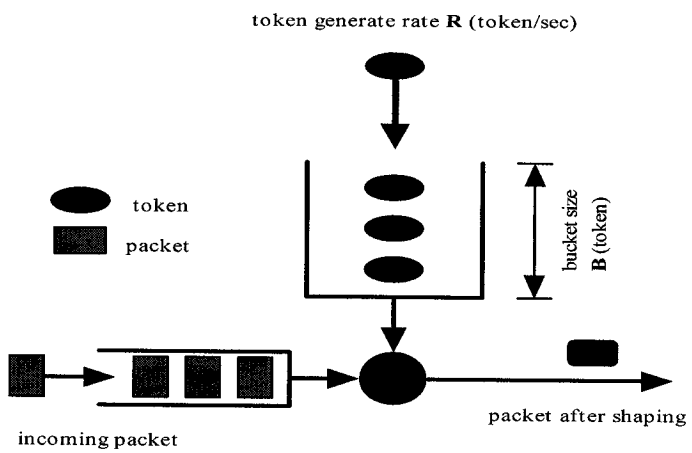


Figure 2.4: Token Bucket Type II

2.5 Monte Carlo Method

The Monte Carlo method (MC) provides a technique for solving mathematical and physical problems by simulating random quantities [27][28]. It is used to obtain numerical solutions to problems which are too complicated to solve analytically. Monte Carlo integration is one of the most common applications of the Monte Carlo method.

In this thesis, we will use the Monte Carlo integration to evaluate expected values of random variables and functions of random processes. Consider an arbitrary function $g(x)$, our aim is to calculate the expected Value (Mean)

$$< G >= \int_a^b g(x)f(x)dx,$$

where $f(x)$ is a p.d.f such that $f(x) \geq 0$ on $[a, b]$ and $\int_a^b f(x)dx = 1$. However, it is difficult to achieve this aim directly. So at first we try to find the following sample mean, which is a Monte Carlo estimator of the expected value:

$$\overline{G}_N = \frac{1}{N}\sum_{i=1}^{N} g(x_i),$$

where the points x_i are taken from the range of integration. An alternate and more efficient approach is to select the points randomly from a given probability distribution. A conventional choice for the points x_i would be a uniform grid.

By the Strong Law of Large Numbers we see that the limit $\overline{G}_N$ almost surely exists. Then according to the fundamental theorem of Monte Carlo, we get

$$< G >= \lim_{N \longrightarrow \infty} \overline{G}_N.$$

Chapter 3

System Model and Control Strategy

In this chapter, we present traffic models, a single dynamic TB model and a complete system model. We also define the relevant control strategies which can govern the operation of the system. At first, we construct traffic models including Poisson process models and a doubly stochastic Poisson (or Cox) process model. We also introduce packet size distribution used in different traffic models. Second, we present a dynamic model of a single TB for access control mechanism. It shows that this kind of TB model can be independently used in various applications. Third, we present a complete dynamic system model, which involves the traffic model, the TB model and the multiplexor model, for access control in the edge of the backbone network. Finally, we propose control polices including an open loop control and a feedback control. To present these models and to define control strategy we need the following notations which are used throughout the thesis:

1. For $x, y \in R : \{x \wedge y\} \equiv Min\{x, y\}$, $\{x \vee y\} \equiv Max\{x, y\}$;

2. For $x = (x_1, ..., x_n), y = (y_1, ..., y_n) \in R^n$:
   $\{x \wedge y\} \equiv \{Min\{x_i, y_i\}, i = 1, 2 \cdots, n\}$, $\{x \vee y\} = \{Max\{x_i, y_i\}, i = 1, 2, \cdots, n\}$;

3. $I(S) = \begin{cases} 1, & \text{if the statement S is true} \\ 0, & \text{otherwise.} \end{cases}$

3.1 Traffic Model

In general, traffic models can be classified into two classes: SRD and LRD [13][14]. The Poisson process has the simplest and most elegant analytical properties. Due to this fact, the Poisson process is often used to model SRD traffic. For the LRD class, we choose a doubly stochastic Poisson (or Cox) process with stochastic intensity. We use FBM to drive the doubly stochastic Poisson (or Cox) process. This is because FBM has fractal or self-similarity properties which is very useful for the simulation of network traffic[13][20].

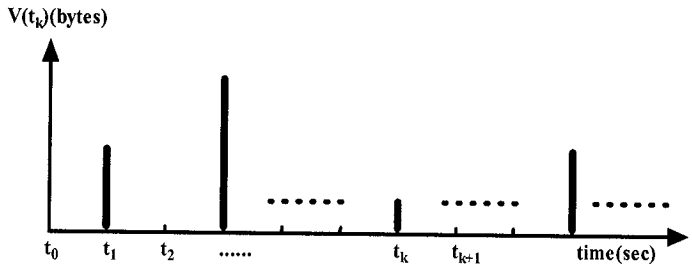


Figure 3.1: Traffic Model

First we give the concept of a general traffic model as shown in Fig 3.1. The arriving traffic denoted by $\{V(t_k), k = 0, 1, 2, ..., K\}$ follows a stochastic process, Poisson process or doubly stochastic Poisson (or Cox) process, observed at time t_k . We partition the time horizon into many equal and small enough non overlapping time intervals $\{I_k\}, k \in N$ given by $I_k \equiv [t_k, t_{k+1})$, and assume that during each of these intervals at most one packet may arrive. We use $V(t_k)$ to denote the size of the packet (or the length of packet), measured in terms of the number of bytes, arriving at time t_k. In other words, $V(t_k)$ represents the length of packet that arrives at any time during the interval $[t_k, t_{k+1})$.

3.1.1 Poisson Process Model

Let $\{T_i, i = 0, 1, 2, ..., n, ..., T_0 = 0\}$ denote the sequence of instants of traffic arrivals. Then, $\{W_n = T_n - T_{n-1}, n > 0\}$ represents the sequence of inter-arrival time process. Let $\{N(t), t \geq 0\}$ be the number of traffic arrivals in the interval $(0, t)$. The Poisson process model is shown in Figure 3.2. According to Poisson process proprieties, inter-arrival times W_n of Poisson process $\{N(t), t \geq 0\}$ are independent and obey the exponential distribution with rate parameter λ:

$$P(W_n \leq t) = \lambda e^{-\lambda t}, t \geq 0.$$

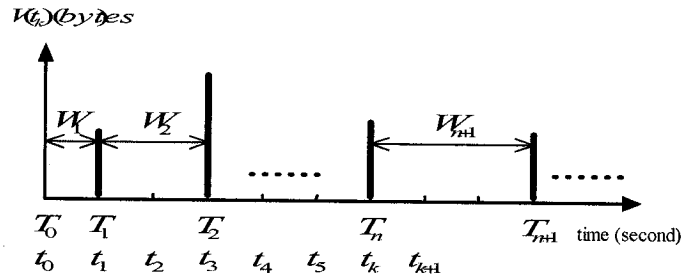


Figure 3.2: Poisson Traffic Model

For a HPP traffic model, the packet arrival rate λ is constant. The number of packets arriving during the interval $(0, t)$ follows the Poisson distribution:

$$P(N(t) = n) = \frac{(\lambda t)^n e^{-\lambda t}}{n!}, t \geq 0.$$

For a NHPP traffic model, λ is a function of time, say, $\lambda = m(t)$, where m is a nonnegative measurable function and locally integrable. In this case the probability that n packets arrive during the interval $(0, t)$ is given by :

$$P(N(t) = n) = \frac{\left(\int_0^t m(s)ds\right)^n e^{-\int_0^t m(s)ds}}{n!}, t \geq 0.$$

3.1.2 Doubly Stochastic Poisson Process Model

A doubly stochastic Poisson (or Cox) process can be considered as a conditional Poisson process, with intensity $\{\lambda(t), t \geq 0\}$ being a nonnegative stochastic process [29]. This can be realized by following two randomization procedures. Given a realization, $\{\lambda(t), t \geq 0\}$, one generates the Poisson process $\{N(t), t \geq 0\}$ using $\{\lambda(t), t \geq 0\}$ as its intensity as in the case of NHPP given above. That is, $\{N(t), t \geq 0\}$ is a (conditional) Poisson process with intensity $\lambda(t)$, where $\lambda(t)$ itself is a stochastic process. In other words,

$$P\{N(t) = n | \lambda(s), s \leq t\} = \frac{\left(\int_0^t \lambda(s)ds\right)^n e^{-\int_0^t \lambda(s)ds}}{n!}.$$

Doubly stochastic Poisson process driven by fractal shot noise (FSN) or fractional Gaussian noise (FGN) is used to simulate network traffic, which has self-similar characteristics [30]. In this thesis, we use fractional Brownian motion, which has fractal (self-similar) properties and is able to capture and incorporate LRD, to generate the intensity $\lambda(t)$ of the Poisson process.

Let $B_H(t)$ denote the FBM, associated with the Hurst parameter (self-similarity parameter) H, which is used to describe the degree of LRD and the burstiness of the traffic. Here, $B_H(t), t \geq 0$, a Gaussian process, is given by the integral transform of the classical Brownian motion,

$$B_H(t) = \int_0^t K_H(t-s)dB(s), t \geq 0,$$

where $\{B(t), t \geq 0\}$, is the standard Brownian motion and $K_H(\cdot)$ is a kernel satisfying certain conditions. We construct two different intensity processes $\lambda(t)$ for the doubly stochastic Poisson (or Cox) process as follows:

$$(1) : \lambda(t) \equiv \{B_H(t) \vee 0\} \equiv \sup\{B_H(t), 0\}, t \geq 0,$$

$$(2) : \lambda(t) \equiv |B_H(t)|, t \geq 0.$$

For our numerical experiments we choose the simple kernel K_H given by

$$K_H(t) = C_H t^{(H-\frac{1}{2})}, \qquad \frac{1}{2} < H < 1,$$

where C_H is any constant. By using the process $\lambda(t)$ as the intensity for a Poisson process, we can generate traffic which exhibits self-similarity and LRD properties.

3.1.3 Packet Size Distribution

The incoming traffic is well characterized by packet inter-arrival time (IAT) and packet size. From the above two sections we see that packet arrival times are determined by different stochastic processes. So we just need to discuss packet size here. Packet size distribution (PSD), one of the characteristics of network traffic, describes the size of packets travelling across the network. In reality, there are a wide range of packets sizes in computer networks. And packet size distribution plays a significant role in the performance of networks. Here two packet size distributions are discussed in order to capture various network traffic better. One is used in the Poisson process and the other is for the doubly stochastic Poisson (or Cox) process .

Packet Size used in Poisson Process

Most studies of packet size distribution can be found in [31][32]. Reference [31] summarized various packet sizes and types occurring most frequently from a total of 342 million packets for Ethernet networks. There are three kinds of size as follows:

1. The packet size is 40 bytes. Such packets are mainly from TCP packets with header flags but no payload, which are typically used at the start of a new TCP session;

2. The packet size is 576 bytes. These packets correspond to the Maximum Segment Size (MSS) of TCP packets from early implementations. MSS defines the largest segment of TCP data that can be transmitted;

3. The packet size is 1500 bytes. These kinds of packets correspond to the Maximum Transmission Unit (MTU) size of an Ethernet network, which is the largest amount of data that can be transferred in one physical frame on the network.

In [31], the complete Imix (Internet mix) of packet size distribution is described, where the average packet size is 427 bytes. In this thesis, we will choose the complete Imix as the packet size distribution of Poisson process traffic. Figure 3.3 shows packet size distribution by packets.

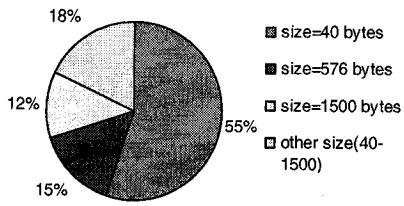


Figure 3.3: Packet Size Distribution (by Packets)

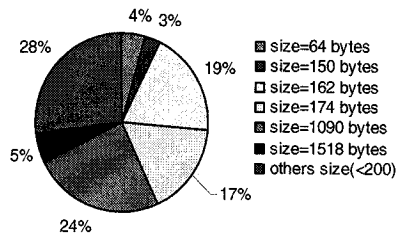


Figure 3.4: BC-pAug89 Packet Size Distribution (by Packets)

Packet Size used in DSPP

BC-pAug89 traffic trace is one of the most typical Ethernet traffic, which has its own packet size distribution. The range of the size of all BC-pAug89 packets is from 64 bytes to 1518 bytes. We investigate the packet size of the first 8-minute of BC-pAug89 and plot its distribution in Figure 3.4. Then, we assume that the packet size distribution of the Cox process (or DSPP) will match that of the first 8-minute of BC-pAug89 with average packet size being 445 bytes.

3.2 A Dynamic Model for Access Control Mechanism

Note that a TB is often used as an access control mechanism in computer communication networks. A new dynamic model for TB Algorithm used in computer networks was proposed in previous work [8]. As in [8], here we use a TB without a waiting buffer to execute only a policing function. Figure 3.5 clearly illustrates a single dynamic TB model as an access control mechanism. This single dynamic TB model can be used in various applications alone to control the maximum traffic rate at the edge of network.

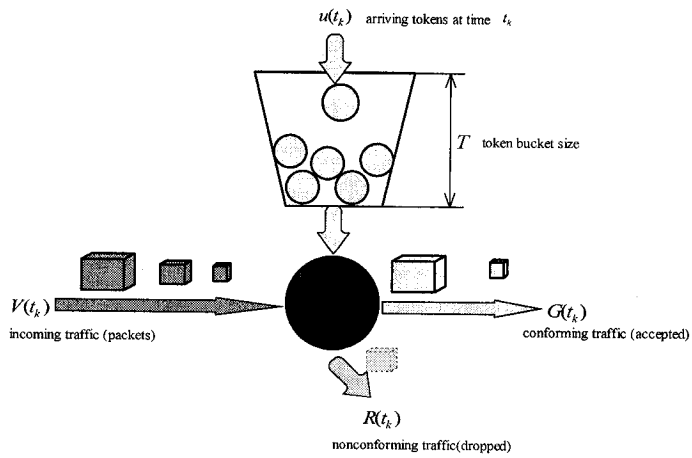


Figure 3.5: A Dynamic TB Model

This dynamic model of TB as policing function follows standard TB terminology introduced in section 2.4. Now we use the state of TB, which is described in terms of the number of tokens in the bucket. Let T be the physical capacity of the TB. The process $u(t_k)$ denotes the arriving tokens at time t_k. In other words, $u(t_k)$ represents the number of tokens that arrive at any time during the interval $[t_k, t_{k+1})$ and $\rho(t_k)$ denotes the state of TB measured in terms of number of tokens in the bucket at time t_k. The state of the TB is governed by the following difference equation:

$$\begin{aligned}\rho(t_{k+1}) = \rho(t_k) + \{u(t_k) \wedge [T - \rho(t_k)]\} \\ - V(t_k) I\{V(t_k) \leq [\rho(t_k) + [u(t_k)] \wedge [T - \rho(t_k)]]\}\end{aligned} \tag{3.1}$$

This expression is based on the fact that the state of each token bucket at time t_{k+1} is decided by its previous state, the number of tokens accepted during the time interval $[t_k, t_{k+1})$ and the number of tokens consumed during the same period $[t_k, t_{k+1})$. From the state equation of TB, we know the evolution of the token population with time.

We can further give the concepts of the conforming traffic and nonconforming traffic. When a packet arrives, if the number of tokens in the TB is larger than or equal to the number of bytes in the arriving packet, this packet can enter the multiplexor in which the packet can be queued up to access the network. This kind of traffic is called conforming traffic. If there are not enough tokens in the token pool, the packet is dropped. This kind of traffic is called nonconforming traffic. By the definition and according to this proposed dynamic TB model, the conforming traffic at time t_k, denoted by $G(t_k)$, can be obtained from the state equation of TB :

$$G(t_k) = V(t_k) I\{V(t_k) \leq [\rho(t_k) + [u(t_k)] \wedge [T - \rho(t_k)]]\} \tag{3.2}$$

From Equation 3.2, the conforming traffic $G(t_k)$ is determined by one condition, that is,

the size of the arriving packet $V(t_k)$ at time $[t_k, t_{k+1})$ is not larger than the number of tokens in the pool at the same time. If the condition is satisfied, then $G(t_k) = V(t_k)$. Otherwise, the arriving packet has to be dropped at the token bucket, implying $G(t_k) = 0$. The state of TB at time t_k and the number of tokens accepted by the TB during the time interval $[t_k, t_{k+1})$ decide the number of tokens at time t_{k+1}.

Hence the nonconforming traffic at time t_k, given by $R(t_k)$, is:

$$R(t_k) = V(t_k) - G(t_k). \tag{3.3}$$

3.3 System Model

We construct a dynamic model for the access control mechanism based on the system model developed in [8]. This is shown in Figure 3.6.

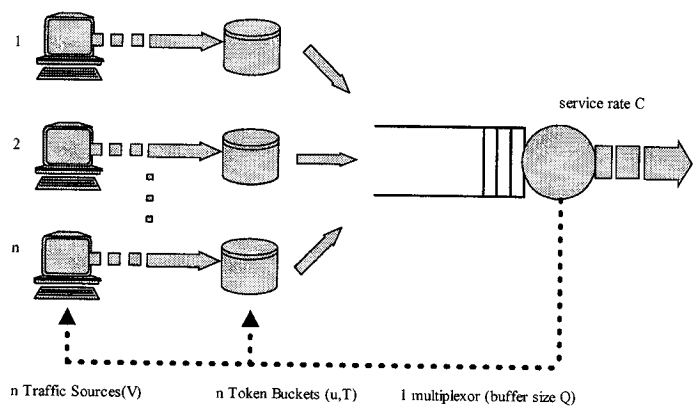


Figure 3.6: Complete System Model

This system model has n token buckets which police the incoming traffic from n traffic sources respectively. All the traffic sources share one and the same multiplexor having a finite buffer size Q, and the outgoing link having the capacity C. Each user is allocated a

bandwidth determined by the network resource limitation, user's request and the network provider's arbitration. When the incoming traffic enters the system, token buckets drop all the nonconforming traffic and accept only the conforming traffic and send them to the multiplexor. The multiplexor accepts the conforming traffic provided the buffer is not full. The conforming traffic accepted by the multiplexor is placed in the queue for access to the network. All actions are dynamically controlled by a controller, which regulates the token supply to each of the token pools. Based on the principle of conservation of information, the system status is described by the occupancy status of all the token buckets and the multiplexor.

At first, we discuss token buckets. The state for one TB is given by the difference equation (3.1) as presented in the previous section. For n token buckets, the state may be described by a n-dimensional vector valued function

$$\rho = (\rho_1, \rho_2, \cdots, \rho_n)'.$$

The size of the token buckets is also a vector denoted by $T = (T_1, T_2, \cdots, T_n)'$ and the token supply to each of the buckets is given by the vector $u = (u_1, u_2, \cdots, u_n)'$. Therefore the status of n token buckets is now governed by the following vector difference equation:

$$\begin{aligned}
\rho_1(t_{k+1}) &= \rho_1(t_k) + \{u_1(t_k) \wedge [T_1 - \rho_1(t_k)]\} \\
&\quad - V_1(t_k) I\{V_1(t_k) \leq [\rho_1(t_k) + [u_1(t_k)] \wedge [T_1 - \rho_1(t_k)]]\}, \\
\rho_2(t_{k+1}) &= \rho_2(t_k) + \{u_2(t_k) \wedge [T_2 - \rho_2(t_k)]\} \\
&\quad - V_2(t_k) I\{V_2(t_k) \leq [\rho_2(t_k) + [u_2(t_k)] \wedge [T_2 - \rho(t_k)]]\}, \\
&\bullet\bullet\bullet \\
\rho_n(t_{k+1}) &= \rho_n(t_k) + \{u_n(t_k) \wedge [T_n - \rho_n(t_k)]\} \\
&\quad - V_n(t_k) I\{V_n(t_k) \leq [\rho_n(t_k) + [u_n(t_k)] \wedge [T_n - \rho_n(t_k)]]\}.
\end{aligned} \tag{3.4}$$

The conforming and the nonconforming traffic are given by the vector-valued functions denoted by $G = (G_1, G_2, \cdots, G_n)'$ and $R = (R_1, R_2, \cdots, R_n)'$ respectively. The conforming traffic of n dimension is give by:

$$G_1(t_k) = V_1(t_k) I\{V_1(t_k) \leq [\rho_1(t_k) + [u_1(t_k)] \wedge [T_1 - \rho_1(t_k)]]\},$$

$$G_2(t_k) = V_2(t_k) I\{V_2(t_k) \leq [\rho_2(t_k) + [u_2(t_k)] \wedge [T_2 - \rho_2(t_k)]]\},$$

$$\bullet\bullet\bullet \tag{3.5}$$

$$G_n(t_k) = V_n(t_k) I\{V_n(t_k) \leq [\rho_n(t_k) + [u_n(t_k)] \wedge [T_n - \rho_n(t_k)]]\},$$

and the vector of nonconforming traffic is given by:

$$R_1(t_k) = V_1(t_k) - G_1(t_k),$$

$$R_2(t_k) = V_2(t_k) - G_2(t_k),$$

$$\bullet\bullet\bullet \tag{3.6}$$

$$R_n(t_k) = V_n(t_k) - G_n(t_k).$$

Now we define the state of the multiplexor at any time t, denoted by $q(t)$, to be the size of the queue waiting for service at time t. Note that the multiplexor of our system is shared by n parallel users at the same time. Fairness is an intuitively desirable property in the allocation of resources to the users. Due to its finite size, it is possible that some time (peak period) the multiplexor will not be able to accept all the conforming traffic from all the sources. For instance, the conforming traffic $G_i(t_k)$, from the i-th user at time t_k, may be accepted or dropped depending on the occupancy status of the multiplexor. In view of this fact, we propose that the multiplexor accepts the conforming traffic in a round robin order. Thus, to complete the dynamics of the whole system, we may now formulate the

status of the multiplexor using this order. Let $q(t_k)$ denote the state of the multiplexor at time t_k. Let $e(t_k)$ denote the end point at which the multiplexor accepts the last (user) conforming traffic at time t_k. Let $g_i(t_k)$ represent the sum of all the conforming traffic up to and including the i-th user. Let $l(t_k)$ denote the largest number of users served during the period $[t_k, t_{k+1})$ and $\chi(t_k)$ the available (empty) space in the buffer at time t_k after part of the traffic has been delivered during the time interval $[t_k, t_{k+1})$. Define

$$r \oplus n \equiv \text{ the remainder of } \quad r \div n.$$

Let h denote the length of each of the time slots $[t_k, t_{k+1})$, for $k = 0, 2, \cdots, K-1$. Using these notations, we justify that the state of the multiplexor is governed by the following difference equation:

$$\begin{aligned} q(t_{k+1}) &= \{[q(t_k) - Ch] \vee 0\} + \Big\{ g_n(t_k) I\{\chi(t_k) \geq g_n(t_k)\} \\ &\quad + \sum_{i=(e(t_{k-1})+1)\oplus n}^{(e(t_{k-1})+n-1)\oplus n} g_i(t_k) I\{g_i(t_k) \leq \chi(t_k) < g_{i+1}(t_k)\} \Big\}. \end{aligned} \tag{3.7}$$

where

$$\begin{aligned} e(t_k) &= [e(t_{k-1}) + l(t_k)] \oplus n \\ l(t_k) &= \sum_{s=1}^{n-1} s I\{g_s(t_k) \leq \chi(t_k) < g_{s+1}(t_k)\} + n I\{\chi(t_k) \geq g_n(t_k)\} \\ g_i(t_k) &= \sum_{j=(e(t_{k-1})+1)\oplus n}^{(e(t_{k-1})+i)\oplus n} G_j(t_k) \quad for \quad i \leq n \\ \chi(t_k) &= Q - [[q(t_k) - Ch] \vee 0]. \end{aligned}$$

We can determine $e(t_k)$ and $l(t_k)$ for any $k = 1, 2, ..., K$ under the initial conditions $e(t_0) = 0$ and $l(t_0) = 0$. The state of the multiplexor, governed by Equation (3.7), at time t_{k+1} is given by the sum of two parts. The first part in (3.7) represents the traffic

that was left in the queue at the end of the previous time period and the second part in (3.7) represents the fresh traffic accepted during the time period $[t_k, t_{k+1})$. Each of the terms can be explained in more details as follows. The first term on the right hand side of Equation (3.7) represents the leftover traffic counted at time t_{k+1} after the traffic in the multiplexor has been delivered into the outgoing link. The second term represents the volume of traffic received from all the sources at time t_{k+1} provided the empty space in the buffer is greater than or equal to the sum of all conforming traffic. The third term describes the traffic accepted by the multiplexor at time t_{k+1} according to round robin principle provided the available space in the queue can not accommodate the sum of all the traffic from all the sources. From Equation (3.7), it is clear that some conforming traffic may be dropped because of the limitation of buffer size and the link capacity. Therefore, the traffic loss in the multiplexor at time t_k denoted by $L(t_k)$ is given by:

$$\begin{aligned} L(t_k) &= \sum_{i=1}^{n} G_i(t_k) - \Big\{ g_n(t_k) I\{\chi(t_k) \geq g_n(t_k)\} \\ &\quad + \sum_{i=(e(t_{k-1})+1)\oplus n}^{(e(t_{k-1})+n-1)\oplus n} g_i(t_k) I\{g_i(t_k) \leq \chi(t_k) < g_{i+1}(t_k)\} \Big\}, \end{aligned} \tag{3.8}$$

where the first part represents the total conforming traffic from all the sources at time t_k , the second part represents the part of conforming traffic that is accepted by the multiplexor. It is clear from this expression that if the available (left) space at time t_k is large enough to accommodate (contain) all the conforming traffic at that time, there will be no multiplexor loss at time t_k. Otherwise, in general some losses would occur. Thus the complete system model is given by the set of $n+1$ difference equations (3.4) and (3.7).

3.4 Control Strategy

Open loop control and close loop control (feedback control) are two kinds of controls used in traffic engineering. Open loop control is a control law that does not use feedback

information from the current network state; it is a blind control. However, close loop control uses feedback information from the network state [24]. Therefore, open loop and feedback control are considered as two different strategies for control (token generation policy) in this thesis.

3.4.1 Open Loop

In this thesis, we consider different cases of simple open loop controls in our numerical experiments. Each case will have a fixed token generation rate. In the first case, the token generation rate is equal to the time average of traffic arrival rate. In the second case, uses the generation rate is greater than the time average of this arrival rate. Finally, in the last case, the generation rate is equal to the peak traffic rate. For any of these given control policies u, we present numerical results and performance evaluation in Chapter 5.

3.4.2 Feedback

Feedback control is regarded as a function of the incoming traffic and the state of the proposed system. In order to give general presentation for feedback control, we first define some notations. Let N_0 denote the set of all nonnegative integers and N_0^d be the Cartesian product, $N_0^d = \underbrace{N \times \cdots \times N}_{d}$. Then we may consider $\mathcal{V} \equiv N_0^n$ as the input space, $X \equiv N_0^{n+1}$ as the state space and $\mathcal{U} \equiv N_0^n$ as the control space. In general, a feedback control law is a nonlinear map from $I \times \mathcal{V} \times X$ to $\mathcal{U}$ indicated by

$$F : I \times \mathcal{V} \times X \longrightarrow \mathcal{U},$$

where I denotes the time index $\{r = 0, 1, 2, 3 \cdots, K\}$.

For this system, abstract feedback control law may be written as follows:

$$u(t_k) \equiv F(t_k, V(t_k), \rho(t_k), q(t_k)) \tag{3.9}$$

where $k = 0, 2, ..., K-1$. The above expression only considers current time. Therefore, the current token generation rate is determined by current input and current system state. However feedback delay may happen. At this time, the present control is not a function of the current state and input rather is a function of the past states (information delay) and inputs. This may be described by

$$u(t_k) \equiv F(t_k - mh, V(t_k - mh), \rho(t_k - mh), q(t_k - mh)) \tag{3.10}$$

for $k = 0$ (no delay), $1, 2, ..., K-1$, where h is the length of the basic interval $[t_k, t_{k+1})$ and m is an integer giving the number of delay intervals. For example, if we choose $m = 2$, it means that feedback delays two basic time intervals (or slot). In other words, present control policy $u(t_k)$ is determined by previous state $(\rho(t_{k-2}), q(t_{k-2}))$ and arrival traffic at that time $V(t_{k-2})$.

Here we use some simple feedback control laws, which are easy to implement. In this thesis, we study feedback control laws without feedback delay in detail. We also discuss feedback control laws with feedback delay. To demonstrate the usefulness and effectiveness of our analysis of the system model, we will compare the results corresponding to the feedback control law with those corresponding to open loop controls based on different traffic.

Feedback Without Delay

In this thesis, we focus on eliminating losses at the multiplexor by using feedback without delay control strategy. In order to achieve the goal, we must consider the maximum permissible allocation of buffer space and the allocation granted by the network provider through application of arbitration. Let Θ_i denote the maximum permissible allocation of

the multiplexor to the i-th user. This is determined as follows:

$$\Theta_i(t_k) \equiv \left\{ \frac{V_i(t_k)}{\sum_{i=1}^{n} V_i(t_k)} \wedge \frac{e_i}{n} \right\} [Q - [(q(t_k) - Ch) \vee 0]], \tag{3.11}$$

where C is the service rate, Q is the buffer size of the multiplexor and h is the length of the time intervals I_k. The factor $e_i \in \{1, 2, 3, \cdots, n\}$ is a parameter controlled by the network provider to determine the grade of permit offered to the i-th user. Note that the first term within the parenthesis does not give priority to any of the users; resources are allocated proportionally to their demands. The second determines the weight of each user permit controlled by the network provider. Thus, resources allocated to each user depend on this parameter. If $e_i = n$ for all i, then every user receives allocation according to its demand. By choosing $1 \leq e_i < n$, the network provider can prevent any user from capturing an excessive amount of resources (for example, bandwidth) by merely increasing the user's demand and thereby starving the other users. The grade of permit is determined by many factors. Such factors include each user's priority, each user's bandwidth requirement, and the capacity of the network. Comparing maximum allowable token permit against the actual traffic, the true allocation for each user at time t_k is obtained as follows:

$$A_i(t_k) = \{\Theta_i(t_k) \wedge V_i(t_k)\}. \tag{3.12}$$

The true allocation for each user is given by the minimum value of the actual traffic and allowable token permit. Thus the true allocation for all the users is equal to or less than the empty space in the buffer of the multiplexor. Based on the above arguments, we may now define the feedback control law without delay as,

$$u_i(t_k) = \{A_i(t_k) - \rho_i(t_k)\} I\{A_i(t_k) \geq \rho_i(t_k), A_i(t_k) = V_i(t_k)\}. \tag{3.13}$$

Following this control policy, we can completely eliminate the cell losses at the multiplexor. The first factor in the expression for equation (3.13), $\{A_i(t_k) - \rho_i(t_k)\}$, represents the

number of additional tokens required by the i-th user at time t_k. This may be either positive or negative . If the allocation $A_i(t_k)$ is not greater than the number of existing tokens in the token bucket, then $u_i(t_k) = 0$. In other words, if there are enough tokens left which can support the true allocation at time t_k, no token is needed to be provided. Thus, the token provision is given by equation (3.13). This control strategy completely eliminates packet losses at the multiplexor.

Feedback With Delay

In view of the general expression for feedback controls with delay given by (3.10), and the particular control law without delay given by (3.13), we use the following control law allowing one step or more steps delay:

$$u_i^d(t_k) \equiv u_i(t_k - [\xi]h) \quad \xi \geq 0, \tag{3.14}$$

where $[\xi]$ means the maximum integer part of ξ. Here we do not consider the delay time within one basic time interval (or slot). In other words, if the delay time is less than the length of the basic interval h, then no feedback delay occurs. In our numerical experiments, we will discuss feedback delay with either one time interval or two time intervals.

Chapter 4

Objective Function and Performance Measures

In this chapter, we shall define objective function by taking into account losses at TBs, losses at the multiplexor and the waiting time in the queue of the multiplexor. And then we will present several possible measures of performance of the overall system. These are network utilization, throughput, average packet losses, congestion, first time to congestion and time spent in the state of congestion.

We shall focus on stochastic traffic, and therefore the objective function and the performance measures are given in terms of expected values of standard deterministic measures of performance. Monte Carlo integration, introduced in Section 2.5, is used to compute the expected values. For convenience, we define M to be the number of sample paths used and $\Omega \equiv \{w_j, j = 1, 2, 3 \cdots, M\}$ denotes the corresponding sample space with finite cardinality M throughout the thesis.

4.1 Objective Function

The objective functional for a network provider is given by:

$$J(u) \equiv E\left\{\sum_{k=0}^{K}\alpha(t_k)L(t_k) + \sum_{i=1}^{n}\sum_{k=0}^{K}\beta_i(t_k)R_i(t_k) + \sum_{k=0}^{K}\gamma(t_k)q(t_k)\right\} \tag{4.1}$$

for any choice of control policy u.

Note that, the objective function J here is not only a function of $u = (u_1, u_2, ..., u_n)'$, the (token) control vector, but also a function of input traffic process, in particular its intensity function λ, where $\lambda = (\lambda_1, \lambda_2, ..., \lambda_n)'$ is the vector of mean traffic arrival rates. These rates are constant for the homogeneous Poisson process and time varying for the nonhomogeneous process and stochastic for the doubly stochastic Poisson process. The first term of the objective functional within the parenthesis describes the weighted packet losses at the multiplexor. The second term represents the weighted losses at token buckets and the last term gives the weighted cost associated with service delay at the multiplexor. This is an approximate measure of delay before being served. Relative weights to various losses are denoted by the parameters $\alpha(t_k), \beta_i(t_k), \gamma(t_k), i = 1, 2, ..., n$, which are nonnegative functions of time and can be assigned to reflect different concerns and scenarios as necessary.

Here we use M sample paths with the sample space $\Omega \equiv \{w_j, j = 1, 2, 3\cdots, M\}$ to estimate the expected values. Note that, for each ω_j, we have the corresponding traffic losses at time t_k : losses at the multiplexor $L(t_k, \omega_j)$, nonconforming traffic $R_i(t_k, \omega_j)$ and service delay at the multiplexor $q(t_k, \omega_j)$. Thus the above expression can be modified as follows:

$$J(u) \cong \frac{1}{M}\sum_{j=1}^{M}\left\{\sum_{k=0}^{K}\alpha(t_k)L(t_k, \omega_j) + \sum_{i=1}^{n}\sum_{k=0}^{K}\beta_i(t_k)R_i(t_k, \omega_j) + \sum_{k=0}^{K}\gamma(t_k)q(t_k, \omega_j)\right\} \tag{4.2}$$

In this thesis, we study how different traffic intensities influence the objective functional based on feedback and open loop control strategies. These will be addressed in later chapters.

4.2 Performance Measures

4.2.1 Network Utilization

Network utilization can be used to measure the efficiency of the access control system. It is defined to be the ratio of the total traffic successfully delivered to the outgoing link and the total traffic that could be served by the link at its full capacity over the observation period. We use η to denote network utilization, which is given by

$$\eta = \frac{\sum_{i=1}^{n}\sum_{k=0}^{K} V_i(t_k) - \left[\sum_{k=0}^{K} L(t_k) + \sum_{i=1}^{n}\sum_{k=0}^{K} R_i(t_k)\right]}{C(t_K - t_0)}. \tag{4.3}$$

We are also interested in its expected value denoted by $\bar{\eta}$. By using the idea of Monte Carlo method, we can compute it and obtain the following expression:

$$\bar{\eta} \cong \frac{1}{M}\sum_{j=1}^{M}\left\{\frac{\sum_{i=1}^{n}\sum_{k=0}^{K} V_i(t_k,\omega_j) - \left[\sum_{k=0}^{K} L(t_k,\omega_j) + \sum_{i=1}^{n}\sum_{k=0}^{K} R_i(t_k,\omega_j)\right]}{C(t_K - t_0)}\right\}. \tag{4.4}$$

4.2.2 Throughput

Throughput is used to measure network efficiency too. It is given by the product of the utilization factor and the link capacity as follows:

$$\Phi = C\eta = \frac{\sum_{i=1}^{n}\sum_{k=0}^{K} V_i(t_k) - \left[\sum_{k=0}^{K} L(t_k) + \sum_{i=1}^{n}\sum_{k=0}^{K} R_i(t_k)\right]}{(t_K - t_0)} \tag{4.5}$$

$$= \frac{1}{t_K - t_0}\sum_{k=1}^{K}\Phi(t_k), \tag{4.6}$$

where $\Phi(t_k)$ denotes the throughput during time slot $[t_k, t_{k+1})$ given by:

$$\Phi(t_k) = g_n(t_k)I\{\chi(t_k) \geq g_n(t_k)\} + \sum_{i=(e(t_{k-1})+1)\oplus n}^{(e(t_{k-1})+n-1)\oplus n} g_i(t_k)I\{g_i(t_k) \leq \chi(t_k) < g_{i+1}(t_k)\}. \tag{4.7}$$

Note that, the throughput measures the amount of bytes (or bits) per second transferred into the network. And the maximum throughput is restricted by link capacity. Again, we focus on the expected value of the throughput $\bar{\Phi}$. It is given by:

$$\bar{\Phi} = C\bar{\eta}. \tag{4.8}$$

4.2.3 Packet Losses

We may also define the measure of performance of the system by its packet losses. We define the percentage (fraction) of packet losses, denoted by σ, as follows:

$$\sigma = \frac{\sum_{k=0}^{K} L(t_k) + \sum_{i=1}^{n}\sum_{k=0}^{K} R_i(t_k)}{\sum_{i=1}^{n}\sum_{k=0}^{K} V_i(t_k)}. \tag{4.9}$$

The numerator of this expression gives the total losses at the token buckets and the multiplexor during the period of observation, and the denominator gives the total traffic from all the users during the same period. Packet losses per unit time is determined by

dividing the cumulative losses by the length of the observation period . Waiting time does not play any role in this expression. Again use Monte Carlo Method, we can calculate the expected cell losses $\bar{\sigma}$.

4.2.4 System Congestion

System congestion is one of the most serious concerns to both the service providers and the network users. In order to define system congestion for the system under consideration, we first introduce the critical region (interval) of the multiplexor denoted by Q_α. It is given by

$$Q_\alpha \equiv [\alpha Q, Q],$$

where $0 < \alpha \leq 1$ and Q is the buffer size of the multiplexor. Then we may define system congestion as follows. The system is declared to be in state of congestion whenever the multiplexor state enters the critical region Q_α. The parameter α can be chosen by the network provider. For example, when the network provider chooses $\alpha = .95$, it means that the system is declared to be in state of congestion whenever ninety five percent of the multiplexor space is filled.

Let $N_c(r)$ be the number of times the system is in state of congestion (blocked) during the period $[t_0, t_r), 0 \leq r \leq K$. We define this by:

$$N_c(r) \equiv \sum_{k=0}^{r} I\left\{q(t_k) \in Q_\alpha\right\}. \tag{4.10}$$

It can be explained as follows. When the multiplexor state $q(t_k)$ at time t_k, for any $k = 0, 1, 2, ...r$, falls into the critical region, we say the system has experienced one time congestion (one time that the system is in the state of congestion).

We are concerned with system congestion for the entire period of observation, $[t_0, t_K)$. In that case this is given by:

$$N_c(K) \equiv \sum_{k=0}^{K} I\left\{q(t_k) \in Q_\alpha\right\}. \tag{4.11}$$

Expected Number of Times in the State of Congestion

Since the system is subject to stochastic traffic, it is natural to consider the expected number of times the system may enter the state of congestion. Thus, we are interested in $E\{N_c(K)\}$ rather than $N_c(K)$ itself. We denote this by

$$\overline{N}_c(K) = E\{N_c(K)\}. \tag{4.12}$$

Note that $N_c(K)$ is a nonnegative random variable depending on the sample path and it is bounded above by K (with probability one). For some sample paths there may be no congestion and some others the entire period may experience congestion. These are the extreme cases. We use Monte Carlo simulation to compute this. Thus the expected number of times that the system is in the state of congestion during the entire period $[t_0, t_K)$ can be computed as follows:

$$\overline{N}_c(K) \equiv E(N_c(K)) \cong \frac{1}{M} \sum_{j=1}^{M} \left(\sum_{k=0}^{K} I\left\{q(t_k)(\omega_j) \in Q_\alpha\right\} \right). \tag{4.13}$$

From the above expression, we find that $\overline{N}_c(K) = E(N_c(K))$ depends not only on the incoming traffic (rate) but also on the length of the period of operation (or observation). Based on this, we can also determine the percentage (or fraction) of time spent in the state of congestion:

$$\phi_K = \frac{E(N_c(K))}{K}. \tag{4.14}$$

Mean First Time to Congestion

In packet switched networks, congestion can occur at any time. Here we are interested in the first time the system hits (experiences) congestion. We define this as follows:

$$\tau_c \equiv \begin{cases} \inf\{r : 0 \le r \le K, N_c(r) \ge 1\}, \\ \infty \quad \text{if no congestion during the period.} \end{cases} \tag{4.15}$$

We note that τ_c is a nonnegative random variable that depends on the demand profile of the day. We will also compute the expected value by using Monte Carlo technique:

$$\overline{\tau}_c \equiv E(\tau_c) \cong \begin{cases} \frac{1}{M'} \sum_{j=1}^{M'} \tau_c(\omega_j), & \text{if } \tau_c(\omega_j) < \infty \\ \infty & \text{otherwise.} \end{cases} \tag{4.16}$$

where $M' \le M$ is the number of sample paths (out of M) for which congestion occurs, that is $\tau_c < \infty$. One of the objectives of traffic engineering is to avoid congestion at network bottlenecks. Thus it is possible to find control strategies, subject to resource constraints, that maximizes the mean first time to congestion.

Chapter 5

Implementation and Numerical Analysis

In this chapter, we shall describe our model implementation, and evaluate its performance corresponding to different stochastic inputs in terms of the cost function, packet losses, utilization, throughput and congestion. In Section 1, we introduce the assumptions, the system parameters, and the specific details of the traffic used for simulation. Then in Sections 2 to 5, we study the system response and performance by using different control strategies subject to HPP (homogeneous Poisson process) traffic, NHPP (non-homogeneous Poisson process) traffic, DSPP (doubly stochastic Poisson process) traffic and BC-pAug89 traffic respectively. In Section 6, we discuss the convergence of the cost function by using Monte Carlo Methods. Finally we summarize the numerical results in Section 7.

5.1 Implementation

In order to use a system approach to analyze network performance and to evaluate the effectiveness of our system, simulation experiments are needed. For simulation, the im-

plementation and some assumptions of the scenario must be introduced. The algorithm introduced in [33] is used to develop the traffic models. The implementation of this algorithm has been done in C++ programming language. We have also implemented our system model along with open loop controls and feedback controls algorithm in C++ programming language. For the implementation of fractional Brownian motion, Matlab has been used.

5.1.1 Assumptions

We will consider a simple scenario comprised of three traffic sources policed by three token buckets with their outputs (conforming packets) multiplexed by a single multiplexor for numerical simulation. For all the numerical experiments, some assumptions are made.

1. Each independent user generates the traffic with the same statistical characteristics and the same rate. Here statistical characteristics mean that all are either HPP traffic, or NHPP traffic, or DSPP traffic or BC-pAug89 traffic;

2. The length of each traffic trace is four seconds;

3. Four seconds are divided into 8000 time slots, whose length is $h = 0.0005$ second. At most one packet may be generated during one time slot for each independent user.

Based on this scenario and assumptions, the objective function and some of the performance measures, introduced in the preceding chapters, are computed for several different traffic.

5.1.2 System Parameters Used for Simulation

We consider the trade-off among three metrics: losses at the token buckets, losses at the multiplexor, and the cost associated with the waiting time. For some applications that

place extra emphasis on avoiding congestion and reducing the waiting time in the queue, it is preferable to drop packets at token buckets rather than at the multiplexor. Based on the above considerations, the relative weights are chosen in our numerical experiments as follows:

- $\alpha(t_k) = 10$, weights of the multiplexor losses for all t_k;
- $\beta_i(t_k) = 5$, weights of the losses at the token buckets for all i and all t_k ;
- $\gamma(t_k) = 1$, weights of the waiting cost for all t_k.

State initialization is set as :

- $\rho_i(t_0) = 0$, initial states of the token buckets for all i;
- $q(t_0) = 0$, initial state of the multiplexor.

System configuration and parameters are chosen in order to evaluate the system performance. We set the size of token bucket $T_i, i = 1, 2, 3$ to be the maximum packet size to guarantee accommodation of the maximum size. For SRD traffic, the maximum packet size is 1500 bytes and for LRD traffic, it is 1518 bytes. If the total (ingress) traffic exceeds the link capacity, the system is in the state of overload. We set the time average of HPP/NHPP traffic load to be 2 Mbps to 8.7 Mbps, and the time average of DSPP/Bellcore traffic to be 1.8 Mbps to 5 Mbps (the details will be presented later on). In general, the available link capacity may be limited. The link capacity can be variable, here we choose two constant values: $C = 5$ Mbps for HPP/NHPP traffic inputs and $C = 8$ Mbps for DSPP/Bellcore traffic inputs. If the link capacity C is too large, there are no losses. The buffer size of the multiplexor is chosen to accommodate three times maximum packet size approximately. Here we select two values, $Q = 4000$ bytes for HPP/NHPP traffic and $Q = 4554$ bytes for DSPP/Bellcore traffic. The length of time slot h is chosen to be sufficiently small so that the probability that more than one packet arrives during any

time slot is very small. By setting $h = 0.0005$ second, we guarantee this condition. The number of time slots is given by $K = 4/h$. In our experiment, we use the Monte Carlo method to compute expected values. For HPP, NHPP and DSPP traffic inputs, we choose $M = 1000$. For Bellcore traffic, we divide the first 8-minute trace into 120 (four-second) traces, therefore we can get $M = 40$. Details are shown in the following table.

Parameters	**HPP Traffic**	**NHPP Traffic**	**DSPP Traffic**	**BC-pAug89 Traffic**
$T_i, i = 1, 2, 3$	1500 Bytes	1500 Bytes	1518 Bytes	1518 Bytes
C	5 Mbps	5 Mbps	8 Mbps	8 Mbps
Q	4000 Bytes	4000 Bytes	4554 Bytes	4554 Bytes
h	0.0005 Sec	0.0005 Sec	0.0005 Sec	0.0005 Sec
K	8000	8000	8000	8000
M	1000	1000	1000	40
$e_i, i = 1, 2, 3$	3	3	3	3
Q_α	0.9*4000	0.9*4000	0.9*4554	0.9*4554

Table 5.1: System Configuration and Parameters

5.1.3 Specification of Traffic

Four different types of four-second (traffic) traces, which have different characteristics and mean rates, are chosen as our traffic sources. The first one is HPP traffic with different, but constant mean rates. The second one is NHPP traffic with a variable, but deterministic rate. The third one is DSPP traffic with an intensity function determined by a self-similar process. The last experiment is carried out with the Bellcore traffic trace (BC-pAug89) obtained from the Bellcore Morristown Research and Engineering facility [10]. In order to show the traffic traces clearly, we plot two-second traces as samples.

Homogeneous Poisson Traffic

We use HPP to generate six different traces with six different constant rates: 200, 300, 400, 500, 800 and 850 packets per second. We plot one of them as a sample shown in Figure 5.1. System response and performance results will be presented later on.

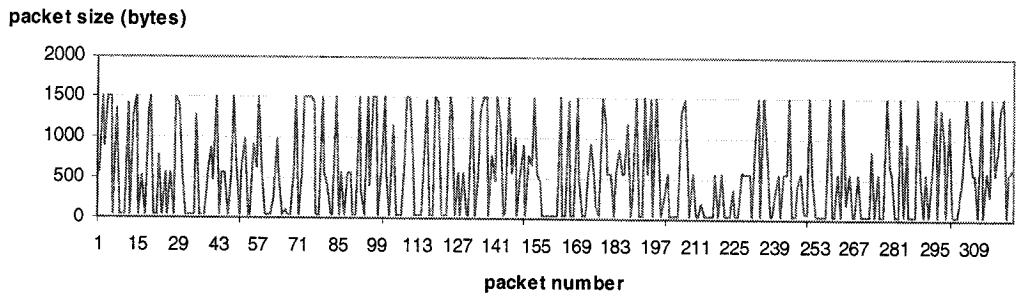


Figure 5.1: 2-Second HPP Traffic Trace

Nonhomogeneous Poisson Traffic

We choose four types of NHPP traffic with different time average of the intensity in simulation experiments. These traffic follow the (traffic rate's) curves shown in Figure 5.2. Table 5.2 shows the details of four traffic rates. One sample of NHPP traffic is shown in Figure 5.3.

Type of Traffic	1	2	3	4
Minimum Rate (packets/sec)	100	200	300	400
Maximum Rate (packets/sec)	300	400	500	600
Time Average Rate (packets/sec)	200	300	400	500

Table 5.2: NHPP Traffic Intensities

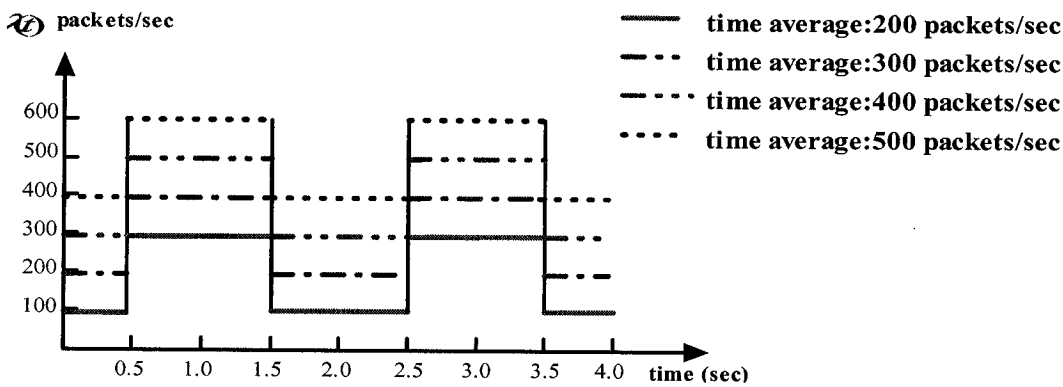


Figure 5.2: NHPP Traffic Intensities

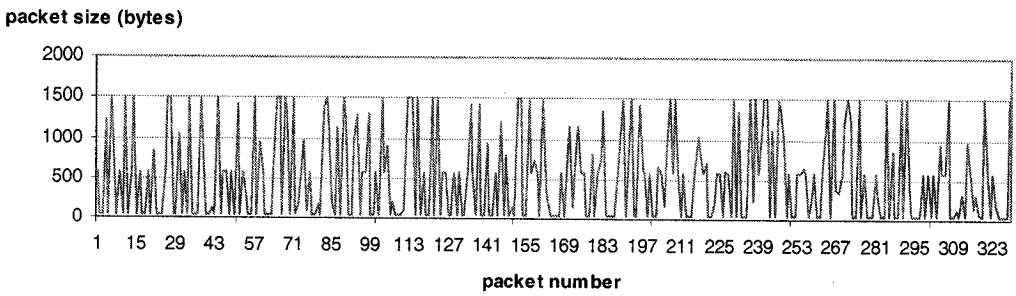


Figure 5.3: 2-Second NHPP Traffic Trace

Doubly Stochastic Poisson Traffic

Here we generate DSPP traffic traces by using the Hurst parameter, where $H = 0.795$ calculated from the first two-minute trace of BC-pAug89. We use the constant C_H appearing in the kernel K_H to define the FBM. In our experiment, we use two different values of C_H and two rules, which can determine the traffic intensity, to generate four types of DSPP traffic. The parameters are shown in Table 5.3.

Type of Traffic	1	2	3	4
H	0.795	0.795	0.795	0.795
C_H	96	96	164	164
λ_H	$B_H(t) \vee 0$	$\lvert B_H(t)\rvert$	$B_H(t) \vee 0$	$\lvert B_H(t)\rvert$
Time Average Rate	170	258	292	445

Table 5.3: DSPP Traffic Intensities

Figure 5.4 shows FBM, B_H with $C_H = 96$ (left) and B_H with $C_H = 164$ (right). Figure 5.5(a) shows intensity processes, $\{\lambda(t) = \{B_H(t) \vee 0\}, t \geq 0\}$, with $C_H = 96$ (type 1) and Figure 5.5(b) indicates intensity processes $\{\lambda(t) = \{B_H(t) \vee 0\}, t \geq 0\}$ with $C_H = 164$ (type 3). Figure 5.6(a) illustrates intensity processes $\{\lambda(t) = |B_H|, t \geq 0\}$ with $C_H = 96$ (type 2) and Figure 5.6(b) is $\{\lambda(t) = |B_H|, t \geq 0\}$ with $C_H = 164$ (type 4). Samples of DSPP traffic are shown in Figure 5.7(a) and (b).

Bellcore Traffic

The trace BC-pAug89, which was measured at the Bellcore Morristown Research and Engineering facility on an Ethernet, collected one million packets that are primarily local traffic mixed with all the traffic between Bellcore and Internet. The trace BC-pAug89 began at 11:25 on August 29, 1989, and lasted about 3142.82 seconds. This trace captured

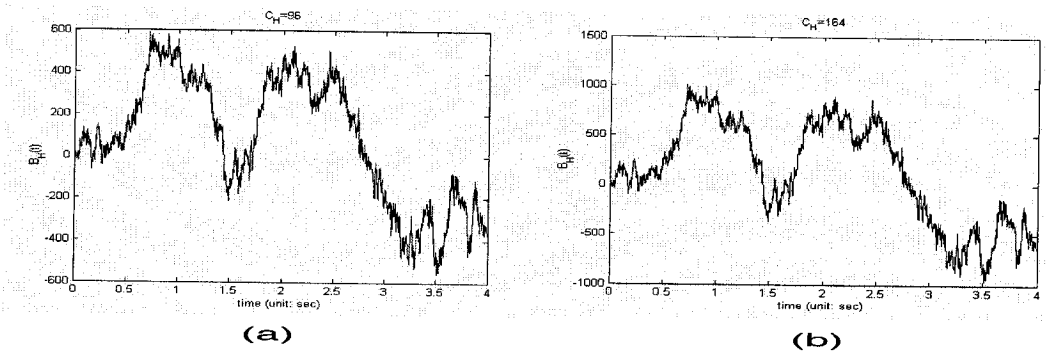


Figure 5.4: FBM $B_H(t)$(a): $C_H = 96$, b): $C_H = 164$)

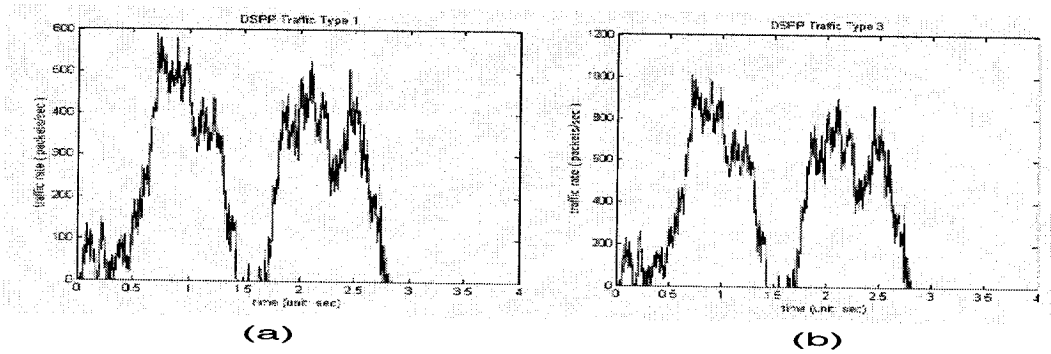


Figure 5.5: DSPP Intensity $\lambda(t) = B_H(t) \vee 0$ (a): type 1, b): type 3)

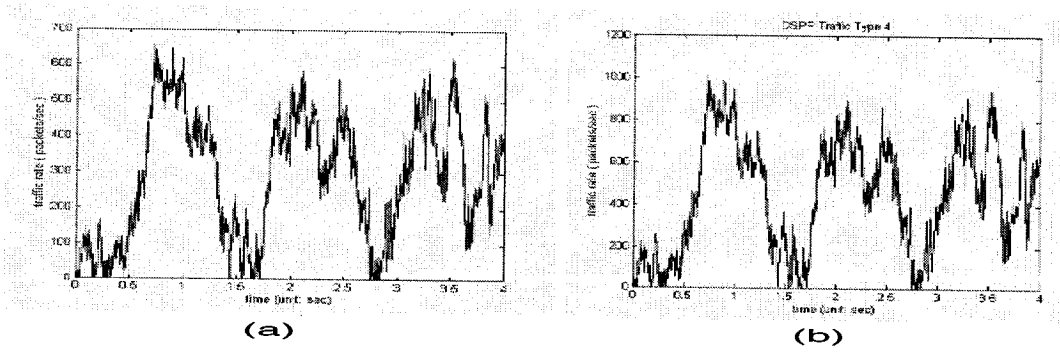


Figure 5.6: DSPP Intensity $\lambda(t) = |B_H(t)|$ (a): type 2, b): type 4)

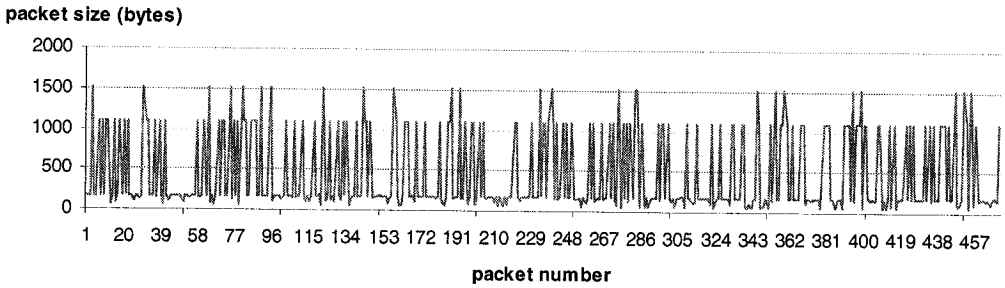


Figure 5.7: 2-Second DSPP Traffic Trace

all Ethernet packets whose size was from 64 bytes to 1518 bytes according to the Ethernet protocol [10]. For the purpose of our study, the first eight-minute trace is used. On the basis of the eight-minute traffic trace, we find that the average packet size is 445 bytes and the mean traffic rate is 356 packets per second. Note that we take the 4-second traffic trace as our traffic source, so we divide the 8-minute trace into 120 (four-second) traces and feed these traces into our system model to compute expected values of all performance measures. We only pick the two-second trace as sample to exhibit the BC-pAug89 trace, as shown in Figure 5.8.

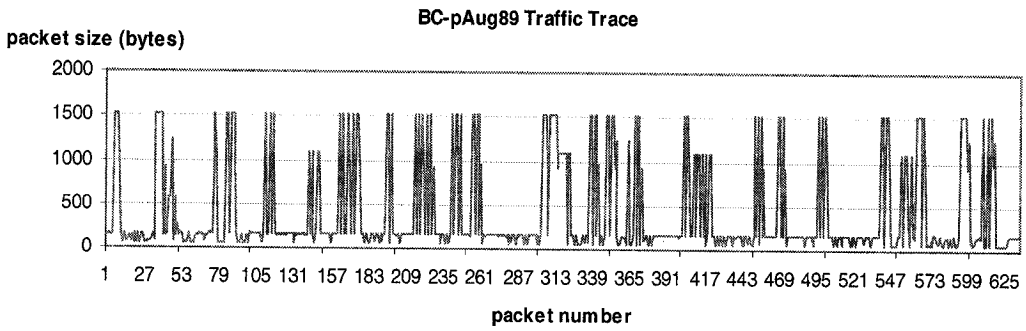


Figure 5.8: 2-Second BC-pAug89 Traffic Trace

5.2 Performance Evaluation based on HPP Traffic

In this section, we will study the dependence of the cost functional, packet losses, utilization, throughput and congestion on traffic rates subject to HPP traffic by carrying out the feedback control strategy without delay. And we will analyze the dependence of system performance on different control polices.

5.2.1 Dependence of Cost on Traffic Rates

The cost function $J(\lambda)$ is plotted as a function of the mean traffic arrival rate λ. Here all the three sources have the same mean arrival rate, though it is not at all necessary. Our results are plotted in Figure 5.9. As expected, the cost increases (in general nonlinearly) with an increasing volume of traffic.

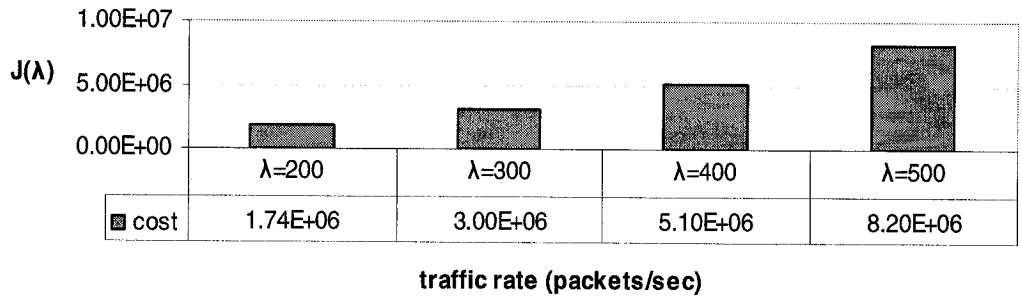


Figure 5.9: Dependence of Cost on HPP Traffic Rates

5.2.2 Dependence of System Losses on Traffic Rates

Figure 5.10 shows the system losses statistics caused by packet losses at the TBs and the cost due to waiting time in the multiplexor. Based on our choice of the feedback control law, packet losses at the multiplexor are completely eliminated. From Figure 5.10, it is clear that the losses at the TBs and waiting cost increase as the traffic increases.

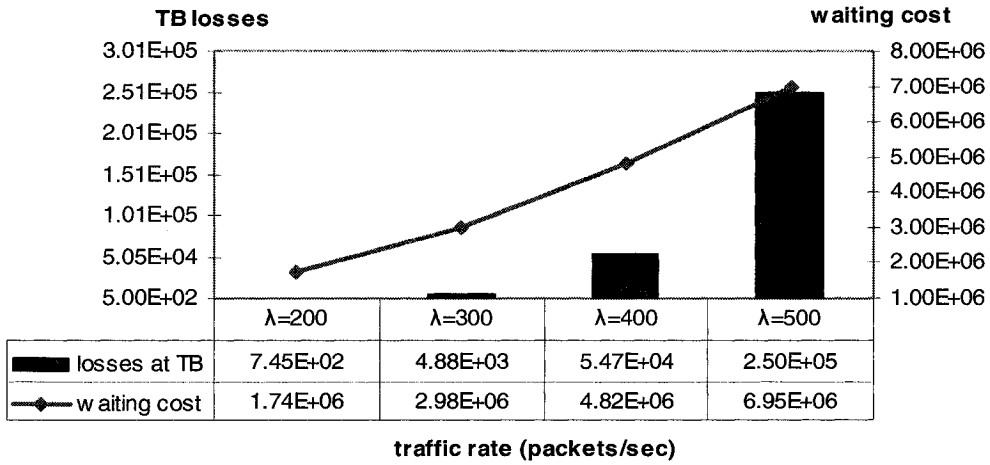


Figure 5.10: Dependence of Losses on HPP Traffic Rates

5.2.3 Dependence of Utilization and Throughput on Traffic Rates

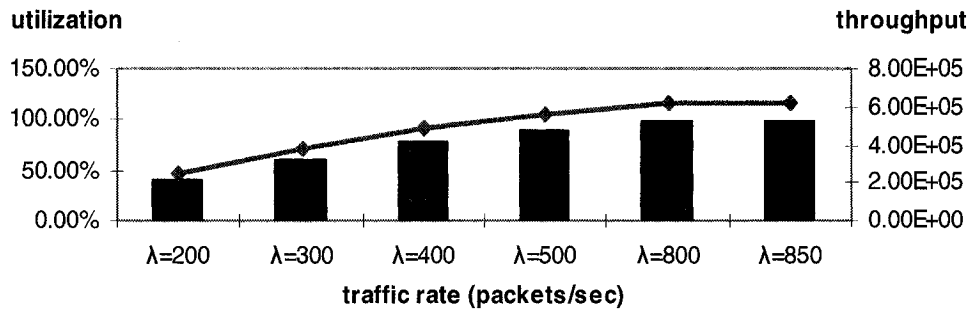


Figure 5.11: Dependence of Utilization and Throughput on HPP Traffic Rates

Dependence of utilization and throughput on HPP traffic rates is plotted in Figure 5.11. Clearly both the utilization and the throughput increase with increasing mean arrival rates. The utilization factor reaches 100 percent as the traffic volume reaches saturation level, which is defined as being the level of conforming traffic equal or greater than the link

capacity, causing saturation of the multiplexor buffer. But this is at the cost of excessive packet losses which lead to degradation of QoS (see Fig 5.10).

5.2.4 Dependence of Congestion on Traffic Rates

In our experiment, we choose $\alpha = 0.9$ and $Q = 4000$ bytes. We declare that the system is in the state of congestion when the buffer of the multiplexor is occupied over $\alpha Q = 3600$ bytes. Two graphs are shown in Figure 5.12. The curve that is monotonically increasing gives the expected (mean) number of times the system enters the state of congestion. The second graph, which is monotonically decreasing, gives the expected first time $E\{\tau_c\}$ when the congestion occurs. Clearly as the volume of traffic increases, these trends are expected. Note that these dependencies are non-linear.

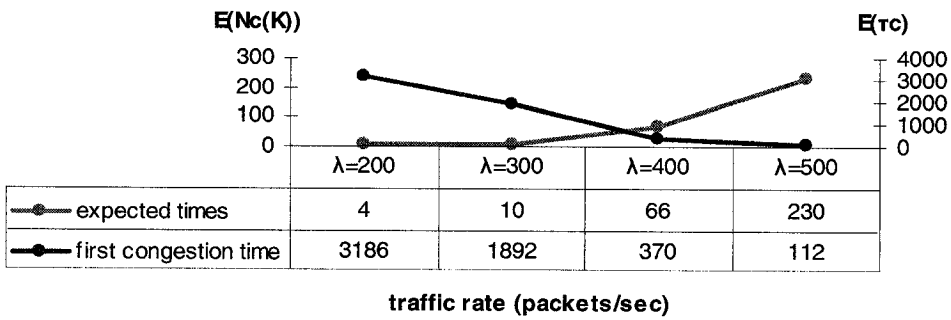


Figure 5.12: Dependence of System Congestion on HPP Traffic Rates

5.2.5 Dependence of Performance on Control Strategies

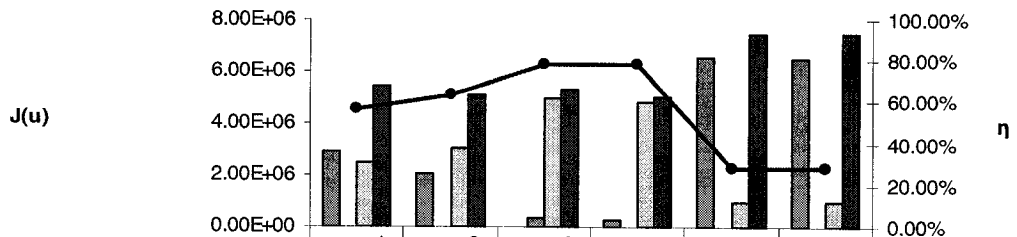
We will consider six cases for four different traffic types (HPP, NHPP, DSPP and Bellcore). The first three cases are for open loop controls with different control laws, and the last three are for feedback controls.

Let μ denote the time average of traffic intensity and μ_p the peak rate of traffic, which is

given by 1500 bytes per time slot for HPP and NHPP traffic or 1518 bytes for DSPP and Bellcore traffic.

- Case 1: open loop $u_i(t) = \mu$ for $i = 1, 2, 3$;
- Case 2: open loop $u_i(t) = 2\mu$ for $i = 1, 2, 3$;
- Case 3: open loop $u_i(t) = \mu_p$ for $i = 1, 2, 3$;
- Case 4: feedback control (no delay);
- Case 5: feedback control (delay 1 time slot);
- Case 6: feedback control (delay 2 time slot).

Here, we input HPP traffic with constant traffic rate 400 packets per second to demonstrate the dependence of system performance on control strategies.



	case 1	case 2	case 3	case 4	case 5	case 6
weighted losses at TB	2.93E+06	2.09E+06	0	2.44E+05	6.52E+06	6.48E+06
weighted losses at Mul	3.23E+03	5.93E+03	3.76E+05	0	4.00E+02	4.00E+02
weighted waiting cost	2.49E+06	3.01E+06	4.96E+06	4.81E+06	9.48E+05	9.59E+05
total cost J(u)	5.42E+06	5.10E+06	5.33E+06	5.05E+06	7.47E+06	7.44E+06
utilization	56.94%	63.67%	78.87%	78.27%	28.22%	28.49%

Figure 5.13: Dependence of Performance on Control Strategies based on HPP

Figure 5.13 shows the performance results for all the six cases above. Case 1 corresponds to the open loop control with control given by traffic mean rate; Case 2 corresponds to the open loop control set equal to two times the mean rate; and Case 3 corresponds to the open loop control set equal to the peak rate, say $\mu_p = 1500$ bytes per unit time slot. Case 4 follows a feedback control law without delay as given by the expression 3.13 and Cases 5 and 6 correspond to a feedback control law with one and two time slot delay respectively as defined as 3.14

The result shows that Case 4 offers the minimum cost. Cases 1-3 are worse. Cases 5 and 6 are the worst of all. The reason for this result is that the open loop strategies do not take into account the demands of the system (blind control). Cases 1 and 2 are too conservative to allow sufficient traffic into the multiplexor and Case 3 is too aggressive so that too much traffic is admitted into the system but the multiplexor fails to process it. The feedback control strategy with delay (even for one time slot) (Cases 5 and 6) can not acquire the current information required to make an allocation decision based on demands. Its current allocation is based on the past state information which, for discrete event systems like ours, may be drastically different causing substantial performance degradation. Here a predictor is necessary which may provide a good estimate of future (one or two time slots ahead) demand useful for control computation. Feedback control without delay seems to be better than other strategies.

From the results corresponding to open loop controls, Cases 1-3 of Figure 5.13, we see that losses at the token buckets decrease while those at the multiplexor increase as the token generating rate u increase. This is because larger token generating rate creates less nonconforming traffic and hence reduces TB losses. But on the other hand, this also brings in more traffic to the multiplexor, thereby increasing the probability of dropping packets there and increasing waiting time. This causes the increase of losses in the multiplexor and the cost associated with waiting time. Note that both Case 3 and Case 4 have almost

identical utilization, but the cost, corresponding to Case 3 is higher than that of Case 4. This is because the open loop policy causes packet losses at the multiplexor (see table in Figure 5.13) which is not desirable in the case of traffic such as video.

5.2.6 Dependence of Performance on Network Provider's Arbitration

The factor e_i, appearing in the expression 3.11 which is used to construct the feedback control law given by 3.13, can be used by the network provider to control network resource sharing and to avoid monopoly from any of the users. The factor e_i is the factor of network provider's arbitration.

Dependence of Packet Losses on e_i

For numerical experiment, only three token buckets are considered. So we can get twenty seven combinations of arbitration given in Table 5.4. The network provider can exercise twenty seven combinations of arbitration to control the token rate according to the feedback control law 3.13.

combination No.	1	2	3	4	5	6	7	8	9
e_1, e_2, e_3	1,1,1	1,1,2	1,1,3	1,2,1	1,2,2	1,2,3	1,3,1	1,3,2	1,3,3
combination No.	10	11	12	13	14	15	16	17	18
e_1, e_2, e_3	2,1,1	2,1,2	2,1,3	2,2,1	2,2,2	2,2,3	2,3,1	2,3,2	2,3,3
combination No.	19	20	21	22	23	24	25	26	27
e_1, e_2, e_3	3,1,1	3,1,2	3,1,3	3,2,1	3,2,2	3,2,3	3,3,1	3,3,2	3,3,3

Table 5.4: Twenty Seven Combinations of Arbitration

Here again, we input the HPP traffic with a constant traffic rate of 400 packets per second

to demonstrate the influence of network provider arbitration. The result plotted in Figure 5.14 shows the mean packet losses as function of the 27 combinations. It is clear from the Figure 5.14 that the network provider can exercise substantial control on the performance of the system. By observing combinations 14,15,17,18,23,24, 26 and 27, in which $e_i \geq 2$ for all $i = 1, 2, 3$, we see that the packet losses are minimum. This result is based on the conditions that all the users have same traffic rate and the link capacity can serve all traffic load, where link capacity is 5Mbps and mean traffic load of system is 4Mbps. Therefore, packet losses will be smaller when the network provider puts less limitation. When the network provider imposes on any one of three users more limitation by adjusting the factor e_i, the packet losses will increase. For example, when the network provider sets $e_i = 1, i = 1, 2, 3$ (see combination 1), the packet losses in this case are much larger than those in other cases.

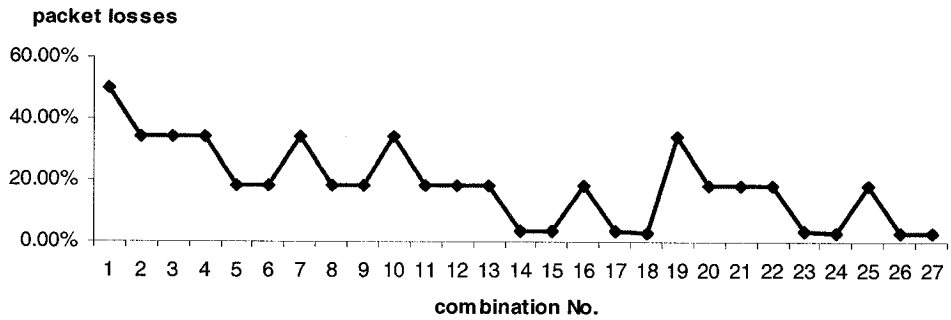


Figure 5.14: Dependence of Packet Losses on e_i based on HPP

Monopoly Avoidance by Network Provider

Now we discuss how to prevent a user from hogging the system resources. When resources are shared according to a demand, and the feedback control law provides the proportional fraction of required demand to each user. In this situation, any one of users can increase his demand and capture more bandwidth thereby starving other users. To avoid this, the

network provider can make use of the arbitration factor $\{e_i\}$.

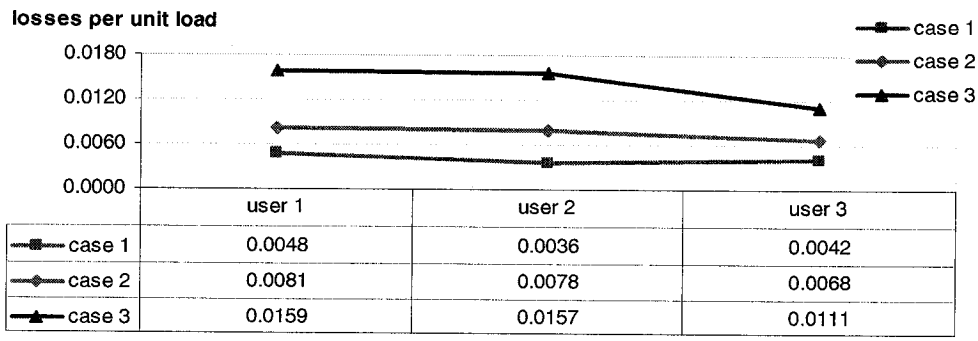
Let λ_1, λ_2 and λ_3 be the traffic rates of the users 1, 2 and 3 respectively. To consider the change of losses per unit (traffic) rate of each user by adjusting the factor e_i, we study three cases, which have different traffic inputs.

- Case 1: $\lambda_1 = 300, \lambda_2 = 300, \lambda_3 = 300$;
- Case 2: $\lambda_1 = 300, \lambda_2 = 300, \lambda_3 = 400$;
- Case 3: $\lambda_1 = 300, \lambda_2 = 300, \lambda_3 = 500$;

In Case 1, all the three users have the same mean rate. In Case 2, the traffic rate of User 3 is increased to 400 packets per second, and in Case 3, it is increased to 500 packets per second.

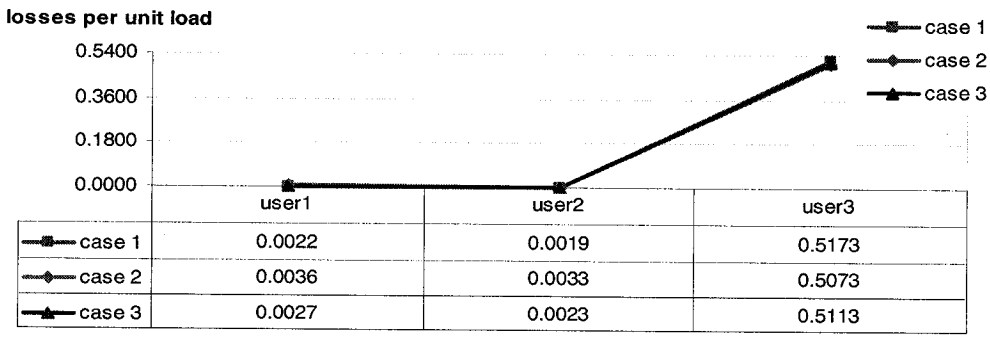
Figure 5.15 illustrates the change of losses per unit (traffic) load of each user with free control, say $e_i = 3$, for $i = 1, 2, 3$. The result shows that the losses per unit load of users 1 and 2 increase as the user 3 increases its volume of traffic. This is because the network provider gives all users free control, the feedback control law allocates resources proportional to the demands (see Equation 3.11). When the user 3 increases the volume of traffic, it grabs more resources starving users 1 and 2.

From Figure 5.15 we see that the losses per unit load for user 3 will also increase a little as the user 3 increases the volume. In order to control this situation, the network provider should adjust the factor e_3 of User 3 by setting $e_3 = 1$. The result is plotted in Figure 5.16. It shows that the losses per unit load of each user remain the same as the user 3 increases its volume of traffic. This is because by decreasing the factor e_3, we prevent User 3 from grabbing the system resources from other users by simply increasing its volume of traffic. Clearly the losses of users increase because it pushes more traffic in without success.



	user 1	user 2	user 3
case 1	0.0048	0.0036	0.0042
case 2	0.0081	0.0078	0.0068
case 3	0.0159	0.0157	0.0111

Figure 5.15: Losses per Unit Load on Free Control (3,3,3) based on HPP



	user1	user2	user3
case 1	0.0022	0.0019	0.5173
case 2	0.0036	0.0033	0.5073
case 3	0.0027	0.0023	0.5113

Figure 5.16: Losses per Unit Load on Control User3 (3,3,1) based on HPP

5.2.7 Dependence of Losses at TB on TB Capacity

Figure 5.17 indicates the dependence of losses on the size of TB for the given feedback control law. We notice that the losses at TBs decrease as the TB capacity increases and once the size of TB reaches a critical value given by the maximum size of packet, which is 1500 bytes here, the cost converges to a certain constant value. It is clear that when the

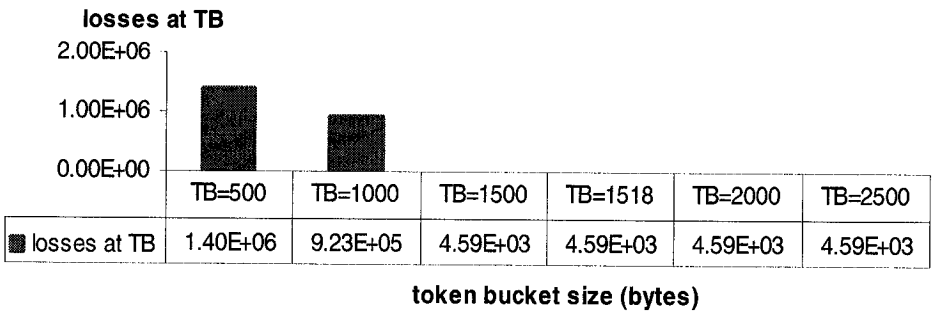


Figure 5.17: Dependence of Cost on TB Capacity based on HPP

size of TBs is too small, those packets with size greater than those of TBs can not pass through and hence will be dropped. Thus in this case the losses at TBs are higher. As we increase the size of TBs beyond the maximum packet, losses at TBs becomes independent of the size of TBs. For example, in our experiment, only 5% and 25% traffic can possibly pass through TBs by setting $T = 500$ and $T = 1000$ bytes according to the packet size distribution of HPP traffic. Therefore most of the traffic is dropped causing more losses. As we increase the size of TBs beyond the maximum packet size, 100% traffic can possibly pass through TBs, thus losses reduce and become independent of the size of TBs.

5.3 Performance Evaluation based on NHPP Traffic

In this section, we will evaluate the system performance by feeding NHPP traffic with different traffic mean rates. We will discuss how the control strategies influence the system

performance based on NHPP traffic.

5.3.1 Dependence of Cost on Mean Traffic Rates

Here we input the same time average of traffic rate into the three sources. The result shown in Figure 5.18 is similar to that for HPP. The cost increases as traffic increase. Comparing Figure 5.9, corresponding to HPP, we can see that the cost of NHPP is higher than that of HPP given the same mean rate. It is because that NHPP traffic is more bursty than HPP traffic when they have the same mean rates. For example, for NHPP traffic, the time average of traffic rate is equal to 300 packets per second, but the rate is changing between 200 and 400 packets per second.

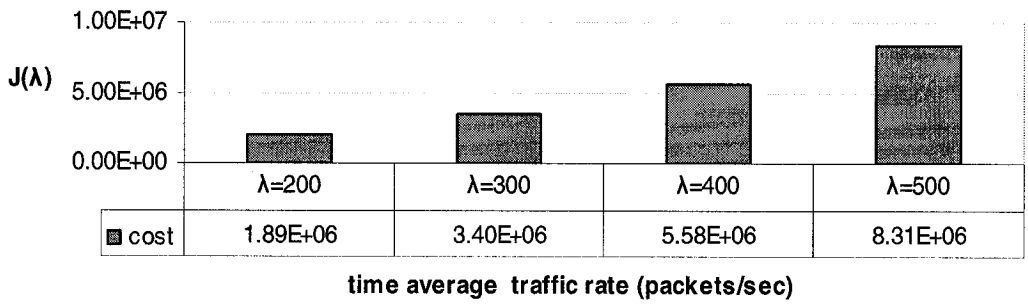


Figure 5.18: Dependence of Cost on NHPP Mean Traffic Rates

5.3.2 Dependence of System Losses on Mean Traffic Rates

The system losses caused by packet losses at the TBs and the cost due to waiting time in the multiplexor are plotted in Figure 5.19. Again, the results show that the losses at the TBs and waiting cost increase as the traffic volume increases.

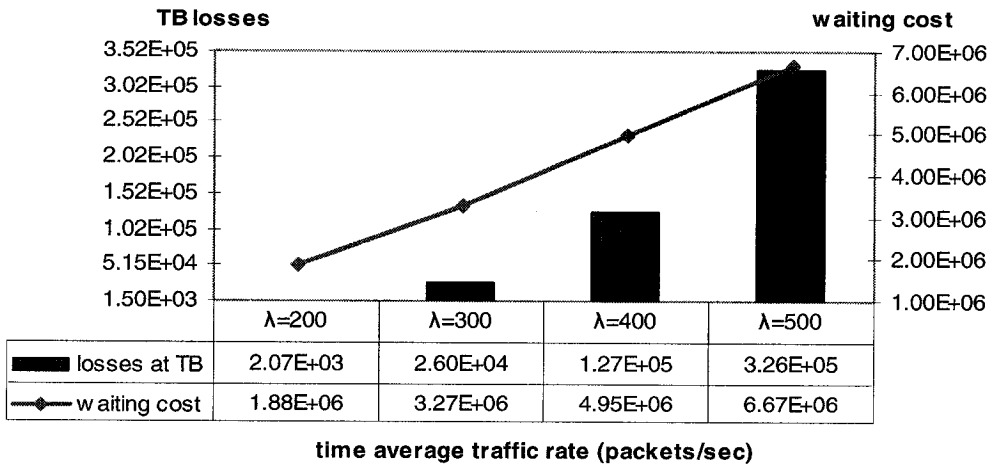


Figure 5.19: Dependence of Losses on NHPP Mean Traffic Rates

5.3.3 Dependence of Utilization and Throughput on Mean Traffic Rates

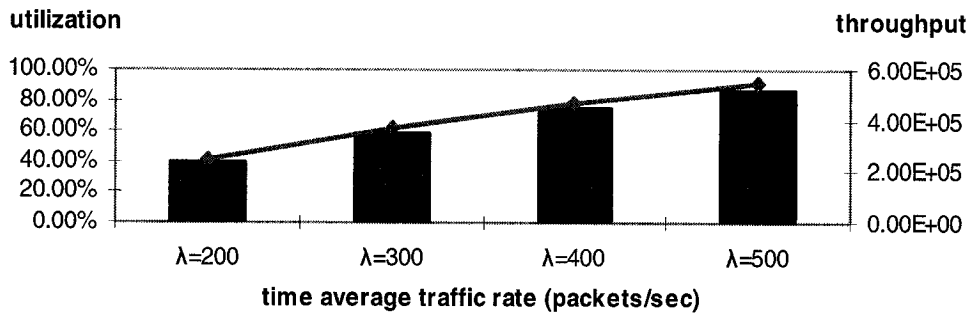


Figure 5.20: Dependence of Utilization and Throughput on NHPP Mean Traffic Rates

Figure 5.20 shows that both the utilization and the throughput increase with increasing mean arrival rates based on NHPP traffic.

5.3.4 Dependence of Congestion on Mean Traffic Rates

System congestion dependent on time average of NHPP traffic rate is plotted in Figure 5.21. The result is similar to that in HPP case. As the volume of traffic increases, the expected number of times the system enters the state of congestion monotonically increases while the mean time where the first congestion occurs monotonically decreases. Comparing HPP (see Figure 5.12) with NHPP (see Figure 5.21), we can see that the first congestion occurs earlier for NHPP traffic and the number of congestion times for NHPP traffic is more than that for HPP traffic based on the same mean rates. This is because the NHPP traffic chosen in this experiment, is more bursty than HPP traffic though their mean rates are equal.

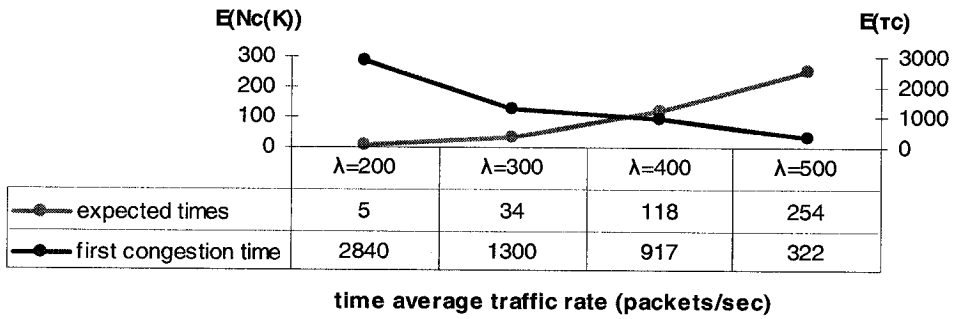
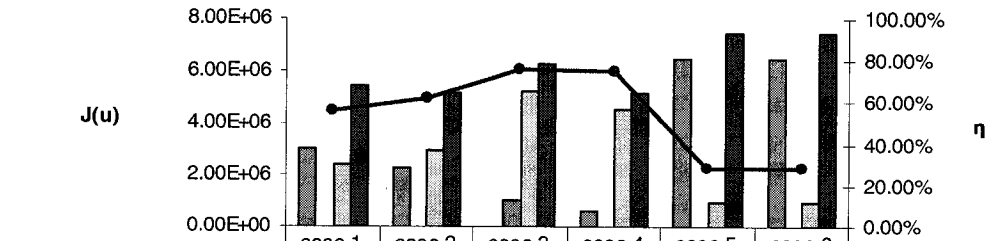


Figure 5.21: Dependence of Congestion on NHPP Mean Traffic Rates

5.3.5 Dependence of Performance on Control Strategies

As in HPP case, we choose the same six cases and input NHPP traffic, whose time averaged traffic rate is 400 packets per second.

Figure 5.22 shows the performance results for all the six cases based on NHPP traffic. The results also show that feedback control without delay (Case 4) does a good job. It obtains the best cost, no losses at the multiplexor, acceptable losses at TBs and waiting



	case 1	case 2	case 3	case 4	case 5	case 6
weighted losses at TB	3.04E+06	2.26E+06	0	6.33E+05	6.50E+06	6.47E+06
weighted losses at Mul	3.64E+03	5.09E+03	1.04E+06	0	1.50E+02	1.50E+02
weighted waiting cost	2.44E+06	2.94E+06	5.26E+06	4.55E+06	9.58E+05	9.73E+05
total cost J(u)	5.48E+06	5.20E+06	6.29E+06	5.18E+06	7.46E+06	7.44E+06
utilization	55.89%	62.13%	76.13%	75.18%	28.20%	28.51%

Figure 5.22: Dependence of Performance on Control Strategies based on NHPP

cost. The open loop controls are either too conservative (Cases 1-2) to cause too many losses at TBs or too aggressive (Case 3) to cause excessive losses in the multiplexor though its utilization is a little bit higher than that of Case 4. The time delay in feedback control policy causes a significant degradation of performance. Cases 5 and 6 are the worst. The results are similar to those in HPP case, but the results related to HPP are a little bit better than those in NHPP. This is because that HPP traffic has constant rate, but NHPP traffic has variable rate, though their mean rates are equal.

5.3.6 Dependence of Performance on Network Provider's Arbitration

Here we study how the arbitration factor e_i influence system performance by feeding NHPP traffic with the time average of traffic rate equal to 400 packets per second.

Dependence of Packet Losses on e_i

For our experiment, there are 27 combinations given in Table 5.4. We plot the mean packet losses corresponding to the 27 combinations shown in Figure 5.23. From the figure it is clear that the network provider can exercise substantial control on packet losses by choosing different factors $\{e_i\}$. This result is similar to the our obtained for HPP case.

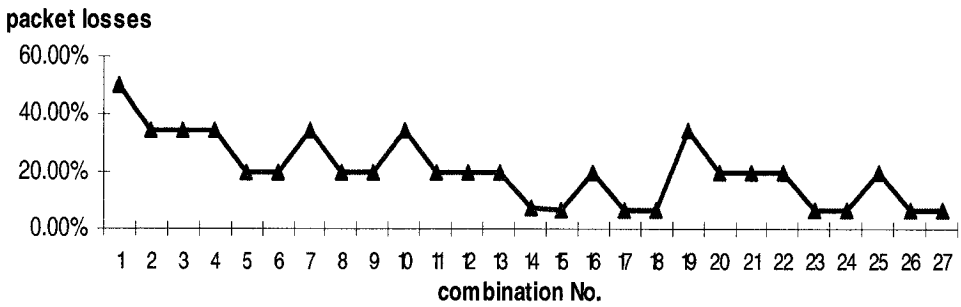
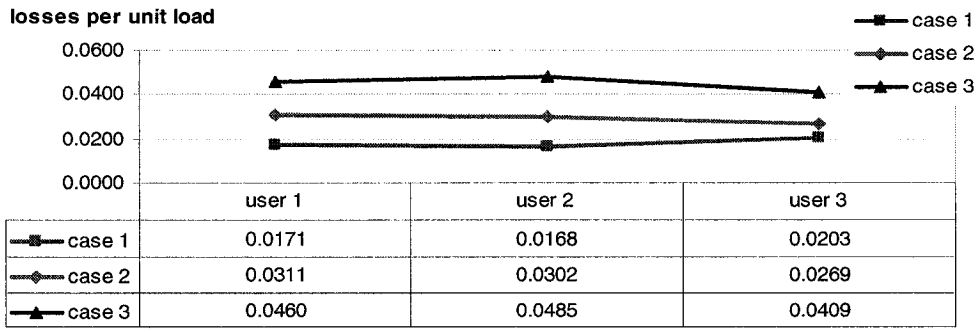


Figure 5.23: Dependence of Packet Losses on e_i based on NHPP

Monopoly Avoidance by Network Provider

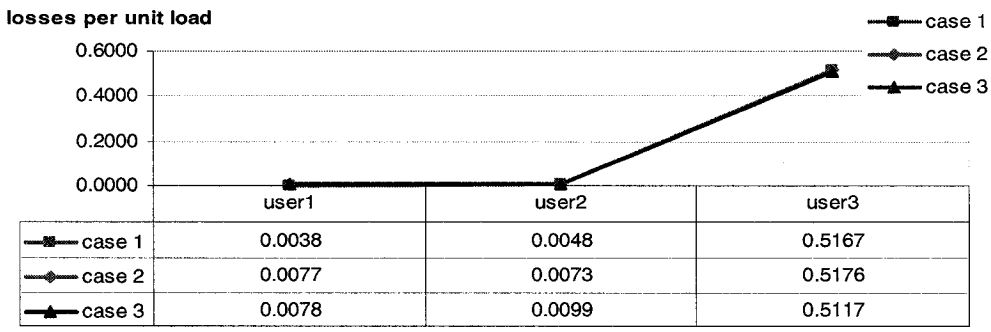
As in the HPP case, in order to prevent a user from hogging the system resource, the network provider also uses the factor e_i in sharing system resources. We also consider three cases: Case 1, all the three users have the same time averaged rates which is 300 packets per second; Case 2, the traffic rate of User 3 is increased to 400 packets per second, and Case 3, it is increased to 500 packets per second.

Figure 5.24 shows the results of losses per unit load of each of the users without the network provider's control, which has combination (3,3,3). Figure 5.25 illustrates the results after imposing control on User 3 by decreasing the factor e_3, giving the combination (3,3,1).



	user 1	user 2	user 3
case 1	0.0171	0.0168	0.0203
case 2	0.0311	0.0302	0.0269
case 3	0.0460	0.0485	0.0409

Figure 5.24: Losses per Unit Load on Free Control (3,3,3) based on NHPP



	user1	user2	user3
case 1	0.0038	0.0048	0.5167
case 2	0.0077	0.0073	0.5176
case 3	0.0078	0.0099	0.5117

Figure 5.25: Losses per Unit Load on Control User3 (3,3,1) based on NHPP

5.3.7 Dependence of Losses at TB on TB Capacity

Figure 5.26 shows the dependence of losses at TB on the size of TB following the feedback control law (no delay). The result is similar to that in HPP case (see Figure 5.17).

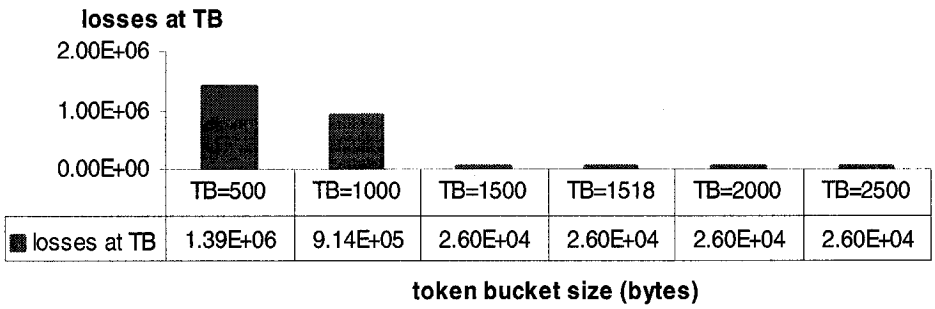


Figure 5.26: Dependence of Cost on TB Capacity based on NHPP

5.4 Performance Evaluation based on DSPP Traffic

In this section, we will study the system response by inputting DSPP traffic with different traffic rates. Again, we will analyze the system performance under different control strategies.

5.4.1 Dependence of Cost on Mean Traffic Rates

For DSPP traffic, the cost depends not only on the volume of traffic but also on the traffic intensity function. Here we use four types of DSPP traffic with different mean rates, shown in Table 5.3. The time average of traffic intensities are used in our experiment as

follows:

$$\lambda_1 = \frac{1}{T}\int_0^T \{B_H(t) \vee 0\}\, dt = 170, C_H = 96$$
$$\lambda_2 = \frac{1}{T}\int_0^T \{|B_H(t)|\}\, dt = 258, C_H = 96$$
$$\lambda_3 = \frac{1}{T}\int_0^T \{B_H(t) \vee 0\}\, dt = 292, C_H = 164$$
$$\lambda_4 = \frac{1}{T}\int_0^T \{|B_H(t)|\}\, dt = 445, C_H = 164$$

The result is plotted in Figure 5.27. Note that the traffic intensity functions of λ_1 and λ_3 are similar, but the values of C_H, which can influence the amplitude of FBM, are different. It is obvious that the cost corresponding to λ_3 is larger than that corresponding to λ_1. By observing the results related to λ_1 and λ_2, we notice that the cost related to λ_1 is better than that to λ_2. It is understood that $B_H \vee 0$ is not larger than $|B_H|$ by giving the same C_H.

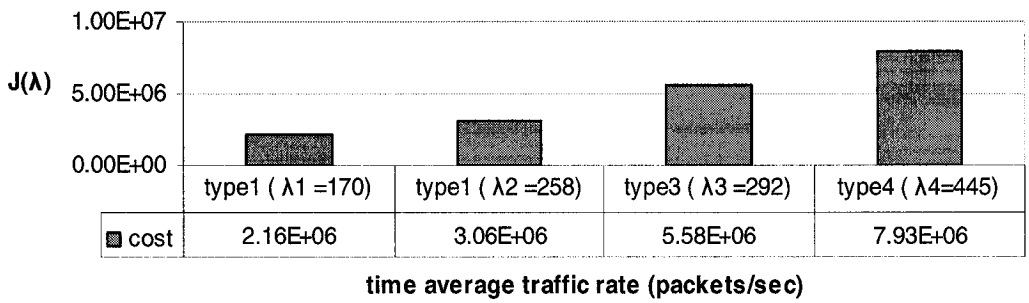


Figure 5.27: Dependence of Cost on DSPP Mean Traffic Rates

5.4.2 Dependence of System Losses on Mean Traffic Rates

Here again, we use the feedback control law (no delay) to eliminate the losses at the multiplexor. Then the system losses only include losses at TBs and the waiting cost. The

result in Figure 5.28 shows that the volume of traffic and the traffic intensity function determine the system losses.

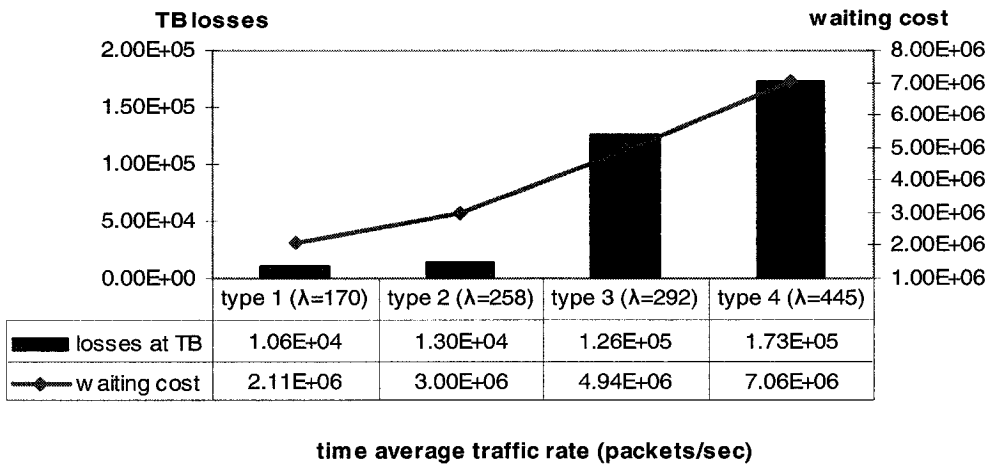


Figure 5.28: Dependence of Losses on DSPP Mean Traffic Rates

5.4.3 Dependence of Utilization and Throughput on Mean Traffic Rates

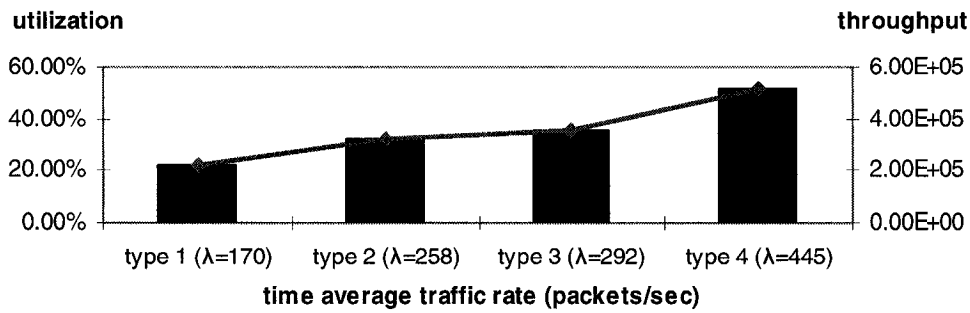


Figure 5.29: Dependence of Utilization and Throughput on DSPP Mean Traffic Rates

Figure 5.29 shows that both the utilization and the throughput increase as the volume of traffic increases.

5.4.4 Dependence of Congestion on Mean Traffic Rates

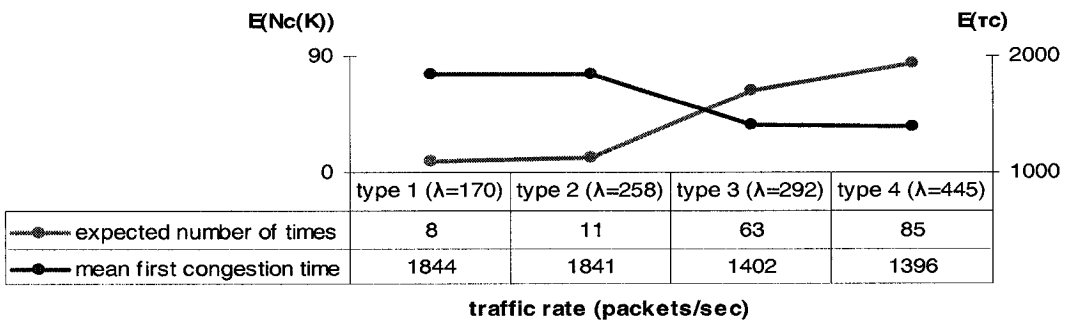
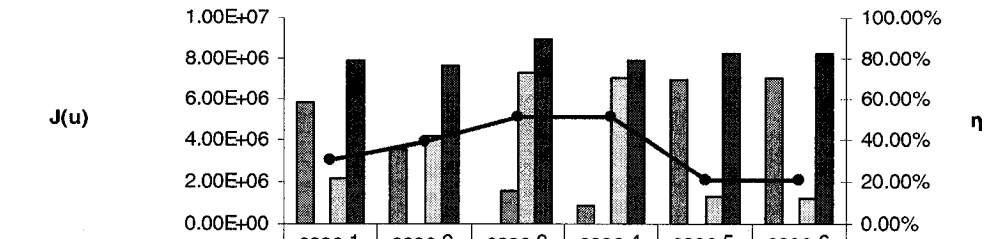


Figure 5.30: Dependence of System Congestion on DSPP Mean Traffic Rates

Here we study the system congestion corresponding to DSPP traffic. The result is illustrated in Figure 5.30. It shows that the expected number of times the system enters the state of congestion increase as the volume of traffic increases. The mean time when the first congestion occurs in Type 1 will be similar as in Type 2. This is because their traffic intensities about the first three seconds are exactly same. The trend of Type 2 is quite similar to that of Type 4 but the later is more bursty than the former. Therefore we can compare them. It is clear that the first hitting time decreases as the volume of traffic increases.

5.4.5 Dependence of Performance on Control Strategies

In order to study dependence of system performance according to different control polices, we consider the six cases same as in HPP case. We use DSPP traffic with the time averaged traffic intensity which is equal to 445 packets per second.



	case 1	case 2	case 3	case 4	case 5	case 6
weighted losses at TB	5.79E+06	3.53E+06	0	8.65E+05	6.94E+06	7.04E+06
weighted losses at Mul	0	0	1.57E+06	0	1.00E+02	0
weighted waiting cost	2.16E+06	4.15E+06	7.34E+06	7.06E+06	1.31E+06	1.24E+06
total cost J(u)	7.95E+06	7.69E+06	8.92E+06	7.93E+06	8.25E+06	8.28E+06
utilization	30.21%	39.53%	51.73%	51.35%	20.97%	20.47%

Figure 5.31: Dependence of Performance on Control Strategies based on DSPP

Figure 5.31 shows the performance results for all the six cases based on DSPP traffic. It is very clear that Case 4 corresponding to the feedback control without delay obtains the best cost and acceptable losses at TB and delay though the network utilization in Case 4 is a little bit lower than that in Case 3. Because the higher losses at the multiplexor in Case 3 is not acceptable at all.

5.4.6 Dependence of Performance on Network Provider's Arbitration

Here we choose DSPP traffic with the time average of traffic rate equal to 445 packets per second to study how the network provider influence the system performance and how to control the shared system by using the arbitration factor e_i.

Dependence of Packet Losses on e_i

The dependence of packet losses on arbitration factor e_i is shown in Figure 5.32. It is

understood that the packet losses where the network provider gives less control is less than those where the network provider gives more control. This is due to the fact that the link capacity is larger than the traffic load so that free control can make more traffic pass successfully.

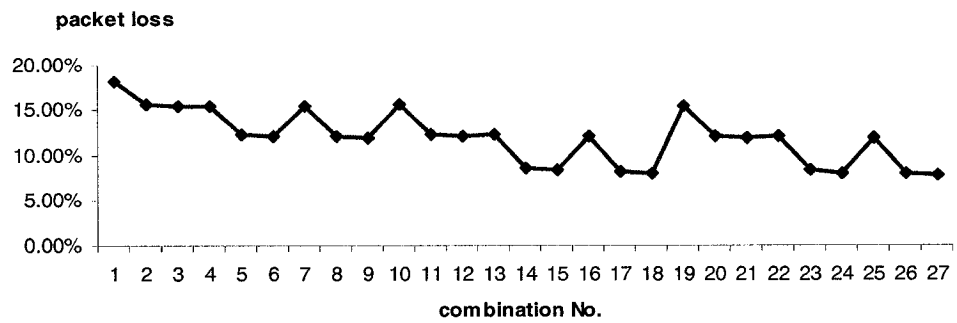
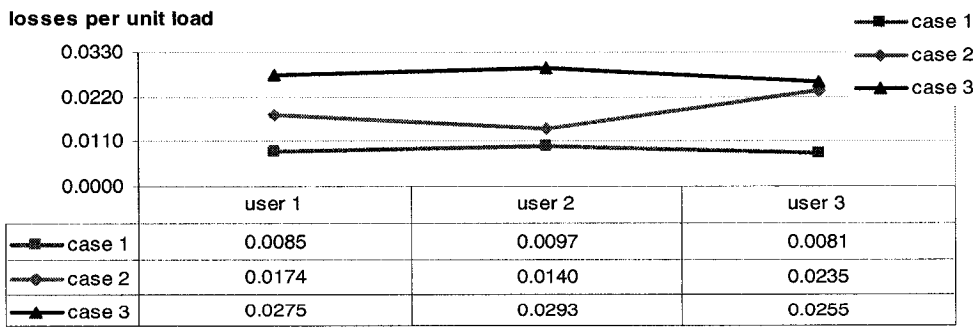


Figure 5.32: Dependence of Packet Losses on e_i based on DSPP

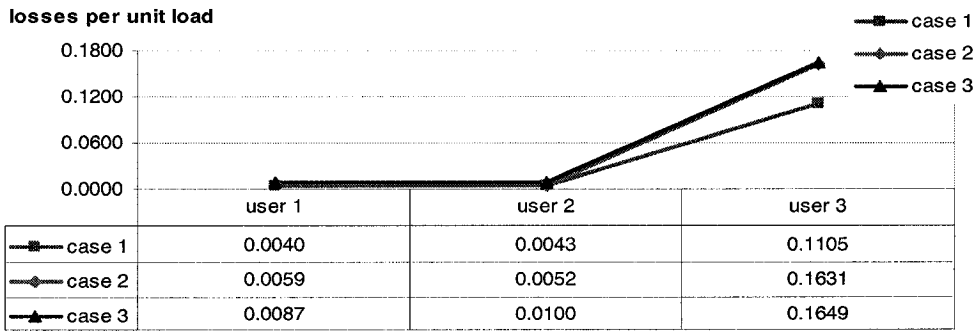
Monopoly Avoidance by Network Provider

Figure 5.24 shows the result where the network provider gives each user free control and Figure 5.25 illustrates the avoidance of the monopoly of User 3 by using the factor e_i.



	user 1	user 2	user 3
case 1	0.0085	0.0097	0.0081
case 2	0.0174	0.0140	0.0235
case 3	0.0275	0.0293	0.0255

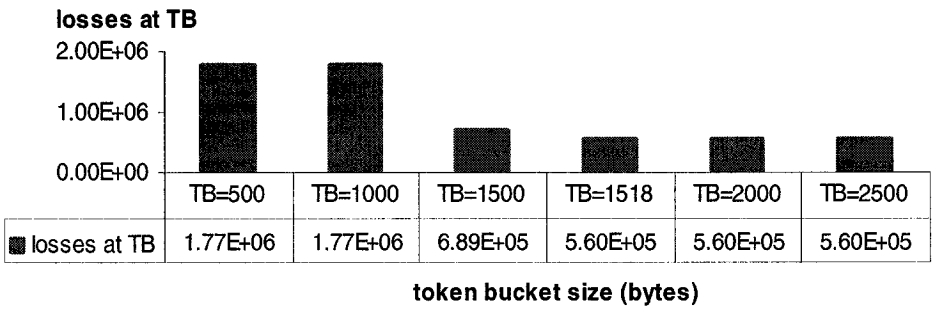
Figure 5.33: Losses per Unit Load on Free Control (3,3,3) based on DSPP



	user 1	user 2	user 3
case 1	0.0040	0.0043	0.1105
case 2	0.0059	0.0052	0.1631
case 3	0.0087	0.0100	0.1649

Figure 5.34: Losses per Unit Load on Control User3 (3,3,1) based on DSPP

5.4.7 Dependence of Losses at TB on TB Capacity



	TB=500	TB=1000	TB=1500	TB=1518	TB=2000	TB=2500
losses at TB	1.77E+06	1.77E+06	6.89E+05	5.60E+05	5.60E+05	5.60E+05

Figure 5.35: Dependence of Cost on TB based on DSPP

As in HPP case, here we also choose the system parameters $C = 5$ Mbps and $Q = 4000$ bytes for DSPP case. We follow the feedback control strategy (no delay). Dependence of losses on TB capacity is plotted in Figure 5.35. In general, the result is similar to that in the HPP case, that is, the losses at TBs decrease as the TB capacity increases and when the size of TB reaches the critical value–the maximum packet size, the losses at TBs converge to a stable value.

There are some differences between the DSPP and HPP cases. The first difference is the critical value. It is 1500 bytes for HPP case while it is 1518 bytes for DSPP case. Another difference is the losses for $T = 500$ and $T = 1000$ bytes. For HPP case, the losses for $T = 1000$ bytes is smaller than those for $T = 500$. However, they are same for DSPP case. This is because that HPP and DSPP traffic have different packet size distributions. For HPP case, between 500 and 1000 bytes, it has packets with size 576 bytes. And for DSPP case, there is no such kind of packets.

5.5 Performance Evaluation based on Bellcore Traffic

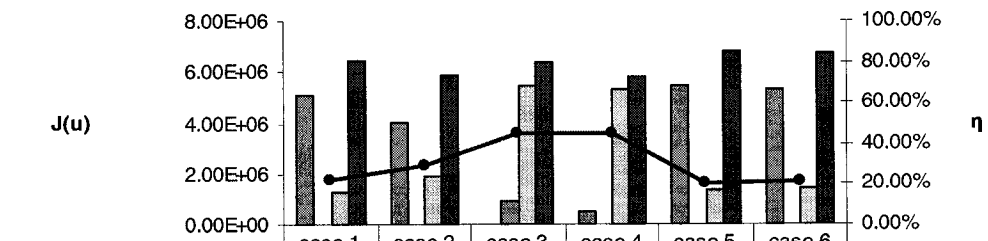
In this section, we will study the dependence of performance on control strategies by feeding BC-pAug89 traffic, and then we will discuss the dependence of the objective function on TB's size and the network parameters such as link capacity C, buffer size Q of the multiplexor by using only the feedback (without delay) control policy given by Equations 3.11- 3.13. And then we elaborate the dependence of cost on the relative weights and convergence of cost using Monte Carlo technique.

5.5.1 Dependence of Performance on Control Strategies

Figure 5.36 shows the performance results for all the six cases based on Bellcore traffic. The results are similar to those of DSPP case.

5.5.2 Dependence of Performance on Network Provider's Arbitration

Figure 5.37 illustrates that the packet losses is a function of the control factor e_i. The result is the same as that in HPP, NHPP and DSPP cases.



	case 1	case 2	case 3	case 4	case 5	case 6
weighted losses at TB	5.11E+06	4.00E+06	0	4.90E+05	5.47E+06	5.34E+06
weighted losses at Mul	0	0	9.02E+05	0	0	0
weighted waiting cost	1.31E+06	1.91E+06	5.45E+06	5.28E+06	1.33E+06	1.40E+06
total cost J(u)	6.42E+06	5.91E+06	6.35E+06	5.77E+06	6.81E+06	6.75E+06
utilization	22.09%	28.96%	45.36%	45.17%	20.26%	20.90%

Figure 5.36: Dependence of Performance on Control Strategies based on Bellcore

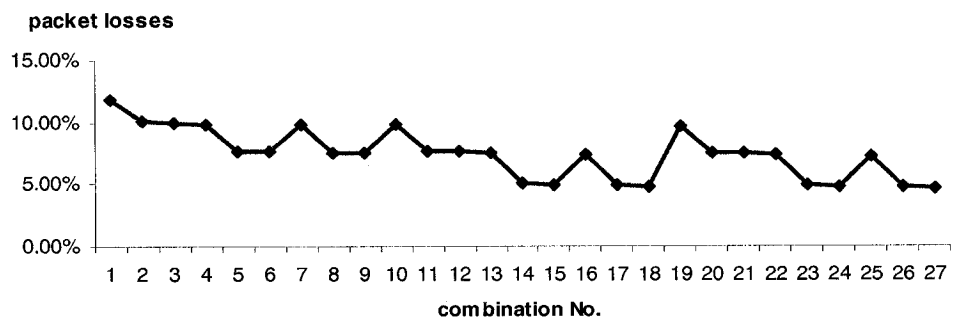


Figure 5.37: Dependence of Packet Losses on e_i based on Bellcore

5.5.3 Dependence of Losses at TB on TB Capacity

Dependence of cost on TB capacity is plotted in Figure 5.38. The parameters and control policy used in Bellcore case are the same as those in DSPP case. In addition, the packet size distribution of DSPP traffic are almost same as that of Bellcore traffic used in this experiment. Therefore, the results of these two cases are similar.

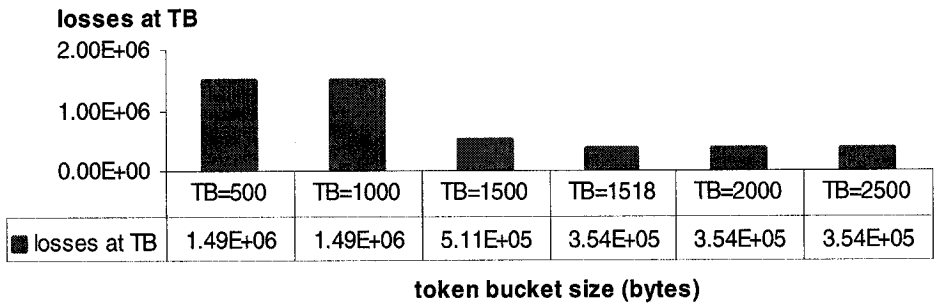


Figure 5.38: Dependence of Cost on TB Capacity based on Bellcore

5.5.4 Dependence of Cost on Network Parameters Q and C

Generally it is expected that the network parameters may affect the performance and hence the cost function. Here we consider the cost function $J = J(C, Q)$ as a function of the link capacity C and the multiplexor size Q. Let $J_Q(C)$ denote a function of C for a given Q and $J_C(Q)$ be a function of Q for a given C. We plot $J_Q(C)$ and $J_C(Q)$ to study the relationship between the cost and network parameters C and Q.

$J_Q(C)$ with different buffer size is plotted in Figure 5.39 . The result shows that for a given buffer size Q, exceeding a certain critical value which can serve the peaks and bursts, the cost will decrease as the link capacity C increases, since increasing link capacity C will increase the service rate, and then reduces the waiting delay and then the cost. Hence:

$$C_1 < C_2 \Rightarrow J_Q(C_1) > J_Q(C_2).$$

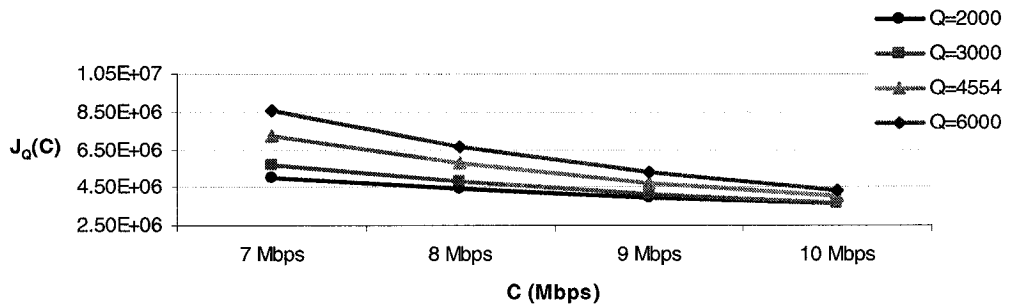


Figure 5.39: Cost Function on C with different Q

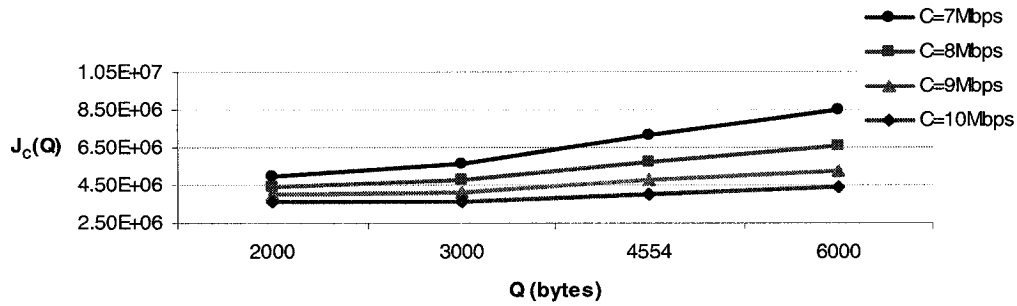


Figure 5.40: Cost Function on Q with different C

Figure 5.40 illustrates $J_C(Q)$ for different link capacity. We obtain the following:

$$Q_1 < Q_2 \Rightarrow J_C(Q_1) < J_C(Q_2).$$

For a given link capacity C, as the buffer size Q increases, the cost $J_C(Q)$ increases. The reason is that increasing buffer size Q causes more and more traffic collection in the buffer waiting for service so that it leads to a higher cost associated with waiting time.

We plot a three-dimensional graph as shown in Figure 5.41. It shows that a relatively

small buffer size Q and a relatively large link capacity C can improve performance of the system.

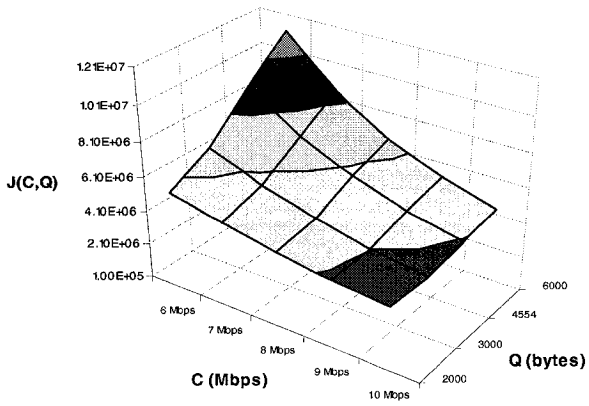


Figure 5.41: Cost Function on Q and C

5.5.5 Dependence of Cost on Relative Weights

Note that we use the buffer occupancy to measure the waiting cost. This is an approximate measure of the waiting cost. Therefore it is very important to choose an appropriate weight for this. In the above sections, we set $\gamma(t_k) = 1$ for any t_k. We see that the major contributor to the total cost is the weighted waiting cost based on the feedback control policy (no delay) shown in Figure 5.36.

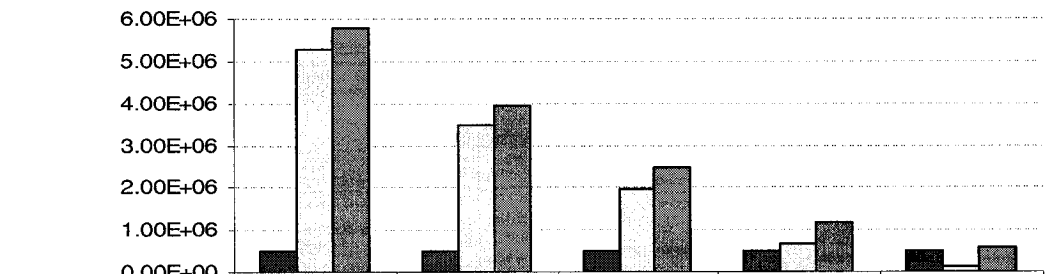
Now we consider that $\gamma(t_k)$ can be dynamically changed. The overload threshold is defined by mQ, where $m \in [0, 1]$ and Q is the buffer size. If the buffer occupancy $q(t_k)$ exceeds the given threshold mQ at time t_k, then we define the weighted waiting cost to be the difference between the current buffer occupancy and the overload threshold, otherwise the weighted waiting cost is set equal to zero, in other words, the waiting time below an

acceptable limit is ignored. Therefore the relative weight for waiting time is:

$$\gamma(t_k) \equiv \left\{1 - \frac{mQ}{q(t_k)}\right\} I\{q(t_k) \geq mQ\}. \tag{5.1}$$

We set different thresholds to compare the influence of waiting cost on the objective function following the feedback control policy (no delay).

- Case 1: $m = 0$, $\gamma(t_k) = 1$ for any t_k;
- Case 2: $m = 0.11$;
- Case 3: $m = 0.25$;
- Case 4: $m = 0.50$;
- Case 5: $m = 0.75$;



	case 1	case 2 (0.11Q)	case 2 (0.25 Q)	case 3 (0.50 Q)	case 4 (0.75Q)
weighted losses at TB	4.90E+05	4.90E+05	4.90E+05	4.90E+05	4.90E+05
weighted waiting cost	5.28E+06	3.47E+06	1.98E+06	6.83E+05	1.16E+05
cost J(u)	5.77E+06	3.96E+06	2.47E+06	1.17E+06	6.06E+05

Figure 5.42: Dependence of Cost on γ based on Bellcore Traffic

The result is shown in Figure 5.42. It is clear from Figure 5.42 that use of uniform weighting factor $\{\gamma(t_k) = 1\}$ for any k makes major contribution to the total cost $J(u)$

(Case 1). Cases 2-5 show that with the increase of the threshold, the total cost decreases. This is because we ignore the waiting cost when buffer occupancy is below the threshold. As the threshold increases, more and more waiting delay is ignored. Therefore the cost decreases.

In our experiment, the link capacity is $C = 8$ Mbps and the length of time slot is 0.0005 second. We know that 500 bytes of traffic can be served within one time slot. We may set the threshold to be 500 bytes. In other words, we ignore the waiting delay when the buffer occupancy does not exceed 500 bytes, which implies $m = \frac{500}{Q} = 0.11$. The result corresponding to $m = 0.11$ is shown in Case 2.

5.6 Convergence of Cost using Monte Carlo Methods

In this section, we will study convergence of cost using Monte Carlo Methods.

First, we study how many samples will result in the convergence of the average value $J_M(u)$ to the expected value $J(u)$. The number of such samples is denoted by M_o, that is, when we use M_o samples, the average value $J_{M_o}(u)$ will be close to the expected value $J(u)$. In this experiment, for HPP traffic, rates 200, 300, 400 and 500 packets per seconds are used. The result is illustrated in Table 5.5. We notice that for the above rates the maximum number of samples required to obtain the expected value is 150.

HPP Traffic Rate (packets/sec)	200	300	400	500
M_o	150	140	135	135
J(u) corresponding to M_o	1.72E+06	3.03E+06	5.08E+06	8.23E+06

Table 5.5: Convergency of Monte Carlo Methods

Now we discuss the number of samples used in Bellcore case. In our experiment, only

40 samples are taken to computer the expected values, since the computer does not have enough memory. The question is whether or not $M = 40$ is good enough. To check this, we use the HPP traffic to compare the above results with those only using $M = 40$. The results are shown in Table 5.6. From the results we see that, although the average value and the expected value are different, but they are close. Hence we conclude that the result by using $M = 40$ is acceptable.

HPP Traffic Rate (packets/sec)	200	300	400	500
$J_M(u)$ **corresponding to** $M = 40$	1.73E+06	3.00E+06	5.10E+06	8.20E+06
J(u)	1.72E+06	3.03E+06	5.08E+06	8.23E+06

Table 5.6: Convergency of Monte Carlo Methods

5.7 Summary of Numerical Results

From the numerical results in the above sections, we reach the following conclusions:

- The characteristics of traffic and the volume of traffic determine the performance of system. In general, the packet losses and the total cost increase as we increase the volume of traffic. The utilization can approach 100% as the traffic volume reach saturation level. This leads to excessive losses thereby the degrading QoS.

- System congestion is also related stochastic inputs. The expected number of times the system enters the state of congestion increase as the volume of traffic increase. Similarly, the mean (or the expected value) of the first time congestion occurs is determined by traffic characteristics.

- The proposed feedback control strategy (3.13) does a good job by reducing the cost functional in case of no feedback delay. Firstly, it eliminates losses at the

multiplexor. We know that losses at the multiplexor are not acceptable for some applications such as video. Second, it can be used to avoid monopoly by the network provider. When resources are shared according to demand, the feedback control law provides the proportional fraction of required demand. In this situation, any one of users can increase his demand and capture more bandwidth thereby starving other users. Network provider can check this by exercising the arbitration factor e_i.

- However, our experimental results show that presence of delay in the feedback control may drastically degrade performance.

- Given the statistical parameters of the traffic, this information can be used to design the access control system by providing the parameters $\{T_i, Q, C\}$. Our numerical result shows that a relatively large link capacity C and a relatively small buffer size Q can improve performance of the system for a given traffic. The link capacity C is related with traffic characteristics, the volume of traffic, the QoS offered by customers and the investment. The buffer size is related to the link capacity and the acceptable buffer delay. We can balance all factors to determine the appropriate parameters C and Q.

- In practical applications, it is very difficult, even impossible, to compute the expected values of random processes and functions of random processes. Therefore we have to choose the Monte Carlo method, which is an empirical method. In order to computer such quantities, we use Monte Carlo technique. The results obtained by using Monte Carlo technique seem to be fairly useful.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

We have analyzed the access control mechanism using stochastic inputs.

First, we have developed traffic models that capture the network traffic characteristic of SRD and LRD respectively and are easily implementable. The Poisson process (including HPP and NHPP) and DSPP, where the intensity functions follow FBM, are used to generate SRD and LRD traffic.

Secondly, we have presented the dynamic system model. This is a more precise model representing some improvement of models in the previous works[8][9]. In the previous model, packet can be cut into pieces, and part of packet may be dropped at the multiplexor. It is not practical. We prevent them. We use round robin packet scheduling to multiplex conforming traffic from all token pools in the system model. We also proposed feedback control algorithm which can reduce cost functional. This feedback control law can eliminate losses at the multiplexor and avoid monopoly too.

Finally, network performance using the system approach has been elaborated. We define

performance measures such as losses at TB, losses at the Multiplexor, waiting delay, expected number of times in the state of congestion, mean first time to congestion. These factors determine QoS required by customers. We elaborate many different performance indices for different traffic types such as HPP, NHPP, DSPP traffic. The numerical results demonstrate that this system and the feedback control police are useful and effective, which can be adapted to any kind of stochastic traffic. The results can serve as a tool for the designers of such controllers to set up different system parameters.

6.2 Future Work

The proposed system approach can reduce traffic losses, improve network performance and still satisfy the QoS requirements. Based on the models, the further investigation issues that could be of interest are as follow: (1). Predict the first time to congestion and exercise preemptive controls to avoid congestion. (2). Develop other simple feedback control strategies (possibly using neural network) that may provide a better performance compared to the one suggested; (3). In view of the performance degradation caused by feedback delay, it is necessary to develop a predictive control law that can estimate the future state (as long as the feedback delay) and provide control actions reducing performance degradation due to delay; (4). Another challenging topic is to develop traffic models which can accurately characterize complex network traffic that has complicated SRD and LRD properties.

Bibliography

[1] S. Jamin, S. J. Shenker, P.B. Danzig, *Comparison of Measurement-based Admission Control Algorithms for Controlled-Load Service*, on-line http://www.ieee-infocom.org/1997/papers/jamin.pdf

[2] H. Ahmadi, R. Gurin, K. Sohraby, *Analysis of Leaky Bucket Access Control Mechanism with Batch Arrival Process*, Globecom 90, (1990), 0344-0349

[3] Nanying Yin and M. G. Hluchyj, *Analysis of the Leaky Bucket Algorithm for On-Off Data sources*, Globecom 91, (1991), 0254-0257

[4] P. P. Tang, T. C. Tai, *Network Traffic Characterization Using Token Bucket Model*, Proc. of the Conference on Computer Communications, New York, Mar. 1999.

[5] C. Wahida, N.U. Ahmed, *Congestion control Using Dynamic Routing and Flow Control*, Stochastic Analysis and applications, 10(2),(1992),123-142

[6] S.Vamvakos and V. Anantharam, *On the Departure Process of a Leaky Bucket System with Long-Range Dependent Input Traffic*, Queueing Systems: Theory and Applications, 28(1-3),(1998), 191-214

[7] M.F. Alam, M. Atiquzzaman and M.A.Karim, *Effects of Source Traffic Shapping for MPEG Video Transmission over Next Generation IP networks*, IEEE (1999), 514-519

[8] N.U. Ahmed, Qun Wang and L.Orozco Barbosa, *A Systems Approach to Modeling the Token Bucket Algorithm in Computer Networks.* Mathematical Problems in Engineering : Theory, Methods and Applications (to appear)

[9] N.U. Ahmed, Bo Li and L.Orozco Barbosa, *Optimization of Computer Network Traffic Controllers using a Dynamic Programming/Genetic Algorithm Approach.* (submitted)

[10] on-line http://ita.ee.lbl.gov/html/contrib/BC.html

[11] A. Pezzuto, *What is Quality of Service?*, QoS Magazine.net, Jan. 2001 on-line http://www.qosmagazine.net/What_is_Quality_of_Service_1.asp

[12] on-line http://www.mitretek.org/pubs/telecom/review99/article1.doc

[13] A.Adas, *Traffic Models in Broadband Networks*,IEEE Communications Magazine, July 1997, on-line http://www.sce.carleton.ca/faculty/devetsikiotis/94581Y_00/Adas.pdf

[14] S.Ma, C.Ji, *Modeling Heterogeneous Network Traffic in Wavelet Domain*, IEEE/ACM transactions on networking, 9(5),(2001)

[15] I. Norros, *On the Use of Fractional Brownian Motion in the Theory of Connectionless Networks*, IEEE JSAC, 13(6), (1995), 953-962

[16] M. Crovella and A. Bestavros, *Self Similarity in World Wide Web Traffic: Evidence and Possible Causes*, IEEE/ACM Trans.Networking, 5

[17] W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson, *On the Self-similar Nature of Ethernet Traffic (Extended Version)*, IEEE/ACM Trans. Networking 2(1), (1994), 1-15

[18] V. Paxson and S. Floyd, *Wide Area Traffic: The failure of Poisson modeling* , IEEE/ACM Transactions on Networking 3, (1995), 226-244

[19] A. Feldmann, A. C. Gilbert. W. Willinger and T. G. Kurtz, *The Changing Nature of Network Traffic: Scaling Phenomena*, Computer Communication Review 28(2),(1998),5-29

[20] Ming Li,Weijia Jia,Wei Zhou, *Simulation of Long-Range Depedent Traffic and A Simulator of TCP Arrival Traffic*, International Journal of Interconnection Networks, on-line http://www.faculty.cs.tamu.edu/zhao/zhao_pub/2001/IJIN_01_MLi.pdf

[21] on-line http://www.cisco.com/univercd/cc/td/doc/product/software/ios120/12cgcr /qos_c/qcpart4/qcpolts.pdf

[22] *Comparing Traffic Policing and Traffic Shaping for Bandwidth Limiting*, on-line http://www.cisco.com/warp/public/105/policevsshape.pdf

[23] D. Awduche, A. Chiu, A. Elwalid, I. Widjaja, X. Xiao, *Overview and Principles of Internet Traffic Engineering*, RFC 3272, May 2002

[24] L. Kalampoukas, *Congestion Management in High Speed Networks*, University of California, Sept.1997 on-line http://citeseer.nj.nec.com/cache/papers/cs/686/ftp: zSzzSzftp.cse.ucsc.eduzSzpubzSzhsnlabzSzlampros-dissertation.pdf/kalampoukas97congestion.pdf

[25] A.S. Tanenbaum, *Computer Networks Third Edition*,", 6th edition, 2000

[26] P.F. Chimento, *Standard Token Bucket Terminology*, May, 2000, on-line http://qbone.internet2.edu/bb/Traffic.pdf

[27] A. Dubi, *Monte Carlo Applications in Systems Engineering*, John Wiley, Sons Ltd, NY, USA 2000

[28] C. P.Robert, G. Casella, *Monte Carlo Statistical Methods*, Springer-Verlag New York, Inc. 1999

[29] H. M. Taylor, S. Karlin, *An Introduction to Stochastic Modeling Third Edition*, Academic Press, 1998

[30] B. Ryu and S. Lowen, *Point Process Models for Self-Similar Network Traffic, with Applications*, Stochastic Models, 14(3) (1998)735-761

[31] *JTC 003 Mixed Packet Size Throughput*, Agilent Technologies, on-line http://advanced. comms.agilent.com/routertester/member/journal/JTC_003.html#1013110

[32] on-line http://wand.cs.waikato.ac.nz/wand/publications/jamie_420/final/report.html

[33] J. Banks, J. S. Carson II, B. L. Nelson, *Discrete-event System Simulation*, Second Edition, Prentice Hall, Upper Saddle River, New Jersey 07458

Appendix I: Stochastic Process

Network traffic can be modelled by different stochastic processes. A stochastic process is a family of random variables defined on a probability space, which contains continuous as well as discrete time processes. Here we introduce several stochastic processes: Poisson process, doubly stochastic Poisson process (DSPP), Gaussian process, and Brownian motion (BM) and fractional Brownian motion (FBM) which are used to generate traffic in this thesis for numerical simulation.

Poisson Process

Poisson process, named after S. Poisson, is one of the most important stochastic process in probability theory. It is assumed that the probability of an arrival in a small time interval depends only on the length of the interval, not on its position. Therefore the system has no memory. It does not care how long since an event last occurred.

A Poisson process [29] denoted by $\{X(t), t \geq 0\}$ with intensity $\lambda > 0$ is a stochastic process such that :

1. $X(t)$ is integer-valued, increasing and $X(0) = 0$;
2. $X(t)$ is independent and has stationary increments;
3. $X(t)$ has the Poisson distribution:

$$P(X(t) = n) = \frac{(\lambda t)^n e^{-\lambda t}}{n!}.$$

Note that, if $X(t)$ is a Poisson process with rate $\lambda > 0$, then the first and second moments are

$$E[X(t)] = \lambda t; \quad Var[X(t)] = \lambda t.$$

A Poisson process has important properties:

1. If $X(t)$ is Poisson distributed then the increments from s to a later point $s+t$ are also Poisson distributed;

2. If $X(t)$ is Poisson with intensity λ and W_n is the duration, then W_n follows exponential distribution: $P(W_n \leq t) = 1 - e^{-\lambda t}$;

3. If W_n is exponential, then W_n is independent of each other and memoryless.

If λ is a constant, it is called a homogenous Poisson process (HPP), and if λ varies with time, $\lambda = m(t)$, it is a nonhomogeneous or nonstationary Poisson process (NHPP), for which probability of n events occurring till time t is giving:

$$P(X(t) = n) = \frac{(\int_0^t m(s)ds)^n e^{-\int_0^t m(t)dt}}{n!}.$$

Doubly Stochastic Poisson Process

In 1955, David Cox introduced a kind of process as models for fibrous threads, which is a NHPP with the rate function itself a stochastic process. Such a process is called a doubly stochastic Poisson process (DSPP) . Now it is often called the Cox process in honor of its discoverer. This process has been widely used to generate stochastic inputs in many areas where the rate of input varies over time, depending on changing and unmeasured factors [29].

Suppose that $X(t)$ $(t \geq 0)$ is a HPP with unit intensity. Let $\lambda(t)$ $(t \geq 0)$ be a non-negative process satisfying $P\{\lambda(t) < \infty\} = 1$ for each $t > 0$. Assume further that $\lambda(t)$ is independent of $X(t)$ and right-continuous. Then we can define a DSPP $N(t)$ as follows:

$$N(t) = X(\lambda(t)), t \geq 0.$$

In this case, DSPP (or Cox process) $N(t)$ is controlled by the stochastic process $\lambda(t)$.

Gaussian Process

A Gaussian random variable is one of the simplest types of random variables, for which the probability distribution with mean μ and standard deviation σ is a normal distribution given by:

$$P\{X = x\} \equiv f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\{-(x-\mu)^2/2\sigma^2\}.$$

By taking $\mu = 0$ and $\sigma = 1$, it is a standard normal distribution.

The Gaussian (normal) distribution is a continuous probability distribution that is useful in characterizing a large variety and type of data. It has many convenient properties. For example, it is a symmetric, bell-shaped distribution, completely determined by its mean μ and standard deviation σ.

A Gaussian random process $\{X(t), t \geq 0\}$ is an indexed family of Gaussian random variables having normal distribute:

$$P\{X(t) = x\} \equiv f(x) = \frac{1}{\sigma(t)\sqrt{2\pi}} \exp\{-(x-m(t))^2/2\sigma^2(t)\},$$

where the mean and the variance are generally time dependent. Those are:

$$E(X(t)) = m(t), \quad E(X(t) - m(t))^2 = \sigma^2(t).$$

Brownian Motion and Fractional Brownian Motion

Brownian motion (BM) is an erratic, zigzag motion of microscopic particles. It was first observed in 1827 by the English botanist Robert Brown.

A random process $\{B_t, t \in [0, T]\}$ is a standard BM if

1. $B_{t+s} - B_t$ is normally distributed with mean 0 and variance s,for each $t > 0$ and $s > 0$;

2. $B_{t+s} - B_t$ is independent of B_t;

3. $B_(t)$ is a continuous function of time and $B_0 = 0$.

Note that BM is also a continuous process, where its mean is equal to zero and its covariance is given by

$$Cov\{B(t), B(s)\} = min\{t, s\}.$$

Fractional Brownian Motion (FBM) consists of steps in a random direction and with a step-length that has some characteristic value. A key feature of FBM is that if you zoom in on any part of the function you will produce a similar random walk in the zoomed in part.

A definition of FBM is as follows: For any Hurst parameter $H \in (0, 1)$ and a (variance) scaling parameter σ^2, a continuous-time stochastic process $\{B_H(t), t \geq 0\}$ is a FBM with Hurst parameter H if

1. $B_H(t)$ is Gaussian;
2. $B_H(0) = 0$ a.s.;
3. $B_H(t)$ has continuous sample paths;
4. $B_H(t)$ has stationary increments;
5. $E[B_H(t)] = 0, t \geq 0$;
6. $E[B_H(t)^2] = \sigma^2 t^{2H}, t \geq 0$.

The autocovariance function $\{cov(B_H(t), B_H(s)), s, t \geq 0\}$ can be obtained directly and

$$Cov\{B_H(t), B_H(s)\} = \frac{\sigma^2}{2}(|t|^{2H} - |t - s|^{2H} + |s|^{2H}).$$

From FBM definition and its properties, it is easily seen that FBM is self-similar with the Hurst parameter H. In particular, when $H = 0.5$ and $\sigma^2 = 1$, $B_H(t) = B(t)$ is the standard BM.

The main difference between FBM and BM is that while the increments in standard BM are independent, they are dependent in FBM. This dependence means that if there is an increasing pattern in the previous steps, then it is likely that the current step will be increasing as well.

Appendix II: Numerical Code

```
/* This code is to generate Homogenous Poisson process traffic.
Traffic rate = 200,300,400,500,800,850 packet/second.
Packet size distribution = complete Imix.*/

#include <iostream.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <math.h>
#include <fstream.h>

//generate random variable between min and max

int GetRandom(int min,int max)
{
   int temp;
   temp = min + rand() % (max - min + 1);
   return temp;
}

//generate randon uniform variable between 0 and 1.

double GetRandomUniform(int count)
{
      double U_random[10000];
      srand( (unsigned)time( NULL ) );
      for (int i=0;i<=count;i++)
      {
            U_random[i]=double(rand())/RAND_MAX;
      }
      return U_random[count];
}


//generate homogenious poisson traffic

void GetHomoPoissonTraffic()
{
      int size=3000;

      for (int k1=0;k1<size;k1++)
      {
            cout<<"size = "<<k1<<endl;
            char buffer[9];
            fstream outfile;
            int i=0,temp=0, k=0, packet=0;
            double t=0.0, lamda=400, h=0.001;
            double U;

            int *V;     //every packet size in every time interval
            double *T; //every T has an arrival packet

            V=new int[10000];      //total size =8000
            T=new double [10000];//total size less than 8000

            for(int index=0;index<10000;index++)
```

```
            {
                  V[index]=0;
                  T[index]=0;
            }

            while(t<=4.0)
            {
                  U=GetRandomUniform(i);      //get probability
                  if( U == 0.00 )        //if U=0, logU is not exist.
                        continue;
                  t=t-(1/lamda)*log(U); //caculate t, at t,a packet
arrival
                  i=i+1;
                  T[i]=t;                //put t into an array
                  k=int(T[i]/h)+1;       //calculate index k of v[k]

                  if(U<=0.18 && U>0)
                  {
                        packet=GetRandom(41,1499);
                  }
                  else if(U<=0.3 && U>0.18)
                  {
                        packet=1500;
                  }
                  else if(U<=0.45 && U>0.3)
                  {
                        packet=576;
                  }
                  else
                  {
                        packet=40;
                  }

                  if(temp!=k)
                  {
                        V[k]=packet;
                  }
                  temp=k;
                  if(k>=8000)
                  {
                        k=4000;
                  }
            }

            sprintf(buffer,"%d.txt",k1);           //name file
            outfile.open(buffer,ios::out);

            outfile<<k<<endl;
            for(int l=0;l<k;l++)
                  outfile<<V[l]<<endl;
            outfile.close();
            delete [] V;
            delete [] T;
      }
}
int main()
```

```
{
      GetHomoPoissonTraffic();
      return 0;
}
```

**/*This code is to generate nonhomogenous Poisson process traffic.
Traffic mean rate = 200,300,400,500 packet/second.
Packet size distribution = complete Imix */**

```
#include <iostream.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <math.h>
#include <fstream.h>

//generate random variable between min and max

int GetRandom(int min,int max)
{
   int temp;
   temp = min + rand() % (max - min + 1);
   return temp;
}

//generate randon uniform variable between 0 and 1.

double GetRandomUniform(int count)
{
      double U_random[100000];
      srand( (unsigned)time( NULL ) );
      for (int i=0;i<=count;i++)
      {
            U_random[i]=double(rand())/RAND_MAX;
      }
      return U_random[count];
}
void Traffic_generator()
{
            int size=3000,seed=2;
            double lamda1=100;
            double lamda2=400;

            for (int k1=0;k1<size;k1++)
            {
                  char buffer[9];
                  fstream outfile;
                  int i=0,temp=0, k=0, packet=0;
                  double t=0.0,h=0.001;
                  double U;

                  int *V;     //every packet size in every time interval
                  double *T; //every T has an arrival packet.

                  V=new int[10000];           //total size =8000
                  T=new double [10000]; //total size less than 8000
```

```
for(int index=0;index<10000;index++)
{
      V[index]=0;
      T[index]=0;
}
while(t<=4)
{
      if(seed==10000)
      {
            seed=0;
      }
      U=GetRandomUniform(seed++);//get probability
      if( U == 0.00 )        //if U=0, logU is not
exist.
            continue;
      if((t<0.5)||(t>=1.5&&t<2.5)||t>=3.5)
      {
            t=t-(1/(double)lamda1)*log(U);
      }
      else
      {
            t=t-(1/(double)lamda2)*log(U);
      }
      i=i+1;
      T[i]=t;
      k=int(T[i]/h)+1;
      if(U<=0.18 && U>0)
      {
            packet=GetRandom(41,1499);
      }
      else if(U<=0.3 && U>0.18)
      {
            packet=1500;
      }
      else if(U<=0.45 && U>0.3)
      {
            packet=576;
      }
      else
      {
            packet=40;
      }
      if(temp!=k)
      {
            V[k]=packet;
      }
      temp=k;
      if(k>=8000)
      {
            k=8000;
      }

}
sprintf(buffer,"%d.txt",k1);
outfile.open(buffer,ios::out);
```

```
                outfile<<8000<<endl;
                for(int l=0;l<8000;l++)
                      outfile<<V[l]<<endl;
                outfile.close();
          }
     }

int main()
{
     Traffic_generator();
     return 0;
}
```

**/* This code is to generate doubly stochastic Poisson process traffic.
Traffic intensity function read from newLamda.txt.
Packet size distribution = Bellcore traffic distribution. */**

```
#include <iostream.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <math.h>
#include <fstream.h>

//generate random variable between min and max
int GetRandom(int min,int max)
{
     int temp;
     temp = min + rand() % (max - min + 1);
     return temp;
}

//generate randon uniform variable between 0 and 1.

double GetRandomUniform(int count)
{
     double U_random[10000];
     srand( (unsigned)time( NULL ) );
     for (int i=0;i<=count;i++)
     {
          U_random[i]=double(rand())/RAND_MAX;
     }
     return U_random[count];
}

void getLamda(double *lamda)
{
     fstream file;
     file.open("newLamda.txt",ios::in);
     for(int i=0;i<400;i++)
     {
          file>>lamda[i];
          if(lamda[i]<10)
          lamda[i]=0;
     }
```

```
      file.close( );
}
void GetDSPPTraffic()
{
      int size=3000,seed=2;
      for (int k1=0;k1<size;k1++)
      {
            char buffer[9];
            fstream outfile;
            int i=0,k=0,temp=0,packet=0,j=0;
            double t=0.0, h=0.0005;
            double lamda[400];
            double U;
            int *V;     //every packet size in every time interval
            double *T;
            V=new int[10000];
            T=new double [10000];
            for(int index=0;index<10000;index++)
            {
                  V[index]=0;
                  T[index]=0;
            }
            getLamda(lamda);
            while(t<=4.00)
            {
                  if(seed==9999)
                        seed=0;
                  U=GetRandomUniform(seed++);
                  if( U == 0.00 )
                        continue;
                  j=int(t/0.01);
                  if(lamda[j]!=0)
                  {
                        t=t-(1/lamda[j])*log(U);
                        i=i+1;
                        T[i]=t;
                        k=int(T[i]/h)+1;
                  }
                  else
                        t= t+0.01;
                  if(U<=0.0449 && U>0)
                  {
                        packet=64;
                  }
                  else if(U<=0.0752 && U>0.0449)
                  {
                        packet=150;
                  }
                  else if(U<=0.2645 && U>0.0752)
                  {
                        packet=162;
                  }
                  else if(U<=0.4317 && U>0.2645)
                  {
                        packet=174;
                  }
```

```
                else if(U<=0.6725 && U>0.4317)
                {
                      packet=1090;
                }
                else if(U<=0.7264 && U>0.6725)
                {
                      packet=1518;
                }
                else
                {
                      packet=GetRandom(65,200);
                }
                if(temp!=k)
                {
                      V[k]=packet;
                }
                temp=k;
                if(k>8000)
                {
                      k=8000;
                }
          }
          sprintf(buffer,"%d.txt",k1);
          outfile.open(buffer,ios::out);
          if( k < 8000 )
                k = 8000;
          outfile<<k<<endl;
          for(int l=0;l<k;l++)
          {
                outfile<<V[l]<<endl;
          }
          outfile.close();
          delete [] V;
          delete [] T;
     }
}

int main()
{
     GetDSPPTraffic();
     return 0;
}
```

/*This code is for fractional Brownian motion. Traffic intensity function is the maximum value of FBM and zero and the absolute value of FBM*/

```
clear all;
clc;
scale1=0.001;
scale2=0.0005;
lambda(1)=0;
n=1;
H=0.795;
tic
```

```
for  x=scale1:scale1:4;
    randn('state',10137);
    sum=0;
     for  y=scale2:scale2:x-scale2;
          sum=sum+randn(1)*sqrt(scale2)*350*(x-y)^(H-0.5);
    end
    if sum<0
     lambda(n)=-sum; //get the absolute value of FBM.
      % lambda(n)=0; //get the maximum value of FBM and zero.
    else
       lambda(n)=sum;
    n=n+1;
   end
end

bb=0:4/(n-1):(4-4/(n-1));
plot(bb,lambda);
xlabel('time (unit: sec)');
ylabel('traffic rate ( packets/sec )')
%ylabel('B_{H}(t)');
toc
```

```
/* This code is to for open loop control algorithm.
For HPP/NHPP C=5Mbps, Q=4000bytes, T=1500bytes.
For DSPP/Bellcore C=8Mbps Q=4554bytes T=1500bytes. */

#include <stdio.h>
#include <iostream.h>
#include <fstream.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>

const int Num =3;
double SERVICERATE=625000;//bytes/second
const double TIMEINTERVAL=0.0005;
const int QUEUESIZE =4000;
int TBSIZE[Num];
int STAGE;
int **states;//[STAGE][Num+1]
int **Traffic;//[STAGE][Num]
int **bandAllocation;
int **trueAllocation;
int confirmedTraffic[Num];
int stageTraffic[Num];
int stage;
int acceptedMul;
int initialState[Num+1];//TB1,TB2,TB3 and multiplexer (total:4 states)
int totalState;
int totalComeTraffic=0;
int lostAtMultiplexer=0;
int lostAtTB=0;
int lostAtM=0;
int startIndex=0;
```

```
double utilization;
int throughput;
int meantotal=0;
int initialControl[3];
void readStage();
void readTraffic(int);
void getTBSize();
int objectiveFunction(int[]);
void confirmTraffic(int []);
int LostAtMultiplexer();
int TotalLostAtTB();
int LostAtTB(int, int);
int WaitingLost();
int AcceptedTrafficByMul();
double variance(int mean,int whichTB);
double StandardDeviation(double variance);

int main()
{
      fstream outfile;
      outfile.open("output_openloop.txt",ios::out);
      int size=1000;
      int average_meanTraffic=0;//average traffic in one time slot
      int average_lostAtTB=0;
      int average_lostAtMul=0;
      int average_waiting=0;
      int average_totalCost=0;
      double average_util=0.0;
      int average_thro=0;
      int average_input=0;
      int average_output=0;
      double average_lossRate=0.0;

      for (int a=0;a<size;a++)
      {
            meantotal=0;
            lostAtTB=0;
            lostAtM=0;
            int lost=0;      //calculate lost J(u) till current stage
            int temp=0;      //calculate waiting lost till current stage
            utilization=0;
            throughput=0;
            totalComeTraffic=0;
            int control[3];
            int stagelost[Num];
            readStage();     //read the number of stage

            // TBs and Mul states in different [tk-1, tk)

            states=new int*[STAGE+1];
            for(int i1=0;i1<STAGE+1;i1++)
            {
                  states[i1]=new int[Num+1];
            }

            // Traffic in different [tk-1, tk) for every source
```

```
            Traffic=new int* [STAGE];
            for(int i2=0;i2<STAGE;i2++)
            {
                  Traffic[i2]=new int[Num];
            }

            // Bandwidth in different [tk-1,tk) for every source
            bandAllocation=new int*[STAGE];
            for(int i3=0;i3<STAGE;i3++)
            {
                  bandAllocation[i3]=new int[Num];
            }

            // True allocation in different [tk-1,tk) for every source

            trueAllocation=new int*[STAGE];
            for(int i4=0;i4<STAGE;i4++)
            {
                  trueAllocation[i4]=new int[Num];
            }

            //read three traffic from three traffic files
            readTraffic(a);

      //get three TB sizes and give initial values of 3 TBs and 1
multiplxer
            getTBSize();

            // run from stage 0 to final stage
            for ( stage=0; stage<STAGE; stage++)
            {
                  for(int i=0;i<Num;i++)
                  {
                        control[i]=initialControl[i];
                  }
//if stage is 0, assign the initial State of control value and
multiplexer
                  if(stage==0)
                  {
                        for(int s=0;s<=Num;s++)
                        {
                              states[stage][s]=initialState[s];
                        }
                  }
// change the control value, if TBs have no enough space to accept all
the incoming tokens
                  for(int l=0; l<Num; l++)
                  {
                        if(states[stage][l]+control[l]>TBSIZE[l])
                        {
                              control[l]=TBSIZE[l]-states[stage][l];
                        }
                  }

                  // calulate the total losses
```

```
// objectiveFunction() is a function to calculate the losses at
current stage

                lost = lost+ objectiveFunction(control);

                //calculate the losses at each TB at current stage

                for(int k2=0;k2<Num;k2++)
                {

     stagelost[k2]=LostAtTB(confirmedTraffic[k2],Traffic[stage][k2]);
                }

                //calculate the total TB losses at current stage

                for(int k3=0;k3<Num;k3++)
                {
                      lostAtTB=lostAtTB+stagelost[k3];
                }

           // calculate the total losses at multiplexer till current
stage

                lostAtM=lostAtM+lostAtMultiplexer;

                // if the final stage
                if(stage==(STAGE-1))
                {
                      states[stage+1][Num]=WaitingLost();
                }

// calculate the states at next stage according to current states and
control
                for( l=0;l<Num;l++)
                {
     states[stage+1][l]=states[stage][l]+control[l]-
confirmedTraffic[l];
                      if (states[stage+1][l]>TBSIZE[l])
                      {
                            states[stage+1][l]=TBSIZE[l];
                      }

                }

                // calculate the states at queue
                states[stage+1][Num]=WaitingLost();
                temp=temp+WaitingLost();
           }

           //calculate utilization and throughout
           utilization=(double)(totalComeTraffic-lostAtM-
                      lostAtTB)/(SERVICERATE*STAGE*TIMEINTERVAL);
           throughput=(int)(SERVICERATE*utilization);

           for(int j1=0;j1<STAGE;++j1)
           {
```

```
                delete [] states[j1];
            }
            delete [] states;

            for(int j2=0;j2<STAGE;j2++)
            {
                delete [] Traffic[j2];
            }
            delete []Traffic;

            for(int j3=0;j3<STAGE;j3++)
            {
                delete [] bandAllocation[j3];
            }

            delete [] bandAllocation;

            for(int j4=0;j4<STAGE;j4++)
            {
                delete [] trueAllocation[j4];
            }

            delete []trueAllocation;
        }

        average_meanTraffic=average_meanTraffic/size;
        average_lostAtTB=average_lostAtTB/size;
        average_lostAtMul=average_lostAtMul/size;
        average_waiting=average_waiting/size;
        average_totalCost=average_totalCost/size;
        average_util=average_util/size;
        average_thro=average_thro/size;
        average_input=average_input/size;
        average_output=average_output/size;
        average_lossRate=average_lossRate/size;
        outfile<<average_meanTraffic<<"\t";
        outfile<<average_lostAtTB<<"\t";
        outfile<<average_lostAtMul<<"\t";
        outfile<<average_waiting<<"\t";
        outfile<<average_totalCost<<"\t";
        outfile<<average_util<<"\t";
        outfile<<average_thro<<"\t";
        outfile<<average_input<<"\t";
        outfile<<average_output<<"\t";
        outfile<<average_lossRate<<endl;
        outfile.close();
        return 0;
}

// read the number of stages of the traffic sources
void readStage()
{
        fstream file1;
        file1.open("1.txt",ios::in);
        file1>>STAGE;
        file1.close();
```

```
}

// read the traffic from a file to an array
void readTraffic(int a)
{
      for(int num=a*3;num<(a+1)*3;num++)
      {
            fstream file1;
            char *buffer;
            buffer=new char[9];
            sprintf(buffer,"%d.txt",num);
            file1.open(buffer,ios::in);
            int c1=num%3;
            file1>>STAGE;
            for(int c2=0;c2<STAGE;c2++)
            {
                  file1>>Traffic[c2][c1];
            }
            file1.close( );
            delete [] buffer;
      }
}

// calculate the TBs' sizes
void getTBSize()
{
      int mean[Num];
      for(int i=0;i<Num;i++)
      {
            int temp=0;// for calculate total traffic of every source

            for(int i1=0;i1<STAGE;i1++)
            {
                  temp=temp+Traffic[i1][i];
            }
            mean[i]=(temp/STAGE); // average bytes in one stage.

            // initialize the open loop control value
            initialControl[i]=171;
            meantotal+=mean[i];
            totalComeTraffic+=temp;
      }
      // calculate the sizes of TBs
      for(int i4=0;i4<Num;i4++)
      {
            // set TB size
            TBSIZE[i4]=1500;

            // initialize the initial states of three TBs
            initialState[i4]=0;//can be changed from 0->T
      }
      // initialize the initial state of multiplexer
      initialState[Num]=0;
}

// calculate the losses
```

```
int objectiveFunction(int control[])
{
      int total=0;
      confirmTraffic(control);
      total = 5*TotalLostAtTB();
      total += 10*LostAtMultiplexer();
      total += WaitingLost();
      return total;
}

// calculate the confirmed traffic
void confirmTraffic(int control[])
{
      for (int i=0; i<Num; i++)
      {
            if (Traffic[stage][i]<=(states[stage][i]+ control[i]))
            {
                  confirmedTraffic[i]=Traffic[stage][i];
            }
            else confirmedTraffic[i]=0;
      }
}

// calculate the traffic accepted by Multiplexer
int AcceptedTrafficByMul()
{
      int totalAcceptedPacket=0;
      int leftSpace=0;
      int totalConfirmedTraffic=0;
      for(int i=0;i<Num;i++)
      {
            totalConfirmedTraffic += confirmedTraffic[i];
      }
      int QueueState = states[stage][Num]-
(int)(SERVICERATE*TIMEINTERVAL);
      if(QueueState<0)
      {
            QueueState =0;
      }
      leftSpace=QUEUESIZE-QueueState;
      if(totalConfirmedTraffic>leftSpace)
      {
            for(int k=0;k<Num;k++)
            {
                  if(leftSpace>=confirmedTraffic[startIndex])
                  {
                        leftSpace=leftSpace-confirmedTraffic[startIndex];

      totalAcceptedPacket+=confirmedTraffic[startIndex];
                        startIndex=(startIndex+1)%Num;
                  }
                  else
                  {
                        break;
                  }
            }
```

```
            return totalAcceptedPacket;
        }
        else
        {
            return totalConfirmedTraffic;
        }
}

// calculate the losses at multiplexer in current stage
int LostAtMultiplexer()
{
        int totalConfirmedTraffic=0;
        for(int i=0;i<Num;i++)
        {
            totalConfirmedTraffic+=confirmedTraffic[i];
        }
        // notes: lostAtMultiplexer is global variable
        lostAtMultiplexer=totalConfirmedTraffic-AcceptedTrafficByMul();
        return lostAtMultiplexer;
}

// calculate the total losses at TBs
int TotalLostAtTB()
{
        int totalLostTB = 0;
        for(int i=0;i<Num;i++)
        {
            totalLostTB += Traffic[stage][i]-confirmedTraffic[i];
        }
        return totalLostTB;
}


// calculate the losses at each TB
int LostAtTB(int confirmedTraffic, int Traffic) //calling by
TotalLostAtTB()
{
        return Traffic - confirmedTraffic;
}
// calculate the waiting losses at the multiplexer
int WaitingLost()
{
        int QueueState = states[stage][Num] -
(int)(SERVICERATE*TIMEINTERVAL);
        int TrafficEnterMul;
        if(QueueState >0)
        {
            QueueState=QueueState;
            TrafficEnterMul=AcceptedTrafficByMul();
        }
        else
        {
            QueueState = 0;
            TrafficEnterMul = AcceptedTrafficByMul();
        }
        return QueueState + TrafficEnterMul;
```

```
}

/* This code is for feedback control algorithm.
For HPP/NHPP C=5Mbps, Q=4000bytes, T=1500bytes.
For DSPP/Bellcore C=8Mbps Q=4554bytes T=1500bytes.*/

#include <stdio.h>
#include <iostream.h>
#include <fstream.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>

const int Num =3;
double SERVICERATE=625000;//bytes/second
const double TIMEINTERVAL=0.0005;
const int QUEUESIZE =4000;
int TBSIZE[Num];
int STAGE;
int **states;//[STAGE][Num+1]
int **Traffic;//[STAGE][Num]
int **bandAllocation;
int **trueAllocation;
int confirmedTraffic[Num];
int stageTraffic[Num];
int stage;
int acceptedMul;
int initialState[Num+1];//TB1,TB2,TB3 and multiplexer (total:4 states)
int totalState;
int totalComeTraffic=0;
int lostAtMultiplexer=0;
int lostAtTB=0;
int lostAtM=0;
int startIndex=0;
double utilization;
int throughput;
int delayStage=0;
int meantotal=0;
void readStage();
void readTraffic(int);
void getTBSize();
int objectiveFunction(int[]);
void confirmTraffic(int []);
int LostAtMultiplexer();
int TotalLostAtTB();
int LostAtTB(int, int);
int WaitingLost();
int AcceptedTrafficByMul();
void TrueAllocation();


int main()
{
      fstream outfile;
      outfile.open("output_delay.txt",ios::out);
```

```
int size=1000;
int average_meanTraffic=0;//average traffic in one time slot
int average_lostAtTB=0;
int average_lostAtMul=0;
int average_waiting=0;
int average_totalCost=0;
double average_util=0.0;
int average_thro=0;
int average_input=0;
int average_output=0;
double average_lossRate=0.0;

for (int a=0;a<size;a++)
{
      meantotal=0;
      lostAtTB=0;
      lostAtM=0;
      int lost=0;       //calculate lost J(u) till current stage
      int temp=0;       //calculate waiting lost till current stage

      utilization=0;
      throughput=0;
      totalComeTraffic=0;
      int control[3];
      int stagelost[Num];
      //read the number of stage
      readStage();

      // TBs and Mul states in different [tk-1, tk)

      states=new int*[STAGE+1];

      for(int i1=0;i1<STAGE+1;i1++)
      {
            states[i1]=new int[Num+1];
      }

      // Traffic in different [tk-1, tk) for every source
      Traffic=new int* [STAGE];
      for(int i2=0;i2<STAGE;i2++)
      {
            Traffic[i2]=new int[Num];
      }

      // Bandwidth in different [tk-1,tk) for every source
      bandAllocation=new int*[STAGE];
      for(int i3=0;i3<STAGE;i3++)
      {
            bandAllocation[i3]=new int[Num];
      }

      // True allocation in different [tk-1,tk) for every source
      trueAllocation=new int*[STAGE];
      for(int i4=0;i4<STAGE;i4++)
      {
            trueAllocation[i4]=new int[Num];
```

```
            }

            //read three traffic from three traffic files
            readTraffic(a);

      //get three TB sizes and give initial values of 3 TBs and 1
multiplxer
            getTBSize();
            // run from stage 0 to final stage
            for ( stage=0; stage<STAGE; stage++)
            {
//if stage is 0, assign the initial State of control value and
multiplexer
                  if(stage==0)
                  {
                        for(int s=0;s<=Num;s++)
                        {
                              states[stage][s]=initialState[s];
                        }
                  }

                  //get actual allocated traffic
                  TrueAllocation();

                  //initialize the control value in current stage
                  for(int ss=0;ss<Num;ss++)
                  {

      if(delayStage<1||(delayStage>=1&&stage<=delayStage))
                        {

      if(trueAllocation[stage][ss]>states[stage][ss])
                              {
                                    control[ss]=trueAllocation[stage][ss]-
                                                states[stage][ss];
                                    if(control[ss]>TBSIZE[ss]-
                                                states[stage][ss])
                                    {
                                          control[ss]=TBSIZE[ss]-
                                                states[stage][ss];
                                    }
                              }
                              else
                              {
                                    control[ss]=0;
                              }
                        }
                        if(delayStage>=1&&stage>delayStage)
                        {
                              if(trueAllocation[stage-
                              delayStage][ss]>=states[stage-
                              delayStage][ss])
                              {
                                    control[ss]=trueAllocation[stage-
                                    delayStage][ss]-states[stage-
                                    delayStage][ss];
```

```
                                    if(control[ss]>=TBSIZE[ss]-
                                                states[stage][ss])
                                    {
                                            control[ss]=TBSIZE[ss]-
                                                    states[stage][ss];
                                    }
                              }
                              else
                              {
                                    control[ss]=0;
                              }
                        }
                  }

                  // calulate the total losses
// objectiveFunction() is a function to calculate the losses at
current stage
                  lost = lost+ objectiveFunction(control);

                  //calculate the losses at each TB at current stage
                  for(int k2=0;k2<Num;k2++)
                  {

      stagelost[k2]=LostAtTB(confirmedTraffic[k2],Traffic[stage][k2]);
                  }

                  //calculate the total TB losses at current stage

                  for(int k3=0;k3<Num;k3++)
                  {
                        lostAtTB=lostAtTB+stagelost[k3];
                  }

            // calculate the total losses at multiplexer till current
stage

                  lostAtM=lostAtM+lostAtMultiplexer;

                  // if the final stage
                  if(stage==(STAGE-1))
                  {
                        states[stage+1][Num]=WaitingLost();
                  }

// calculate the states at next stage according to current states and
control
                  for(int l=0;l<Num;l++)
                  {
                        states[stage+1][l]=states[stage][l]+control[l]-
                                          confirmedTraffic[l];
                        if (states[stage+1][l]>TBSIZE[l])
                        {
                              states[stage+1][l]=TBSIZE[l];
                        }

                  }
```

```
                    // calculate the states at queue
                    states[stage+1][Num]=WaitingLost();

                    temp=temp+WaitingLost();
               }

               //calculate utilization and throughput
               utilization=(double)(totalComeTraffic-lostAtM-
                    lostAtTB)/(SERVICERATE*STAGE*TIMEINTERVAL);
               throughput=(int)(SERVICERATE*utilization);
               average_meanTraffic+=meantotal/3;
               average_lostAtTB+=lostAtTB;
               average_lostAtMul+=lostAtM;
               average_waiting+=temp;
               average_totalCost+=lost;
               average_util+=utilization;
               average_thro+=throughput;
               average_input+=totalComeTraffic;
               average_output+=totalComeTraffic-lostAtTB-lostAtM;

     average_lossRate+=(double)(lostAtTB+lostAtM)/(double)totalComeTra
ffic;

               for(int j1=0;j1<STAGE;++j1)
               {
                    delete [] states[j1];
               }
               delete [] states;

               for(int j2=0;j2<STAGE;j2++)
               {
                    delete [] Traffic[j2];
               }
               delete []Traffic;

               for(int j3=0;j3<STAGE;j3++)
               {
                    delete [] bandAllocation[j3];
               }
               delete [] bandAllocation;

               for(int j4=0;j4<STAGE;j4++)
               {
                    delete [] trueAllocation[j4];
               }
               delete []trueAllocation;
     }

     average_meanTraffic=average_meanTraffic/size;
     average_lostAtTB=average_lostAtTB/size;
     average_lostAtMul=average_lostAtMul/size;
     average_waiting=average_waiting/size;
     average_totalCost=average_totalCost/size;
     average_util=average_util/size;
     average_thro=average_thro/size;
```

```
	average_input=average_input/size;
	average_output=average_output/size;
	average_lossRate=average_lossRate/size;

	outfile<<average_meanTraffic<<"\t";
	outfile<<average_lostAtTB<<"\t";
	outfile<<average_lostAtMul<<"\t";
	outfile<<average_waiting<<"\t";
	outfile<<average_totalCost<<"\t";
	outfile<<average_util<<"\t";
	outfile<<average_thro<<"\t";
	outfile<<average_input<<"\t";
	outfile<<average_output<<"\t";
	outfile<<average_lossRate<<endl;
	outfile.close();
	return 0;
}

// read the number of stages of the traffic sources
void readStage()
{
	fstream file1;
	file1.open("1.txt",ios::in);
	file1>>STAGE;
	file1.close();
}

// read the traffic from a file to an array
void readTraffic(int a)
{
	for(int num=a*3;num<(a+1)*3;num++)
	{
		fstream file1;
		char *buffer;
		buffer=new char[9];
		sprintf(buffer,"%d.txt",num);
		file1.open(buffer,ios::in);
		int c1=num%3;
		file1>>STAGE;
		for(int c2=0;c2<STAGE;c2++)
		{
			file1>>Traffic[c2][c1];
		}
		file1.close( );
		delete [] buffer;
	}
}

// calculate the TBs' sizes
void getTBSize()
{
	int mean[Num];
	for(int i=0;i<Num;i++)
	{
		int temp=0;// for calculate total traffic of every source
		for(int i1=0;i1<STAGE;i1++)
```

```
            {
                  temp=temp+Traffic[i1][i];
            }
            mean[i]=(temp/STAGE); // average bytes in one stage.
            meantotal+=mean[i];
            totalComeTraffic+=temp;
      }

      // calculate the sizes of TBs
      for(int i4=0;i4<Num;i4++)
      {
            // set TB size
            TBSIZE[i4]=1500;
            // initialize the initial states of three TBs
            initialState[i4]=0;//can be changed from 0->T
      }
      // initialize the initial state of multiplexer
      initialState[Num]=0;
}
void TrueAllocation()
{
      int TotalTrafficInAStage=0;
      int leftSpace=0;

      double ProviderControl[Num]={3,3,3};
      for(int i=0;i<Num;i++)
      {

      TotalTrafficInAStage=TotalTrafficInAStage+Traffic[stage][i];
      }
      int QueueState = states[stage][Num]-
(int)(SERVICERATE*TIMEINTERVAL);
      if(QueueState<0)
      {
            QueueState =0;
      }
      leftSpace=QUEUESIZE-QueueState;

      for(int j=0;j<Num;j++)
      {
            if(TotalTrafficInAStage!=0)
            {
            double
            t1=(double)Traffic[stage][j]/(double)TotalTrafficInAStage;
                  double t2=ProviderControl[j]/(double)Num;

                  if(t1>t2)
                  {
                        bandAllocation[stage][j]=(int)(t2*leftSpace);
                  }
                  else
                        bandAllocation[stage][j]=(int)(t1*leftSpace);
            }
            else
            {
                  bandAllocation[stage][j]=0;
```

```
            }

            if(bandAllocation[stage][j]>=Traffic[stage][j])
            {
                  trueAllocation[stage][j]=Traffic[stage][j];
            }
            else
            {
                  trueAllocation[stage][j]=bandAllocation[stage][j];
            }
      }

      return ;
}


// calculate the losses
int objectiveFunction(int control [])
{
      int total=0;
      confirmTraffic(control);
      total=5*TotalLostAtTB()+10*LostAtMultiplexer()+WaitingLost();
      return total;
}

// calculate the confirmed traffic
void confirmTraffic(int control[])
{
      for (int i=0; i<Num; i++)
      {
            if (Traffic[stage][i]<=(states[stage][i]+ control[i]))
            {
                  confirmedTraffic[i]=Traffic[stage][i];
            }
            else confirmedTraffic[i]=0;
      }

}

// calculate the traffic accepted by Multiplexer
int AcceptedTrafficByMul()
{
      int totalAcceptedPacket=0;
      int leftSpace=0;

      int totalConfirmedTraffic=0;

      for(int i=0;i<Num;i++)
      {
            totalConfirmedTraffic += confirmedTraffic[i];
      }


      int QueueState = states[stage][Num]-
(int)(SERVICERATE*TIMEINTERVAL);
      if(QueueState<0)
```

```
        {
                QueueState =0;
        }

        leftSpace=QUEUESIZE-QueueState;

        if(totalConfirmedTraffic>leftSpace)
        {
                for(int k=0;k<Num;k++)
                {
                        if(leftSpace>=confirmedTraffic[startIndex])
                        {
                                leftSpace=leftSpace-confirmedTraffic[startIndex];

        totalAcceptedPacket+=confirmedTraffic[startIndex];
                                startIndex=(startIndex+1)%Num;
                        }
                        else
                        {
                                break;
                        }
                }
                return totalAcceptedPacket;
        }
        else
        {
                return totalConfirmedTraffic;
        }

}

// calculate the losses at multiplexer
int LostAtMultiplexer()
{
        int totalComfirmedTraffic=0;
        for(int i=0;i<Num;i++)
        {
                totalComfirmedTraffic+=confirmedTraffic[i];
        }
        // notes: lostAtMultiplexer is global variable

        lostAtMultiplexer=totalComfirmedTraffic-AcceptedTrafficByMul();
        return lostAtMultiplexer;
}

// calculate the total losses at TBs
int TotalLostAtTB()
{
        int totalLostTB = 0;
        for(int i=0;i<Num;i++)
        {
                totalLostTB += LostAtTB(confirmedTraffic[i],
Traffic[stage][i]);
        }
        return totalLostTB;
}
```

```
// calculate the losses at each TB
int LostAtTB(int confirmedTraffic, int Traffic) //calling by
TotalLostAtTB()
{
      return Traffic - confirmedTraffic;
}

// calculate the waiting losses at the multiplexer
int WaitingLost()
{
      int QueueState = states[stage][Num] -
(int)(SERVICERATE*TIMEINTERVAL);
      int TrafficEnterMul;
      if(QueueState >0)
      {
            QueueState=QueueState;
            TrafficEnterMul=AcceptedTrafficByMul();
      }
      else
      {
            QueueState = 0;
            TrafficEnterMul = AcceptedTrafficByMul();
      }
      return QueueState + TrafficEnterMul;
}
```

/* This code is for congestion.
For HPP/NHPP C=5Mbps, Q=4000bytes, T=1500bytes.
For DSPP/Bellcore C=8Mbps Q=4554bytes T=1500bytes.*/

```
#include <cstdio>
#include <iostream>
#include <fstream>
#include <cmath>
#include <string>
#include <cstdlib>
using namespace std;
const int Num =3;
double SERVICERATE=625000;//bytes/second
const double TIMEINTERVAL=0.0005;
const int QUEUESIZE =4000;
int TBSIZE[Num];
int STAGE;
int **states;//[STAGE][Num+1]
int **Traffic;//[STAGE][Num]
int **bandAllocation;
int **trueAllocation;
int confirmedTraffic[Num];
int stageTraffic[Num];
int stage;
int acceptedMul;
int initialState[Num+1];//TB1,TB2,TB3 and multiplexer (total:4 states)
```

```
int totalState;
int totalComeTraffic=0;
int lostAtMultiplexer=0;
int lostAtTB=0;
int lostAtM=0;
int startIndex=0;
double utilization;
double throughput;

void readStage();
void readTraffic(int a);
void getTBSize();
void confirmTraffic(int []);
int WaitingLost();
int AcceptedTrafficByMul();
void TrueAllocation();


int main()
{
      int size=1000;
      fstream outfile;
      outfile.open("output_congestion.txt",ios::out);
      for (int a=0; a<size;a++)
      {
            int control[3];
            int congestionCount=0;

            //read the number of stage
            readStage();

            // TBs and Mul states in different [tk-1, tk]

            states=new int*[STAGE+1];

            for(int i1=0;i1<STAGE+1;i1++)
            {
                  states[i1]=new int[Num+1];
            }

            // Traffic in different [tk-1, tk] for every source
            Traffic=new int* [STAGE];
            for(int i2=0;i2<STAGE;i2++)
            {
                  Traffic[i2]=new int[Num];
            }

            // Bandwidth in different [tk-1,tk] for every source
            bandAllocation=new int*[STAGE];
            for(int i3=0;i3<STAGE;i3++)
            {
                  bandAllocation[i3]=new int[Num];
            }

            // True allocation in different [tk-1,tk] for every source
```

```
            trueAllocation=new int*[STAGE];
            for(int i4=0;i4<STAGE;i4++)
            {
                  trueAllocation[i4]=new int[Num];
            }
            //read three traffic from three traffic files
            readTraffic(a);

//get three TB sizes and give initial values of 3 TBs and 1 multiplxer
            getTBSize();
            // run from stage 0 to final stage
            for ( stage=0; stage<STAGE; stage++)
            {
//if stage is 0, assign the initial State of control value and
multiplexer
            if(stage==0)
            {
                  for(int s=0;s<=Num;s++)
                  {
                        states[stage][s]=initialState[s];
                  }
            }

            //get actual allocated traffic
            TrueAllocation();

            //initialize the control value in current stage

            for(int ss=0;ss<Num;ss++)
            {
                  if(trueAllocation[stage][ss]>=states[stage][ss])
                  {
                        control[ss]=trueAllocation[stage][ss]-
                                    states[stage][ss];
                        if(control[ss]>=TBSIZE[ss]-states[stage][ss])
                        {
                              control[ss]=TBSIZE[ss]-states[stage][ss];
                        }
                  }
                  else
                  {
                        control[ss]=0;
                  }
            }
            confirmTraffic(control);
            // if the final stage
            if(stage==(STAGE-1))
            {
                  states[stage+1][Num]=WaitingLost();
            }

// calculate the states at next stage according to current states and
control
            for(int l=0;l<Num;l++)
            {
```

```
                    states[stage+1][l]=states[stage][l]+control[l]-
                                        confirmedTraffic[l];
                    if (states[stage+1][l]>TBSIZE[l])
                    {
                          states[stage+1][l]=TBSIZE[l];
                    }

              }

              // calculate the states at queue

              states[stage+1][Num]=WaitingLost();
              //outfile<<"waitinglost = "<<WaitingLost()<<"\t";
if(states[stage+1][Num]<=QUEUESIZE&&states[stage+1][Num]>=0.9*QUEUESIZ
E)
              {
                    congestionCount=congestionCount+1;
                    if(congestionCount==1)
                          outfile<<a<<"\t"<<stage<<"\t";
              }
        }
        if(congestionCount!=0)
        {
              outfile<<congestionCount<<endl;
        }
        for(int j1=0;j1<STAGE;++j1)
        {
              delete [] states[j1];
        }
        delete [] states;

        for(int j2=0;j2<STAGE;j2++)
        {
              delete [] Traffic[j2];
        }
        delete []Traffic;


        for(int j3=0;j3<STAGE;j3++)
        {
              delete [] bandAllocation[j3];
        }

        delete [] bandAllocation;


        for(int j4=0;j4<STAGE;j4++)
        {
              delete [] trueAllocation[j4];
        }

        delete []trueAllocation;
        }
        outfile.close();
        return 0;
}
```

```
// read the number of stages of the traffic sources
void readStage()
{
      fstream file1;
      file1.open("4.txt",ios::in);
      file1>>STAGE;
      file1.close();
}

// read the traffic from a file to an array
void readTraffic(int a)
{

      for(int num=a*3;num<(a+1)*3;num++)
      {
            fstream file1;
            char *buffer;
            buffer=new char[9];
            sprintf(buffer,"%d.txt",num);
            file1.open(buffer,ios::in);
            int c1=num%3;

            file1>>STAGE;
            for(int c2=0;c2<STAGE;c2++)
            {
                  file1>>Traffic[c2][c1];

            }
            file1.close( );
            delete [] buffer;
      }

}

// calculate the TBs' sizes
void getTBSize()
{
      for(int i4=0;i4<Num;i4++)
      {
            TBSIZE[i4]=1500;
            initialState[i4]=TBSIZE[i4];
      }
      initialState[Num]=0;
}

void TrueAllocation()
{
      int TotalTrafficInAStage=0;
      int leftSpace=0;

      double ProviderControl[Num]={3,3,3};
      for(int i=0;i<Num;i++)
      {
```

```
      TotalTrafficInAStage=TotalTrafficInAStage+Traffic[stage][i];
      }
      int QueueState = states[stage][Num]-
(int)(SERVICERATE*TIMEINTERVAL);
      if(QueueState<0)
      {
            QueueState =0;
      }
      leftSpace=QUEUESIZE-QueueState;

      for(int j=0;j<Num;j++)
      {
            if(TotalTrafficInAStage!=0)
            {
            double
t1=(double)Traffic[stage][j]/(double)TotalTrafficInAStage;
                  double t2=ProviderControl[j]/(double)Num;

                  if(t1>t2)
                  {

      bandAllocation[stage][j]=(int)(ProviderControl[j]*leftSpace/Num);
                  }
                  else
                        bandAllocation[stage][j]=(int)(Traffic[stage][j])
                  *leftSpace/TotalTrafficInAStage;
            }
            else
            {
                  bandAllocation[stage][j]=0;
            }

            if(bandAllocation[stage][j]>=Traffic[stage][j])
            {
                  trueAllocation[stage][j]=Traffic[stage][j];
            }
            else
            {
                  trueAllocation[stage][j]=bandAllocation[stage][j];
            }
      }

      return ;
}

// calculate the confirmed traffic
void confirmTraffic(int control[])
{
      for (int i=0; i<Num; i++)
      {
            if (Traffic[stage][i]<=(states[stage][i]+ control[i]))
            {
                  confirmedTraffic[i]=Traffic[stage][i];
            }
            else confirmedTraffic[i]=0;
```

```
	}

}

// calculate the traffic accepted by Multiplexer
int AcceptedTrafficByMul()
{
	int totalAcceptedPacket=0;
	int leftSpace=0;

	int totalConfirmedTraffic=0;

	for(int i=0;i<Num;i++)
	{
		totalConfirmedTraffic += confirmedTraffic[i];
	}

	int QueueState = states[stage][Num]-
(int)(SERVICERATE*TIMEINTERVAL);
	if(QueueState<0)
	{
		QueueState =0;
	}
	leftSpace=QUEUESIZE-QueueState;

	if(totalConfirmedTraffic>leftSpace)
	{
		for(int k=0;k<Num;k++)
		{
			if(leftSpace>=confirmedTraffic[startIndex])
			{
				leftSpace=leftSpace-confirmedTraffic[startIndex];

	totalAcceptedPacket+=confirmedTraffic[startIndex];
				startIndex=(startIndex+1)%Num;
			}
			else
			{
				break;
			}
		}
		return totalAcceptedPacket;
	}
	else
	{
		return totalConfirmedTraffic;
	}
}

// calculate the waiting losses at the multiplexer
int WaitingLost()
{
	int QueueState = states[stage][Num] -
(int)(SERVICERATE*TIMEINTERVAL);

	int TrafficEnterMul;
```

```
        if(QueueState >0)
        {
              QueueState=QueueState;
              TrafficEnterMul=AcceptedTrafficByMul();
        }
        else
        {
              QueueState = 0;
              TrafficEnterMul = AcceptedTrafficByMul();
        }
        return QueueState + TrafficEnterMul;
}
```