INFORMATION TO USERS

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

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

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

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

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

UMI®

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:	LI, Bo
ADRESSE POSTALE / MAILING ADDRESS:	717-1339 MEADOWLANDS DRIVE NEPEAN ON K2E7B4
GRADE / DEGREE: M.Sc. (Systems Science)	**ANNÉE D'OBTENTION / YEAR GRANTED** 2003
TITRE DE LA THÈSE / TITLE OF THESIS: ON THE OPTIMIZATION OF THE TOKEN BUCKET CONTROL MECHANISM	

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

Jan. 7th. 2003

DATE

Li Bo

(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

LI, Bo
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

On the Optimization of the Token Bucket Control Mechanism

Luis Orozco-Barbosa
DIRECTEUR DE LA THÈSE - THESIS SUPERVISOR

EXAMINATEURS DE LA THÈSE - THESIS EXAMINERS

T.H. Yeap

J.-M. Thizy

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

On the Optimization of the Token Bucket Control Mechanism

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

Bo Li

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

© Bo Li, Ottawa, Canada, 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.

0-612-76536-9

Canadä

// ACKNOWLEDGEMENTS

I would like to thank my thesis supervisors, Dr. N.U.Ahmed and Dr L. Orozco-Barbosa, for their support, understanding and guidance throughout the completion of my thesis. Their constructive suggestions was a constant source of motivation.

I would also like to thank Dr. Charalambos Charalambous and Dr. Jean-Michel Thizy. Their lectures on control and optimization were great sources of my inspiration.

I am grateful to Dr. X. Hua, Mr. Q. Wang, Ms. H. Yan and other colleagues for their valuable discussions and suggestions.

DEDICATION

To my parents, my husband Henry and my 5-years old daughter Amy, I wish to express my gratitude for their extraordinary encouragements, patience and support.

ABSTRACT

During the past years, there has been an increasing interest in the design and development of network traffic controllers capable of ensuring the QoS requirements of a wide range of applications.

In this thesis, based on previous work, we use a systems approach to construct a dynamic model for the token bucket mechanism : a traffic controller widely used in various QoS-aware protocol architectures. An enhanced model of multiplexor is also added into the multiple token buckets system model. In this way, the model represents a complete system at the access node of the network.

We then develop an optimization algorithm based on a dynamic programming and genetic algorithm approach. Applying two MPEG-1 video traces and two self-similar traffic traces, we conduct an extensive campaign of numerical experiments allowing us to gain insight into the operation of the controller and evaluate the benefits of using a genetic algorithm approach to speed up the computation process based on dynamic programming. Our results show that the optimization is not only capable of getting the best cost, but also balancing the costs corresponding to different aspects. The results also shows that the use of the genetic algorithm proves particular useful in reducing the computation time required to optimize the operation of a system consisting of multiple token-bucket regulated sources.

Contents

ACKNOWLEDGEMENTS i

DEDICATION ii

ABSTRACT iii

Contents iii

List of Figures v

List of Tables vii

List of Acronyms ix

List of Symbols x

1 Introduction and Motivation 1

1.1 Background 1

1.2 Motivation and Objective 2

1.3 Thesis Organization 3

2 Basic Principles and Literature Review 4

2.1 Traffic Regulation at Access Node 4

2.2 Optimization Methodologies 8

2.2.1 Dynamic Programming Principle (DP) 9
2.2.2 Genetic Algorithms 12
2.3 Final Remarks 15

3 Computer Networks System Model **16**
3.1 Traffic Model 17
3.2 System Model 17

4 Optimization of System Performance **22**
4.1 Dynamic Programming Equation for Token Bucket Mechanism 23
4.2 Solution of the Dynamic Programming Equation(DPE) 25
4.3 Reduction of Search Time Using Genetic Algorithms 27

5 Implementation and Numerical Results **33**
5.1 Network Efficiency Metrics 34
5.2 Specification of the Traffic Traces 34
5.2.1 MPEG Traffic Trace 35
5.2.2 Self-similar Traffic Trace 36
5.3 Experimental Results and Analysis 38
5.3.1 Experiment Using MPEG Traffic 39
5.3.2 Experiment Using Self-similar Traffic 53
5.3.3 Optimization Using Genetic Algorithm 55

6 Conclusion **60**

Bibliography **62**

Appendix: Source Code **65**

List of Figures

2.1 Bufferless token bucket 6

2.2 Buffered token bucket 7

2.3 An example of crossover operation 14

3.1 Traffic model 17

3.2 System model 18

4.1 Multiple stage 24

4.2 Multi-stage process of token bucket control problem 26

4.3 Optimal control token bucket algorithm flow chart(Part 1) 28

4.4 Optimal control token bucket algorithm flow chart (Part 2) 29

4.5 Formation of genome and population 30

5.1 MPEG-1 traffic traces 35

5.2 Self-similar traffic traces 37

5.3 Dependence of cost on control strategy(C=1.456 Mbps) 41

5.4 Dependence of cost on control strategy (C=2.184 Mbps) 41

5.5 Incoming traffic vs. conforming traffic (fixed u=mean traffic) 43

5.6 Incoming traffic vs. conforming traffic (fixed u=1.3x mean traffic) 44

5.7 Incoming traffic vs. conforming traffic (fixed u=mean traffic). 46

5.8 Incoming traffic vs. conforming traffic (fixed u = 1.6 x mean traffic). . . . 47

5.9 Optimal token generation rates 48

5.10 Dependence of cost on Q and C. 50
5.11 Dependence of cost on initial state. 51
5.12 Dependence of cost on initial state. 52
5.13 Dependence of losses on control strategy. 55
5.14 Dependence of running time T(p,g) on p and g (Case 1). 57
5.15 Dependence of running time T(p,g) on p and g (Case 2). 58
5.16 Dependence of cost J(p,g) on p and g (Case 1). 58
5.17 Dependence of cost J(p,g) on p and g (Case 2). 59

List of Tables

5.1 Summary of MPEG-1 traffic traces specification 36

5.2 Statistics of self-similar traffic traces 38

5.3 Average control rate 50

List of Acronyms

CBR	Constant Bit Rate
DP	Dynamic Programming
DPE	Dynamic Programming Equation
GA	Genetic Algorithm
HGA	Hybrid Genetic Algorithm
IDP	Iterative Dynamic Programming
LRD	Long Rang Dependence
MPEG	Moving Picture Experts Group
QoS	Quality of Service
RSVP	Resource Reservation Protocol
TB	Token Bucket
VBR	Variable Bit Rate

List of Symbols

c	Link rate from TB to multiplexor
C	Network link rate
F_ρ	Transition function of TB state
F_q	Transition function of buffer state
$g(t_k)$	Conforming traffic at time $[t_k, t_{k+1})$
J	Total cost
$L(t_k)$	Losses at multiplexor at time $[t_k, t_{k+1})$
$q(t_k)$	Multiplexor buffer state at time $[t_k, t_{k+1})$
Q	Multiplexor Buffer capacity
$r(t_k)$	Non-conforming traffic at time $[t_k, t_{k+1})$
T	Token bucket capacity
$u(t_k)$	Number of tokens generated at time $[t_k, t_{k+1})$
$v(t_k)$	Incoming packet size at time $[t_k, t_{k+1})$
V	Value function
X	State variable
α	Weight given to multiplexor losses
β	Weight given to TB losses
γ	Weight given to waiting losses
$\rho(t_k)$	Token bucket state at t_k
τ	Time interval

Chapter 1

Introduction and Motivation

1.1 Background

During the last years, there has been an increasing interest in the definition of simple but yet effective traffic control schemes able to ensure the grade of service required by a wide variety of applications in computer network. By effective, it is understood that the control mechanism should also ensure a proper level of utilization of the network resources, a fundamental condition to ensure the profitability of operation. The token bucket mechanism has become one of the most effective traffic control mechanism which has been widely used in various applications and studied in literature.

In [2], a dynamic model was developed for the Token Bucket Algorithm. The overall model consisted of a number of sources and corresponding TB-s connected to a multiplexor at the edge of the network. Two objective functions have been defined in terms of the QoS from the point of view of the network providers. A simple feedback control has been employed to control the token generation rate in order to improve QoS and network utilization. The feedback control law can dynamically provide a suitable token generation rate and allocate available bandwidth based on the incoming traffic, and states of both, the TBs and the

multiplexor. The numerical results showed the effectiveness of the feedback control law in meeting the QoS requirements.

1.2 Motivation and Objective

The motivation of this thesis is based on the following three major points. First, most studies in the past [1][3][9][15] have focused on the study of a single token bucket without considering the other traffic sources or multiplexor which are involved in the whole system at the access node. The work in [2] has been proved effective by constructing a multiple TB model and a shared multiplexor model. But the system model has its limitation. Second, previous works have not attempted to optimize the token generation rate in a multiple-token environment. According to the numerical results and the analysis in [2], the feedback control law is apparently powerful to reduce the losses, while increasing the network utilization. The optimization of the system operation should prove beneficial in understanding the TB operation. Last, past studies have been conducted assuming a particular type of application (e.g., video or voice) or traffic statistics (e.g. ON/OFF, LRD dependence). But in the real world, the traffic sources most likely emit various types of data instead of only one. The TB control mechanism that is designed for only one type of data might not be able to control other types of data effectively. Therefore, the design of the traffic controller should be take into account various types of traffic.

Motivated by the major problems of traffic control mechanisms as described above. This thesis addresses the following points. To complete the system model in [2], we define the relationship between the TB states and the link capacity that constraints the traffic transmission from the TB to the multiplexor. A new set of mathematical equations taking into account this relationship is formulated. On the other hand, to get the best cost regarding to the overall packet losses from multiple token buckets, the losses at the

multiplexor and the waiting delay, we find an optimal token generation rate which can improve the QoS and the network utilization by using system approach. Considering the system as a whole, we minimize the cost functional subject to the system constraints. The design of the optimal token bucket control law considers the adaptation of various types of data and scenarios. In this thesis, we look at ways to reduce the computation time of dynamic programming by using genetic algorithms. These algorithms prove useful by considerably reducing the search process, particularly as the number of token buckets increases.

1.3 Thesis Organization

The thesis is organized as follows. Chapter 2 reviews the basic notation of the token bucket mechanism, others' works related to the topic, the principle of dynamic programming, and genetic algorithms. In Chapter 3, we formally define a dynamic model of the token bucket including the shared resource, multiplexor, and give a detailed description of the model. In Chapter 4, we define the objective functional and provide the details of the dynamic programming/genetic algorithm approach proposed for the optimization of a network control system consisting of multiple token buckets. In Chapter 5, we describe the implementation of the algorithm, then conduct a set of numerical experiments to illustrate the effectiveness of the proposed approach and gain the insight into the operation of the token bucket. In Chapter 6, we conclude the practical issues of the optimization process, analyze the benefits of the proposed methodology by assessing the computation time reduction and effectiveness of the optimal control as applied to token bucket mechanism.

Chapter 2

Basic Principles and Literature Review

In this chapter, we review some notations and principles on which the following chapters are based. We also briefly overview the research efforts and principles in the area of network access control and optimization methodologies.

2.1 Traffic Regulation at Access Node

There are various types of services which are carried out in nowadays networks, such as data, voice and video. The traffic patterns generated by these services can fall into two main categories : 1) Constant bit rate (CBR) traffic which is widely used in current telephone system, such as T1 lines and voice grade lines; 2) Variable bit rate (VBR) traffic which is widely used in computer network and requires efficient traffic controls, to meet the application's QoS requirements. Our work focus on the control of VBR traffic.

To control the VBR traffic at the edge of the network, there are two important traffic regulation mechanisms: policing and shaping. These two mechanisms allow the conforming

traffic to get into the network. Conforming traffic is defined as the traffic which are conformed by token bucket to be able to get into the network. Traffic policing manages the maximum rate of the traffic. In the most common traffic policing forms, Token buckets (TBs) are responsible to regulate the maximum rate. One can adjust the TB configurations, such as TB size and token generation rate, to suit the network and user needs. In our work, a newly designed token bucket model and optimal control algorithm are formulated. Token bucket is a common algorithm used to police the traffic rate. A specific configured traffic rate or a dynamic derived traffic rate based on the congestion can be specified to a token bucket. If a buffer is added to a TB, some of the non-conforming traffic would wait in the buffer. This kind of operation can be also described as traffic shaping.

Since Turner designed a first version of the TB mechanism [1] in 1986, the TB mechanism has been extensively studied and a wide variety of different versions of the TB mechanism have been developed and employed in various control areas in computer networks, such as admission control and traffic shaping among others.

The basic architecture of the TB [1] is pretty easy to understand. In its simplest form, a token bucket is characterized by its capacity expressed in tokens, a token generation rate created and placed in the bucket. A TB mechanism is associated with each network subscriber, from now on referred as a user. The rate at which the user may send his traffic into the network is regulated by the TB mechanism. The TB builds up tokens of fixed size, typically one byte each token, at a constant rate. In the most general form, a TB has a fixed capacity. Upon a packet arrival, if the number of tokens is equal to or greater than the packet size, the packet is sent into the network by consuming corresponding number of tokens. On the contrary, if no enough tokens are available, the packet is classified as belonging to the non-conforming traffic which will be discarded by the TB. The tokens in the bucket can be saved up when the user does not have anything to send. If the bucket

becomes full, the newly generated tokens will be discarded. The TB is shown in Figure 2.1. Besides the TB models described above, some other models define that one token takes on one packet, or more than one byte instead of just one byte.

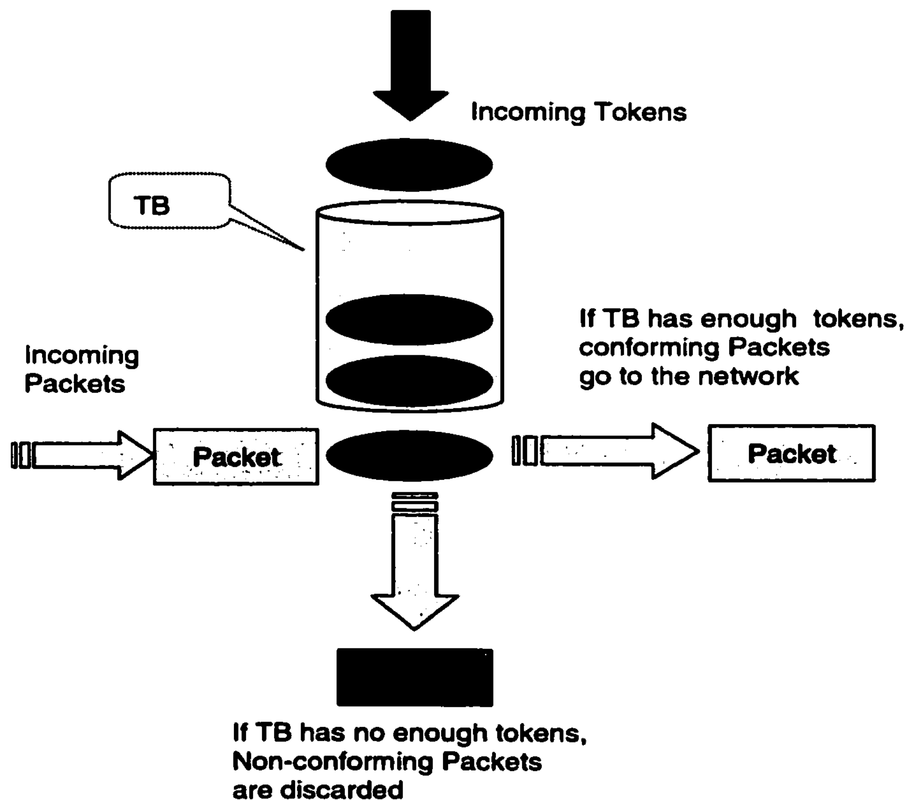


Figure 2.1: Bufferless token bucket

The TB can be also designed as a buffer based model, in which a buffer is added at the TB (see figure 2.2). If the arrival packet can not be served immediately by the available tokens in the TB, it is stored in the buffer and waits for new generated tokens. The packets are admitted into the network as soon as enough tokens are made available. If the buffer is full, the incoming traffic that can not be accepted by the buffer is discarded. This form of operation can decrease the packet losses at TB. In this case, the TB acts as

a shaper as well.

The token bucket mechanism has been studied widely in the literature. In recent years, various authors have used the token bucket model to characterize the traffic generated by different types of sources, such as video and voice.

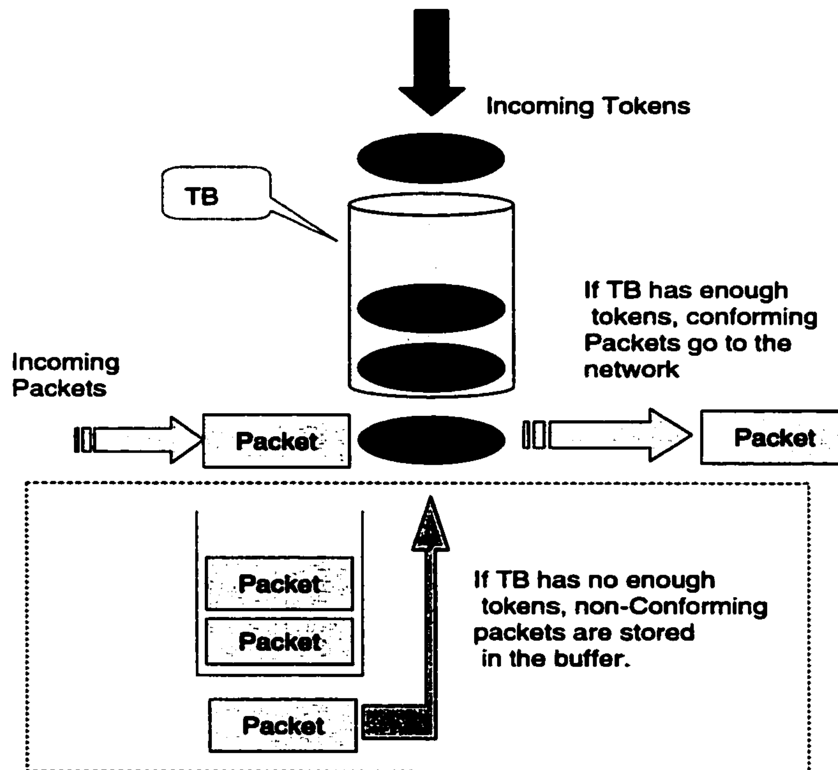


Figure 2.2: Buffered token bucket

In [18], Lombardo et al. have used the Token Bucket model to characterize the traffic generated by pre-recorded MPEG video. The main objective of their work has been to provide an analytical methodology to compute the traffic specifications used by RSVP: the signaling protocols of the IntServ protocol architecture [23]. It combines a smoother

followed by a token bucket used as a traffic shaper.

In [19], Procissi et al. have also used the token bucket to characterize traffic exhibiting long-range dependence (LRD). Their numerical results have shown the effectiveness of their analytical approach for the proper sizing of the token bucket parameters for LRD traffic.

In [15], Bruni and Scoglio have also used the token bucket model to optimize the traffic generation rate of video sources when satisfying the token bucket controls. In the article, an optimal control problem is formulated. A minimum packet losses and packet delay are attainted by optimizing the emitting bit rate of the VBR MPEG traffic at the server side. The goal is to ensure, as much as possible, a good video quality and avoid network congestion.

Besides the above applications of token bucket, various token bucket algorithms have been defined by some other authors as well. For example, In [20], Bruno et al. have estimated the token bucket parameters for voice over IP traffic. In [5], Tang and Tai have conducted a network traffic characterization using the token bucket algorithm. Some authors have also analyzed the operation and performance of a token bucket using a buffer to store the incoming traffic.

2.2 Optimization Methodologies

Optimal control theory, such as the Principle of Dynamic Programming , the Minimum Principle of Pontryagin, or neural-networks, can be used to minimize the cost functional. In this thesis, we use the Bellman's principle of dynamic programming to solve the main

problem and use a genetic algorithm (GA) to reduce the search time.

2.2.1 Dynamic Programming Principle (DP)

Richard Bellman [5], first introduced dynamic programming in 1957. Since then, there has been a great deal of interest in using the idea which is becoming a popular and a powerful optimization methodology.

Bellman described the way of solving problems in order to find the best decisions one after another. According to Bellman's Principle of Optimality [5]:

An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.

We can divide the overall problem into sub-problems without violating the optimal policy. Dynamic programming can make the complicated problem easier to solve. A complicated problem can be seen as a "dynamic" problem which is composed of time dependent stages. The state variable is defined in such a way that it can describe the process at a specific stage. Given the state of the process at the beginning of a stage, a decision, which is known as u (the control variable), is made. That gives a return at the stage and transform the process to the ending stage. The decision u must be optimal such that the overall return of all stages from current stage to the ending stage must be the minimum or maximum which depends on the requirements of the objective function.

Considering a system governed by a state equation with $X(k)$ denoting the state variable at time stage k and $u(k)$ the control variable or the decision variable, we formulate the following system equations:

$$\begin{cases} X(k+1) \equiv F(k, X(k), u(k)), k = 0, 1, 2, \cdots, N-1 \\ X(0) = X_0 \end{cases} \tag{2.1}$$

where $X(k) \in R^N$ and $U(k) \in R^N$.

The controls belong to a specified set $\mathcal{U}_{ad}$ called the class of admissible controls. The cost functional is defined as:

$$J(u) \equiv \sum_{k=0}^{N-1} \ell(k, X(k), u(k)) + W(N, X(N)) \tag{2.2}$$

where the first term represents the running cost and the last term determines the terminal cost. The objective is to find a control policy from the admissible class $\mathcal{U}_{ad}$ that minimizes the cost functional J. We formulate this as a dynamic programming problem. Given the state $X(r)$ at time r, we may define

$$J(r, X(r), u) \equiv \sum_{k=r}^{N-1} \ell(k, X(k), u(k)) + W(N, X(N)) \tag{2.3}$$

as the cost of operating the system from time r starting from state $X(r)$ and using the control policy $u \in \mathcal{U}_{ad}$, where $\mathcal{U}$ is a closed bounded subset of R^m. We need to find a control policy u^* such that $J(u^*) \leq J(u)$. Let $I = 0, 1, 2, ..., N$ and $r \in I$. Now we may define

$$V(r, X(r)) \equiv \inf\{J(r, X(r), u), u \in \mathcal{U}_{ad}\} \tag{2.4}$$

as the value function, where $V(r, X(r))$ gives the best performance at time r. Clearly this gives the minimal cost to run the system starting from state $X(r)$ at time r until the end of the whole process N. Using the basic arguments of dynamic programming one can readily prove the dynamic equation 2.8as follows:

Proof:

According to equation 2.3 and 2.4, we know

$$\begin{aligned} V(r, X(r)) &\equiv inf_{u\in U_{ad}}\{l(r, X(r), u(r)) + \sum_{k=r+1}^{N-1} \ell(k, X(k), u(k)) + W(N, X(N))\} \\ &\leq l(r, X(r), u(r)) + inf_{u\in U_{ad}}\{ \sum_{k=r+1}^{N-1} \ell(k, X(k), u(k)) + W(N, X(N))\} \\ &\leq l(r, X(r), u(r)) + V(r+1, X(r+1)). \end{aligned}$$

This implies the following:

$$V(r, X(r)) \leq inf_{u\in U_{ad}}\{l(r, X(r), u(r))\} + V(r+1, X(r+1)). \tag{2.5}$$

For $\forall\varepsilon > 0, \exists u$, s.t

$$\begin{aligned} V(r, X(r)) &\geq l(r, X(r), u(r)) + \sum_{k=r+1}^{N-1} \ell(k, X(k), u(k)) + W(N, X(N)) - \varepsilon \\ &\geq l(r, X(r), u(r)) + V(r+1, X(r+1)) - \varepsilon \\ &\geq inf_{u\in U_{ad}}\{l(r, X(r), u(r))\} + V(r+1, X(r+1)) - \varepsilon \end{aligned}$$

Let $\varepsilon \longrightarrow 0$, we get

$$V(r, X(r)) \geq inf_{u\in U_{ad}}\{l(r, X(r), u(r))\} + V(r+1, X(r+1)). \tag{2.6}$$

Combining (2.5) and (2.6), we have,

$$V(r, X(r)) \equiv inf_{u\in U_{ad}}\{l(r, X(r), u(r))\} + V(r+1, X(r+1)). \tag{2.7}$$

Substituting X(r+1) in the above equation (2.7) by the right hand side of the equation (2.2), the value function V must satisfy the the recursive equation 2.8.

$$\begin{cases} V(k, x(k)) \equiv Inf_{u\in Uad}\left\{\ell(k, x(k), u(k)) + V(k+1, F(k, x(k), u^*(k))\right\}, k = 0, 1, \cdots N-1 \\ V(N, x(N) \equiv W(N, x(N)) \end{cases} \tag{2.8}$$

Equation (2.8) is known as the Bellman equation of dynamic programming.

To solve the equation, several methods developed by various scientists have been proved effective, such as Lagrange multipliers in [5][6], tabular computation in [6], or most easily a quadratic programming algorithm in [5][6]. Some other algorithms, such as IDP (Iterative Dynamic Programming)[7], have also been used to shorten the computation time required by the optimization process.

In our work, due to the high dimension of the problem, the state and control sets are spread over a very large range of multiple integers. Therefore the traditional tabular computation, which iterates all the admissible U_{ad} under each state, would take an unacceptable long time. For this reason, we consider the use of Genetic Algorithms [8] which should allow us to find the optimal solution without having to explore all the possible solutions.

2.2.2 Genetic Algorithms

The genetic algorithm was invented by John Holland and his colleagues at the University of Michigan in 1970's. It has become a part of evolutionary computing applied to a rapidly growing research area. Since the early 1980's, The genetic algorithms (GA) have been experienced an abundance of applications which spread across a large variety of disciplines. GA or the variations of GA has been applied in computer network and telecommunication area to attain the best utilization of the available resource. In [21], Kassotakis, Markaki and Vasilakos have applied hybrid Genetic algorithm in channel sharing/reusing method to accomplish the establishment of a maximal number of connections with minimal number of isochronous channels. A study of the simulation results show that when the problem space increases the HGA algorithm is more efficient than others.

Genetic algorithms work very well on combinatorial problems. They are less susceptible

to falling into local optima than other search methods, such as gradient search or Random Search, since the genetic algorithms traverse the search space using the genotype rather than the phenotype [12]. However, GAs tend to be computationally expensive. A genetic algorithm is a search procedure that optimizes a certain objective function by maintaining a population of candidate solutions. It employs some operations that are inspired by Darwin's theory about evolution to generate a new population based on the better fitness from the previous one.

A typical GA consists of the following [8]:

1. Encoding is a scheme that describes how to represent the value of a solution in a chromosome. The most common encoding scheme is binary encoding that is to convert the value to a string of bits, which could be its binary representative or some other alternatives in binary form. And permutation encoding and value encoding have been used with some success as well.

2. A population is a group of guesses of the solution to the problem. It is a small set of solutions

3. Within a population, each individual or solution is called a chromosome, or genome, which can be represented by a set of bits, string, or some other structures. Genome contains genes which represent the values of unknowns.

4. Fitness is a method of calculating how good or bad the solutions within the population are.

5. A selection method is scheme used to choose the best solutions to form a new population generation.

6. A crossover is an operator to generate children from the parent chromosomes, or say solutions. An example of crossover operation is shown in Figure 2.3. The encoding

scheme of the example is binary encoding.

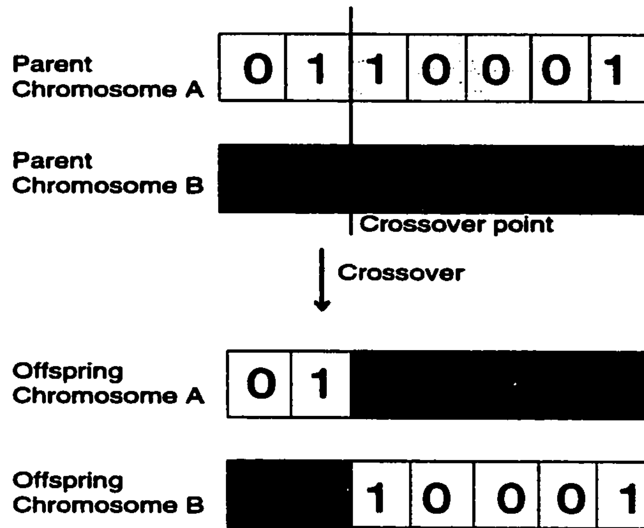


Figure 2.3: An example of crossover operation

7. A mutation operator is to maintain the diversity of the solutions. At each locus of the chromosome, the value of that locus might be flipped according to a user specified mutation probability. In a binary encoded chromosome, flipping means change 0 to 1, or change 1 to 0.

The basic operation of a GA starts from the random selection of its initial population, which is also known as the population at the first generation, with a specific size from a large search space. The fitness of each chromosome in the population is evaluated by user defined fitness function. Two parent chromosomes of the offspring are randomly selected from the current population according to some selection schemes. The better fitness of the chromosome, the higher chance to be chosen. The crossover is done between two parent chromosomes, then two new chromosomes are generated. The number of crossover points could be one or more. To improve the diversity of the chromosomes being generated, mutations are applied to some locus of each chromosome according to a mutation probability.

The newly generated offspring is put into the next generation. This operation continues until the population size of the next generation is full. After a number of generations of operation, the best solution are obtained.

GAs have several variations. The two most common GAs are 1) simple GA, also called generational GA; 2 Steady-state GA. Simple GA is the original algorithm based on the idea of John Holland. When the simple genetic algorithm reproduces, the entire set of individuals is replaced by their children. Even with an elitism strategy, many of the best individuals that have been found may not reproduce. Another drawback of simple GA is that the number of generation replacement makes the number of fitness evaluation relatively large. Different from simple GA, steady state GA only replace a few individuals in each generation. The number or the percentage of replacement can be user specified. This type of replacement is often referred to as overlapping populations [12]. To avoid the lost of the best individuals and speed up the search speed. We apply steady-state GA to solve the optimal traffic control problem.

2.3 Final Remarks

In this chapter, we have introduced the work done in the analysis of the token bucket. This has allowed us to have a clear understanding of the application and issues involved in the design of traffic controllers. It is clear that the success of these traffic controllers depend on their effectiveness. Therefore, it is important to optimize their operation. We have also reviewed the basis of discrete time dynamic programming and genetic algorithms. These will be used in the later chapters towards the optimization of the non-linear, multi-stage TB operation.

Chapter 3

Computer Networks System Model

In this chapter, we present a dynamic model based on the basic philosophy of IP networks. This model includes some improvement over the previous system model developed in [2]. Traffic model, TB model and multiplexor model are all involved in the system model introduced below, which resent a new methodology to solve the optimal control problem in computer communication network. Before describing the system model, let us define the following symbols which will be used throughout the thesis.

1. $\{x \wedge y\} = Min\{x, y\}$, $\{x \vee y\} = Max\{x, y\}, x, y \in R$;
2. For $X, Y \in R^n$ the components of the vector $Z \equiv \{X \wedge Y\}$ are given by $z_i \equiv x_i \wedge y_i, i = 1, 2 \cdots n$, where $X = \{x_i, i = 1, 2, \cdots, n\}$ and $Y = \{y_i, i =, 1, 2, \cdots, n\}$. Similarly for $Z \equiv X \vee Y$ we write $z_i \equiv x_i \vee y_i$.
3. $I(s) = \begin{cases} 1, & \text{if the predicate S is true} \\ 0, & \text{otherwise.} \end{cases}$.

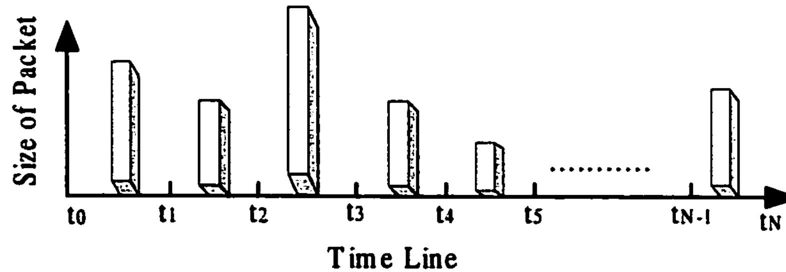


Figure 3.1: Traffic model

3.1 Traffic Model

The traffic model considered here, which is shown is Fig 3.1, is built up according to the VBR traffic pattern. It remains the same as in [2]. The incoming traffic is denoted by$\{v(t_k), k = 1, 2, ...N\}$. Since during each time interval $[t_k, t_{k+1})$ at most one packet may arrive, $v(t_k)$ then represents the length or the size of the packet. The time interval $[t_k, t_{k+1})$ must be small enough to ensure that at most one variable-sized packet arrives.

The VBR traffic can be very bursty. Especially for voice or video transmission, the peak rate could be far larger than the average rate. Therefore the bandwidth allocation to the traffic is hard to determine. The peak rate allocation may lead to future congestion, while the mean rate or lower than the mean rate allocation would result in losses, waste of network resources, and the QoS may not be guaranteed.

3.2 System Model

The system model which is considered here lays on the access node of a network. In order to model a typical network access node, we consider that the system involves n traffic sources, and each source is assigned a TB. Figure 3.2 shows the system model.

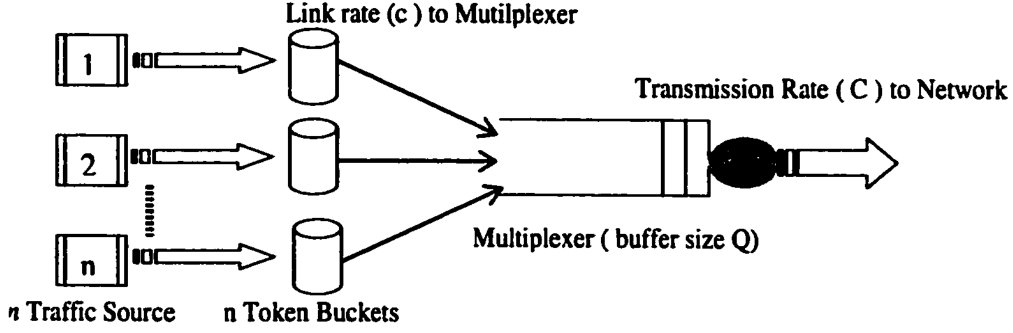


Figure 3.2: System model

Following [2], the TB model described herein is a bufferless token bucket, which is modified to take into account the link rate c_i, of the input link i, where $i = 1, 2, \ldots n$. c_i is the link rate between the $i - th$ token bucket and the multiplexor. Considering the multiple source nature of an access node to the network, we use c to denote the vector of the input link rates, where $c = (c_i, i = 1, 2, ..., n)$.

The system model is formally given by the following vector difference equation:

$$\rho(t_k) = \rho(t_{k-1}) + \{u(t_{k-1}) \wedge [T - \rho(t_{k-1})]\}$$

$$- v(t_{k-1}) I\Big(v(t_{k-1}) \leq \Big\{[\rho(t_{k-1}) + [u(t_{k-1}) \wedge (T - \rho(t_{k-1}))]] \wedge c\tau\Big\}\Big) \quad (3.1)$$

where the vector $\rho(t_k) = (\rho_i(t_k), i = 1, 2, ..., n)$ denotes the status of token occupancy of all the token buckets at time t_k, that is, $\rho_i(t_k)$is the number of tokens in the ith TB at time t_k. $u(t_{k-1})$ is also a vector $(u_i(t_{k-1}), i = 1, 2, ..., n)$, which denotes the incoming tokens to each TB at the start of the time interval $[t_{k-1}, t_k)$. The vector $T = (T_i, i = 1, 2, ..., n)$ denotes the size (maximum capacity) of all the TB-s , $v(t_{k-1}) = (v_i(t_{k-1}), i = 1, 2, ..., n)$ is a vector denoting the size of the arriving packets from each source during the time interval $[t_{k-1}, t_k)$, and τ is the length of the time interval.

The conforming traffic at time $[t_k, t_{k+1})$, (that is, the traffic matching with the available tokens in the token banks) from each of the TB-s is then given by:

$$g_i(t_k) = v_i(t_k) I \{v_i(t_k) \leq [\rho_i(t_k) + (u_i(t_k) \wedge (T_i - \rho_i(t_k)))] \wedge c_i \tau\} \tag{3.2}$$

Clearly this means that the non-conforming traffic, denoted by $r_i(t_k)$, is given by

$$r_i(t_k) = v_i(t_k) - g_i(t_k) \tag{3.3}$$

In this study, we assume that the (traffic) sources share the same multiplexor in an access node at the edge of the network. The multiplexor collects all the conforming traffic that can be accepted in its buffer before they are launched on to the outgoing link depending on the available bandwidth or capacity at the time. Thus, to complete the dynamic model of the whole system, one must also include the temporal variation of the queue at the multiplexor. For simplicity we may assume that the link rate is piecewise constant on each of the intervals $[t_{k-1}, t_k)$ and it is denoted by $C(t_{k-1})$, and the multiplexor has a finite buffer capacity Q. Let the size of the queue, $q(t_k)$, which is the total traffic in the buffer waiting for service at the multiplexor at time t_k, denotes the state of the multiplexor. The dynamics of the multiplexor queue is then given by the following balance equation:

$$q(t_k) = \{[q(t_{k-1}) - C(t_{k-1})\tau] \vee 0\} + \left\{ \left[\sum_{i=1}^{n} g_i(t_{k-1}) \right] \wedge [Q - ([q(t_{k-1}) - C(t_{k-1})\tau] \vee 0)] \right\}. \tag{3.4}$$

The first term on the right hand side of the expression (3.4) describes the leftover traffic in the queue at time t_k after the traffic has been injected into the network. The second term represents all the conforming traffic accepted by the multiplexor during the same period of time. If the network capacity is large enough to serve all the traffic in the queue at time t_{k-1}, the state of the queue or the multiplexor is given by all the incoming traffic

accepted. But in the case that the network could not serve all the waiting traffic, the unserved traffic is stored in the queue. If the buffer with size Q does not have enough space to store all the unserved traffic the excess is discarded.

Because of the limitation of the buffer size and the output link capacity, some traffic may be dropped at the multiplexor. Thus the traffic losses at the multiplexor at time $[t_k, t_{k+1})$is described by:

$$L(t_k) = \left[\sum_{i=1}^{n} g_i(t_k)\right] - \left\{\left[\sum_{i=1}^{n} g_i(t_k)\right] \wedge [Q - ([q(t_k) - C(t_k)\tau] \vee 0)]\right\} \tag{3.5}$$

The term $\left\{\left[\sum_{i=1}^{n} g_i(t_k)\right] \wedge [Q - ([q(t_k) - C(t_k)\tau] \vee 0)]\right\}$ represents the conforming traffic accepted by the multiplexor. If the buffer space, Q, is large enough to accept all the conforming traffic, no multiplexor losses would occur during the time interval $[t_k, t_{k+1})$. Otherwise, some part of the conforming traffic must be dropped.

According to equations (3.1) and (3.4), we may formally write the abstract model, which will be used later in our algorithm and computation, as follows. Let $X = \begin{pmatrix} \rho \\ q \end{pmatrix} \in R^{n+1}$ denote the state of the system consisting of n token banks and one multiplexor. Then the system dynamics can be described compactly by the following system of difference equations,

$$X(k+1) = F(k, X(k), u(k)), k = 0, 1, 2, \cdots, N-1 \tag{3.6}$$

where $F(k, X(k), u(k)) \in R^{n+1}$ is the state transition function. It represents all the expressions on the right hand side of the system of token buckets given by equation (3.1) and those of the multiplexor given by equation (3.4). We shall denote the components of this vector $F(k, X(k), U(k))$ by $F_{\rho_i}(k, \rho(k), u(k)), i = 1, 2, \cdots, n$ and $F_q(k, q(k), u(k))$. Let N_0 denotes the set of all nonnegative integers and N_0^k denote the Cartesian product

of k copies of N_0. It is important to note that

$$F : N_0 \times N_0^{n+1} \times N_0^n \longrightarrow N_0^{n+1}$$

and consequently our system is governed by a vector difference equation in the state space N_0^{n+1}. This fact makes optimization computationally expensive.

Chapter 4

Optimization of System Performance

The basic objective of any computer communication network is the transport of information expressed in various formats (voice, video, data) from source to destination promptly, efficiently and reliably without failure. Before being able to optimize the network operation, we must define an appropriate objective functional that is reflective of these attributes. In view of the basic objectives , it is important to include in the functional all the possible losses that may occur in the network. However, since our research focuses on the traffic regulation at the access node of the edge of the network, the objective functional given in function (4.1) involves all the possible losses at the access node, including the losses at the TB-s and the multiplexor. In addition to these losses, there are delays in service due to waiting times at the multiplexor, which is also included.

$$J(u) = \sum_{k=0}^{K} \alpha(t_k) L(t_k) + \sum_{k=0}^{K} \sum_{i=1}^{n} \beta_i(t_k) r_i(t_k) + \sum_{k=0}^{K} \gamma(t_k) q(t_k). \tag{4.1}$$

The first term of the objective function (4.1) represents the weighted traffic losses at the multiplexor. The second term gives the sum of weighted losses at the TBs and the last term represents the weighted cost of waiting time. The parameters $\alpha(t_k), \beta_i(t_k), \gamma(t_k), i = 1, 2, ...n$ are the weights to which different values can be assigned according to different scenarios and criteria. Note that, the objective functional is a function of u, which is the control

vector. Our goal is to find an optimal control u that minimizes the cost function (4.1). We may recall that by the control u we mean the token supply to each of the individual TB-s during each of the intervals of time $[t_{k-1}, t_k)$ over the entire period of operation from $[t_0, t_K]$.

In this thesis, we use the Principle of Dynamic Programming, introduced by Bellman [5] supplemented by a genetic algorithm to solve the optimization problem. Because of the recursive nature of our system model, the problem can be easily broken up into simpler sub-problems. Since our system evolves in the lattice N_0^{n+1}, we have to consider integer constraints and use a tabular computation in our implementation. In our work, due to the high dimension of the problem, the state and control sets are spread over a very large range of multiple integers, the traditional tabular computation, which iterates all the admissible U_{ad} under each state, would take unacceptable long time. Thus we consider the use of Genetic Algorithms [8] which should allow us to reduce the time required to find the optimal solution. GA can reduce the search time for control values in each sub-problem of dynamic programming, and save computer memory allocation as well. By combining the power of both the dynamic programming and the genetic algorithm, we can determine the optimal control values effectively without using excessive iterations involving all the admissible control values. In fact genetic algorithms can greatly improve the performance of dynamic programming.

4.1 Dynamic Programming Equation for Token Bucket Mechanism

Note that, in our system, there are n traffic sources that result in n states associated with token buckets. There is still one state associated with the multiplexor queue. Thus totally $n + 1$ state variables are given. We use $X(k)$ to represent the state vector where

k is the stage variable that keeps track of the stage index. The control $u(k)$ in dynamic programming is defined as the vector of token generation rates. Because n TBs are considered, each TB is controlled by its own token generation policy. Thus $u(k) \in N_0^n$ is a n tuple vector as well.

To solve the optimal control problem from time t_0 to t_N, we divide the problem into multiple stages indexed from 0 to N, refer to Figure 4.1. Let $k = 0, 1 \ldots N$ be the stage variable. Each stage takes one time interval $[t_k, t_{k+1})$. Therefore, stage $k = 0$ represents time interval $[t_0, t_1), \ldots$, $k = N - 1$ represents $[t_{N-1}, t_N)$ and k=N represents t_N respectively.

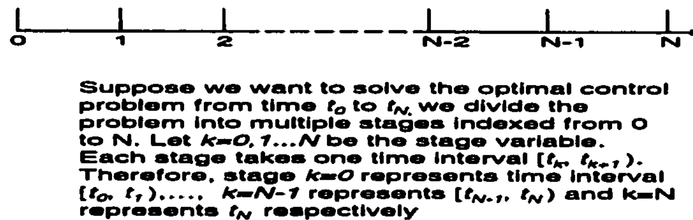


Figure 4.1: Multiple stage

Our task is to study the high dimensional Integer Optimization problem using the discrete time dynamic programming as discussed in the previous section. Accordingly, the complete set of equations required to solve the problem consists of the Bellman equation, the cost functional and the state equation. These are all collected at one position for the convenience of the reader as follows:

$$V(k, X(k)) \equiv inf_{u\in\mathcal{U}_{ad}} \{\ell(k) + V(k+1, F(k, X(k), u(k)))\}, \tag{4.2}$$

$$V(N, X(N)) = 0, \tag{4.3}$$

$$\ell(k) = \alpha(k)L(k) + \sum_{i=1}^{n} \beta(k)r_i(k) + \gamma(k)q(k), \tag{4.4}$$

$$X(k) \equiv (\rho(k), q(k)), F \equiv (F_\rho, F_q), \quad X(k+1) = F(k, X(k), u(k)), \tag{4.5}$$

$$\begin{aligned} F_\rho(k, \rho(k), u(k)) \equiv \rho(t_k) &+ u(t_k) \wedge (T - \rho(t_k)) \\ &- v(t_k) I\Big(v(t_k) \le [\rho(t_k) + u(t_k) \wedge (T - \rho(t_k))] \wedge c\tau\Big), \end{aligned} \tag{4.6}$$

$$\begin{aligned} F_q(k, \rho(k), q(k), u(k)) &\equiv \{(q(t_k) - C\tau) \vee 0\} \\ &+ \left[\sum_{i=1}^{n} g_i(t_k)\right] \wedge [Q - (q(t_k) - C\tau) \vee 0]. \end{aligned} \tag{4.7}$$

4.2 Solution of the Dynamic Programming Equation(DPE)

By using the Dynamic Programming Equation (4.2) along with the terminal condition (4.3), we solve our optimal traffic control problem step by step. This is a sequential approach which eventually leads to the optimal control policy u^*. Before describing the computational sequence, we discuss how the optimal value can be obtained by using dynamic programming. In our network traffic control system, the multi-stage process is shown in Figure 4.2.

The rectangle represents the transition function that contains two operations F_ρ and F_q where ρ and q denote the state variables indicating the states of the TBs and the multiplexor with u denoting the decision variable representing token supply. From Figure 4.2, we can see that the output of one stage is the input of the next. To find the minimum cost (equivalently the value) during time $[t_0, t_N]$, we must find the optimal control value u^* which minimizes the cost of the whole process from stage 0 to stage N giving $V(0, X(0))$. It is clear from the (DPE) (4.2) that the optimum cost $V(0, X(0))$ is the sum

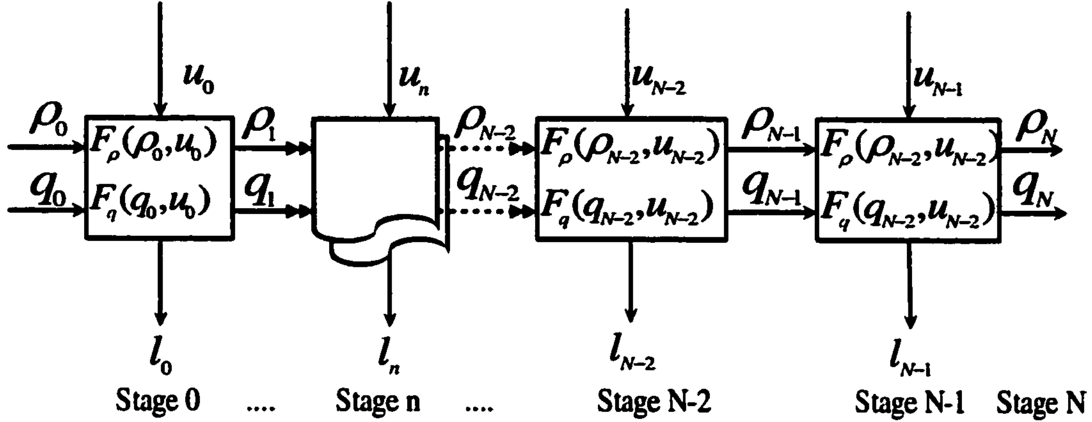


Figure 4.2: Multi-stage process of token bucket control problem

of $\ell(0)$ and $V(1, X(1))$. Thus to compute $V(0, X(0))$ we must know $V(1, X(1))$ before we can find the optimal $u^*(0)$ that minimizes the cost to go from $X(0)$ giving $V(0, X(0))$. This is true for every stage. Hence the DPE must be solved backward in time starting from the last stage. At the terminal time N, $V(N, X(N)) = W(N, X(N))$ for a given function W. To minimize the terminal cost one must choose $u(N-1)$ that minimizes the functional $W(N, F(N-1, X(N-1), u(N-1)))$. Clearly optimal $u(N-1)$ depends on the state at stage $N-1$ and so it may be denoted by $u(N-1, X(n-1))$. This is continued till the initial stage is reached. If the terminal cost is zero this process begins from stage $N-1$. That is why the DPE shown in (4.2) has to be solved from the final stage.

Since we solve the problem backward. Before the whole process finishes, we do not know which state vector at stage k (k=1,2,...N-1) can finally satisfies the given initial state X' at time 0 and which is involved in global optimal state sequence to get the global optima. We have to compute the optimal cost for each admissible state vector $X_i(k)$, where $i = 0, 1, ..., \prod_{j=1}^{n}(T_j + 1)$ denotes the index of the admissible state vectors. During those computations, GA, which is described in next subsection, is applied to search the

optimal solution $u^*(X(k))$. Therefore, at each stage form $k = 2$ to $N-1$, each admissible $X_i(k)$ and its corresponding $u^*(X_i(k))$ must be recorded. The corresponding best cost for each $X_i(k)$ is not necessary to be recorded during the running of the whole process. However, during the running of each stage k, all corresponding $J(k+1, X_i(k+1))$ at $k+1$ stage are required in order to compute the best cost at current stage k. So we must store at least 2 stages of corresponding costs for their state vectors at each stage, until it reaches stage 0. Because the initial state is given as X', at the stage 0, we simply find the $u^*(X')$ which must be the optimal control for stage 0 that minimizes the cost to go from stage 0 to N, and the Optimal cost is finally attained as well. But to get the optimal control sequence, we have to trace back to the terminal stage starting from stage 0. We can easily compute the optimal state $X^*(1)$ and find the corresponding $u^*(X(1))$ by looking up the pre-stored table. This operation continues stage after stage until the end. The whole process is clearly illustrated in Figure 4.3 and Figure 4.4.

4.3 Reduction of Search Time Using Genetic Algorithms

The high-dimensional feature of the token bucket control system makes the number of combinations of states very large. The search space for the control values for each state at each stage is also large. For each TB, the admissible control values are $0, 1, \ldots(T_i - \rho_i(t_k))$. Thus there are $(T_i - \rho_i(t_k) + 1)$ choices for the $i-th$ token bank at time t_k. If there are n TBs, the total number of admissible control values is given by

$$\prod_{i=1}^{n}(T_i - \rho_i(t_k) + 1).$$

For 3 TBs with the same maximum capacity 2000, the average search space of each TB would be 1001, the total number of admissible controls would be 10013 = 1003003001. Obviously, to find an optimal control value of each state vector from such a large set is

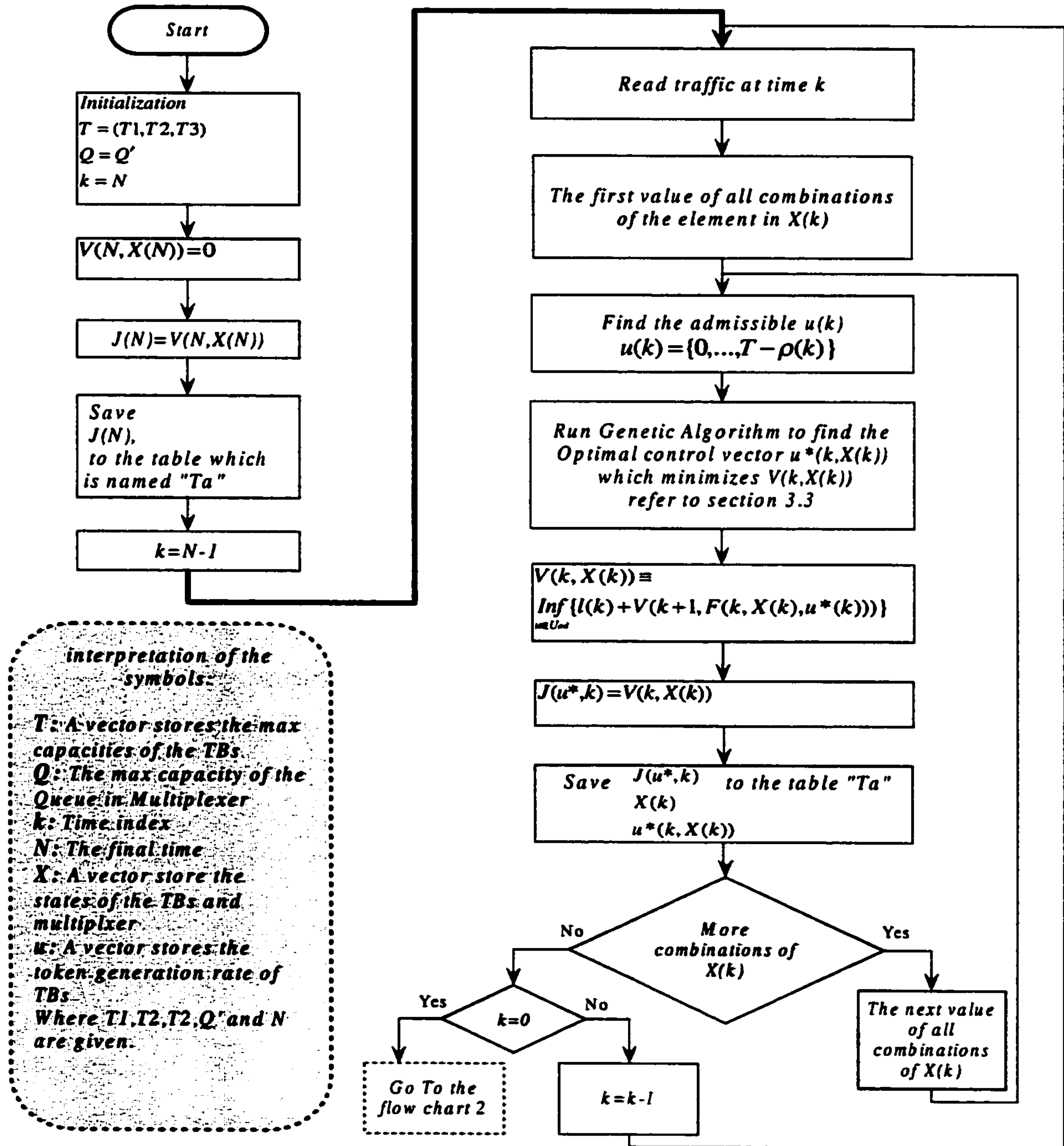


Figure 4.3: Optimal control token bucket algorithm flow chart(Part 1)

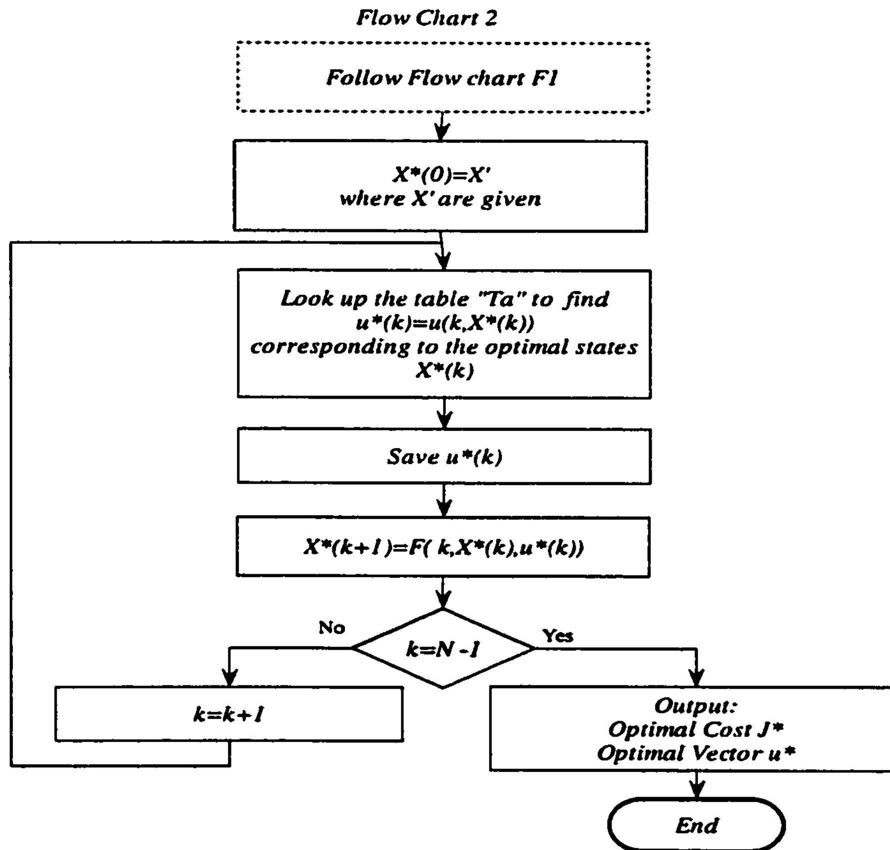


Figure 4.4: Optimal control token bucket algorithm flow chart (Part 2)

not easy. Genetic Algorithms can solve this problem very efficiently.

In our implementation, to avoid the loss of the best individuals and speed up the search process, we use the steady-state GA to solve the optimal traffic control problem. Different from simple GA, steady state GA only replaces a few individuals in each generation. The number or the percentage of replacement can be user specified. This type of replacement is often referred to as overlapping populations [12].

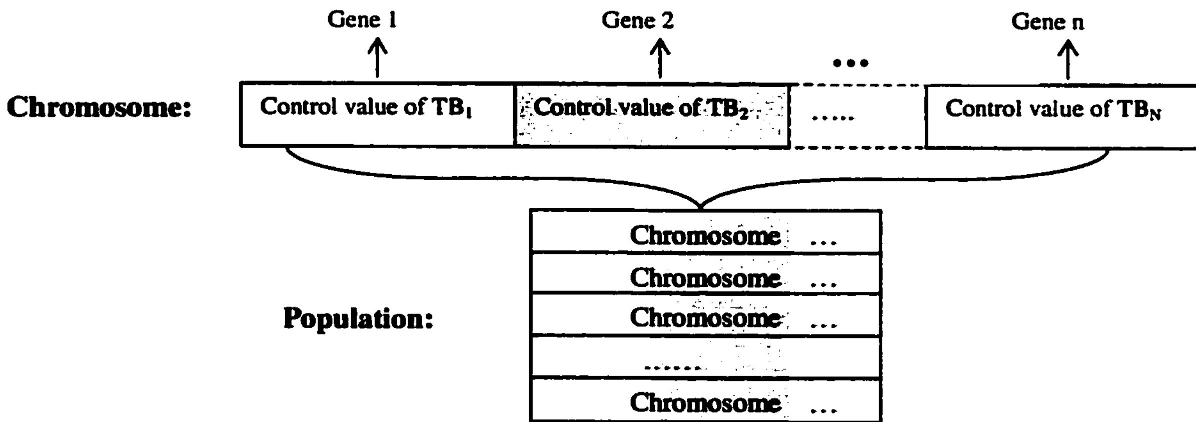


Figure 4.5: Formation of genome and population

For the control problem, the search space is the entire set of admissible control vectors. A population is composed of a small group of selections from the whole control set. Each individual (chromosome) within a population represents a combination of each control value of each TB. If there are n TBs, each individual contains n genes, each gene maps one control value of one TB. The formation of chromosome and the formation of population are shown in Figure 4.5. The fitness function is formulated based on the value function (4.2) as follows:

$$f(k, X(k), u(k, X(k))) = \ell(k, X(k), u(k, x(k))) + V(k+1, F(k, X(k), u^*(k))), k = 0, 1, ..., N-1. \tag{4.8}$$

From the above equation (4.8), we know that the fitness function is the sum of 2 terms:

the first term is the current cost at $k-th$ stage; and the second term is the best cost from stage $k+1$ to $N-1$ which only satisfies the current state vector and the current control vector. The value of the first term is easy to compute, and the value of the second term can be searched from the pre-stored cost table by knowing the corresponding state vector $X(k+1)$. This state vector can be calculated by the state transition function (4.5) and (4.6). Note that, even though the second term of the fitness function is the best cost of current state and control, the value of the fitness function may not be the best, because the randomly selected control vector may not be the optimal one. Only after the GA finds the best solution $u^*(k, X(k))$, the value of the fitness function of $u^*(k, X(k))$ becomes the optimal cost $J(u^*, k, X(k))$ which only corresponds to the current state vector.

Before concluding this section, we present an outline of the Steady-State Genetic Algorithm [10][12] we are using in our computation. We use binary encoding in our implementation:

1 **[Start]** Generate random population of M chromosomes (n vectors of admissible u).

2 **[Clone]** Copy the previous population to be the temporary population of the next generation.

3 **[Replace]** Replace some of the individuals by repeating following steps until the user specified number of replacement or percentage of replacement is reached.

- **3.1** **[Select]** select two parent chromosomes from the population according to their fitness (the better fitness, the bigger chance to be selected).
- **3.2** **[Crossover]** With a crossover probability and single crossover point in our case, cross over the parents to form a new offspring (children).
- **3.3** **[Mutate]** With a mutation probability, mutate new offspring at each locus (position in chromosome).

3.4 **[Return]** Place new offspring into the temporary population in the new generation.

3.5 **[Destroy]** Destroy the worst chromosomes to keep the specified population size.

4 **[Generate new population]** Use new generated population for a further run of algorithm.

5 **[Test]** If the end condition is reached, **stop**, and return the best solution in current population, else go to step 2.

The running of the above process requires some pre-defined parameters, such as the population size, the number of generations, the number of replacement, crossover probability, and the mutation probability among others. The proper selection of those parameters could be very helpful to find the the optimal cost and optimal control in the shortest time. An improper parameter selection may result in a longer running time, a slower convergence, and even the loss of the optimal value. Among those parameters, the definitions of the number of generations and the population size are most important of all. We will discuss how these two parameters affect the running time and cost in Chapter 5.

Chapter 5

Implementation and Numerical Results

The implementation of the above algorithm is done in the C++ programming language. It uses the Galib genetic algorithm package [10], written by Matthew Wall at the Massachusetts Institute of Technology. The source code was downloaded from [10]. Two main programs were developed: Program 1 is used to compute the optimal control and the optimal cost; Program 2 is used to compute the cost by applying the constant rate open loop control. The source code is attached in Appendix.

In this chapter we apply the algorithm introduced in Chapter 4 and the implementation scheme outlined above to some real cases. Because the dynamic programming is computationally very intensive, we consider a scenario comprised of only two traffic sources policed by two token buckets. The computations are carried out, based on MPEG traffic transmission and self-similar traffic transmission.

5.1 Network Efficiency Metrics

Before considering numerical experiments and their analysis, we must define some very important network efficiency metrics. These measures are useful for the design of network parameters. We define the network utilization η, as follows:

$$\eta = \frac{\sum_{i=1}^{n}\sum_{k=0}^{K} v_i(t_k) - \left[\sum_{k=0}^{K} L(t_k) + \sum_{i=1}^{n}\sum_{k=0}^{K} r_i(t_k)\right]}{C(t_K - t_0)}. \tag{5.1}$$

The network average throughput is given by

$$\Upsilon = C\eta = \frac{\sum_{i=1}^{n}\sum_{k=0}^{K} v_i(t_k) - \left[\sum_{k=0}^{K} L(t_k) + \sum_{i=1}^{n}\sum_{k=0}^{K} r_i(t_k)\right]}{(t_K - t_0)} \tag{5.2}$$

$$= (1/(t_K - t_0)) \sum_{k=0}^{K} \Upsilon(t_k) \tag{5.3}$$

$$= (1/(t_K - t_0)) \sum_{k=0}^{K} \left\{ \left[\sum_{i=1}^{n} g_i(t_k)\right] \wedge [Q - ([q(t_k) - C\tau] \vee 0)] \right\} \tag{5.4}$$

where $\Upsilon(t_k)$ denotes the throughput during time interval$[t_k, t_{k+1})$.

The definitions above will be used in the analysis later.

5.2 Specification of the Traffic Traces

In our experiments, we first apply a group of MPEG traffic traces to test the efficiency of our algorithm. Then we use a group of self-similar traffic traces to verify our control mechanism and illustrate how the mechanism satisfies the need of various traffic types.

5.2.1 MPEG Traffic Trace

Multimedia services tend to be highly demanding in terms of network resources. In the near future, a major part of the traffic will be produced by multimedia services, integrating media, such as voice and video. Traffic flows generated by video applications have the properties of high peak rate and strong variability, presenting critical problems for a network management to maintain quality service. In this section, we observe two MPEG-1 encoded traffic traces, and apply those traces to test our algorithm in the next section.

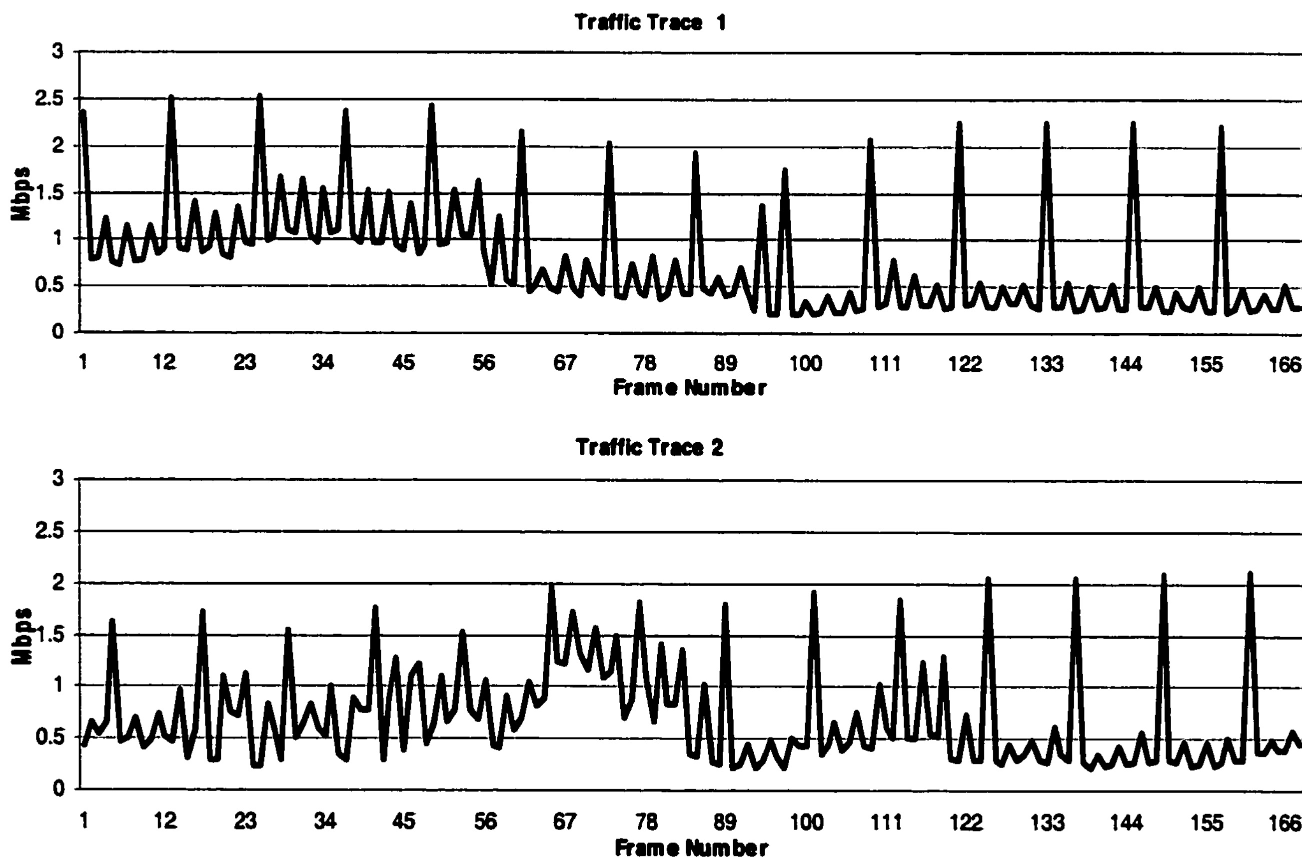


Figure 5.1: MPEG-1 traffic traces

The traces used in our experiments correspond to the traffic statistics of two MPEG-1 sequences encoded using the Berkeley MPEG-encoder (version 1.3) [13] [14]. Each group of pictures (GoP) contains 12 frames with a frame rate of 24 frames/sec. Trace 1 contains the frames of 7 seconds of "Mr. Bean", and Trace 2 contains the frames of 7 seconds of "The Simpsons". The GoP pattern can be characterized as *IBBPBBPBBPBB*, where I-frames are encoded with a moderate compression ratio, P-frames have a higher compression ratio than I-frames, and B-frames have the highest compression ratio[13][15]. The traffic traces for both video streams are shown in Figure 5.1.

From the figure the GoP pattern is clearly identified. An I-frame appears at a regular interval of one every 12 frames. The video traffic statistics are given in Table 5.1.

Unit: Mbps

Traffic Trace	**Peak Rate (Pi)**	**Average Rate(Mi)**	**Standard Deviation (Si)**
Trace 1	2.54 Mbps	0.759 Mbps	0.5784 Mbps
Trace 2	2.104 Mbps	0.696 Mbps	0.477 Mbps

Table 5.1: Summary of MPEG-1 traffic traces specification

5.2.2 Self-similar Traffic Trace

The fast development of broadband communications is characterized by a non-homogeneous traffic mix with highly bursty nature, such as ftp, video, audio, web. Traditional traffic models, such as Poisson, have been put into question, and in fact, such mixed traffic has been confirmed to be self-similar traffic. It has some very important properties [22]: 1) The distributions of the actual traffic processes have heavy tailed nature; 2) Correlations

exhibit a long range dependence.

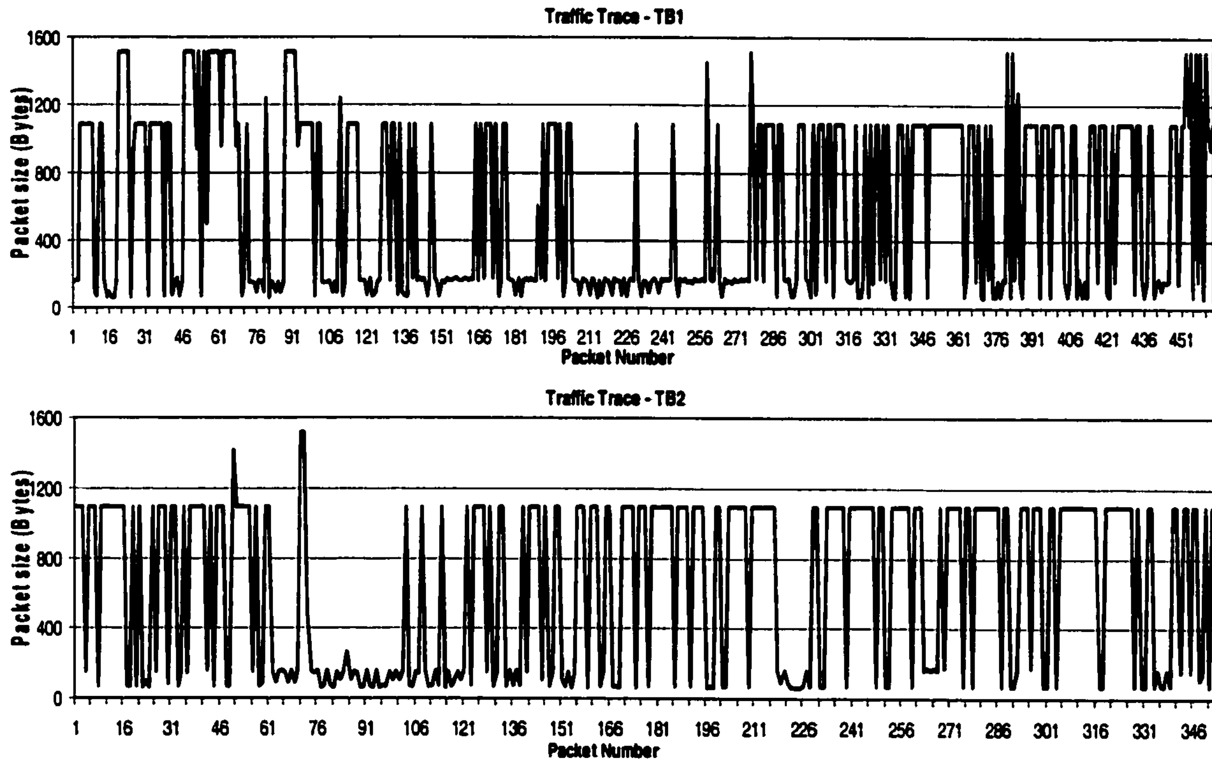


Figure 5.2: Self-similar traffic traces

The self-similar traces in the experiments were downloaded from "Internet Traffic Achieve" provided by Bellcore [24]. The traffic traces, which are studied widely, have been proved to be typical self-similar traffic in [22]. The packet size of the traffic trace is varying from 64 bytes to 1518 bytes. The traces are plotted in Figure 5.2, which shows the number of packets in X-Axis and packet sizes in Y-Axis. Table 5.2 provides the traffic statistics of the two traces:

Traffic Trace	Mean Rate (Mi)	Average Packet Size
Trace 1	2.24 Mbps	599 bytes
Trace 2	2.29 Mbps	648 bytes

Table 5.2: Statistics of self-similar traffic traces

5.3 Experimental Results and Analysis

Throughout the experiment, relative weights are assigned to the different losses as follows:

- $\alpha(t_i) = 10$, weight given to the multiplexor losses;
- $\beta(t_i) = 5$, weight assigned to the losses at the TB ;
- $\gamma(t_i) = 1$, weight given to the waiting losses.

Under the scenario considered herein, by assigning more weight to the losses at the multiplexor, we are considering preferable to reject the traffic at the TB (the entry point to the network) than to have to drop the packets in excess at the multiplexor. This is in line with the IETF philosophy that once the (conforming) traffic has been admitted into the network, enough resources have to be guaranteed for its transmission. On the other hand, the losses at TB can be reduced by implementing a waiting queue at TB which could be done in a future work. We also consider the trade-off between the losses at the TB-s and the waiting losses. If the multiplexor has enough space to accept more traffic, the traffic may be accepted even though it may experience at times longer delays. The delay time can be controlled by adjusting the buffer capacity or the TB capacity. Therefore, the weight assigned to the losses at TB is larger than the weight given to the delay caused by the waiting losses. It is understood that. depending on the application and policy implemented in a particular setup, the relative weights will have to be set up

accordingly. For some data, which have much more impact on the delay than the losses at the receiving end, we may adjust the weight for buffer delay to a larger value.

5.3.1 Experiment Using MPEG Traffic

For the transmission of the video data through the network, we assume the use of UDP/IP protocol stack. Each frame is divided into one or more packets with varying size between 64 bytes to 1500 bytes. According to our traffic model,one time interval only can be fit at most one packet. Without loss of generality, we further consider that one token takes on 100 bytes of traffic. Each traffic trace we described in section 5.2.1 is applied to a traffic source which is governed by a TB. There are total 2 TB-s are considered in this experiment.

Dependence of cost on control strategies.

We have conducted a set of experiments to observe the dependence of the cost on different control strategies. The total cost computed by equation (4.1), total losses at TB-s, losses at the multiplexer, and the waiting losses, corresponding to the optimal control, will be compared with those corresponding to some other open loop controls with constant token generation rates. More importantly, traces of the conforming traffic and token generation rates will allow us to understand the operation of the token bucket mechanism. By studying these traces, we should be able to gain insight in the mode of operation of all the control strategies under consideration. The system configurations are specified as follows:

- $T_i = 2000 Bytes, i = 1, 2$: the token bucket has been set large enough to be able to accommodate the maximum packet length,
- $C = 1.456$ and $2.184 Mbps$: these two values have been chosen to evaluate a system configuration overloaded or heavily loaded,

- $c_i = 2.56Mbps$: The rate of the link joining the TB to the multiplexor is assumed large enough to deliver the conforming traffic at the rate it is being generated,
- $Q = 4000bytes$: the multiplexor has been dimensioned to accommodate up to two 1500 bytes packets,
- $\tau = 4.63m\sec$: this interval corresponds to the arrival of a packet, since frames are generated at a rate of 24 frames per second, we assume that a video frame is segmented into up to nine packets,
- $\rho_i(0) = T_i$: the token buckets are assumed to be initially full.
- $Q = 0$: the multiplexor buffer is assumed to be empty initially.

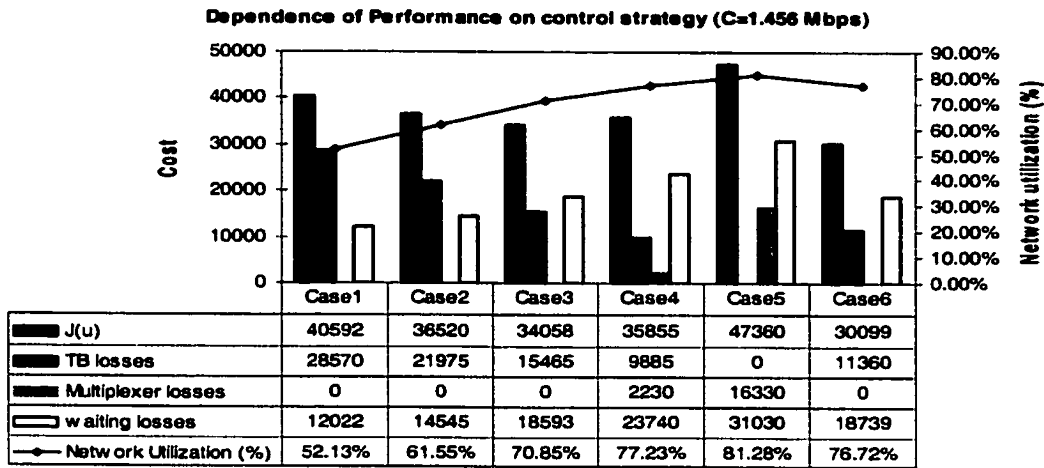
The following cases are studied:
Open loop controls without optimization:

- Case 1: $u_i(t_k) = 0.7 \times M_i$;
- Case 2: $u_i(t_k) = M_i$;
- Case 3: $u_i(t_k) = 1.3 \times M_i$;
- Case 4: $u_i(t_k) = 1.6 \times M_i$;
- Case 5: $u_i(t_k) = P_i$; (Peak Rate of the *ith* Traffic).

Optimal Control:

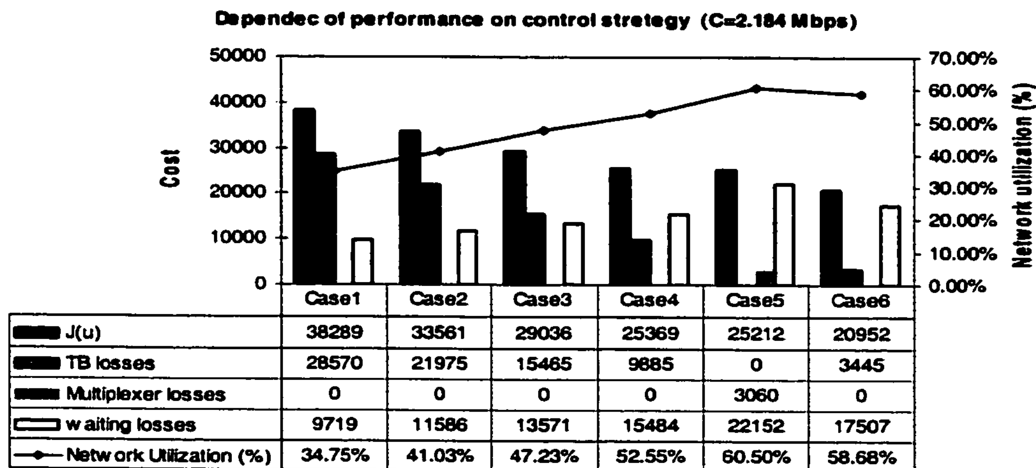
- Case 6: with variable bit rate control

Numerical results are shown in Figure 5.3 and Figure 5.4.
Figures 5.3 shows us that when the output link service rate C = 1.456 Mbps, Strategy 5, which corresponds to the open loop control with a fixed token generation rate equal to the traffic's peak rate, provides the worst performance of all. Even though it provides



	Case1	Case2	Case3	Case4	Case5	Case6
J(u)	40592	36520	34058	35855	47360	30099
TB losses	28570	21975	15465	9885	0	11360
Multiplexer losses	0	0	0	2230	16330	0
waiting losses	12022	14545	18593	23740	31030	18739
Network Utilization (%)	52.13%	61.55%	70.85%	77.23%	81.28%	76.72%

Figure 5.3: Dependence of cost on control strategy(C=1.456 Mbps)



	Case1	Case2	Case3	Case4	Case5	Case6
J(u)	38289	33561	29036	25369	25212	20952
TB losses	28570	21975	15465	9885	0	3445
Multiplexer losses	0	0	0	0	3060	0
waiting losses	9719	11586	13571	15484	22152	17507
Network Utilization (%)	34.75%	41.03%	47.23%	52.55%	60.50%	58.68%

Figure 5.4: Dependence of cost on control strategy (C=2.184 Mbps)

the lowest TB losses and the highest network utilization, which can be clearly seen from the data table in Figure 5.3, the losses at the multiplexor is much larger than others, and the waiting delay is remarkable long as well. Therefore, this strategy is so aggressive. It is understood that the traffic losses at the multiplexor are harder to control than the losses at the source such as at the TB. In applications, such as video communications, the losses at the multiplexor may result in low quality of the video at the receiving end or more delay due to lost packet re-transmission. Long waiting time of the packet at the multiplexor may also cause unacceptable delay of video frames. In some applications, such as packetized voice, long delays may considerable affect the quality of service of the end application. It is also clear that Strategies 1 and 2 are too conservative resulting in a high number of packets being dropped at the TB. That results in a high waste of network resource. The high packet losses at TB also affect the quality of the video at the receiving end. Even strategy 3, which is the best of all constant rate open loop strategies, can not beat the network performance of case 6 that is under the control of optimal token generation rate. By optimizing the operation of the overall system, Strategy 6 balances the packet losses while ensuring a proper level of utilization.

We increase the output link service rate to C = 2.184 Mbps, other parameters remain the same. The system performance is summarized in Figure 5.4, which shows that despite the output link capacity increase, Strategies 1, 2, 3, 4, 5 exhibit the same performance at TB-s. TB losses cannot be reduced by simply increasing the output link capacity, C. That implies the constant rate control can not dynamically allocate the available network resources. On the contrary, the optimal control strategy 6 treats the system as a whole. It balances the trade-off of the TB losses, multiplexor losses and the waiting delay according to the available resources and makes the total cost the best ever.

Dependence of the conforming traffic of control strategies

In this section, we observe the insight on how the control strategy affects the conforming traffic at TB and analyze the performance at frame level.

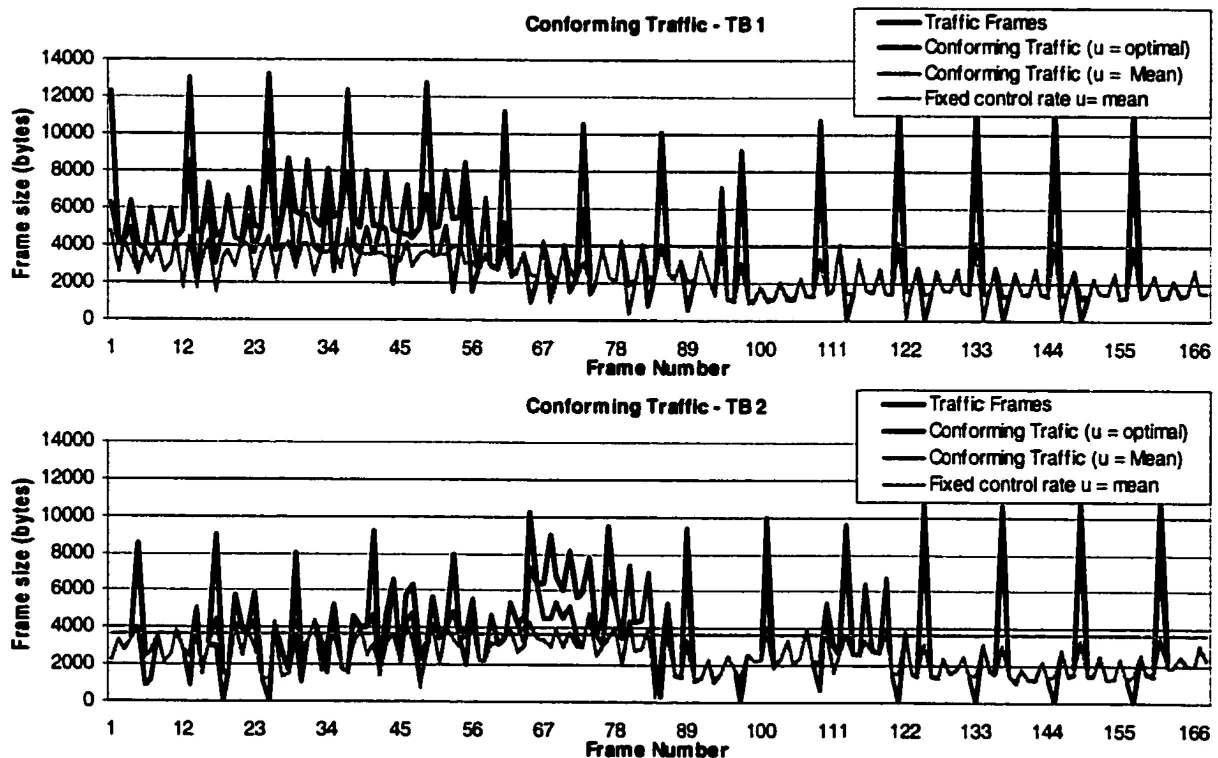


Figure 5.5: Incoming traffic vs. conforming traffic (fixed u=mean traffic)

Figure 5.5 provides the traces of the incoming traffic, together with the conforming traffic resulting when applying Strategies 2 (token generation rate = mean rate of the incoming traffic) and Strategy 6 (the optimal case), when the output link service rate C = 1.456 Mbps. For reference purposes, the figures also show the mean rate of the input traffic,

which is also the token generation rate.

As expected, it is clear that by fixing the token generation rate equal to the mean rate of the sources, there is considerable waste of resources. The losses at TB are very heavy, especially during the I-frame transmissions. Those heavy TB losses must result in a low quality of the video at the receiving end.

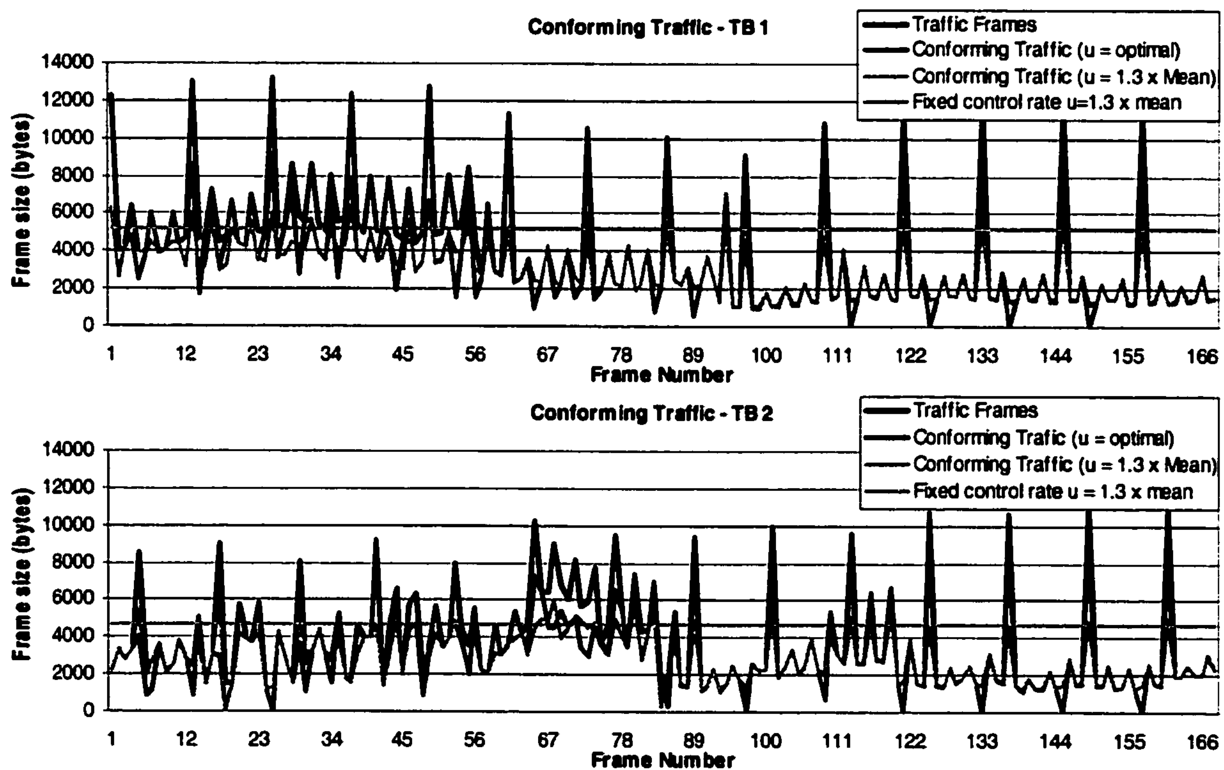


Figure 5.6: Incoming traffic vs. conforming traffic (fixed u=1.3x mean traffic)

In the case of the optimal control strategy 6, we can see that the TB performance is much better, compared with applying the fixed control rate u which equals the mean

traffic rate. The resources are dynamically distributed under optimal control strategy. We notice however that at some points of time, the rate of the conforming traffic exhibits very low values or completely shut down. This happens at instance when one source transmits at a much higher rate than the other one. For instance, at frame 108, 110, or 122, we can clearly observe that there is a considerable discrepancy between the actual incoming traffic rate and the rate of the conforming traffic. This results from the fact that the optimization has been done for the overall system as a whole. It considers the overall performance, even it is not fair for one or the other, the overall system cost must be optimal. Once again, a satisfactory solution to this unfairness condition will depend on the application or policy in place. If there are some special requirements regarding to the fairness, one can add more constrains during the computation process to fulfill the requirements of the specific application or policy. From Figure 5.9, we have seen that the token generation rate of Source 2 exhibits this same discrepancy with respect to the input traffic rate. For instance, for the results form the particular setup considered herein, we see that the sources take turns in gaining a bigger share of the resources, i.e., channel bandwidth.

Figure 5.6 shows similar results as Figure 5.5, except that in this case the token generation rate has been fixed to 1.3 times the mean of the input traffic rate. But the losses at B-frame and P-frame transmissions are considerably reduced. At most cases, the token generation rate can satisfy the B-frame and P-frame transmission rate, resulting in very low or no packet losses. From the figure, it is clear that by setting the token generation rate at a higher rate may help during period where the rate of a source remains high during a long period of time. For instance, during the frames 64 to 72, Source 2 exhibits a high degree of activity. In the case when the token generation rate is set to 1.3 times the mean traffic rate, the conforming traffic exhibits at some points even higher values than the ones obtained for the optimal case. However, during I-frame transmissions, this token

generation rate is still not sufficient to deal with those relatively highest rate traffic frames.

Figures 5.7 and 5.8 show the traces of the conforming traffic and sources for the case when the capacity of the link C is increased to 2.184 Mbps.

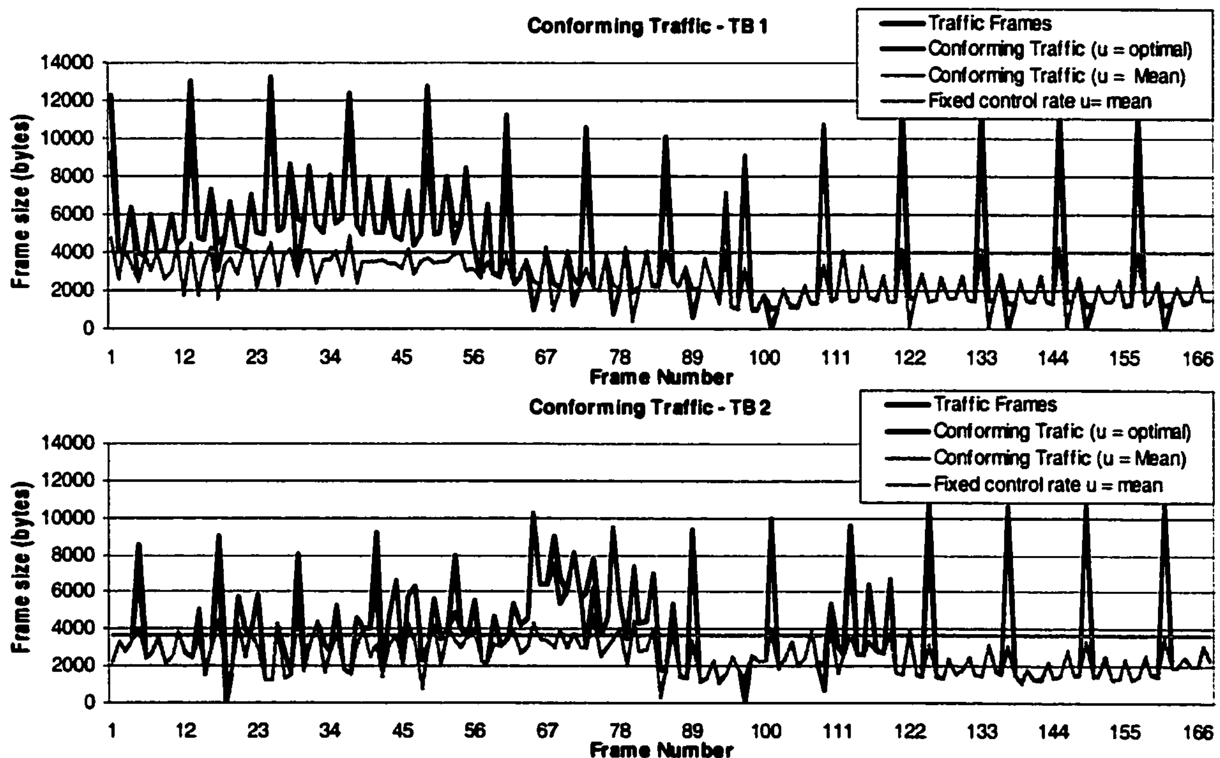


Figure 5.7: Incoming traffic vs. conforming traffic (fixed u=mean traffic).

Because of the increasing of the output link capacity, we see the tremendous improvement of the TB performance under optimal control strategy. Even though there are still some losses, the losses are much more acceptable than the TB losses of the previous case with C = 1.456 Mbps. In Figure 5.7, when the output link rate increases to 2.184 Mbps, there

are still some instances when the conforming traffic is completely shut down or at a very low rate. However, the number of this kind of instance is much less. Therefore, in overall Strategy 6 is able to take advantage of the extra capacity reducing considerable the overall losses while maintaining a good level of utilization.

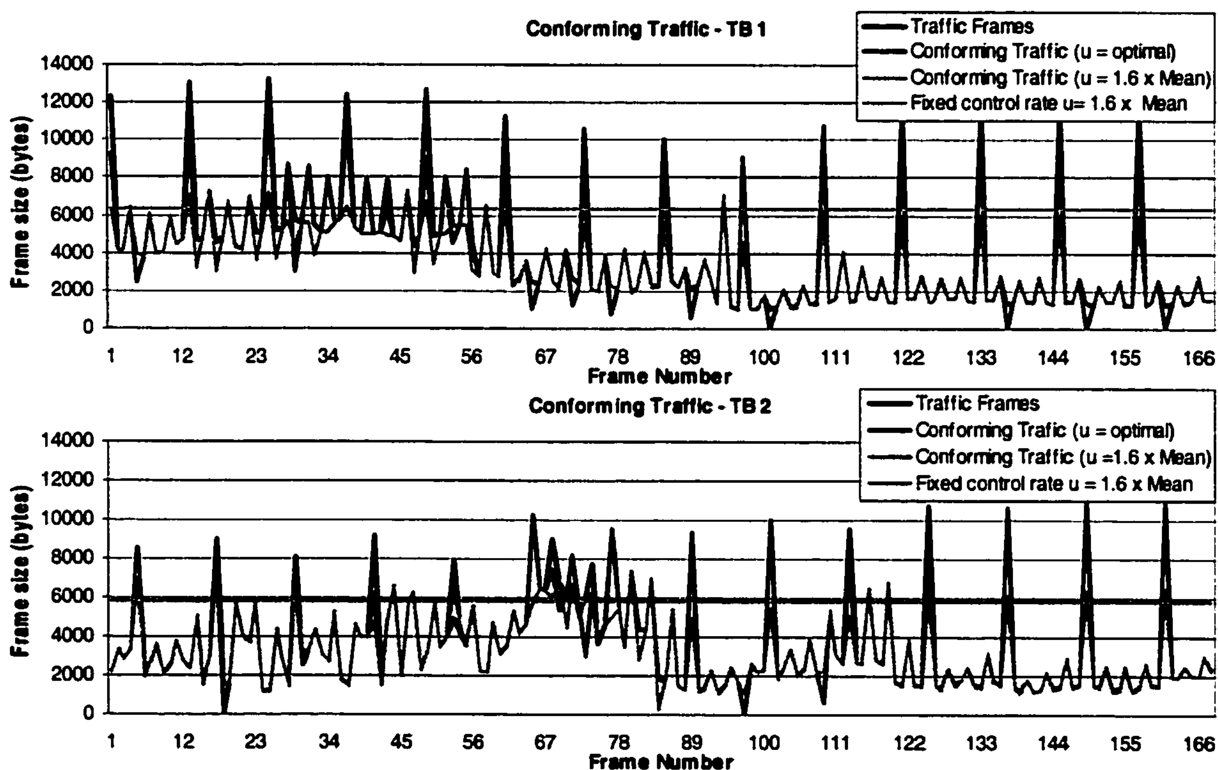


Figure 5.8: Incoming traffic vs. conforming traffic (fixed u = 1.6 x mean traffic).

Furthermore, during period of high activity, it is able to make use of the extra capacity. For instance, during the frames 64 to 72, the conforming traffic corresponding to Source 2 follows very closely the pattern of the incoming traffic. In the case of fixed token generation, even a token generation rate of up to 1.6 times the mean incoming traffic rate is

not able to cope with the demand during this period of time as good as the optimal case which is plotted in Figure 5.8.

Dependence of optimal control rate on C

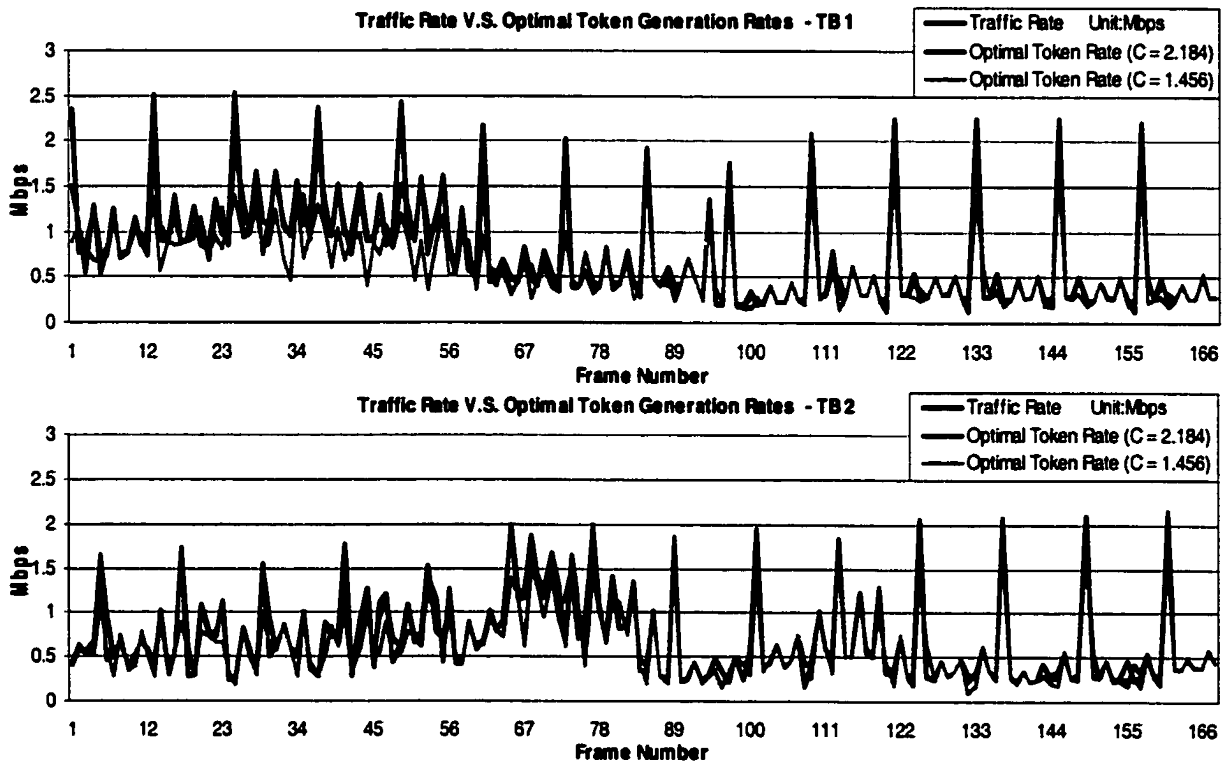


Figure 5.9: Optimal token generation rates

Figure 5.9 depicts the optimal token generation rates and traffic rates for both TB-s. We observe that the token generation rates follow the same pattern as the traffic traces. In situations when both sources exhibit higher rates at the same time, the token generation rates are limited by the output channel rate. For instance, from the figure we see that

at the large spike around frame 50, the rates of both sources add up to about 4 Mbps. However, the token generation of both sources are clearly limited by the channel output rate. For both channel rates, the token generation rate of Source 1 is lower than the corresponding traffic rate. This will obviously translate into packets that will have to be discarded at the token bucket.

Figure 5.9 also shows that in the case when one of the sources transmits at much higher rates than the other one, the token generation rate of the high activity source will follow accordingly. In the mean time, the token generation rate of the low activity source may not be able to generate the required tokens to transmit all its packets. For instance, at frame 109, the traffic rate of Source 1 exceeds the 2 Mbps while Source 2 is transmitting at a rate of about 0.4 Mbps. As seen from the figures, the token generation rate of Source 1 is slightly higher than 2 Mbps, while the token generation rate of Source 2 shows a significant discrepancy with respect to the source rate.

We see that there are situations when the conforming traffic gets shut down (see Figure 5.5). For instance, at frame 122, the conforming traffic of Source 2 is completely shut down, and this despite the fact that there are tokens being generated during this frame at its corresponding token bucket (refer to TB2 in Figure 5.9). However, since the packet length being used can be as large as 1500 bytes, there may not be enough tokens present at the token bucket to take on the whole packet. According to our model, in this case, the whole packet is discarded at the token bucket. In the figure, there are several instances where this situation arises for both sources.

As expected, when the output link rate C increases, the control rate increases accordingly. Even though, the control rate when C= 2.184 Mbps is not always higher than the control rate when C= 1.456 Mbps during the whole transmission process. Some exceptions can

be observed from Figure 5.9, for instance, at frame 100 and 159. We can see the overall token generation rate by observing the average rate of the whole process, which is shown in Table 5.3. We get a conclusion from the table that $Mu_i^*(C = 1.456Mpbs) < Mu_i^*(C = 2.184Mbps), i = 1, 2$, where Mu_i^* denotes the average rate of optimal control u^* at $i-th$ TB.

C→ TB↓	C=1.456 Mpbs	C=2.184 Mbps
TB1	0.614 Mbps	0.709 Mbps
TB2	0.613 Mpbs	0.697 Mbps

Table 5.3: Average control rate

Dependance of optimal cost J on Q and C

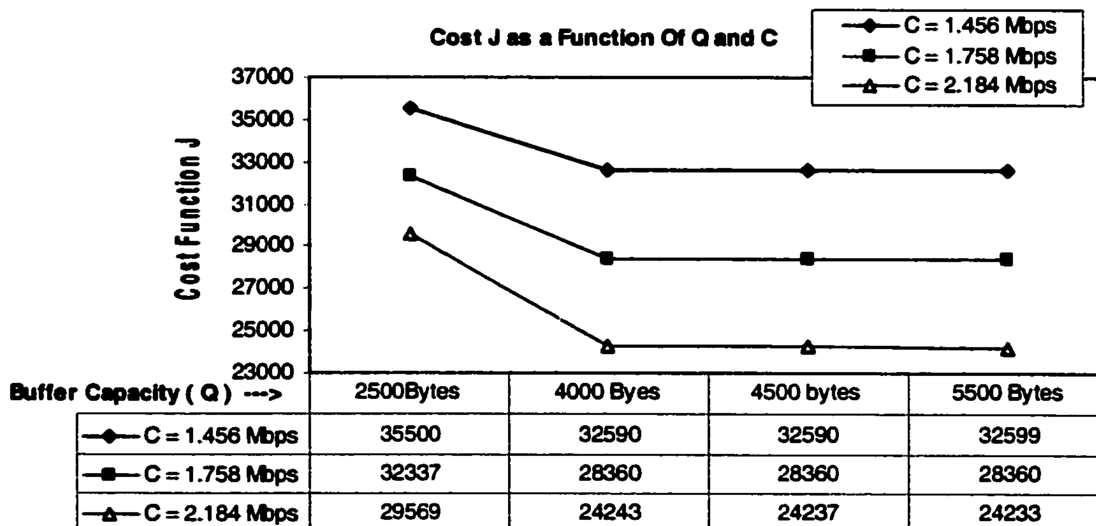
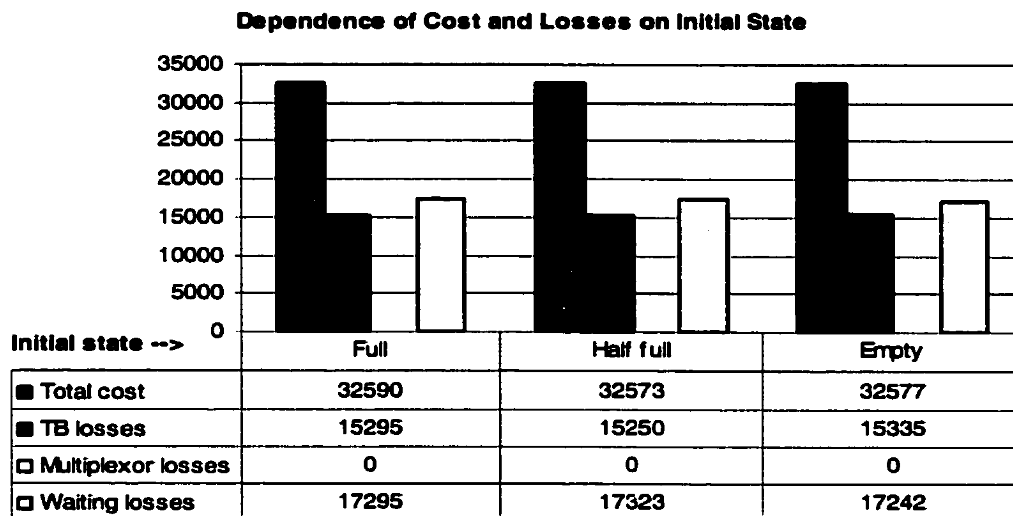


Figure 5.10: Dependence of cost on Q and C.

We now study the effect of some of the network parameters, such as Q and C, over the

cost functions for the optimal control. In this set of experiments, we change the values of the buffer size (queue capacity) and the values of the output link service rate C, the values of the parameters and the corresponding costs are summarized in Figure 5.10.

From Figure 5.10, as expected, increasing the output capacity C decreases the cost. However, increasing the buffer size Q beyond a certain size does not further improve the system performance. Except for the case when the buffer size is set to be 2500 bytes, all other buffer sizes show similar cost values for a given output capacity rate. This is not surprising if we realize that the token bucket size has been set to 2000 bytes, allowing the sources to transmit up to 2000 bytes back to back. Therefore a minimum buffer size of 4000 has to be provided to accommodate the conforming traffic. Furthermore, it is important to point out that the waiting losses do not play a major role as the multiplexor buffer is increased. This is due to the fact the amount of conforming traffic present in the multiplexor at a given time will be limited by size of the token buckets.



Initial state -->	Full	Half full	Empty
■ Total cost	32590	32573	32577
■ TB losses	15295	15250	15335
□ Multiplexor losses	0	0	0
□ Waiting losses	17295	17323	17242

Figure 5.11: Dependence of cost on initial state.

Dependence of cost on initial state

In this section, we study the effects of the initial state to the system performance under the control of the optimal strategy. Three values of initial state are considered herein:

- $\rho_i(0) = T_i$, Both TB-s are full of tokens;
- $\rho_i(0) = 0.5 \times T_i$, Both TB-s have half full of tokens;
- $\rho_i(0) = 0$, Both TB-s are empty, no tokens in the TB-s initially.

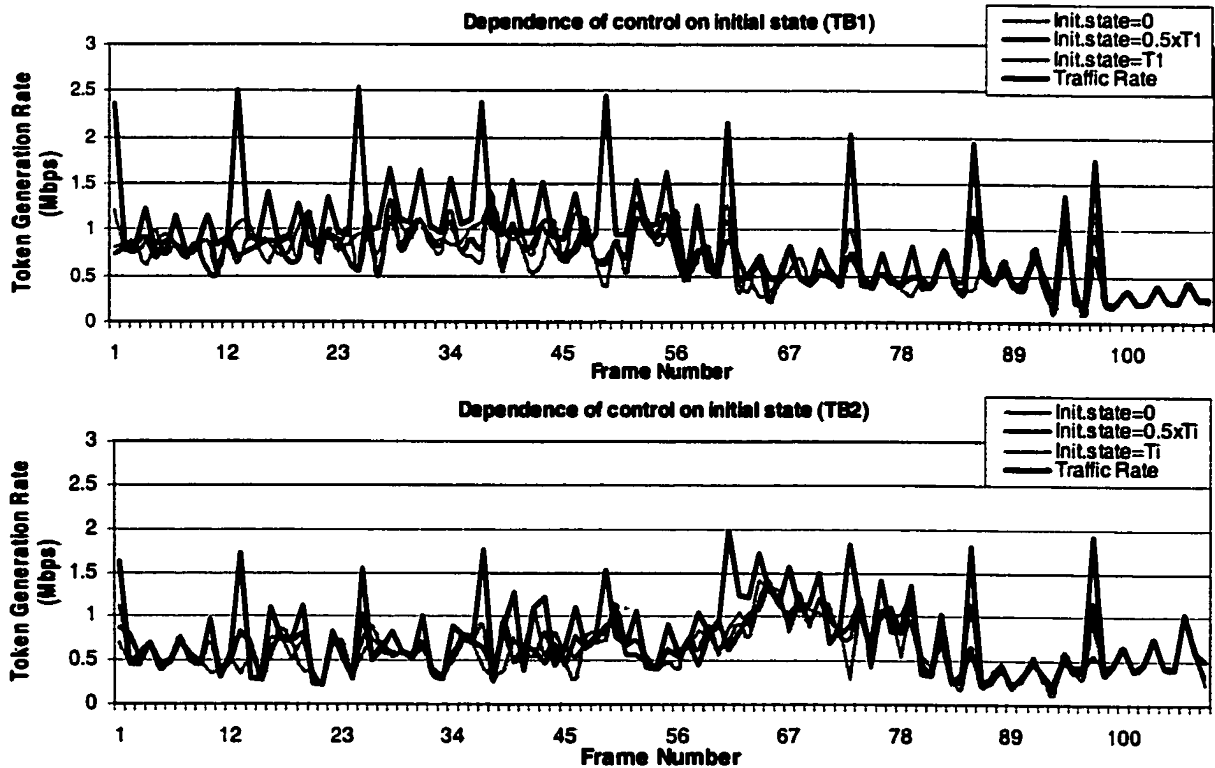


Figure 5.12: Dependence of cost on initial state.

According to the numerical results that we plot in the graph shown in Figure 5.11. we know that under optimal control strategy the initial state does not affect the cost and loss rate. The costs are very similar no matter the initial state. The same as the cost, neither the losses of TB, nor the waiting cost changes significantly. The losses at the multiplexer remains 0 while the initial state changes.

Nevertheless, the initial state affects the optimal control sequence especially at the beginning of the transmission. The optimal control values are totally different when we change the value of the initial states as shown in Figure 5.12. Therefore, it proves that different initial state results in different control sequence.

5.3.2 Experiment Using Self-similar Traffic

In this section, our experiments have been conducted using two pre-downloaded self-similar traffic traces which are described in Section 5.2. Since the behavior of our optimal control algorithm has been analyzed in detail in the previous subsection 5.3.1, we only illustrate how efficient the algorithm is if some other traffic types are transmitted rather than MPEG-1 traffic by comparing the cost and losses of the system controlled by our newly designed algorithm with those controlled by other strategies. This time, we specify that 1 token takes on 64 bytes traffic. The system configuration is redefined as following:

- $T_i = 2500 Bytes$;
- $C = 6.144 Mbps$;
- $c_i = 2.56 Mbps$;
- $Q = 4500 bytes$;

- $\tau = 0,0005\,\text{sec}$;
- $\rho_i(0) = T_i$;
- $q = 0$.

where $i = 1, 2$. The following cases are studied:

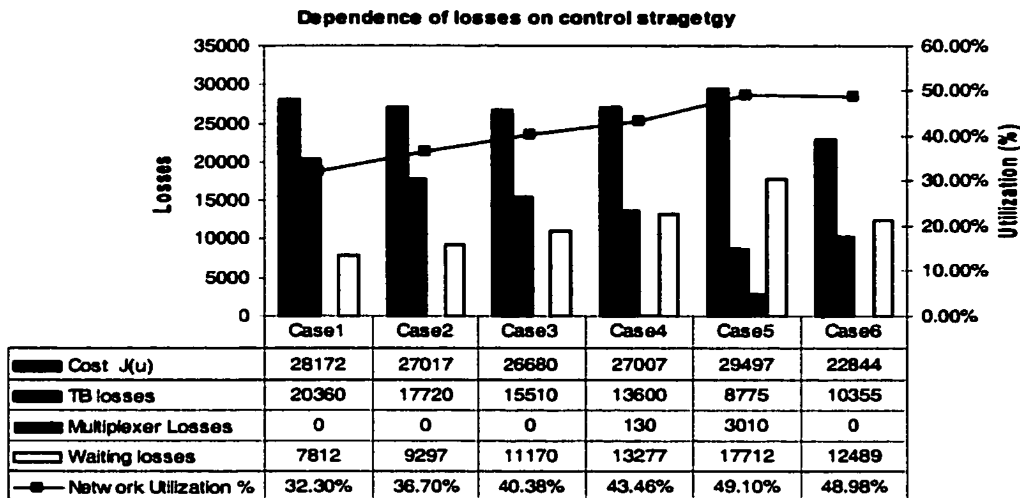
Constant rate open loop controls without optimization:

- Case 1: $u_i(t_k) = 0.7 \times M_i$;
- Case 2: $u_i(t_k) = M_i$;
- Case 3: $u_i(t_k) = 1.3 \times M_i$;
- Case 4: $u_i(t_k) = 1.6 \times M_i$;
- Case 5: $u_i(t_k) = 2.5 \times M_i$;

where $i = 1, 2$, and the last case

- Case 6: optimal control with variable bit rate.

Figure 5.13 depicts the results for all six strategies under consideration. In fact, self-similarity captures most of the characteristics of nowadays network traffics. Figure 5.13 illustrates some similar results as Figure 5.3 and Figure 5.4 in Section 5.3.1. Strategy 6, which applies the optimal control, gives the best performance which is 14% better than the cost J and 21.3% higher than the network utilization μ controlled by strategy 3 that offers the best cost among those non-optimal strategies. And it provides almost the same percentage of network utilization as the highest one which is offered by Case 5 as well. However, Case 4 and Case 5 are too aggressive, such that the some of the conforming packets are discarded at the multiplexor and some of the conforming packets have too much delay in the multiplexor buffer. It is also obvious that Case 1 and Case 2 make the network under utilized. Case 6 totally eliminates the packet losses at the multiplexor and shortens the packet delay.



	Case1	Case2	Case3	Case4	Case5	Case6
Cost J(u)	28172	27017	26680	27007	29497	22844
TB losses	20360	17720	15510	13600	8775	10355
Multiplexer Losses	0	0	0	130	3010	0
Waiting losses	7812	9297	11170	13277	17712	12489
Network Utilization %	32.30%	36.70%	40.38%	43.46%	49.10%	48.98%

Figure 5.13: Dependence of losses on control strategy.

5.3.3 Optimization Using Genetic Algorithm

We present in this section two examples to illustrate the benefits of using an approach based on Genetic Algorithms to shorten the running time of Dynamic Programming. It has been known that discrete time dynamic programming have very high impact on computer system resource and the computation takes two much time. The involvement of GA plays an important role to make dynamic programming more practical.

Before experiments, we define some terms which would be used in the numerical result.

- **p**: population size;
- **g**: number of generation;
- **T (p,g)** : Running time of the computation;
- **J (p,g)** : Cost as a function of p and g.

In our experiments,we have used a Sun Solaris Server with 4GB memory and 466-MHz UltraSPARC-II processor. We have kept track of the running time of each experiment. The following two cases were considered:

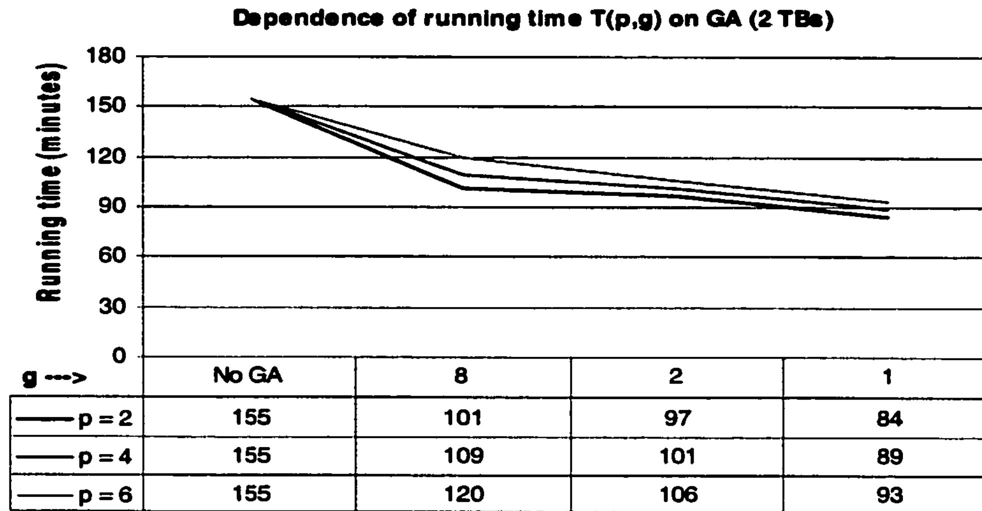
Case 1:

- Total Number of TB-s: 2
- Token bucket size: 2500 Bytes
- 1 token corresponds to 100 Bytes
- Total number of stage (number of time interval): 108
- Number of admissible state vectors per stage: 33800
- Maximum number of admissible control vector corresponding to each state vector: 676
- Average number of admissible control vector corresponding to each state vector: 338

Case 2:

- Total Number of TB-s: 3
- Token bucket size: 2500 Bytes
- 1 token corresponds to 100 Bytes
- Total number of stage (number of time interval): 9
- Number of admissible state vectors per stage: 790920
- Maximum number of admissible control vector corresponding to each state vector: 17576

- Average number of admissible control vector corresponding to each state vector: 8788

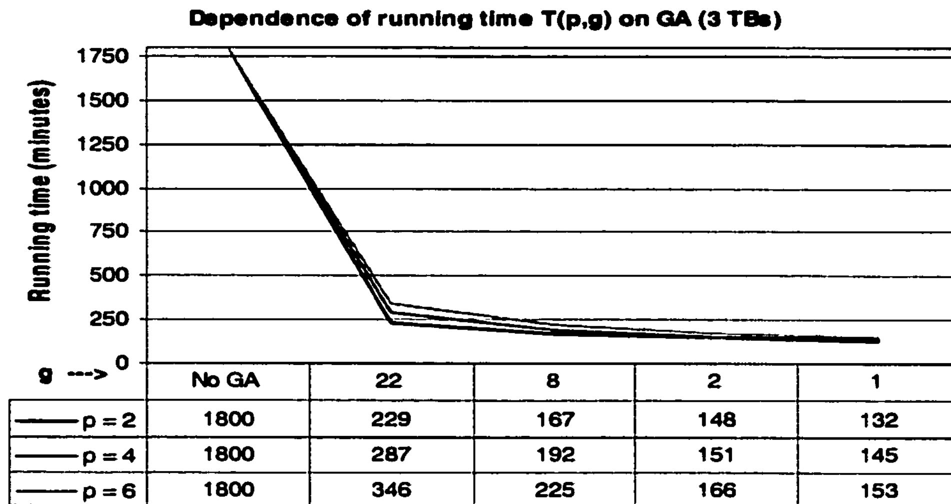


g --->	No GA	8	2	1
p = 2	155	101	97	84
p = 4	155	109	101	89
p = 6	155	120	106	93

Figure 5.14: Dependence of running time T(p,g) on p and g (Case 1).

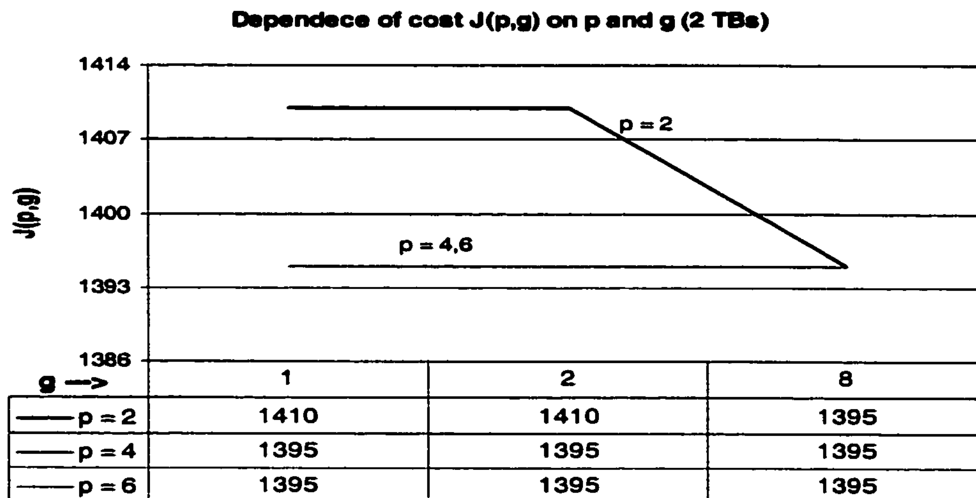
Figures 5.14 to 5.17 show the time required by the optimization procedure as well as the cost function as a function of the size of the population, p, and number of generation, g. As seen from the figures, the use of the Genetic Algorithm reduces considerably the computation time. This reduction is far more significant as the number of token buckets increases (see Figure 5.15). As shown in this figure, the computation time of the cost function without the use of the Genetic Algorithm is 1800 minutes, while by using the Genetic Algorithm this time can be reduced to 145 minutes when using a population of 4 and 1 generation to get the optimal cost (see Figure 5.17).

Figure 5.16 and Figure 5.17 also shows how the optimal value converges. It provides us the guidelines on how to choose parameters p and g, in order to reduce the computation time.



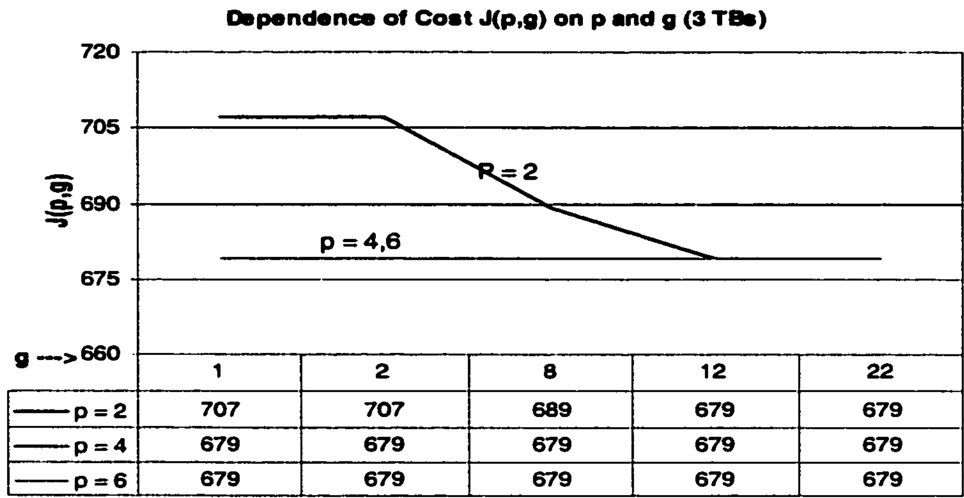
g --->	No GA	22	8	2	1
p = 2	1800	229	167	148	132
p = 4	1800	287	192	151	145
p = 6	1800	346	225	166	153

Figure 5.15: Dependence of running time T(p,g) on p and g (Case 2).



g --->	1	2	8
p = 2	1410	1410	1395
p = 4	1395	1395	1395
p = 6	1395	1395	1395

Figure 5.16: Dependence of cost J(p,g) on p and g (Case 1).



	1	2	8	12	22
p = 2	707	707	689	679	679
p = 4	679	679	679	679	679
p = 6	679	679	679	679	679

Figure 5.17: Dependence of cost J(p,g) on p and g (Case 2).

Chapter 6

Conclusion

The optimization of the token bucket control mechanism presented in this thesis have allowed us to gain insight into the operation of the token bucket algorithm. The numerical results and the system analysis have shown the effectiveness of dynamic programming, genetic algorithm and our system model in the study of the operation and performance of the token bucket. They also set some preliminary guidelines for the design of the system parameters, such as the buffer capacity, Q.

Dynamic programming has been proved to be a powerful tool in finding the optimal control and minimum cost. However, we could not ignore the practical issues of dynamic programming. The processing time of dynamic programming algorithm is considerably long and highly memory demanding.

By recording the running time and the memory allocation of each experiment, we have obtained the following statistics, for a system consisting of 2 token buckets:

- Total number of stages (number of time intervals): 1512

- Average number of admissible state vector during each stage:17741
- Average number of admissible control vector corresponding to each state vector: 310
- Average running time: 10.7 hours
- Average memory allocation: 211 Mbytes

Since a real system will be characterized by a large number of sources. We must find ways to improve the computational requirements. In our implementation, some efforts have been taken to reduce the running time by using GA.

To further reduce the running time, we propose the use of parallel computing. We have not explored any applicable methods to reduce the memory allocation problem. However, the use of Check point theory, which has been recently presented in some papers [15][16] might be able to solve the problem.

To make the system and the algorithm more effective and practical, future work may focus on the following points: 1. enhance the dynamic model to take into account issues, such as fairness, particular application requirements and packet dropping policies; 2. improve the computation and memory requirements of the optimization algorithm presented in this thesis; 3. design of an optimal feed back control strategy based on the lessons learnt herein.

Bibliography

[1] A.S. Tanenbaum, *Computer Networks Third Edition*, Prentice Hall PTR Upper Saddle River, New Jersey,1996

[2] N.U. Ahmed, Qun Wang and L.Orozco Barbosa, *A Systems approach to modelling the Token Bucket Algorithm in computer Networks.* Mathematical Problems in Engineering (Theory, Methods and Applications).

[3] Nanying Yin and Michael G. Muchyj, *Analysis of the Leaky Bucket Algorithm for On-Off DataSources*, Globecom 91, 1991, pp. 9.2.1-9.2.7

[4] C. Wahida, N.U. Ahmed, *Congestion control Using Dynamic Routing and Flow Control*, Stochastic Analysis and applications 1992

[5] Richard Bellman, *Dynamic Programming*, Princeton University Press, Princeton, New Jersey,1957

[6] George L. Nemhauser, *Introduction to Dynamic Programming*, John Wiley & sons, 1966

[7] R. Luus, *Iterative Dynamic Programming: From Curiosity to a Practical Optimization Procedure*, Control and Intelligent Systems, Vol. 26, No.1, 1998, pp. 1-8.

[8] David A Cloley, *An Introduction to Genetic Algorithms for Scientists and Engineers*, World Scientific, 1999

[9] Puqi Perry Tang, Tsung-Yuan Charles Tai, *Network Traffic Characterization Using Token Bucket Model*, Proceedings IEEE INFOCOM '99, the Conference on Computer Communications, New York, vol. 1, Mar. 1999. pp. 51-62.

[10] Mstrk Obitko, *Introduction to Genetic Algorithms*, on-line,`http://cs.felk.cvut.cz/~xobitko/ga/`,1998

[11] Matthew Wall, *A C++ Library of Genetic Algorithm Components*, on - line `http://lancet.mit.edu/ga/`http://lancet.mit.edu/ga/

[12] Matthew Wall, *Introduction to Genetic Algorithms*, on-line `http://lancet.mit.edu/\protect$\relax\sim$mbwall/presentations/IntroToGAs`

[13] O. Rose, *Statistical Properties of MPEG Video Traffic and their impact on Traffic modeling in ATM Systems*, on – line `http://nero.informatik.uni-wuerzburg.de/MPEG/`

[14] on – line `http://nero.informatik.uni-wuerzburg.de/MPEG/`

[15] Carlo Bruni, Carterina Scoglio, *An optimal rate control algorithm for guaranteed services in broadband network*, Computer Networks 37, 2001, pp.331-344

[16] Christopher Tarnas, Richard Hughey, *Reduced space hidden markov model training*, on- line `http://www.soe.ucsc.edu/research/compbio/sam.html`

[17] J Alicia Grice, Richard Hughey, Don Speck, *Reduced space Sequence Alignment*, on-line `http://www.soe.ucsc.edu/research/compbio/sam.html`

[18] A. Lombardo, G. Schembra and G. Morabito, *Traffic Specifications for the Transmission of Stored MPEG Video on the Internet*, IEEE Transactions on Multimedia, vol. 3, No.1, March 2001, pp.5-17.

[19] Gregorio Procissi, Anurag Garg, Mario Gerla, M.Y. Sanadidi, *Token bucket characterization of long-range dependent traffic*, Computer Communications, 2002, pp.1009-1017

[20] R. Bruno, R.G. Garroppo and S. Giordano, *Estimation of Token Bucket Parameters of VoIP Traffic*, Proc. IEEE ICC'99, 1999,pp.353-356

[21] I.E.Dasotakis, M.E.Markaki, and A.V. Vasilakos, *A Hybrid Genetic Approach for Chanel Reuse in Multiple Access Telemcommunication Networks*,IEEE Selected Areas in Communications, Vol. 18, NO.2, Feb 2000, pp.234-243

[22] W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson, *On the self-similar nature of ethernet traffic (Extended Version)*, IEEE/ACM Transactions on Networking, vol. 2, no.1,1994, pp.1-15,.

[23] R.braden, D.clark, S. shenker, *Integrated services in the internet architecture: an overview*", Internet RFC 1633, 1994

[24] on – line `http://ita.ee.lbl.gov/html/contrib/BC.html`

Appendix: Source Code

Appendix A: Optimal Control Source Code

Appendix B: Constant Open Loop Control Source Code

Appendix A: Optimal Control Source Code

```
/*************************************************************************
     This is the implementation of the optimal control mechanism - 2 TBs
     ==============================================================

     It uses the Galib genetic algorithm package [10],
     written by Matthew Wall at the Massachusetts Institute of
     Technology. The source code was downloaded from [10].

     ==============================================================
     Written by: Bo Li
     August 2002

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

 #include <stdio.h>
 #include <iostream.h>
 #include <fstream.h>
 #include <ga/ga.h>
 #include <math.h>

 const short int Num =2; // the number of token buckets
 double SERVICERATE; // the output link service rate
 unsigned short A=0; // the weight alpha
 unsigned short B=0; // the weight betta
 unsigned short C=0; // the weight gamma
 double TIMEINTERVAL;// the time between two stages
 unsigned short QUEUESIZE;  // the queue capacity of the multiplexor
 unsigned short LINKCAPACITY[Num]; //the link rates from TB to multiplexor
 unsigned short TBSIZE[Num];  // the capacity of TBs
 unsigned short STAGE; // record the total number of stages
 unsigned short **states; // all states
 unsigned short **Traffic; //all traffic from all TBs and all stages
 unsigned short confirmedTraffic[Num]; // conforming traffic of each state
 int **twoStageOptimal; // keep the optimal values of this stage and the next
 stage
 short stage; // keep the change of stage
 unsigned short acceptedMul; // the accepted bytes by the multiplexor
 unsigned short initialState[Num+1]; // store the inital states of TB and
 Queue
 int totalState=1; // keeps track of the number of possible states
 int optimalIndex=0; //keeps track of the array index of the optimal value
 unsigned short int population; // the GA population size
 unsigned short int generation; // number of generations of GA

 /* read number of stages from a file*/
 void readStage();
 /* read traffic traces from a file*/
 void readTraffic();
```

```
/* read all parameters from a file*/
void readParameters();
/* compute the variance of the traffic trace*/
double variance(unsigned short,unsigned short);
/* compute the standard Deviation of the traffic trace*/
double StandardDeviation(double);
/* compute objective function without running GA*/
float Objective1(unsigned short G[]);
/* compute objective function with GA*/
float objective(GAGenome & c);
/* compute the current conforming traffic*/
unsigned short confirmTraffic(unsigned short []);
/* search the optimal value for the table */
int searchNextV(unsigned short []);
/* compute the losses at the multiplexor*/
int LostAtMultiplexer(unsigned short);
/* compute the total losses at TBs*/
int TotalLostAtTB();
/* compute the losses at a single TB*/
int LostAtTB(unsigned short, unsigned short);
/* compute the waiting losses at the Queue*/
int WaitingLost(unsigned short,unsigned short);
/* compute the accepted traffic by multiplexer
   calling by WaitingLost() and LostAtMultiplier()*/
int AcceptedTrafficByMul(unsigned short ,unsigned short);

/* trace the optimal value from the initial state*/
void traceBack();



//int lostAtTB=0;


/* define a stucture which can maintain the element of the value table*/
struct tableElement
{
    unsigned short states[Num+1];
    unsigned short control[Num];
};

/* define a two dimentional value table */
struct tableElement **possibleStates;


/*
==============================================================================
                start of the main program
=============================================================================*/

int main(int argc, char **argv)
{
```

```
unsigned int seed = 0;

/* read a seed value from the console argument*/
for(int i=1; i<argc; i++) {
    if(strcmp(argv[i++],"seed") == 0) {
        seed = atoi(argv[i]);
    }
}
/* read the number of stages */
readStage();

/* allocate enough memory for all states*/
states=new unsigned short *[STAGE];
for(short int i1=0;i1<STAGE;i1++)
{
    states[i1]=new unsigned short[Num+1];
}

/* allocate enough memory for traffic traces*/
Traffic=new unsigned short* [STAGE];
for(short int i2=0;i2<STAGE;i2++)
{
    Traffic[i2]=new unsigned short[Num];
}

/* read traffic traces to the two dimentional array*/
readTraffic();
/* read all parameters from a file*/
readParameters();

/* allocate enough memory for the value table*/
possibleStates=new pair *[STAGE];
for(int i5=0;i5<STAGE;i5++)
{
    possibleStates[i5]=new pair[totalState];
}

/* allocate enough memory for the optimal values of current stage
   and the next stage*/
twoStageOptimal=new int*[2];
for(int i6=0;i6<2;i6++)
{
    twoStageOptimal[i6]=new int[totalState];//totalState];
}

/* create GAParameterList object and set ga paremeters */
GAParameterList params;
GASteadyStateGA::registerDefaultParameters(params);
params.set(gaNpopulationSize, population);
params.set(gaNelitism,gaTrue);
```

```
params.set(gaNpCrossover, 0.9);
params.set(gaNpMutation, 0.04);
params.set(gaNnGenerations, generation);
params.set(gaNselectScores,GAStatistics::Minimum);
params.parse(argc, argv, gaFalse);

/* IMPORTANT ==the running of the whole optimal control process tarts ==*/
for ( stage=STAGE-1; stage>=0; stage--)
{
    cout<<"running stage :"<<stage<<endl;
    int n=0;  //keeps the index of the totalstates
    int localOptimal = 9999999;

    /* find where the optimal values of the next stage stores.*/
    if(optimalIndex==0)
    {
        optimalIndex=1;
    }
    else optimalIndex=0;

    /* the comparation of the best value starts */
    for (unsigned short p1 = 0; p1<= TBSIZE[0] ; p1++)
    {
        unsigned short i = 0;

        states[stage][0] =  p1;
        for (unsigned short p2 = 0; p2<=TBSIZE[1]; p2++)
        {
            states[stage][1] = p2;

            for (unsigned short q=0; q <= QUEUESIZE; q++)
            {
                states[stage][2] = q;

                /*  find the boundary for control u  */
                /* Declare MaxU[3] to store the boundary for u */
                unsigned short MaxU[Num];

                /* define and initialize if it will run ga*/
                bool runGa=true;

                /* initially define the number of admissible control
                    values*/
                unsigned short totalU=1;

                /* an array of control values */
                unsigned short G[Num];

                float o;

                /* compute the Maximum number of token generating  */
                for(unsigned short j=0; j<Num; j++)
```

```
{
    /* the Maxcimum number of token generating can't
       exceed the available space in TB */
    MaxU[j] = TBSIZE[j]*1 - states[stage][j];
}

/* compute the total number of admissable control values */
for(unsigned short u=0;u<Num;u++)
{
    totalU=totalU*(MaxU[u]+1);
}

/* if the search space smaller than 300, we do not use GA
   to find the optimal value, since from several testings,
   we found that running speed is faster without using GA
   if the search space is smaller than 300*/
if(totalU<300)
{
    runGa=false;
}

/* run GA */
if(runGa==true)
{

    /* define the search space */
    GAAlleleSet<unsigned short> a[Num];
    GAAlleleSetArray<unsigned short> alleles;
    for(unsigned short n1=0;n1<Num;n1++)
    {
        for(unsigned short h=0; h<=MaxU[n1]; h++)
        {
            a[n1].add(h);
        }
        alleles.add(a[n1]);
    }

    /* define a genome object */
    GA1DArrayAlleleGenome<unsigned short>
                                genome(alleles, objective);

    /* define a GA object */
    GASteadyStateGA ga(genome);
    /* add all ga parameters to ga*/
    ga.parameters(params);
    /* tell ga compute the minimum value*/
    ga.minimize();
    /* initilaize the first generation of population*/
    ga.initialize(seed);

    /* run ga until reach the number of generation
       pre-defined*/
```

```
        int g=0;
        while(g<generation)
        {
            ga.step();
            g++;
        }

        /* assign the value of the best individual to genome*/
        genome=ga.population().best();

        /* assign the each value of control to corresponding
           TBs to an array*/
        for(unsigned short i=0;i<Num;i++)
        {
            G[i]= genome.gene(i);
        }

        /* compute the best value corresponding to
           the best genome GA found*/

        o=objective1(G);
        localOptimal = o;
    }
    /* if the search space is small, find the best value
       without GA*/
    else
    {
        unsigned short index=0;
        struct tableElement c[totalU];
        int optimalV[totalU];
        int min=1000000;
        int minIndex=0;

        /* go through all the admissable control values*/
        for(int a=0;a<=MaxU[0];a++)
        {
            for(unsigned short a1=0;a1<=MaxU[1];a1++)
            {
                c[index].control[0]=a;
                c[index].control[1]=a1;
                /* compute the cost of current control value*/
                optimalV[index]=objective1(c[index].control);

                /* compare if the cost is smaller than the min*/
                if(optimalV[index] <=min)
                {
                    min=optimalV[index];
                    minIndex=index;
                }
                index++;

            }
```

```
                    }
                    /* assign the optimal control value to an array*/
                    for(unsigned short a4=0;a4<Num;a4++)
                    {
                        G[a4]=c[minIndex].control[a4];
                    }

                    localOptimal=optimalV[minIndex];
                }

                /* save the optimal control for the current state
                   and the current state to the table */
                for(unsigned short s=0;s<Num;s++)
                {
                    possibleStates[stage][n].states[s]=states[stage][s];
                    possibleStates[stage][n].control[s]=G[s];
                }
                possibleStates[stage][n].states[Num]=states[stage][2];

                /* save the optimal cost corresponding to current state*/
                twoStageOptimal[optimalIndex][n]=localOptimal;

                n++;

            }

        }
    }
    cout<<"finished stage "<<stage<<endl;
}
/* =========== End of the optimal control process ===================*/

/* trace the optimal value from the initial state to the end */
traceBack();

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 j=0; j<STAGE; j++)
{
    delete [] possibleStates[j];
}
delete [] possibleStates;
```

```
    for(int j6=0;j6<2;j6++)
    {
        delete[] twoStageOptimal[j6];
    }
    delete [] twoStageOptimal;

    return 0;
}
/*==============================================================================
                            END OF THE MAIN FUNCTION
==============================================================================*/


/* read how many stages */
void readStage()
{
    fstream file;
    file.open("Traffic.txt",ios::in);
    file>>STAGE;
    file.close();
}

void readTraffic()
{
    fstream file;
    file.open("Traffic.txt",ios::in);
    file>>STAGE;

    for(unsigned short l=0;l<STAGE;l++)
    {
        for(unsigned short l1=0;l1<Num;l1++)
        {
            file>>Traffic[l][l1];
            cout<<Traffic[l][l1]<<",";
        }
        cout<<endl;
    }
    file.close( );
}

double variance(short int mean,short int source)
{

    int v=0;
    for(int i=0; i<STAGE;i++)
    {
        v=v+ pow((mean-Traffic[i][source]),2);
    }

    return v/STAGE;
}
```

```
double StandardDeviation(double variance)
{
    return sqrt(variance);
}

void readParameters()
{
    fstream f;
    f.open("parameters.txt",ios::in);
    char l[50];
    f.getline(l,50);
    f>>population;
    f>>generation;
    f.getline(l,50);
    f.getline(l,50);
    f>>A;
    f>>B;
    f>>C;
    f.getline(l,50);
    cout<<"===Alpha "<<A <<", beta="<<B<<", gamma="<<C<<endl;
    f.getline(l,50);
    cout<<l<<endl;

    f>>TIMEINTERVAL;
    f.getline(l,50);
    f.getline(l,50);
    cout<<l<<endl;
    f>>SERVICERATE;
    f.getline(l,50);
    f.getline(l,50);
    cout<<l<<endl;
    f>>QUEUESIZE;
    f.getline(l,50);
    f.getline(l,50);
    cout<<l<<endl;
    cout<<"******"<<QUEUESIZE<<endl;
    for(int i=0;i<Num;i++)
    {

        f>>TBSIZE[i];
        cout<<TBSIZE[i]<<".";
    }
    cout<<endl;
    f.getline(l,50);
    f.getline(l,50);
    cout<<l<<endl;
    for(int i2=0;i2<Num;i2++)
    {
        f>>LINKCAPACITY[i2];
        cout<<LINKCAPACITY[i2]<<",";
    }
```

```
    cout<<endl;
    f.getline(l,50);
    f.getline(l,50);
    cout<<l<<endl;
    for(int i1=0;i1<Num;i1++)
    {
        f>>initialState[i1];
    }

    f>>initialState[Num];

    for(short int i3=0;i3<Num;i3++)
    {
        totalState=totalState*(TBSIZE[i3]+1);
    }
    totalState=totalState*(QUEUESIZE+1);
    cout<<"total state "<<totalState<<endl;
    f.close();

}

/* this function is used by no GA computation*/
float objective1(unsigned short G[])
{

    float total=0; // it stores the total cost

    /* define and initial the state of the next stage correspondin to
       the current state*/
    unsigned short nextState[Num+1]={0,0,0};

    /* copmute the conforming traffic*/
    unsigned short totalConfirmed=confirmTraffic(G);

    /* compute the cost current stage*/
    total =B*TotalLostAtTB()+A*LostAtMultiplexer(totalConfirmed);
    unsigned short waiting=WaitingLost(totalConfirmed,states[stage][Num]);

    total= total+C*waiting;


    /* seach the corresponding optimal cost from the next stage to
       the terminal stage*/
    if(stage!=STAGE-1)
    {
        /* compute the state of the next stage according to the
           state transition function */
        for(unsigned short l=0;l<Num;l++)
        {
            nextState[l]=states[stage][l]+G[l]-confirmedTraffic[l];

            if (nextState[l]>TBSIZE[l])
```

```
            {
                nextState[1]=TBSIZE[1];
            }
        }

        nextState[Num]=waiting;
        unsigned short otherIndex;

        if(optimalIndex==0) otherIndex=1;
        else otherIndex=0;

        /* compute the total cost from current stage to the terminal stage*/
        total=total + twoStageOptimal[otherIndex][searchNextV(nextState)];
    }
    return total;
}

unsigned short confirmTraffic(unsigned short Genome[])
{
    unsigned short totalConfirmed=0;
    for (unsigned short i=0 ;i<Num;i++)
    {
        if (Traffic[stage][i]<=(states[stage][i]+ Genome[i]))
        {
            confirmedTraffic[i]=Traffic[stage][i];
        }

        else confirmedTraffic[i]=0;//states[stage][i]+Genome[i];
        if (confirmedTraffic[i]>TIMEINTERVAL*LINKCAPACITY[i])
        {
            confirmedTraffic[i]=0;
        }
        totalConfirmed=totalConfirmed+confirmedTraffic[i];
    }
    return totalConfirmed;

}

/* this function used by GA */
float objective(GAGenome & c)
{
    GA1DArrayAlleleGenome<unsigned short>
                        &genome=(GA1DArrayAlleleGenome<unsigned short>&)c;
    unsigned short nextState[Num+1]={0,0,0};

    float total=0;
    unsigned short G[Num];

    for(unsigned short i=0;i<Num;i++)
    {
        G[i]=genome.gene(i);
    }
```

```
    /* comupute the conforming traffic*/
    unsigned short totalConfirmed=confirmTraffic(G);

    /* compute the cost of current stage */
    total =B*TotalLostAtTB()+A*LostAtMultiplexer(totalConfirmed);
    unsigned short waiting=WaitingLost(totalConfirmed,states[stage][Num]);

    total = total+C*waiting;

    /* search the optimal total cost from next stage to the terminal stage*/
    if(stage!=STAGE-1)
    {
        /* computet the state of the next stage by state transition function
        */
        for(unsigned short l=0;l<Num;l++)
        {
            nextState[l]=states[stage][l]+G[l]-confirmedTraffic[l];
            if (nextState[l]>TBSIZE[l])
            {
                nextState[l]=TBSIZE[l];
            }
        }
        nextState[Num]=waiting;
        unsigned short otherIndex;
        if(optimalIndex==0) otherIndex=1;
        else otherIndex=0;

        /* search the optimal cost from the next stage to the terminal stage
        */
        unsigned short val
                       =twoStageOptimal[otherIndex][searchNextV(nextState)];

        total=total + val;
    }
    return total;
}


/* search the index of the optimal value at next stage corresponding to
   the current state of the current stage*/
int searchNextV(unsigned short nextState[])
{
    unsigned short depth=0;
    int startIndex=0;

    int endIndex=0;

    /* find the range of the index */
    while(depth<Num)
    {
```

```
        int numOfStates=1;
        for(int i=depth+1;i<Num;i++)
        {
            numOfStates=numOfStates*(TBSIZE[i]+1);
        }
        numOfStates=numOfStates*(QUEUESIZE+1);

        startIndex=startIndex+nextState[depth]*numOfStates;

        endIndex=startIndex+numOfStates-1;

        depth++;

    }
    int in;

    /* find the exact index */
    for(int index=startIndex;index<=endIndex;index++)
    {
        if(possibleStates[stage+1][index].states[Num]==nextState[Num])
        {
            in=index;
            break;

        }

    }
    return in;
}


int LostAtMultiplexer(unsigned short totalConfirmed)
{
    return totalConfirmed-
AcceptedTrafficByMul(totalConfirmed,states[stage][Num]);
}

int TotalLostAtTB()
{
    int total = 0;
    for(unsigned short i=0;i<Num;i++)
    {
        total += LostAtTB(confirmedTraffic[i], Traffic[stage][i]);
    }
    return total;
}


int LostAtTB(unsigned short confirmedTraffic, unsigned short Traffic)
//calling by TotalLostAtTB()
{
    return Traffic - confirmedTraffic;
```

```
}


int AcceptedTrafficByMul(unsigned short totalConfirmed,unsigned short
stateAtMul)
{

    unsigned short leftSpace=0;

    int QueueState = stateAtMul-SERVICERATE*TIMEINTERVAL;

    if(QueueState<0)
    {
        QueueState =0;
    }

    leftSpace=QUEUESIZE-QueueState;
    if(totalConfirmed>leftSpace)

        return leftSpace;
    else return totalConfirmed;
}

int WaitingLost(unsigned short confirmed, unsigned short stateAtMul)
{

    int QueueState = stateAtMul - SERVICERATE*TIMEINTERVAL;

    unsigned short goTraffic;

    if(QueueState >0)
    {
        QueueState=QueueState;
    }

    else
    {
        QueueState = 0;
    }

    goTraffic=AcceptedTrafficByMul(confirmed,stateAtMul);

    return QueueState + goTraffic;

}


void traceBack()
{
    bool toNextStage=false;
    fstream f;
```

```
/* open a result file in which the computation result will be writen*/
f.open("result.txt",ios::out);


/* define optimalControl and optimalState to store real optiaml value*/
unsigned short **optimalControl;
unsigned short **optimalState;

optimalControl=new unsigned short *[STAGE];

for(short int i3=0;i3<STAGE;i3++)
{
    optimalControl[i3]=new unsigned short[Num];
}

optimalState=new unsigned short*[STAGE];
for(unsigned short i4=0;i4<STAGE;i4++)
{
    optimalState[i4]=new unsigned short[Num+1];
}


/* initialize the state varibles*/
for(unsigned short j=0;j<=Num;j++)
{
    optimalState[0][j]=initialState[j];
}

int totallost=0;
int stagelost[Num];
int totalOptimal;

/* start tracing */
for(unsigned short i=0;i<STAGE;i++)
{
    unsigned short confirmed[Num];
    unsigned short totalConfirmed=0;

    int index=0;

    /* find the index of the real optimal cost */
    index=searchNextV(optimalState[i]);

    if (i==0)
    {
        totalOptimal=twoStageOptimal[optimalIndex][index];

        /* find the final optimal cost and write it to result file*/
        f<<"***** OPTIMAL VALUE : "<<twoStageOptimal[optimalIndex][index]
         <<endl;
```

```
    cout<<"optimal Value :"<<twoStageOptimal[optimalIndex][index]
         <<endl;

    for(short int k=0;k<Num;k++)
    {
       cout<<possibleStates[i][index].states[k]<<",";
    }
    cout<<endl;

}

f<<"==============At stage "<<i<<":"<<endl;;

/* find the corresponding optimal control from the the table*/
for(short int k=0;k<Num;k++)
{
    optimalControl[i][k]=possibleStates[i][index].control[k];

    f<<"control:"<<optimalControl[i][k]<<";";
}
f<<endl;


for(short int k1=0;k1<=Num;k1++)
{

    f<<"states:"<< optimalState[i][k1]<<";";

}

f<<endl;


/* compute the conforming traffic when both state and the control are
   optimal*/
for (short int s=0 ;s<Num;s++)
{

    if (Traffic[i][s]<=(optimalState[i][s]+ optimalControl[i][s]))
    {
       confirmed[s]=Traffic[i][s];
    }
    else confirmed[s]=0;

    if(confirmed[s]>LINKCAPACITY[s]*TIMEINTERVAL)
    {
       confirmed[s]=0;
    }
    totalConfirmed=totalConfirmed+confirmed[s];

}
```

```
    f<<"totalConfirmed :"<< totalConfirmed<<endl;

    unsigned short waiting=0;

    /* compute waiting losses under optimal condition*/
    waiting=WaitingLost(totalConfirmed,optimalState[i][Num]);


    /* compute the losses at TBs*/
    for(short int k2=0;k2<Num;k2++)
    {

        stagelost[k2]=LostAtTB(confirmed[k2],Traffic[i][k2]);
        f<<"lost TB:"<<stagelost[k2]<<",";
    }
    f<<endl;

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

    f<<"totallost1: "<<totallost<<endl;

    totallost=totallost+totalConfirmed
              -AcceptedTrafficByMul(totalConfirmed,optimalState[i][Num]);

    f<<"totallost:"<<totallost<<endl;

    /* if the final stage is reached, finish tracing*/
    if(i==STAGE-1)
    {
        f<<"waiting : "<<waiting<<endl;
        break;
    }

    /* compute the optimal state at the next stage
       according to the state transition function*/
    for(short int l=0;l<Num;l++)
    {
        optimalState[i+1][l]=optimalState[i][l]+optimalControl[i][l]-
                                                          confirmed[l];
        if (optimalState[i+1][l]>TBSIZE[l])
        {
            optimalState[i+1][l]=TBSIZE[l];
        }
    }
    optimalState[i+1][Num]=waiting;
    state++;
}
```

```
    int totalTraffic=0;
    for(int t=0;t<STAGE;t++)
    {
        for(int t1=0;t1<Num;t1++)
        {
            totalTraffic=totalTraffic+Traffic[t][t1];
        }
    }

    /* write the result to the file */
    int waiting=totalOptimal-A*(totallost-lostAtTB)-B*lostAtTB;
    f<<"TOAL OPTIMAL VALUE ==============> "<<totalOptimal<<endl;
    f<<"None waiting losses =============> "<<totallost<<endl;
    f<<"REAL lost at TB =================> "<<lostAtTB<<endl;
    f<<"Weighted lost at TB =============> "<<B*lostAtTB<<endl;
    f<<"REAL lost at Multiplexer ========> "<<(totallost-lostAtTB)<<endl;
    f<<"Weighted lost at Multiplexer ====> "<<A*(totallost-lostAtTB)<<endl;
    f<<"REAL waiting lost ===============> "<<waiting/C<<endl;
    f<<"Weighted wating lost ============> "<<waiting<<endl;
    f<<"Throughtput =====================> "<<(totalTraffic-totallost)
                                 /(SERVICERATE*TIMEINTERVAL*STAGE)<<endl;

    f.close();

    fstream fcontrol;
    fcontrol.open("control.txt",ios::out);

    cout<<"optimal control"<<endl;

    for(int j6=0;j6<STAGE;j6++)
    {
        for(int j5=0;j5<Num;j5++)
        {
            fcontrol<<optimalControl[j6][j5]<<endl;
            cout<<optimalControl[j6][j5]<<", ";
        }
    }
    fcontrol.close();

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

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

} // END OF THE OPTIMAL CONTROL
```

Appendix B: Constant Rate Open Loop Control Source Code

```
/***********************************************************************
      This is the implementation of the constant rate open loop
      control mechanism - 2 TBs

      Written by: Bo Li
      July 5th, 2002

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

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

/* read number of stages from file*/
void readStage();
/* read traffic trace from traffic file*/
void readTraffic();
/* set open loop control rate*/
int controlRate();
/* compute the cost*/
float objective(int[]);
/* compute the conforming traffic*/
void confirmTraffic(int []);
/* compute the lost at the multiplexor*/
int LostAtMultiplexer();
/* compute the total losses at TB*/
int TotalLostAtTB();
/* compute the TB losses at on TB, calls by TotalLostAtTB()*/
int LostAtTB(int, int);
/* compute the waiting losses*/
int WaitingLost();
/* compute the accepted traffic by the buffer
   calling by WaitingLost() and LostAtMultiplier()*/
int AcceptedTrafficByMul();
/* compute the variance */
double variance(int mean,int source);
/* compute standard deveiation, given the variance*/
double StandardDeviation(double variance);
/* read */
void readParameters();

const int Num=2; // define the number of tbs
int A=0;// define the weight alpha
```

```
int B=0;// define the weight beta
int C=0;// define the weight gamma
int confirmedTraffic[Num];
double SERVICERATE;
double TIMEINTERVAL;
int QUEUESIZE; // define the queue capacity
int LINKCAPACITY[Num];
int TBSIZE[Num];
int STAGE;  //define the total number of stages
int **states; // define a dynamic array to store the states
int **Traffic;
int intControl[3];
int **optimalControl;
int **optimalState;
int stage; // stage index
int acceptedMul;
int totalOptimal=0;
int initialState[Num+1];
int totalState;
int lostAtMultiplexer=0;
int lostAtTB=0;
int lostAtM=0;
float weight=0; // the weight given to the traffic mean

int main(int argc, char **argv)
{


  int control[3];

  /* take the control weight from the argument*/
  for(int i=1; i<argc; i++) {
        weight = atof(argv[i]);
  }


      int stagelost[Num];
      int totallost=0;

      /* read the number of stages from a file*/
      readStage();

      /* creat an array to store the states*/
      states=new int*[STAGE+1];
      for(int i1=0;i1<STAGE+1;i1++)
      {
            states[i1]=new int[Num+1];
      }

      /* create an array to store the traffic*/
```

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

/* read traffic from a file*/
readTraffic();

/* calculate token generation rate*/
controlRate();

/* read parameters from a file*/
readParameters();

fstream confirming;
fstream tbstate;
fstream controlf;
confirming.open("confirming.txt",ios::out);
controlf.open("control.txt",ios::out);
tbstate.open("tbstate.txt",ios::out);
int lost=0;

/* start the constant rate open loop control transmission*/
for ( stage=0; stage<STAGE; stage++)
{
      /* assign control values */
      for(int ss=0;ss<Num;ss++)
      {
            control[ss]=intControl[ss];
      }

      /* if the stage is 0, initialize the TB size*/
      if(stage==0)
      {
            for(int s=0;s<=Num;s++)
                  states[stage][s]=initialState[s];
      }
      cout<<"===============stage :"<<stage<<endl;

      /* write the current states into a file*/
      for(int l1=0;l1<=Num;l1++)
      {
            cout<<"states :"<<states[stage][l1]<<",";
            tbstate<<states[stage][l1]<<" ";
      }
      cout<<endl;
      tbstate<<endl;

      /* if the tb can not occopy the token generated, only put
```

```
   a number of tokens equal to the left space of TB*/
for(int l2=0;l2<Num;l2++)
{
      if(states[stage][l2]+control[l2]>TBSIZE[l2])
      {
            control[l2]=TBSIZE[l2]-states[stage][l2];

      }
      cout<<"control:"<<control[l2]<<",";
      controlf<<control[l2]<<" ";
}
cout<<endl;
controlf<<endl;

/* calculate the current total losses*/
lost = lost+ objective(control);

/* write the conforming traffic at the current stage into a
file*/
for(int c=0;c<Num;c++)
{
      confirming<<confirmedTraffic[c]<<" ";
}
confirming<<endl;

for(int k2=0;k2<Num;k2++)
{
stagelost[k2]=LostAtTB(confirmedTraffic[k2],Traffic[stage][k2]);
      cout<<"lost TB:"<<stagelost[k2]<<",";
}
cout<<endl;

/* calculate the total losses at TB until current stage*/
for(int k3=0;k3<Num;k3++)
{
      lostAtTB=lostAtTB+stagelost[k3];
}

/* calcualte the total losses at multiplexer til current stage*/
lostAtM=lostAtM+lostAtMultiplexer;
cout<<"the lost is "<<lost<<endl;

/* if it is the final stage, only assign the value to the
   state of multiplexor*/
if(stage==(STAGE-1))
{
      states[stage+1][Num]=WaitingLost();
      break;
}
```

```
        /* if it is not the final stage, assign the values to TB states
*/
        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];
              }
        }
        states[stage+1][Num]=WaitingLost();
  }

  /* calculate the total traffic */
  int totalTraffic =0;
  for(int t=0;t<STAGE;t++)
  {
        for(int t1=0;t1<Num;t1++)
        {
              totalTraffic=totalTraffic+Traffic[t][t1];
        }
  }

  /* calculate the total waiting losses*/
  int waitingLosses=(lost-A*lostAtM-B*lostAtTB)/C;

  /* output statistics */
  cout<<"Non waiting lost =============> "<<lostAtM+lostAtTB<<endl;
  cout<<"REAL lost At TB ==============> "<<lostAtTB<<endl;
  cout<<"weighted lost at TB ==========> "<<B*lostAtTB<<endl;
  cout<<"REAL lost at Multiplexer =====> "<<lostAtM<<endl;
  cout<<"weighted lost at Multiplexer => "<<A*lostAtM<<endl;
  cout<<"REAL waiting losses ==========> "<<waitingLosses<<endl;
  cout<<"Weighted waiting losses ======> "<<C*waitingLosses<<endl;
  cout<<"total traffic=================> "<<totalTraffic<<endl;
  cout<<"network utilization===========> "<<(totalTraffic-lostAtM-
                        lostAtTB)/(SERVICERATE*TIMEINTERVAL*STAGE)<<end
                        l;
  cout<<weight<<endl;

  confirming.close();
  tbstate.close();
  controlf.close();

  /* release the memory allocation */
  for(int j1=0;j1<STAGE;++j1)
  {
        delete [] states[j1];
  }
```

```
	delete [] states;

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

	return 0;
}

/* read total number of stage from a file */
void readStage()
{
	fstream file;
	file.open("Traffic.txt",ios::in);
	file>>STAGE;
	file.close();
}

/* read the traffic traces from a file */
void readTraffic()
{
	fstream file;

	file.open("Traffic.txt",ios::in);
	file>>STAGE;

	for(int l=0;l<STAGE;l++)
	{
		for(int l1=0;l1<Num;l1++)
		{
			file>>Traffic[l][l1];
		}
	}
	file.close( );
}

/* assign the constance control rate*/
void controlRate()
{

	int mean[Num];

	for(short int i=0;i<Num;i++)
	{
		short int temp=0;
		for(short int i1=0;i1<STAGE;i1++)
		{
			temp=temp+Traffic[i1][i];
```

```
            }
            cout<<"temp "<<temp<<endl;

            /* calculate the traffic mean rate*/
            mean[i]=(temp/STAGE);
            if (weight==0)
            {
                  intControl[i]=15;
            }

            /* assign a weight to the control value */
            else intControl[i]=ceil(mean[i]*weight);
            cout<<"intControl :"<<intControl[i]<<","<<endl;
            cout<<"mean traffic "<<i<<":"<<mean[i]<<endl;
      }
}

/* compute the variance of the packet size*/
double variance(short int mean,short int source)
{
      int v=0;
      for(int i=0; i<STAGE;i++)
      {
            v=v+ pow((mean-Traffic[i][source]),2);
      }
      return v/STAGE;
}

/* compute the standard deveiation of the packet size*/
double StandardDeviation(double variance)
{
      return sqrt(variance);
}

/* read all parameters from a file */
void readParameters()
{
      fstream f;
      f.open("parameters.txt",ios::in);
      char l[50];
      f.getline(l,50);
      f>>A;
      f>>B;
      f>>C;
      f.getline(l,50);
      cout<<"===Alpha "<<A <<", beta="<<B<<", gamma="<<C<<endl;
      f.getline(l,50);
      cout<<l<<endl;

      f>>TIMEINTERVAL;
```

```
	f.getline(l,50);
	f.getline(l,50);
	cout<<l<<endl;
	f>>SERVICERATE;
	f.getline(l,50);
	f.getline(l,50);
	cout<<l<<endl;
	f>>QUEUESIZE;
	f.getline(l,50);
	f.getline(l,50);
	cout<<l<<endl;
	cout<<"******"<<QUEUESIZE<<endl;
	for(int i=0;i<Num;i++)
	{

		f>>TBSIZE[i];
		cout<<TBSIZE[i]<<".";
	}
	cout<<endl;
	f.getline(l,50);
	f.getline(l,50);
	cout<<l<<endl;
	for(int i2=0;i2<Num;i2++)
	{
		f>>LINKCAPACITY[i2];
		cout<<LINKCAPACITY[i2]<<",";
	}
	cout<<endl;
	f.getline(l,50);
	f.getline(l,50);
	cout<<l<<endl;
	for(int i1=0;i1<Num;i1++)
	{
		f>>initialState[i1];
	}

	f>>initialState[Num];

	for(short int i3=0;i3<Num;i3++)
	{
		totalState=totalState*(TBSIZE[i3]+1);
	}
	totalState=totalState*(QUEUESIZE+1);
	cout<<"total state "<<totalState<<endl;
	f.close();

}

/* compute the objective function */
float objective(int Genome[])
```

```
{
      float total=0;

      /* compute the conforming traffic*/
      confirmTraffic(Genome);

      total = total+A*LostAtMultiplexer();
      total = total+C*WaitingLost();
      total = total+B*TotalLostAtTB();

      return total;
}

/* compute the conforming traffic*/
void confirmTraffic(int Genome[])
{

      for (int i=0 ;i<Num;i++)
      {
            if (Traffic[stage][i]<=(states[stage][i]+ Genome[i]))
            {
                  confirmedTraffic[i]=Traffic[stage][i];
            }
            else confirmedTraffic[i]=0;
            if (confirmedTraffic[i]>TIMEINTERVAL*LINKCAPACITY[i])
            {
                  confirmedTraffic[i]=0;
            }
      }
}

/* compute accepted traffic by the multiplexor*/
int AcceptedTrafficByMul()
{
      int totalT=0;
      int leftSpace=0;

      /* compute the total comforming traffic */
      for(int i=0;i<Num;i++)
      {
            totalT += confirmedTraffic[i];
      }

      /* compute the queue state */
      int QueueState = states[stage][Num]-SERVICERATE*TIMEINTERVAL;
      if(QueueState<0)
      {
            QueueState =0;
      }
```

```
	/* compute the left space */
	leftSpace = QUEUESIZE-QueueState;

	/* return the number of accepeted traffic by multiplexor*/
	if(totalT>leftSpace)
		return leftSpace;
	else return totalT;
}

/* compute the losses at the multiplexor*/
int LostAtMultiplexer()
{
	int totalConfirmed=0;
	for(int i=0;i<Num;i++)
	{
		totalConfirmed+=confirmedTraffic[i];
	}
     lostAtMultiplexer=totalConfirmed-AcceptedTrafficByMul();
	return lostAtMultiplexer;

}

/* compute the total losses at TB*/
int TotalLostAtTB()
{
	int total = 0;
	for(int i=0;i<Num;i++)
	{
		total += LostAtTB(confirmedTraffic[i], Traffic[stage][i]);
	}
	return total;
}

/* compute the losses at one TB*/
int LostAtTB(int confirmedTraffic, int Traffic)
{
	return Traffic - confirmedTraffic;
}

/* compute the waiting losses at the queue*/
int WaitingLost()
{
	int QueueState = states[stage][Num] - SERVICERATE*TIMEINTERVAL;
	int goTraffic;
	if(QueueState >0)
	{
		QueueState=QueueState;
		goTraffic=AcceptedTrafficByMul();
	}
	else
```

```
	{
		QueueState = 0;
		goTraffic = AcceptedTrafficByMul();
	}
	return QueueState + goTraffic;

}

/*****************************************************************************
		END OF THE CONSTANT RATE OPEN CONTROAL PROGRAM
*****************************************************************************/
```