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.

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

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

Université d'Ottawa · University of Ottawa

A DYNAMIC MODEL OF THE TOKEN BUCKET CONTROL MECHANISM IN COMPUTER NETWORKS

By

Wang, Qun

A thesis submitted to
The School of Graduate Studies and Research
University of Ottawa
In partial fulfillment of the requirements
For the degree of

Master of Applied Science
In Electrical Engineering

Ottawa-Carleton Institute of Electrical Engineering
School of Information Technology and Engineering
Faculty of Engineering
University of Ottawa
Ottawa, Ontario, CANADA

December 2001

ACKNOWLEDGMENTS

I would like to thank my supervisors, Dr. NasirUddin Ahmed and Dr. Luis Orozco-Barbosa, for their constant guidance and encouragement through this research and thesis work.

I also would like to acknowledge the financial support of National Science and Engineering Research Council.

DEDICATION

To my family in TianJin for their understanding and support.

ABSTRACT

In order to provide Quality of Service (QoS), the network must enforce traffic management such as traffic classification, admission control, access control (shaping and policing), flow control and congestion control. Applying the policies and mechanisms of traffic management, the Internet Engineering Task Force (IETF) has proposed various QoS models, Integrated Service (IntServ), Differentiated Service (DiffServ) and Multiple Protocol Label Switching (MPLS), for the next-generation Internet in order to deliver emerging multimedia applications, which have stringent QoS requirements. It is interesting to note that the Token Bucket (TB) algorithm as a control mechanism has been employed popularly in them.

In this thesis, we construct a new dynamic model for the TB control mechanism and apply a system approach to its analysis. This model is then augmented by adding a dynamic model of a multiplexer in an access node, where the TB exercises a policing function. In the model, traffic policing, traffic multiplexing and network utilization are formally defined. Based on the model, the issues such as QoS, traffic sizing and network dimensioning are studied. And also we propose an algorithm using feedback control to improve QoS and network utilization.

Applying MPEG video traces as the input traffic to the model, the numerical computing results very much agree to the analytical model. The model gives a guideline to study the interactions among the traffics and elements of TB-based access control to achieve desired QoS and network utilization.

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

TABLE OF ACRONYMS

AF	Assured Forwarding
ATM	Asynchronous Transfer Mode
CBS	Committed Burst Size
CBR	Constant Bit Rate
CIR	Committed Information Rate
EBS	Excess Burst Size
ECN	Explicit Congestion Notification
EF	Expedited Forwarding
FIFO	First In First Out
IETF	Internet Engineering Task Force
InteServ	Integrated Service
IP	Internet Protocol
ISP	Internet Service Provider
LAN	Local Area Network
LBAP	Linear Bounded Arrival Processes
LDP	Label Distribution Protocol
LSR	Label Switched Routers
LSP	Label Switched Path
MPEG	Motion Picture Engineering Group
MPLS	Multiple Protocol Label Switching
PHB	Per-Hop Behavior
PIR	Peak Information Rate
QoS	Quality of Service
RAS	Rate Adaptive Shaper
RED	Random Early Detection
RSVP	Reservation Protocol
SRTCM	Single Rate Three Color Marker
TB	Token Bucket

TRTCM	Two Rate Three Color Marker
UDP	User Datagram Protocol
VBR	Variable Bit Rate

CHAPTER 1

Introduction and Motivation

1.1 Background Information

Currently, the Internet provides "best effort" packet delivery services, where all kinds of traffics such as data, voice and video are treated in the same manner. This simplicity has made the Internet scalable and therefore successful. However, as more and more users and applications have been connecting to the network, the network service demand eventually exceeds its capacity. Instead of blocking the overloaded demand, the network degrades its services gracefully over all applications. Although the resulting variability in delivery delays, jitter and packet loss do not adversely affect typical Internet applications such as email, file transfer and Web applications, other applications cannot adapt to this inconsistent service levels. For example, delivery delays and jitters cause problems for applications with real-time requirements, such as real-time video and audio.

Although some argue that increasing bandwidth may accommodate these real-time applications, it is still not enough to avoid jitter during traffic bursts. Even on a relatively unloaded network, delivery delays can vary enough to continue adversely affecting real-time applications [1]. Moreover, the Internet is dependent on the end systems' cooperation and there is no way to prevent non-adaptive sources from gaining greater shares of network bandwidth and thereby starving other, well-behaved sources.

In order to deal with these issues over the Internet, it is necessary to add some level of quantitative or qualitative determinism on the existing services. This requires implementing some intelligence to the Internet to differentiate traffic: one with strict

timing requirements and the other that can tolerate delay, jitter and loss. That is what QoS mechanisms are designed to do. QoS mechanisms do not create bandwidth, but manage to make it work more effectively to meet the wide range of application requirements. The goal of QoS mechanisms is to enhance the Internet with some level of predictability on top of "best effort" services.

IETF has been proposing and working on such new service models as the Integrated Service (IntServ) [2], Differentiated Service (DiffServ) [3] and Multiple Protocol Label Switching (MPLS) [4] as the solutions to provide QoS for the next-generation Internet.

1.1.1 Integrated Service

IntServ is a flow-oriented QoS service model. A flow is defined as a stream of packets originating from a specific user activity, for example, a single application session. A flow may be identified by different policies: for instance, IPv4 uses source and destination IP address and the destination port number. IntServ reserves bandwidth for individual flows to provide guaranteed services. Two main service categories are defined based on the delay and loss requirements.

(1) Guaranteed Delay service [5] provides absolute guarantees on the delay and loss experienced by a flow. Packets are neither lost nor experiencing delay exceeding the specified bound. These firm guarantees are provided using resource reservations.

(2) Controlled Load service [6] provides the service that is equivalent to that experienced on an unloaded network. Most packets are neither lost nor experiencing queuing delay. However, they are not provided with specific quantitative guarantees.

IntServ requires that resources be reserved for flows in order to provide the requested QoS. Reservation Protocol (RSVP) [7] is a protocol designed to provide resource reservations and it is designed to work with IntServ, though it can be used with other service models as well.

IntServ has some limitations, during which the biggest concern is whether it can scale to large networks such as backbone networks, where there are a large number of individual flows. For making resource reservation for these flows, there need to be a huge amount of control messages that require a lot of processing power. Likewise, maintaining state information for all the flows can require a lot of storage capacity. Moreover, to classify a large number of packets and schedule numerous queues, the mechanisms to implement at the routers become extremely complex. Furthermore, policy and security issues need to be resolved as well to determine who can make reservations and prevent unauthorized users from making a reservation.

However, for small networks such as Intranets where there are less number of flows, IntServ is an appropriate model to provide QoS. Large backbone networks will need more scalable mechanisms to differentiate traffic and provide differentiated services to them.

1.1.2 Differentiated Service

The DiffServ architecture is a class-oriented QoS service model. It is based on a simple architecture where traffic entering a network is classified and possibly conditioned at the boundaries of the network, and assigned to different behavior aggregates. Each behavior aggregate is identified by a single DiffServ (DS) codepoint. Within the core of the network, packets are forwarded according to the Per-Hop Behavior (PHB) associated with the DS codepoint. The per-hop behavior is the externally observable forwarding behavior applied at a DS-compliant node to a DS behavior aggregate. By supporting differentiated service in Internet, the network service providers could offer different types or grades of services to different customers (e.g. real-time video and audio require low delay while file transmission require high throughput). In this way, the customers willing to pay more could get better services.

In order to provide different classes of services to different customers, a 6-bit "DS Field" in IPv4 and IPv6 is dedicated for DiffServ use. Different values in this field indicate different types of services the customer may get. A value in this field is also called a DS codepoint. The DS codepoint could be set by the traffic sources or could also be set at the boundary nodes of every DS domain the packets enter.

The key features of DiffServ that overcome some of the limitations of IntServ are:

(1) Coarse differentiation: Instead differentiating per flow traffic, DiffServ defines a small number of well-defined classes.

(2) Complexity in the edges and simplicity in the core (good scalability): In IntServ with RSVP, each router in the network implements packet classification in order to provide different levels of service. To make networksF more scalable, in DiffServ, packet classification is moved to the edge of the network. Edge routers classify and mark packets appropriately. Interior routers simply process packets based on these markings. This implies that interior routers do not recognize individual flows; instead, they deal with aggregate classes.

Beside "best effort" service, there are two other classes of services in DiffServ-capable network: Assured Forwarding (AF) service, Expedited Forwarding (EF) service.

(1) Assured Forwarding service [8] is a service that allows the Internet Service Provider (ISP) to offer different levels of forwarding assurances for IP packets received from a customer. The idea behind AF is to give the customer the assurance of a minimum throughput, even during periods of congestion, while allowing users to consume more bandwidth when the network load is low. Thus, a connection using the AF service should achieve a throughput equal to the subscribed minimum rate, also called target rate, plus some share of the remaining bandwidth gained by competing with all the active "best effort" connections.

In AF class, IP packets can be marked with one of the three possible drop precedence. In case of congestion, the drop precedence of a packet determines the relative importance of the packet within the AF class. A congested DS node tries to protect packets with lower drop precedence from being lost by discarding packets with higher drop precedence. By using the drop precedence, we can effectively control non-adaptive sources (e.g. UDP sources) from getting more than their fair share of network resources.

(2) Expedited Forwarding [9] service provides low loss, low latency, low jitter, and bandwidth guarantees on end-to-end basis through DS domain. Examples of applications that might use this service are real time applications such as video and audio based applications.

Loss, latency and jitter are all due to the queues traffic traverses while transmitting the network. Therefore providing low loss, latency and jitter for some traffic aggregate means ensuring that the traffic sees no (or very small) queues. This can be achieved by ensuring that, at any time, the output capacity is higher or equal to the input capacity of a given queue.

However, there are still some limitations in DiffServ. The major disadvantage of DiffServ is that providing QoS to traffic flows on a per-hop basis often cannot guarantee end-to-end QoS. Moreover, providing QoS to aggregate flows cannot guarantee per-flow QoS within.

1.1.3 Multi-Protocol Label Switching

MPLS was originally designed to improve the forwarding speed of routers but is now emerging as a technology of traffic engineering. In some respects, MPLS is similar to DiffServ, as it also marks traffic at ingress boundaries in a network, and un-marks at egress points. But the difference is that DiffServ uses the marking to determine priority within a router and MPLS uses markings (labels) to determine the next router hop.

When IP packets enter an MPLS capable network, the MPLS edge router analyzes the contents of the IP header and selects an appropriate label to encapsulate the packet. This short fixed-length label acts as a shorthand representation of an IP packet's header. Part of the great power of MPLS comes from the fact that, in contrast to conventional IP routing, this analysis can be based on more than just the destination address carried in the IP header.

In MPLS network, packets with fixed labels can be forwarded very efficiently because MPLS routers would not look at the entire packet header, rather only at the label and use that to forward the packet. The label not only allows packets to be forwarded more quickly, but also allows the paths to be set up in a various ways, for example, the path could represent the normal destination-based routing path, a policy-based explicit route, or a reservation-based flow path. Ingress routers classify incoming packets and wrap them in an MPLS header that carries the appropriate label for forwarding by the interior routers. The labels are distributed by a dynamic Label Distribution Protocol (LDP) [10], which effectively sets up a Label Switched Path (LSP) along the Label Switched Routers (LSR).

1.1.4 Complete Network QoS Model

The three QoS models, IntServ, DiffServ and MPLS, are not mutually exclusive to one another. On the contrary, they complement each other nicely to provide top-to-bottom and end-to-end QoS between senders and receivers. IntServ tries to solve the problems by resource reservation. But it fails due to scalability issues. DiffServ employs prioritization and aggregation to limit the number of traffic classes in the backbone. But DiffServ also fails because end-to-end QoS is difficult to be guaranteed by aggregating flows. MPLS is more of a traffic-engineering protocol than a QoS protocol by itself supporting routing in the edges using IP philosophy where IntServ can

be used, and switching at the core using ATM philosophy by using DiffServ techniques. IETF is actively exploring the integration of these QoS models together for the future Internet [11-13] where IntServ works in the edges, DiffServ works in the core , and MPLS conducts fast forwarding and traffic engineering.

1.2 Thesis Motivation and Objectives

From the proposals of IntServ, DiffServ and MPLS from IETF [14-17], it is interesting to note that the TB algorithm as a control mechanism has been employed to implement these QoS models popularly in the edge of the network. There is a number of previous research devoted to the TB algorithm analysis. For example, [18] presents two models - "exact model" and "fluid flow model" for the TB algorithm and applies ON and OFF traffic model to study selectivity of TB, i.e., the effectiveness of the device in detecting critical source behavior, and the filtering properties, i.e., the ability of controlling the load offered to the network. The contribution of [19] is that it analyzes the throughput and blocking of packets for a general Markovian arrival process and the robustness of the throughput to unknown arrival processes as well. [20] assumes that time is slotted and arrival in each time slot is considered to be a batch process. The distribution of batch sizes is arbitrary while the time between arrivals of successive batches is taken to be geometrically distributed. By using Matrix Analytic techniques, the queue length distribution at the access point is obtained. [37] studys how leacky bucket algorithm affects long-rang dependent traffic. [38] develops an analytical model for token bucket and leaky bucket. In sum, all of the previous works either make a strong assumption on the traffic, the models are static, or only study one of the aspects of traffic management and network performance. So far, there is no a unified and consistent approach to the TB related problem even though TB algorithm has been employed extensively in both

academic research and industries in computer networks.

In this thesis, we construct a new dynamic model for the TB control mechanism and use a system approach for its analysis. This model is then augmented by adding a dynamic model for a multiplexer at an access node where the TB exercises a policing function. In the model, traffic policing, traffic multiplexing and network utilization are formally defined. Based on the model, we study such issues as QoS, traffic sizing and network dimensioning. Furthermore, we develop an algorithm using feedback control to improve QoS and network utilization using the TB as the underlying control mechanism.

By applying MPEG video traces as the input traffics to the model, we verify the effectiveness of the models and algorithms. The models set the guidelines to study the interactions among the traffics and elements of TB-based access control to achieve optimal QoS and network utilization.

1.3 Thesis Organization

The rest of the thesis is structured as follows. Chapter 2 introduces the traffic management and applications of TB algorithm in traffic management and DiffServ capable network. Chapter 3 presents a dynamic or equivalently a state space model for TB based access control to be developed at the edge node consisting of a number of TB-s connecting to a multiplexer. Applying a system approach, we provide a formal definition of traffic policing, multiplexing and network utilization . In Chapter 4, we define the objective functions in terms of QoS for a network provider and employ a feedback control dynamically allocating available bandwidth over the network users to improve QoS and network utilization. Chapter 5 undertakes the study using MPEG video traces as input traffic to verify the effectiveness and practicability of the models.

Chapter 6 draws the conclusions and mentions certain outstanding issues for future work.

CHAPTER 2

Traffic Management and Applications of Token Bucket Algorithm

This chapter introduces the principles and elements of traffic management and its components. And then we present different versions of TB and their applications in traffic management. Finally, we introduce the application of TB algorithm in DiffServ QoS model proposed from IETF.

2.1 Traffic Management

Traffic management refers to a set of polices and mechanisms that makes a network to provide QoS to selected network traffic over various underlying technologies including X.25, Frame Relay, Asynchronous Transfer Mode (ATM), IEEE 802 Local Area Networks (LANs) and IP-based networks etc., while attempting to make maximum use of network resources. In this thesis, we only study traffic management on IP-based networks or the Internet. There is a significant similarity among traffic management on the different network technologies. In particular, traffic management is composed of such essential components as traffic classification, admission control, access control (shaping and policing), flow control and congestion control.

2.1.1 Traffic Classification

Traffic classification is the first step toward solving performance problems when network traffic contend for limited bandwidth. In order to provide the requested QoS that can be measured by a matrix of delay, jitter, loss, bandwidth and availability (probability

of service blocking), it is critical to classify traffics to enable different QoS treatment. And traffic classes also represent the types of service provided by the network. Generally, we partition network services into two fundamental classes: guaranteed service and "best effort" service, depending on whether it provides a performance guarantee to the traffic. This can be done based on various fields in IP headers (e.g., source, destination addresses and protocol type) and higher layer protocol headers (e.g., source and destination port numbers for TCP or UDP).

2.1.2 Admission Control

Admission control determines whether a requested "connection" is allowed to be carried by the Internet. The main considerations behind this decision are whether admitting the " connection" would reduce the service quality of existing "connections", or whether the incoming "connection" QoS requirement can be met. If either of these conditions holds, the "connection" either delayed until resources are available, or rejected. Thus, admission control plays a crucial rule in ensuring that the Internet meets its QoS requirements. The algorithms for admission control is simple for "best effort" and Constant Bit Rate (CBR) traffic but very complex for real-time VBR (Variable Bit Rate) traffic [21][34]. The algorithm for CBR admission control can be comparing peak rate of CBR and available bandwidth and "best effort" traffic can be shaped to CBR and use the same method. The admission control for VBR is hard because VBR traffic is inherently bursty. That is, it is characterized by the periods where the rate can be much greater than the average rate. Peak rate allocation would result in under utilization of the system resources, while average rate allocation would result in delay and loss. Current available methods for VBR traffic admission control are using renegotiation(feedback control) and effective bandwidth[22-24]. For QoS enabled IP networks, Admission Control, for example, could be performed in the setting up of

RSVP flows or MPLS paths.

2.1.3 Traffic Shaping and Policing

In QoS enabled IP networks, it is necessary to specify the traffic profile for a "connection" to decide how to allocate various network resources. Traffic shaping and policing ensures that traffic entering at an edge or a core node adheres to the profile specified. Typically, this mechanism is used to reduce the burstiness of a traffic stream. Generally, shaping may be implemented at end systems or at the edge of the network. This involves a key tradeoff between benefits of shaping (e.g., loss in downstream network) and the shaping delay. Policing often happens at the edge of the network. Leaky Bucket and Token Bucket based traffic shaping and policing are examples of these mechanisms.

2.1.4 Flow Control

When a server transfers a file to a client after fragmenting it into packets, the server faces the nontrivial problem of choosing the rate at which to inject these packets into the network. If it sends all the packets at once, it may send them faster than the network or the client can process them, leading to congestion and packets loss. Flow control refers to a set of techniques that enable a data source to match its transmission rate to the currently available service rate at a receiver and in the network. Flow control is often confused with congestion control. Congestion control includes not only flow control but also the actions being taken after congestion happens.

We can divide flow control techniques into three broad categories: open loop, closed loop and hybrid [21]. In open-loop flow control, during "connection" establishment, a source describes its behavior with a set of parameters called traffic descriptors and negotiates bandwidth and buffer reservations with network elements along the path to

its destination. The network elements examine this description and decides whether it can support the "connection". If it can, it forwards the setup request to the next element along the path. Otherwise, it negotiates the parameters down to an acceptable value, or blocks the call. The difficulty in open-loop flow control is for a source to capture its entire future behavior with a handful of parameters, i.e. traffic descriptors, because the network's admission-control algorithm uses these parameters to decide whether to admit the source or not. The examples of traffic descriptors are such as average rate, peak rate and Linear Bounded Arrival Processes (LBAP)[25].

Open-loop flow control works best when a source can describe its traffic well with a small number of parameters. Otherwise, the source is better off with closed-loop and hybrid flow control. In closed-loop flow control, source transmission rate is adjusted dynamically in response to feedback signals, so that the ensemble of sources does not overload the network. If closed-loop flow control is ineffective, sources either suffer excessive packet loss or underutilize network resources.

In hybrid flow control, a source reserves some minimum capacity, but may obtain more if other sources are inactive. Thus, the source must go through admission control during the "connection" setup phase, during which every network element must test whether the minimum bandwidth is available. Subsequently, sources adjust their demand in response to the bandwidth available in the network.

2.1.5 Congestion Control

All networks have a finite capacity. Too much traffic injected to a network will cause congestion. Once congestion happens, the network applications will suffer packet delay and loss. For QoS guaranteed networks to operate in a stable and efficient fashion, it is essential that they have viable and robust congestion control capabilities. These capabilities refer to the ability to flow control as described in the last part and shed

excessive traffic during the periods of congestion.

If a router runs out of buffer, it must make a decision on which packets to discard. This depends on the application and on the error strategy used in the data link layer. Older packets will be worse to throw away for a file transfer than newer ones, since they will cause a gap in the received packets and a go-back-n strategy may be employed by the data link layer. For real-time voice or video it is probably better to throw away old data that are already obsolete and keep new packets [26]. It appears to be better for routers to start dropping packets as soon as congestion seems likely, rather than wait for congestion to take over. Random Early Detection (RED)[27] and Explicit Congestion Notification (ECN)[28] are two of the proposed capabilities.

Many design parameters at the data link, network and transport layers affect congestion. Setting each of these with a mind to preventing congestion helps to eliminate the problem. For example, a flooding routing algorithm is fast, but generates a lot of redundant packets. Timers in the data link layer that are set too tight may cause needless retransmission of packets [26].

One can try to prevent congestion by careful design, or by using a more dynamic approach, sensing the presence of congestion and then either increasing subnet capacity (spare lines and routers) or decreasing the traffic offered to the subnet. These methods are feedback strategies.

2.2 Applications of Token Bucket Algorithm in Traffic Management

A TB is actually a traffic filter. There are two parameters concerned: one is token bucket size, and the other one is token rate in the algorithm. Tokens, which are credits enabling a user to send a certain number of bytes in packets into the network, are filled

into the bucket at a certain "token rate". The bucket itself has a specified capacity, "token bucket size", over which newly arriving tokens are discarded. To transmit a packet, the algorithm will remove from the bucket a number of tokens equal to the packet size.

Depending on what enforcement action is taken and whether or not there is a "shaping" buffer, four different versions of TB schemes are summarized in [36]: If there are not enough tokens in the bucket to send a packet, the newly arriving packet is either dropped (**figure 1**), or considered nonconforming and thus sent out with a lower priority tag (**figure 2**). The tagging packets will be dropped first once the network experiences congestion. Alternatively, the newly arriving packet can be temporally stored in a finite shaping buffer and waiting for tokens to be generated before being sent out. When the buffer is full, either the newly arriving packet is dropped (**figure 3**), or buffered packets are chosen , tagged with lower priority and sent out. At the same time, the newly arriving one is stored (**figure 4**).

In addition, there is another scheme called "leaky bucket" as shown in **figure 5**. Leaky bucket is actually a simple shaper, shaping arriving packet at a desired rate. There is no tokens involved in this scheme.

A TB has different functions in the different segments of traffic management. In admission control, the parameters of TB are used as a set of traffic descriptors for network management to make a decision in regards to acceptance or rejection of a traffic by comparing the effective bandwidth of the traffic and the available bandwidth in the network. After the network accepts the traffic, TB works as a shaper to smooth the traffic in order to force it to conform to the contract between the user and the network. Basically, the token bucket mechanism used for traffic shaping has both a token bucket and a shaping buffer. If it does not have a shaping buffer, it would be a policer that drops or tags the nonconforming packets in the edge of the network. For

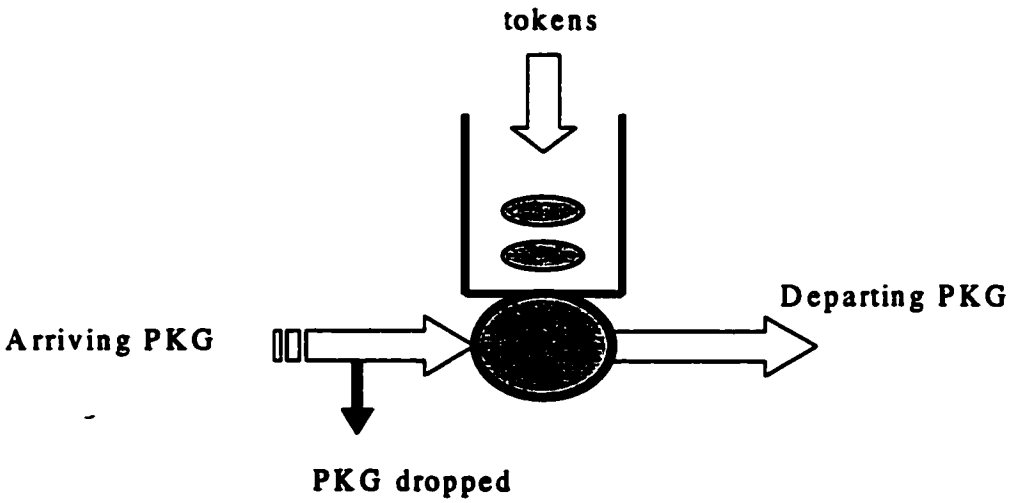

Figure 1 Token bucket version 1

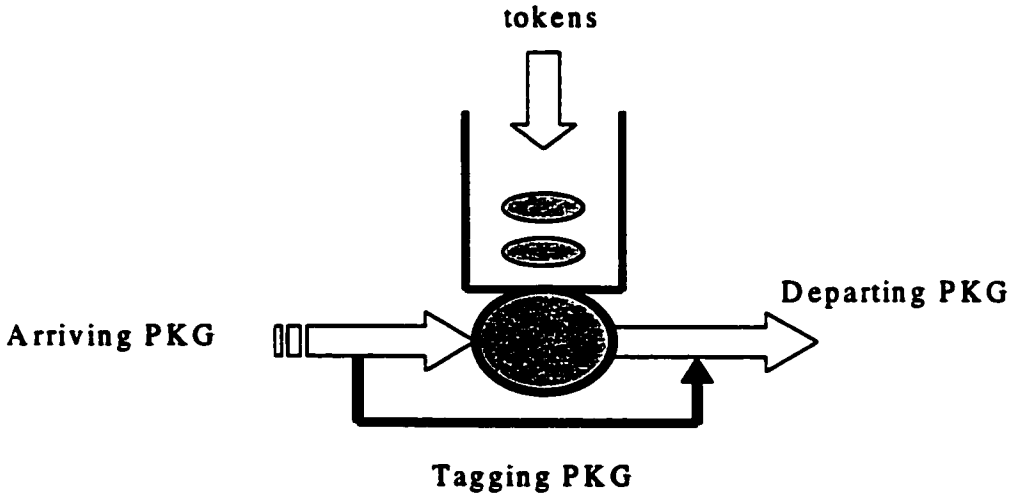

Figure 2 Token bucket version 2

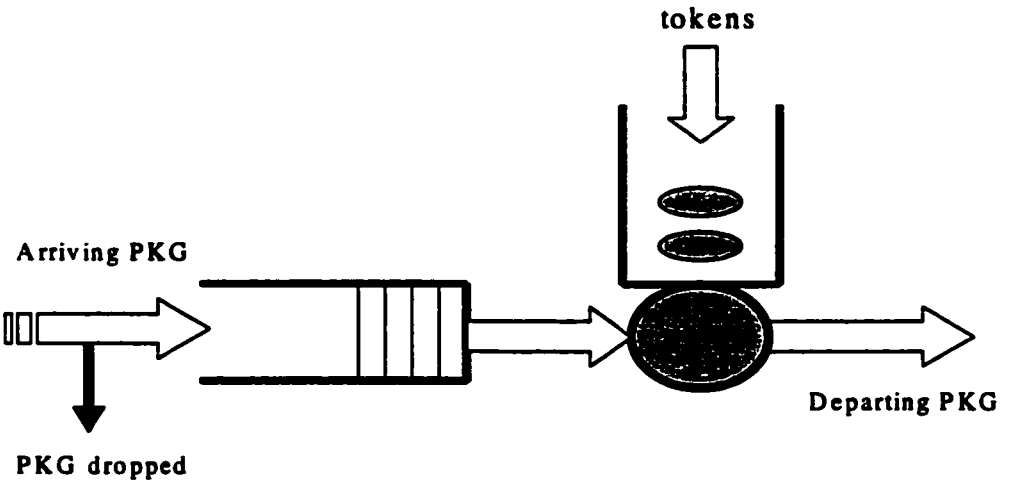

Figure 3 Token bucket version 3

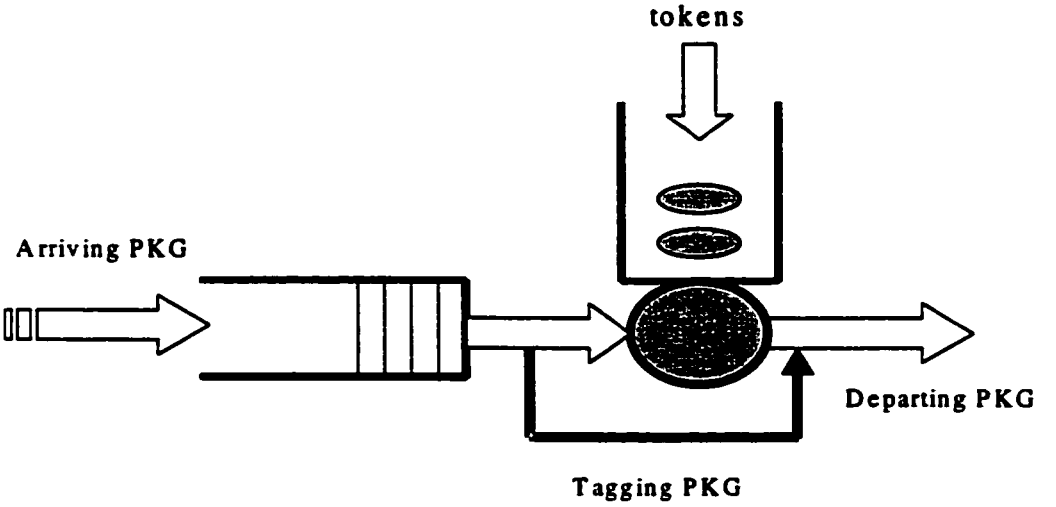

Figure 4 Token bucket version 4

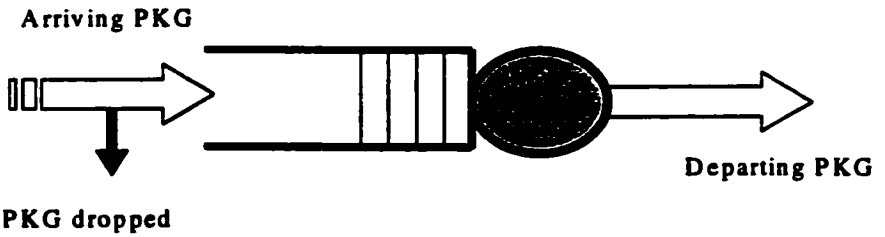

Figure 5 Leaky bucket version

bursty traffic, sometimes renegotiation or feedback control is necessary to achieve the acceptable QoS and network utilization. Thus, the traffic sources and the parameters of TB need to be adjusted dynamically by the feedback information from the network. Next, the TB applications in DiffServ are given in detail and the reader may refer to [14] and [17] for its applications in IntServ and MPLS.

2.3 Application of TB in DiffServ

DiffServ capable networks can comprise one or more DS (DiffServ) domains. A DS domain is a contiguous set of DS nodes that operate with the same service provisioning policy on each node. A DS domain consists of DS boundary nodes and DS interior nodes. DS boundary nodes interconnect the DS domain to other DS or non-DS-capable domains, while DS interior nodes only connect to other DS interior or boundary nodes

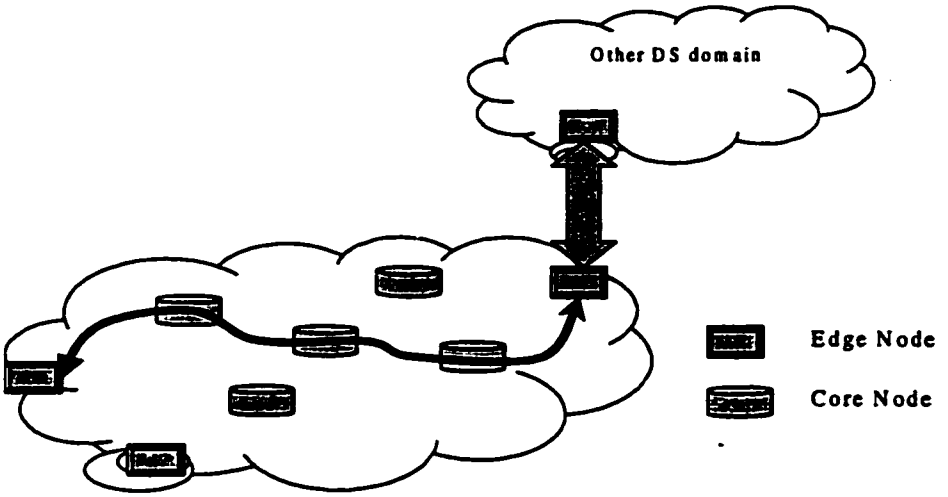

Figure 6 DiffServ Capable Network Domains

within the same DS domain. One of the important features of DS domains is that most of the complexity, related to the support of differentiated services, is located in boundary nodes while interior nodes are kept simple. This means that interior nodes offer services only for aggregated traffic rather than on a per-flow basis. It is the boundary nodes' responsibility to classify the packets into several behavior aggregates, meter the traffic against profiles, mark/remark packets, and shape or drop the packets, etc. Interior nodes just forward the aggregated traffic according to the codepoints (set by boundary nodes or flow sources) in the packets' headers. **Figure 6** shows interconnection between DS domains and within a DS domain.

The packet classification policy identifies the subset of traffic that may receive a differentiated service by being conditioned and/or mapped to one or more behavior aggregates (i.e. codepoint marking) within the DS domain. Traffic conditioning consists of metering, shaping, policing and marking, ensuring that the traffic entering the DS

domain conforms to the rules specified in the profile in accordance with the domain's service provisioning policy. **Figure 7** shows the logical view of a packet classifier and traffic conditioner at a DS boundary node.

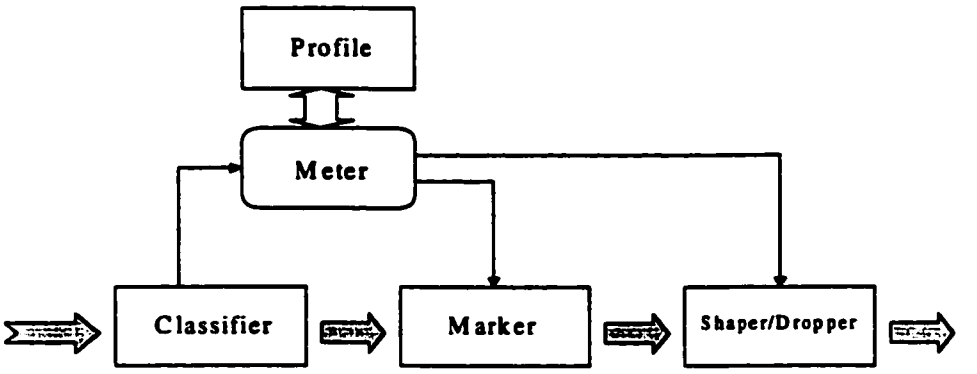

Figure 7 Edge Node of DiffServ Network

In a boundary node of a DS domain, a flow of packets is classified into several classes by a classifier. The classifier classifies the traffic by reading the codepoint in the packet's header if this packet has already been pre-marked; otherwise, it reads other fields in the packet's header, such as source/destination IP address and protocol identifier. The meter at the boundary nodes then measures the traffic stream against a traffic profile, which specifies the temporal properties of this stream selected by a classifier. A profile based on a token bucket may look like "codepoint = X, use token bucket u, T". This profile means that all the packets with codepoint X will be measured against a token bucket meter with rate u and burst size T. Out of profile packets are those packets that arrive when insufficient tokens are available in the bucket. These

out of profile packets may be marked with a lower priority by a marker, shaped by a shaper, or dropped by a dropper at the boundary nodes. Packet markers set the DS field of a packet to a particular codepoint, adding the marked packet to a particular DS behavior aggregate. Shapers will delay some or all of the packets in a traffic stream in order to bring the stream into compliance with a traffic profile. Droppers discard some or all of the packets in a traffic stream in order to bring the stream into compliance with a traffic profile.

There are three token-bucket based metering algorithms proposed by IETF up to date, which are A Single Rate Three Color Marker[15], A Two Rate Three Color Marker[16] and A Rate Adaptive Shaper for Differentiated Services[29]. The first two are actually token bucket based schemes and last one is the combination of a shaper with the token bucket based schemes.

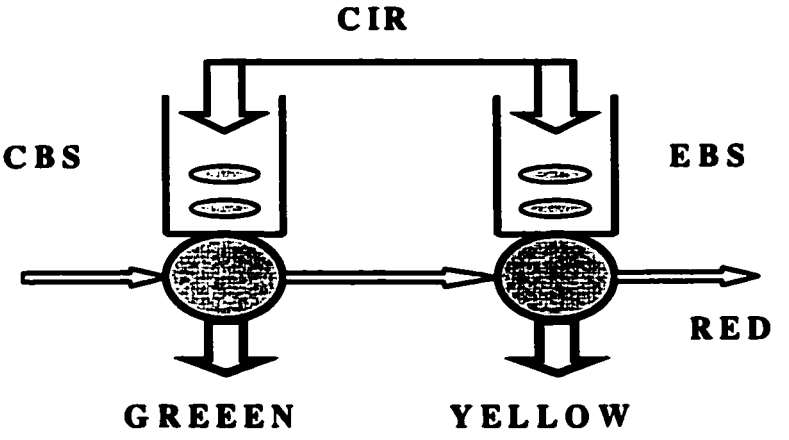

Figure 8 SRTCM

The Three Color Marker algorithms (Single Rate Three Color Marker - SRTCM

in **Figure 8** and Two Rate Three Color Marker - TRTCM in **Figure 9**) are a kind of token bucket algorithm. They meter an IP packet stream and mark its packets green, yellow, or red. Marking is based on a Committed Information Rate (CIR) in SRTCM, or CIR and the Peak Information Rate (PIR) in TRTCM, and two associated burst sizes, Committed Burst Size (CBS) and Excess Burst Size (EBS). The rates are measured in bytes per second for IP packets. The CBS and EBS are measured in bytes. A packet will be marked as green if it does not exceed the CBS, yellow if it does exceed the CBS, but not the EBS, and red otherwise.

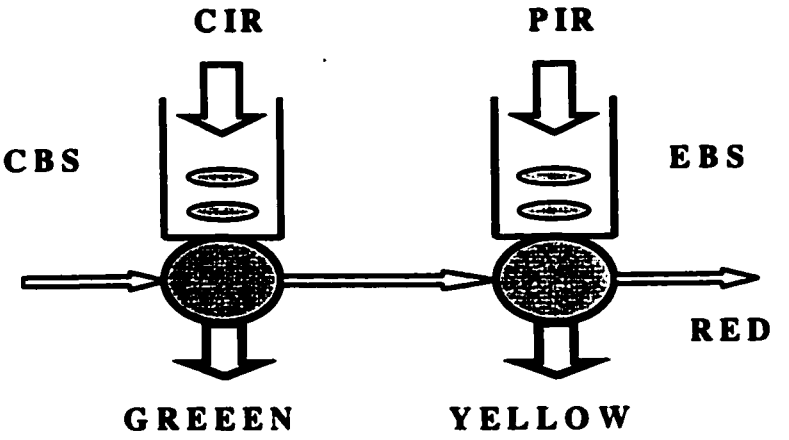

Figure 9 TRTCM

The rate adaptive shaper (RAS) is a scheme that shapes the arriving traffic before it is metered in order to increase the percentage of in-profile packets and ultimately the overall goodput of the users attached to such a shaper. RAS is typically used as shown in **figure 10** where the meter and the marker are the Three Color Makers proposed in [15] and [16]. From the previous introduction, we may see it is actually a combination

of a Leaky Bucket and Token Buckets.

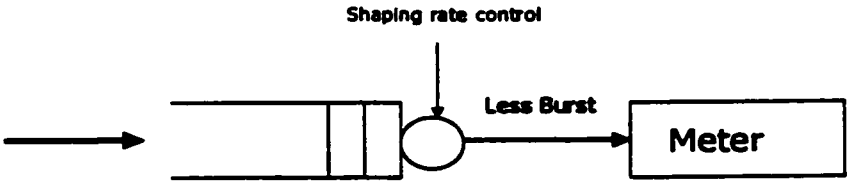

Figure 10 Rate Adaptive Shaper

The main objective of the shaper is to produce at its output a traffic that is less bursty than the input traffic, but the shaper avoids discarding packets in contrast with classical token bucket based shapers. The shaper itself consists of a tail-drop FIFO queue that is emptied at a variable rate. The shaping rate, i.e. the rate at which the queue is emptied, is a function of the occupancy of the FIFO queue. If the queue occupancy increases, the shaping rate will also increase in order to prevent loss and too large delays through the shaper. The shaping rate is also a function of the average rate of the incoming traffic and as well as if there are tokens in the token bucket.

CHAPTER 3

Dynamic Models of TB Based Access Control

This chapter presents all models used in this thesis. First, we present a general model of traffic as the input of TB based access control. And then we present a new dynamic model of a single TB that exercises traffic policing in the edge of the network. It is easy to demonstrate the usefulness of this model alone in various applications. Based on the models of traffic and TB, we construct a complete dynamic model of access control in the edge of the network where there is a multiplicity of traffic governed by a TB model of policing function (one for each source) accessing a multiplexer in an access node.

3.1 A General Model of Traffic

In multi-service integrated networks, a scenario of future networks, we can classify all applications into three kinds of network services: data, voice and video. The traffic behavior of the three application categories are very different. Generally, we can classify them into two, CBR traffic and VBR traffic. The characteristic of CBR traffic is that the source sends it at a fixed rate so that it can be simply described by this fixed rate. Thus, CBR traffic can be allocated by fixed bandwidth corresponding to the rate for a congestion-free transmission. On the other hand, the characteristics of VBR traffic is that the source sends it at a various rate and sometimes it could be very bursty, i.e., the average rate of the traffic is a small fraction of its peak rate. The fixed bandwidth allocation for VBR traffic is not applicable because peak rate allocation would result in underutilization of the network resources, while average rate allocation would result in losses. Feedback control, introduced in the next chapter is one of the common

techniques to allocate network resources for VBR traffic to balance QoS and network utilization.

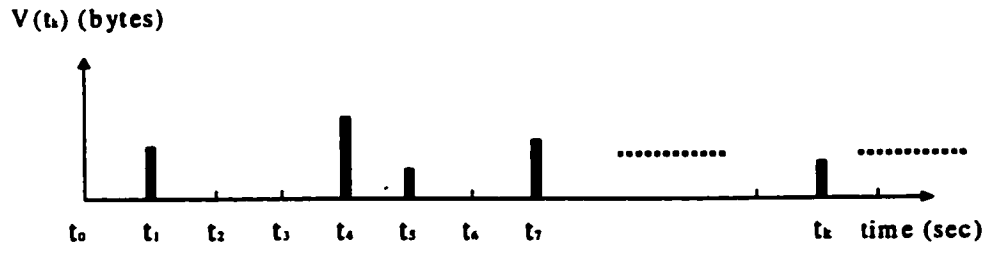

Figure 11 Traffic model

Now, let us consider a general traffic model $\{V(t_k), k = 1, 2, \cdots\}$ as shown in **Figure 11**, which could be thought as any one of data, voice, or video traffic. The traffic, denoted by $\{V(t_k), k = 1, 2, \cdots\}$, is a random process observed at times t_k for k=1,2,...,K. $V(t_k)$ represents the size of the arriving packet at time t_k. The time intervals $[t_{k-1}, t_k)$ are the one of time slots of observation periods and we can assume they are equal and small enough during which only one packet may arrive at the observation point.

3.2 A Dynamic Model of Token Bucket as Policing Function

The TB that executes a policing function is a TB without a buffer as shown in **Figure 12**. When a packet arrives at the TB, if there are enough tokens in the bucket, the

packet is considered conforming and a corresponding number of tokens is withdrawn from the token bucket. In case where there are not enough tokens in the bucket, it is considered nonconforming and the token bucket stays the same. Let us denote the arriving tokens by $u(t_k)$ during the time interval $[t_{k-1}, t_k)$. If the bucket is already full, newly arriving tokens are rejected.

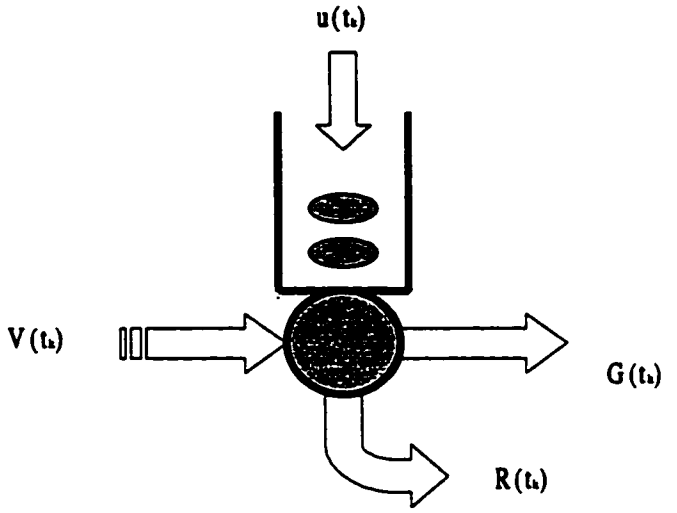

Figure 12 Token bucket model

Let us define some symbols throughout the thesis. The unit of the symbols is measured in bytes or bytes per second when applicable:

1. $\{a \wedge b\}$= Min{ a, b};$\{a \vee b\}$= Max{ a, b};
2. I is an indicator function defined by:

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

3. T: physical size of TB;

The state of the token bucket at time t_k is given by the number of tokens in the bucket and let us denote this by $\rho(t_k)$. The evolution of token population with time is governed by the following difference equation which is based on a similar mathematical approach used in previous work on dynamic routing[30-31]:

$$\rho(t_k) = \rho(t_{k-1}) + \{u(t_k) \wedge [T - \rho(t_{k-1})]\} \\ - V(t_k) I\Big\{V(t_k) \leq \rho(t_{k-1}) + \Big[u(t_k) \wedge [T - \rho(t_{k-1})]\Big]\Big\}. \quad (1)$$

Fundamentally, this is a balance equation, where the first term on the right hand side denotes the number of tokens present at the previous time instance t_{k-1}, the second term represents the number of tokens received, more precisely, accepted by the bucket during the time period $[t_{k-1}, t_k)$, and finally the last term represents the number of tokens consumed at time t_k. The balance is the current token population. The quantity $u(t_k)$ represents the number of tokens transmitted to the TB. This is treated as a control variable determined by the network management based on the status of the network. This will be further discussed when defining the model of edge node and feedback control.

From the state equation of TB, we can further specify the conforming traffic, denoted by $G(t_k)$ at time t_k, :

$$G(t_k) = V(t_k) I\Big\{V(t_k) \leq \rho(t_{k-1}) + \Big[u(t_k) \wedge [T - \rho(t_{k-1})]\Big]\Big\} \quad (2)$$

where the item $\Big[u(t_k) \wedge [T - \rho(t_k)]\Big]$represents the number of tokens accepted by the TB during the time interval $[t_{k-1}, t_k)$ and the item in $\{...\}$ represents the condition that the size of the arriving packet is not larger than the number of tokens in the TB at time t_k.

The nonconforming traffic $R(t_k)$ is then given by:

$$R(t_k) = V(t_k) - G(t_k). \tag{3}$$

3.3 Model of Edge Node

Based on the above model, let us consider a more complex model of an edge node as shown in **Figure 13.** There are n sources accessing the same multiplexer in the edge node and each of them is governed by a TB policing function. The multiplexer has a finite buffer size Q and service rate (or link capacity) C. The TB-s in the edge of the network shape the traffic by dropping the nonconforming packets. There is a controller in the access node dynamically controlling the access rate of each traffic by adjusting the traffic sources and the token supply at each TB to achieve maximum utilization of the multiplexer under the constraint that the QoS of each accessing traffic is satisfied.

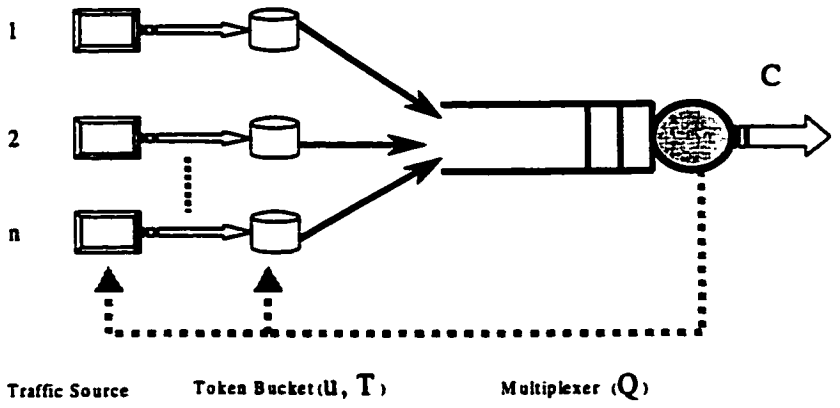

Figure 13 System model

The state of the system (edge node) is now described by the status of all the TB-s and the multiplexer. The status of the token buckets as discussed before is now given by a vector valued function $\rho \equiv (\rho_1, \rho_2 \cdots, \rho_n)'$ of dimension n as follows:

$$\begin{aligned}
\rho_1(t_k) &= \rho_1(t_{k-1}) + \{u_1(t_k) \wedge [T_1 - \rho_1(t_{k-1})]\} \\
&\quad - V_1(t_k) I\Big\{V_1(t_k) \leq \rho_1(t_{k-1}) + \Big[u_1(t_k) \wedge [T_1 - \rho_1(t_{k-1})]\Big]\Big\} \\
\\
\rho_2(t_k) &= \rho_2(t_{k-1}) + \{u_2(t_k) \wedge [T_2 - \rho_2(t_{k-1})]\} \\
&\quad - V_2(t_k) I\Big\{V_2(t_k) \leq \rho_2(t_{k-1}) + \Big[u_2(t_k) \wedge [T_2 - \rho_2(t_{k-1})]\Big]\Big\} \\
\\
&\quad \bullet\bullet\bullet \\
\\
\rho_n(t_k) &= \rho_n(t_{k-1}) + \{u_n(t_k) \wedge [T_n - \rho_n(t_{k-1})]\} \\
&\quad - V_n(t_k) I\Big\{V_n(t_k) \leq \rho_n(t_{k-1}) + \Big[u_n(t_k) \wedge [T_n - \rho_n(t_{k-1})]\Big]\Big\}.
\end{aligned} \tag{4}$$

This is a system of difference equations describing the dynamics of token population in each of the TB-s which are similar to equation (1). In these equations there are also expressions for the conforming traffic. Again the conforming traffic is also given by a vector valued function of dimension n, denoted by $G \equiv (G_1, G_2, \cdots, G_n)'$. These are given by:

$$\begin{aligned}
G_1(t_k) &= V_1(t_k) I\Big\{V_1(t_k) \leq \rho_1(t_{k-1}) + \Big[u_1(t_k) \wedge [T_1 - \rho_1(t_{k-1})]\Big]\Big\} \\
G_2(t_k) &= V_2(t_k) I\Big\{V_2(t_k) \leq \rho_2(t_{k-1}) + \Big[u_2(t_k) \wedge [T_2 - \rho_2(t_{k-1})]\Big]\Big\} \\
&\quad \bullet\bullet\bullet
\end{aligned} \tag{5}$$

$$G_n(t_k) = V_n(t_k) I\Big\{V_n(t_k) \leq \rho_n(t_{k-1}) + \Big[u_n(t_k) \wedge [T_n - \rho_n(t_{k-1})]\Big]\Big\}.$$

Thus the vector of nonconforming traffic is given by:

$$\begin{aligned} R_1(t_k) &= V_1(t_k) - G_1(t_k) \\ R_2(t_k) &= V_2(t_k) - G_2(t_k) \\ &\bullet\bullet\bullet \\ R_n(t_k) &= V_n(t_k) - G_n(t_k). \end{aligned} \tag{6}$$

Again these equations are similar to the equations (2) and (3). These vectors will be used later in formulating the objective function.

In order to complete the dynamics of the whole system, we must formulate the status of the multiplexer. Let $q(t)$, at time t, denote the size of the queue at the multiplexer waiting for service, that is, for onward transmission into the network. This variable is taken as the state of the multiplexer. It is governed by the following difference equation:

$$\begin{aligned} q(t_k) = \Big\{[q(t_{k-1}) - C\tau] \vee 0\Big\} + \\ \Big\{\Big[\sum_{i=1}^{n} G_i(t_k)\Big] \wedge \Big[Q - \Big[[q(t_{k-1}) - C\tau] \vee 0\Big]\Big]\Big\}, \end{aligned} \tag{7}$$

where τ denotes the length of the time intervals $[t_{k-1}, t_k), k = 1, 2, \cdots,$ K.

The first term on the right hand side of the above expression describes the leftover traffic at time t_k after the multiplexer has injected traffic into the network during the period $[t_{k-1}, t_k)$. The second term represents the traffic accepted from all the sources during the same period. It is clear from this expression that the traffic accepted by the multiplexer is given by the smaller of the available (empty) space in the buffer and the sum of all the conforming traffic. Thus the complete system dynamics is given by the set of $n + 1$ difference equations (4),(7). Even though the system of equations (4)

appear to be mathematically uncoupled, they are actually coupled through equation (7) which contains all the conforming traffic from all the TB-s. Further, coupling is also present through the controls which are dependent on the status of the multiplexer and other network variables such as the incoming traffic and the status of the TB-s. This will be clear later as we define the (feedback) control strategies.

It is expected that the multiplexer may not always be able to serve all the conforming traffic because of the limitation of bandwidth and buffer size. Thus the traffic lost at (or rejected by) the multiplexer is given by:

$$L(t_k) = \Big[\sum_{i=1}^{n} G_i(t_k)\Big] - \Big\{\Big[\sum_{i=1}^{n} G_i(t_k)\Big] \wedge \Big[Q - [[q(t_{k-1}) - C\tau] \vee 0]\Big]\Big\}. \tag{8}$$

From this equation and equation(6), we may define network utilization as:

$$\eta = \frac{\sum_{i=1}^{n}\sum_{k=0}^{K} V_i(t_k) - [\sum_{k=0}^{K} L(t_k) + \sum_{i=1}^{n}\sum_{k=0}^{K} R_i(t_k)]}{C(t_K - t_0)}. \tag{9}$$

This expression gives the level of network utilization under different (token) control policies. Another measure of network performance is the throughput which is intimately related to the network utilization as defined above.

$$\begin{aligned}\Upsilon = C\eta &= \frac{\sum_{i=1}^{n}\sum_{k=0}^{K} V_i(t_k) - [\sum_{k=0}^{K} L(t_k) + \sum_{i=1}^{n}\sum_{k=0}^{K} R_i(t_k)]}{(t_K - t_0)} \\ &= (1/(t_K - t_0)) \sum_{k=0}^{K} \Upsilon(t_k) \\ &= (1/(t_K - t_0)) \sum_{k=0}^{K} \Big\{\Big[\sum_{i=1}^{n} G_i(t_k)\Big] \wedge \Big[Q - [[q(t_{k-1}) - C\tau] \vee 0]\Big]\Big\}\end{aligned} \tag{10}$$

where $\Upsilon(t_k)$ denotes the throughput during the period $(t_{k-1}, t_k]$.

CHAPTER 4

Objective Functions and Control Strategy

This chapter introduces the objective functions. They are defined taking into account the losses at the multiplexer, losses at the TB-s and a measure of service delay at the multiplexer. We give a feedback control making a tradeoff between QoS and network utilization in order to minimize the objective functions.

4.1 General Objective Function

By considering only QoS (loss and delay), a general objective function designed for a network service provider is given by:

$$J(u) = \sum_{k=0}^{K} \alpha(t_k) L(t_k) + \sum_{i=1}^{n} \sum_{k=0}^{K} \beta_i(t_k) R_i(t_k) + \sum_{k=0}^{K} \gamma(t_k) q(t_k) \tag{11}$$

where $u = (u_1, u_2, \cdots, u_n)'$ is the control vector that appears in the dynamic model of the TB-s. The parameters $\{\alpha, \beta_i, \gamma, i = 1, 2, \cdots, n\}$ are nonnegative functions of time assigning relative weight. The first term in the function penalizes losses in the multiplexer, the second term penalizes lost traffic at the TB's and the third term is an approximate measure of waiting time or delay at the multiplexer before being served. The problem is how to find a control strategy to minimize the objective function. There are three options regarding control strategies: open loop, feedback or hybrid. In the case of feedback controls, one may visualize the feedback as a function of the state of the system and the user demand (incoming traffic). Therefore the control vector may be given by:

$$u(t_k) = \Gamma(V(t_k), q(t_{k-1}), \rho(t_{k-1})) \tag{12}$$

for $k \in \{1, 2, 3 \cdots, K\}$, where Γ is a suitable feedback control law.

4.2 Optimal Control

To minimize the objective function (11), we must find an optimal control law $u(t_k)$ under some constraints from the system. In general, the constraint set for the control, denoted by U, is by itself a function of the state variables $\{\rho, q\}$ and the input traffic vector V. Since the controls are n-tuples of nonnegative integers, it is clear that at any time, say, t_k, $U \equiv U(V(t_k), \rho(t_{k-1}), q(t_{k-1}))$ is a subset of N_0^n. Thus the optimal control problem becomes to choose at time t_k, for every $k \in D \equiv \{0, 1, 2, \cdots, K\}$, a control $u(t_k) \in U(V(t_k), \rho(t_{k-1}), q(t_{k-1}))$ that minimizes the function (11) subject to the dynamic constraints (4) and (7). This is a problem that can be solved by the use of dynamic programming principle and genetic algorithm. In this thesis, a simple and practical proportional feedback control law is proposed which is much easier to implement compared to complex control laws that may result from rigorous optimization.

4.3 Feedback Control Law

In this part, a simple feedback control law, proportional feedback control, is proposed, which is easy to implement. Towards this goal, let us consider some special versions of the objective function(11). Assume that all the users (sources) can manage traffic at different QoS levels and that minimum QoS is guaranteed by admission control. Furthermore, let us set $\alpha(t_k)$ in equation (11) equal to infinity in order to eliminate losses in the multiplexer which may cause unpredictable QoS for each user. This means

that losses at the multiplexer must be forced to zero. We may achieve this by imposing appropriate constraints on the controls as presented below.

4.3.1 Objective Function J_1

Let us first assume that the delay in the multiplexer is negligible. The objective function is then given by

$$J_1(u) = \sum_{k=0}^{K} \alpha(t_k) L(t_k) + \sum_{i=1}^{n} \sum_{k=0}^{K} \beta_i(t_k) R_i(t_k). \tag{13}$$

This is the cost associated with traffic losses at the token buckets. Now let consider two cases: proportion and proportion with priority.

Case A (Proportion)

In order to reduce the losses in the multiplexer to zero, let us define

$$\Theta_i(t_k) \equiv \frac{V_i(t_k)\{Q - [(q(t_{k-1}) - C\tau) \vee 0]\}}{\sum_{i=1}^{n} V_i(t_k)} \tag{14}$$

as the maximum allowable token permit for the i-th user. This can be further interpreted as the maximum permissible allocation of multiplexer (buffer) space to the i-th user. Note that no priority is given to any of the users. Resources are allocated proportionally to their demands. Weighting this against the actual traffic or demand, we can obtain the true allocation

$$A_i(t_k) = \{\Theta_i(t_k) \wedge V_i(t_k)\} \tag{15}$$

Note that

$$\sum_{i=1}^{n} A_i(t_k) \leq \{Q - [(q(t_{k-1}) - C\tau) \vee 0]\} \tag{16}$$

and therefore the losses in the multiplexer is completely eliminated for all $\{t_k\}$.

Thus we can now define the control policy as

$$u_i(t_k) = \Big\{[A_i(t_k) - \rho_i(t_{k-1})]\Big\} I\{A_i(t_k) \geq \rho_i(t_{k-1})\} \tag{17}$$

The amount of tokens filled into token bucket are determined by the difference between the volume of accepted incoming traffic (or available bandwidth) and existing tokens in the token bucket. If the latter is larger than the former, no tokens need to fill and this is realized by use of the indicator function multiplying the expression within the bracket.

Case B (Proportion With Priority)

In certain circumstances, for example, the traffic have different resource demand or simply some users are willing to pay more for the time and the volume of traffic, it is necessary to assign special priorities to some users. In accordance with this situation, the definition of proportional allocation given by (14) is modified to allocation with priority defined by the following expression,

$$\hat{\Theta}_i(t_k) = \frac{\delta_i(t_k) V_i(t_k)\{Q - [(q(t_{k-1}) - C\tau) \vee 0]\}}{\sum_{i=1}^{n} \delta_i(t_k) V_i(t_k)} \tag{18}$$

where

$$0 \leq \delta_i(t_k) \leq 1, \text{ and } \sum_{i=1}^{n} \delta_i(t_k) = 1, \ \forall\, t_k.$$

Note that the priority index δ may vary with time from user to user. For example, if we wish to provide absolute priority to user j, for certain periods of time, say, P, we may set

$$\delta_j(t_k) = 1, \ \forall\, t_k \in P, P \subset \{t_r, r = 1, 2, 3, \cdots, K\}.$$

Again we can define the true allocation as in (15) by

$$\hat{A}_i(t_k) = \{\hat{\Theta}_i(t_k) \wedge V_i(t_k)\}, \tag{19}$$

and note that, exactly as in (16), once again we have

$$\sum_{i=1}^{n} \hat{A}_i(t_k) \leq \{Q - [(q(t_{k-1}) - C\tau) \vee 0]\}, \tag{20}$$

to ensure zero loss at the multiplexer. In this case the control law given by (17) is slightly modified as presented below,

$$\hat{u}_i(t_k) = \Big\{[\hat{A}_i(t_k) - \rho_i(t_{k-1})]\Big\} I\{\hat{A}_i(t_k) \geq \rho_i(t_{k-1})\} \tag{21}$$

4.3.2 Objective Function $J_2 = J$

Now, let us add the component of the waiting time or equivalently, the delay to objective function J_1. Thus, the objective function is now given by

$$J_2(u) \equiv J = \sum_{k=0}^{K} \alpha(t_k) L(t_k) + \sum_{i=1}^{n} \sum_{k=0}^{K} \beta_i(t_k) R_i(t_k) + \sum_{k=1}^{K} \gamma(t_k) q(t_k). \tag{22}$$

Again, let us consider both the cases, proportion and proportion with priority. The control laws for both cases remain exactly the same as before, i.e. equations (17) and (21). Since having included the delay, we can expect the objective function $J_2 \geq J_1$.

4.4 Dependence of Objective Functions on Network Parameters Q and C

In the preceding section, feedback control laws have been proposed that eliminate multiplexer losses. We can compute the objective functions corresponding to these control laws. Furthermore, we may also fix these control laws in (17) and (21) and compute the objective functions with different network parameters such as the multiplexer size Q and the link capacity C. In particular, we consider J_1 as a function of Q and C and

denote this by $J_1(C, Q)$. Similarly, we consider J_2 also as a function of these parameters and denote this by $J_2(C, Q)$. For a given link capacity C, we may plot $J_{1,C}(Q), J_{2,C}(Q)$ as functions of the multiplexer size Q. Similarly, for a given multiplexer size Q, we may plot $J_{1,Q}(C), J_{2,Q}(C)$ as functions of C. Finally, we may also present 3D plots of these functions for several traffic profiles. These results will be discussed in detail in the numerical computing part. These results will clearly demonstrate the effectiveness of the dynamic models and the proposed control laws.

CHAPTER 5

Application of Feedback Control in MPEG Video Transmission

In this chapter, we apply the feedback control discussed in the last chapter to MPEG video transmission. MPEG video traffic is inherently very bursty over many time scales and claimed difficult for traffic management [32-33]. Applying MPEG video traffic to the models and employing feedback control to the traffic can demonstrate the effectiveness of the models discussed in the last chapters. We implement the models along with the feedback control algorithm in OPNET. We consider a scenario comprised of three traffic sources policed by three TB-s whose conforming outputs are multiplexed into a single finite buffer.

Unit: M bytes/sec

	Peak Rate (Pi)	Average Rate (Mi)	Standard Deviation ($\sum$i)
Trace 1	4.32	0.64	0.65
Trace 2	2.18	0.28	0.14
Trace 3	5.64	0.87	0.64

Table 1 Summary of video traces specification

5.1 Configuration of Numerical Computing

We choose three independent five-minute MPEG video traces from cartoon, movie and sports as the traffic sources. **Table 1** summarizes the specification of the traces. In the table, we denote by $\{P_i, M_i, \Sigma_i\}$ the peak rate, the mean rate, and the standard deviation of the i-th traffic source respectively.

The configuration for the computing being conducted later is as follows:

System configurations:

- $T_i = \Sigma_i, i = 1, 2, 3$;
- $C = \sum_{1 \le i \le 3} M_i = 1.79 Kbytes$;
- Q= 3 $Mbytes$;
- τ= 0.5 seconds (interval of GoP[1]);

State initialization:

- $\rho_i(t_0) = T_i$;
- $q(t_0) = 0$;

Relative weight:

For the weights given to the components in the cost functional J, we choose $\alpha(t_k) = 10$, for all k, $\beta_i(t_k) = 1$ for all $\{i, t_k\}$, and $\gamma(t_k) = 1$ for all k. Thus, the weight on the losses at multiplexer are ten times larger than the other losses. In the following part, we compute the objective functions of J_1 and J_2 corresponding to these weights.

[1]Group of Picture (GoP): MPEG video trace is broken into GoPs and each contains 15 video frames. In NTSC, the video is transmitted at 30 frames per second.

5.2 Results and Analysis

In the first part of the computing, applying different control strategies (open loop control with different parameters and feedback control) over the three video traffics, we compare the costs (objective functions) resulting from each control strategy. In the second part of the computing, by solely using feedback control, we try to see the dependence of objective functions on network parameters (the link speed and buffer size of the multiplexer).

5.2.1 Numerical Computing with Different Control Strategies

In this part, we conduct six different control strategies: the feedback control as given by (17) and five open loop control strategies as described below.

Open Loop Control:

- Case 1: $u_i(t_k) = M_i$;
- Case 2: $u_i(t_k) = 1.01M_i$;
- Case 3: $u_i(t_k) = 1.04M_i$;
- Case 4: $u_i(t_k) = 1.05M_i$;
- Case 5: $u_i(t_k) = P_i$ for $i \in \{1, 2, 3\}$;

Feedback Control:

- Case 6 with control law given by equation (17);

Remark:

1. The system is in a heavy-load situation during the computing. 2. The buffer size of a multiplexer is determined by delay constraints of the traffic.

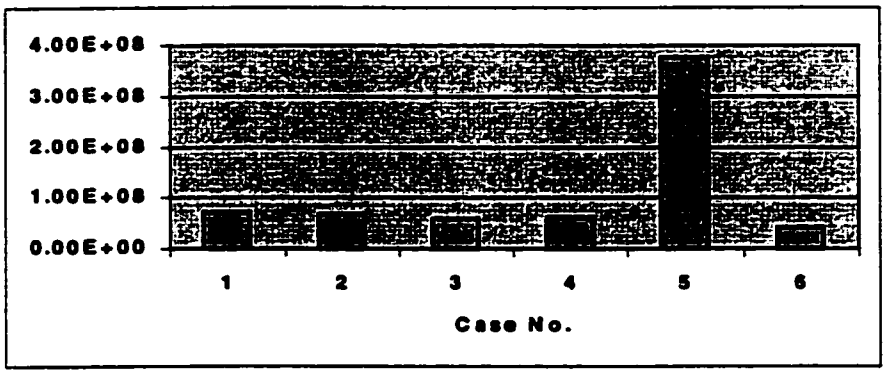

Figure 14 Objective function J_1

Figure 14 gives a summary of the objective function of J_1 for all the six control strategies from case 1 to case 6. Obviously, the numerical computing results show that the case 6 of feedback control offers the minimum cost among all strategies. The case 5 of open loop control with "peak rate" is the worst of all. Open loop controls are either too "aggressive (case 5)" causing losses in the multiplexer or too "conservative(cases 1-4)"degrading QoS by under-utilization of available network resources.

With the same configurations, we compute the full objective function: $J_2 = J$, which has one component of multiplexer delay more than J_1. Obviously, $J_2 \geq J_1$. The computing results summarized in **Figure 15** illustrates a different scenario. The cost of case 6 of feedback control is larger than those of from case 1 to case 4, but still less than that of case 5 of open loop control. This change is because the open loop strategies do not allow traffic sufficiently access to the multiplexer ("too conservative"), thereby cause less losses at multiplexer, and consequently degrade throughput as seen

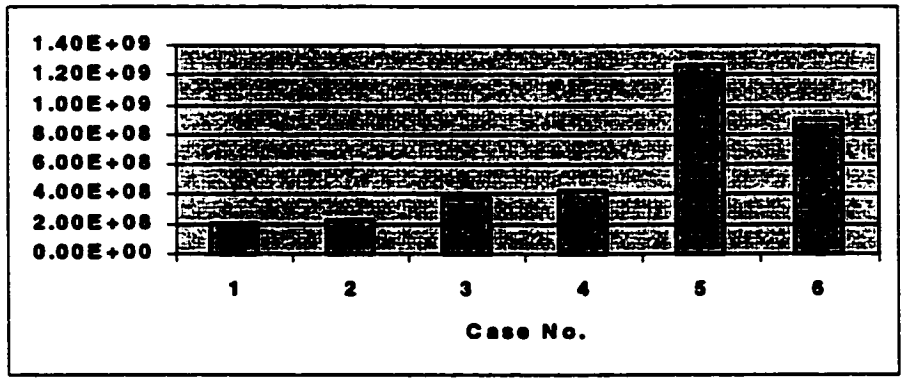

Figure 15 Objective function J_1

in **Figure 16**. Note that the throughput of case 1 to case 4 is lower than those of case 5 and case 6. Comparing the figures with respect to the cases 5 (peak control) and 6 (feedback control), we may figure out that case 6 beats case 5 in terms of cost (**Figure 15**). Furthermore, the throughput (**Figure 16**) of case 6 is very close to that of case 5.

5.2.2 Different Network Parameters

In this part, we study the dependence of the objective functions J_1 and J_2 on the network design parameters such as (output) link capacity C and buffer size Q of the multiplexer by solely applying fixed feedback control strategy given by (17).

Through the intensive computing, we summarize the results as follows. **Figure 17** plots the objective function J_1 as a function of Q ($J_{1,C}(Q)$) corresponding to different link capacities C but constant in each computing; **Figure 18** plots the objective

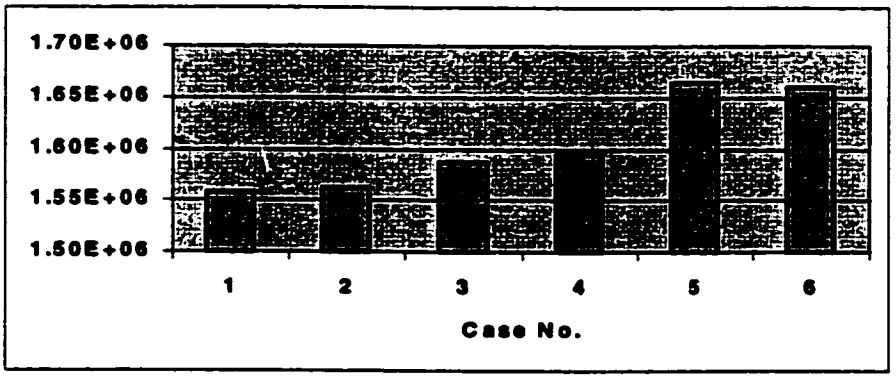

Figure 16 Throughput (bytes/sec)

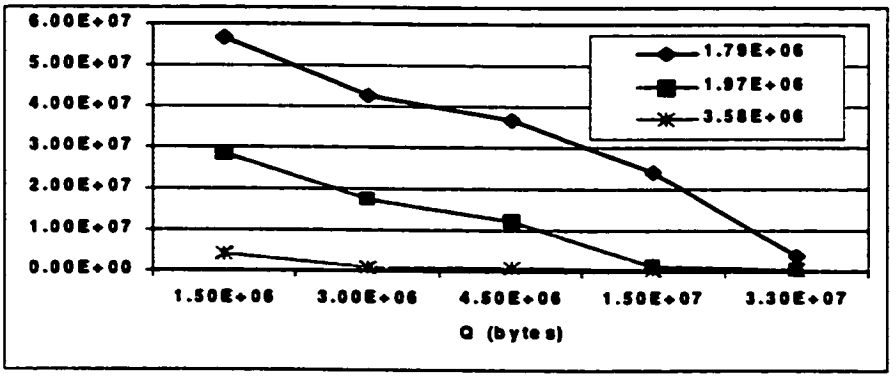

Figure 17 Objective function $J_i(Q)$ with different C-s

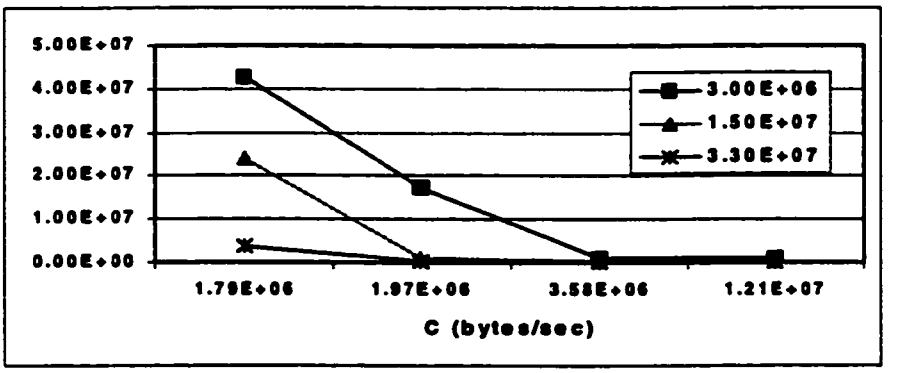

Figure 18 Objective function $J_1(C)$ with different Q-s

function J_1 as a function of C ($J_{1,Q}(C)$) for different values of the multiplexer size Q. As expected, increasing buffer size Q and link capacity C can decrease the cost. From **figure 17**, we may also conclude that

$$C_1 \leq C_2 \leq C_3 \Longrightarrow J_{1,C_1}(Q) \geq J_{1,C_2}(Q) \geq J_{1,C_3}(Q), \ \forall\, Q > 0. \tag{23}$$

Moreover, as Q increases beyond certain value so that the buffer can accommodate the bursts or even the peaks of all the traffics, the cost reaches a plateau, here zero, thereby eliminating all losses and QoS degradation. Likewise, from **figure 18**, we may also conclude that

$$Q_1 \leq Q_2 \leq Q_3 \Longrightarrow J_{1,Q_1}(C) \geq J_{1,Q_2}(C) \geq J_{1,Q_3}(C), \ \forall\, C > 0. \tag{24}$$

. In this case, the similar explanations hold.

Now, let us consider the full objective function $J_2 = J$. **Figure 19** plots the objective function of J_2 as a function of Q for different values of capacity C_i. This

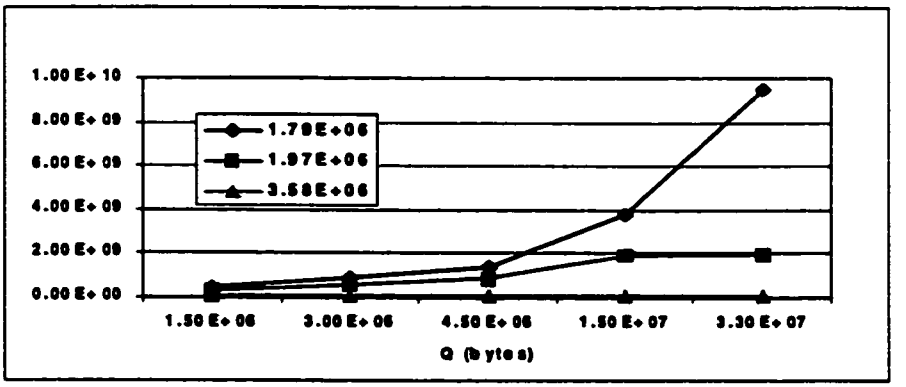

Figure 19 Objective function $J_2(Q)$ with different C-s

picture illustrates a different profile comparing with **figure 17**. However the order of dominance remains the same as given below.

$$C_1 \leq C_2 \leq C_3 \Longrightarrow J_{2,C_1}(Q) \geq J_{2,C_2}(Q) \geq J_{2,C_3}(Q), \ \forall \ Q > 0. \tag{25}$$

A correct explanation of the increasing cost with increasing Q, for any fixed C, is that when the link capacity C is small, increasing multiplexer size Q increases queue size and hence the delay that causes higher cost. Thus for any fixed C, $J_{2,C}(Q)$ is a nondecreasing function of Q.

Again, **Figure 20** illustrates the objective function of J_2 as a function of C for different values of Q. We obtain the fofollowing order relation,

$$Q_1 \leq Q_2 \leq Q_3 \Longrightarrow J_{2,Q_1}(C) \leq J_{2,Q_2}(C) \leq J_{2,Q_3}(C), \ \forall \ Q > 0. \tag{26}$$

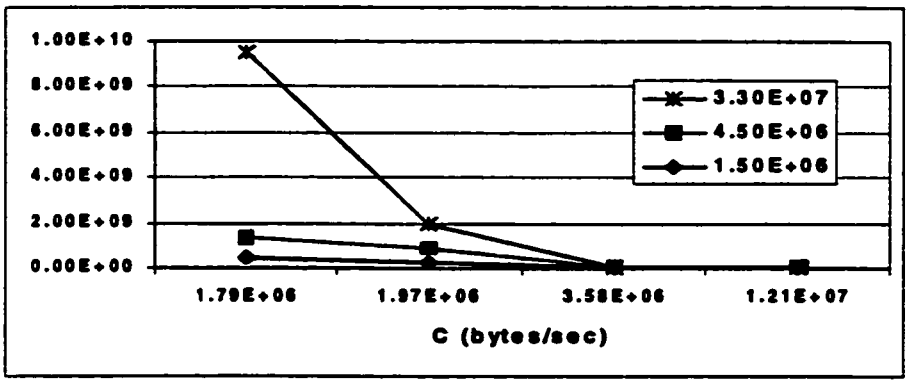

Figure 20 Objective function $J_2(C)$ with different Q-s

The result that $J_{2,Q}(C)$ is a decreasing function of C. follows from the fact that, for any fixed Q, increasing link capacity C increases the service rate, which reduces the delay and thereby the cost.

To present a whole image of dependence of objective functions on network design parameters of Q and C, we plot three-dimensional pictures showed in **Figure 21** and the **Figure 22** i.e. $J_1(C,Q)$ and $J_2(C,Q)$. From the previous analysis, these figures are self-explanatory.

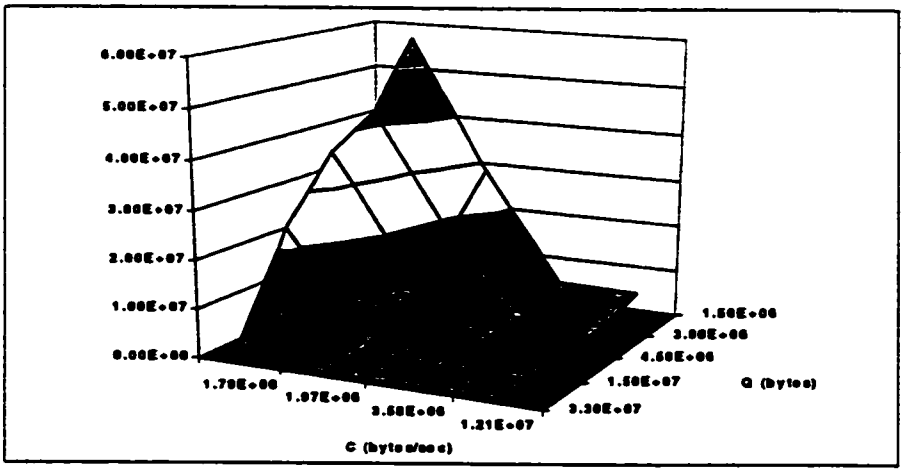

Figure 21 Objective function $J_1(C, Q)$

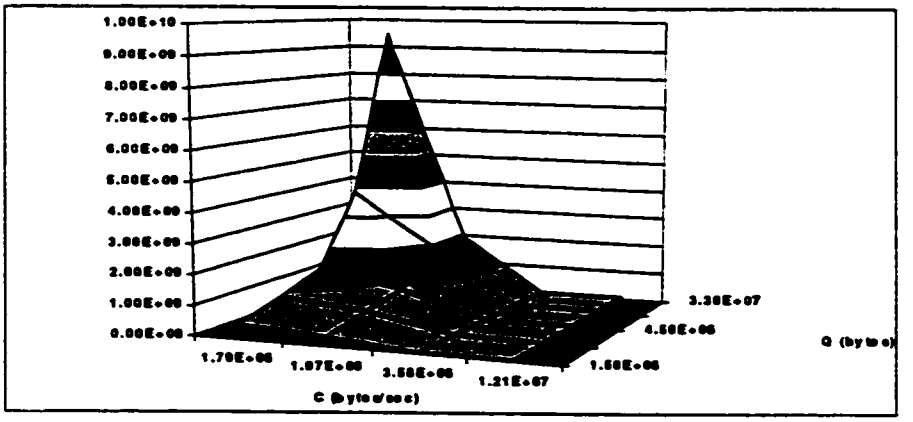

Figure 22 Objective function $J_2(C, Q)$

CHAPTER 6

Conclusion

QoS provision is a crucial issue for the next generation Internet: a multi-service integrated network. Although there is a variety of service models that have been proposed, essentially the traffic control is to be implemented to achieve the goal of guaranteed data delivery. The TB control mechanism is popularly employed in traffic management and new Internet service models.

In the thesis, a dynamic model is proposed that formally defines the TB control mechanism and its applications in computer networks. Using the model, we study the dynamics of TB based access control in the edge of the network. This dynamic model gives a simple infrastructure to the problems of TB control mechanism. The model also gives a guideline to traffic sizing, network dimensioning and trade-off over loss, delay and network utilization etc. The future work based on the model could fall into three areas: first, employ feasible optimization techniques to obtain optimal feedback control strategies; second, extend the model to a complete system of interconnected network and study end-to-end performance; third, use the model to analyze current network architectures and applications.

Reference

[1] Ramayya Krishnan, Haebin Kim and Jee Hae Kwoo, "Project Proposal for Telecommunication Management",www.andrew.cmu.edu/haebin/proposal-telecom.html.

[2] R. Braden, D. Clark, and S. Shenker, "Integrated Services in the Internet Architecture: an Overview", RFC 1633, Internet Engineering Task Force, June 1994.

[3] S.Blake, D. Clark, M. Carlson, E. Davies, Z. Wang, and W. Weiss, "An Architecture for Differentiated Services", RFC 2475, Internet Engineering Task Force, December 1998.

[4] E. Roseo, A. Viswanathao, and R. Callon, "Multi protocol Label Switching Architecture", RFC 3031, Internet Engineering Task Force, January 2001.

[5] S. Shenker, C. Partridge and R. Guerin, "Specification of Guaranteed Quality of Service", RFC 2212, Internet Engineering Task Force, September 1997.

[6] J. Wroclawski, "Specification of the Controlled-Load Network Element Service", RFC 2211, Internet Engineering Task Force, September 1997.

[7] R. Braden, L. Zhang, S. Berson, S. Herzog and S. Jamin, "Resource ReSerVation Protocol (RSVP) - Version 1 Functional Specification", RFC 2205, Internet Engineering Task Force, September 1997.

[8] Juha Heinanen, Fred Baker, Walter Iiss and John Wroclawski, "Assured Forwarding PHB Group", RFC 2597, Internet Engineering Task Force, June 1999.

[9] Van Jacobson, Kathleen Nichols and Kedarnath Poduri, "Expedited Forwarding PHB", RFC 2598, Internet Engineering Task Force, June 1999.

[10] Andersson L., Doolan P., Feldman N., Fredette A. and R. Thomas, "LDP Specification", RFC 3036, Internet Engineering Task Force, January 2001.

[11] Francois Le Faucheur, Liwen Wu,Bruce Davie, " MPLS Support of Differentiated Services", IETF Internet Draft, draft-ietf-mpls-diff-ext-09.txt, April, 2001

[12] F. Tommasi, S. Molendini and A. Tricco, " Integrated Services across MPLS domains using CR-LDP signaling", Internet Draft,draft-tommasi-mpls-intserv-01.txt, May 2001.

[13] Syed, Hamid,"The DS marking Capability Negotiation: A Usage Case for the RSVP CAP Object", Internet Draft,draft-hamid-issll-rsvp-cap-dsmark-00, February, 2001.

[14] S. Shenker, J. Wroclawski, "General Characterization Parameters for Integrated Service Network Elements", RFC 2215, Internet Engineering Task Force, September 1997.

[15] J. Heinanen, R. Guerin, "A Single Rate Three Color Marker", RFC 2697, Internet Engineering Task Force, September 1999.

[16] J. Heinanen, R. Guerin, "A Two Rate Three Color Marker", RFC 2698, Internet Engineering Task Force, September 1999.

[17] O. Aboul-Magd and B. Jamoussi, "A Framework for Service Definition and Interworking Using CR-LDP", Internet Draft, draft-aboulmagd-srvc-def-crldp-00.txt, Internet Engineering Task Force, October 1999.

[18] Milena Butto, Elisa Cavallero, and Alberto Tonietti, "Effectiveness of the Leaky Bucket Policing Mechanism in ATM Networks", IEEE Journal on Selected Areas in Communications, VOL. 9, NO. 3, pp. 335-342., April 1991.

[19] A. W. Berger, "Performance Analysis of a Rate Control Where Tokens and Jobs Queue", IEEE Journal on Selected Areas in Communications, VOL. 9, NO. 2, pp.165-170., February 1991.

[20] Hamid Ahmadi, Roch Guerin and Khosrow Sohraby, "Analysis of Leaky Bucket Access Control Mechanism with Bacth Arrival Process", Globecom 90, pp. 0344-0349., 1990.

[21] S. Keshav, "An Engineering Approach to Computer Networking", Chapter 13, zAddison-Wesley, 1997.

[22] J.W. Roberts, "Performance Evaluation and Design of Multiservice Networks", COST 224 Final Report, Commission of the European Communities, Brussels, 1992.

[23] R. Guerin and L. Gun, " A Unified Approach to Bandwidth Allocation and Access Control in Fast Packet-Switched Networks", Proceedings of IEEE INFOCOM 1992.

[24] R.J. Gibbens and P. J Hunt, "Effective Bandwidth of Multi-Type UAS Channel", Queuing Systems, Vol. 29, No. 9, pp. 17-28., October 1991.

[25] R. L. Cruz, "A Calculus for Network Delay and a Note on Topologies of Interconnection Networks", Ph.D. Thesis, University of Illinois, 1987.

[26] Christopher Lefelhocz, Bryan Lyles, Scott Shenker and Lixia Zhang, " Congestion Control for Best-Effort Service",IEEE Network, Volume 10, Number 1, , pp. 10-19., January/February 1996.

[27] S. Floyd aod V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance," IEEE ACM Transactions on Networking, Vol. 1, No. 4, pp. 397-413.,

August 1993.

[28] K.K. Ramakrishnan, S. Floyd, "A proposal to add Explicit Congestion Notification (ECN) to Ipv6 and to TCP", IETF Internet Draft, November 1997.

[29] O. Bonaventure, S. De Cnodder, "A Rate Adaptive Shaper for Differentiated Services", RFC 2963, Internet Engineering Task Force, October 2000.

[30] C. Wahida, N. U. Ahmed, "Congestion control Using Dynamic Routing and Flow Control", Stochastic Analysis and Applications, Vol. 10 No. 2, pp. 123-142., 1992.

[31] N. U. Ahmed, T.E. Dabbous and Y.E. Lee,"Dynamic Routing for Computer Queuing Networks", INT. J. Systems SCL, VOL. 19, No. 6, pp. 967-977., 1988.

[32] J. Beran, R. Sherman, M. Taqqu, and W. Willinger, "Long-rang Dependence in Variable Bit-Rate Video Traffic", IEEE Transactions on Communications, Vol. 43, pp. 1566-1579., 1995.

[33] Alfio Lombardo, Giacomo Morabito and Fiovanni Schembra, "Traffic specification for MPEG video transmission over the Internet", International Conference on Communications, IEEE Communications Society, New Orleans, June, 2000.

[34] Qun Wang, Xiangan Jiang, "Traffic Control in the Edges of DiffServ Capable Internet", Project Report of Traffic and Switch Theory, December, 2000.

[35] N. U. Ahmed, Qun Wang, L. Orozco-Barbosa, "A System Approach to Token Bucket Algorithm in Computer Networks", submitted to IEEE Transactions on Systems, Man and Cybernetics, 2001.

[36] H. Jonathan Chao, "Design of Leacky Bucket Access Control Schemes in ATM Networks", ICC, Denver, CO., June, 1991.

[37] S. Vamvakos and V. Anantharam, " On the Departure Process of a Leaky Bucket System with Long-Range Dependent Input Traffic", Queueing Systems : Theory and Applications, Vol. 28, Nos. 1-3, pp. 191 -214, May 1998.

[39] M.F. Alam, M. Atiquzzaman, M.A. Karim, " Traffic Shaping for MPEG Video Transmission over the Next Generation Internet", Computer Communications, 23, pp. 1336-1348., 2000.

Appendix: Numerical Code

1. **CODE OF TRAFFIC SOURCE (MPEG VIDEO)**
2. **CODE OF TOKEN BUCKET**
3. **CODE OF FEEDBACK CONTROLLER**

1. CODE OF TRAFFIC SOURCE (MPEG VIDEO)

```
/* Process model C form file: T_pro_VBR_source_sub1.pr.c */
/* Portions of this file copyright 1992, 1996, 1998 by MIL 3, Inc. */



/* This variable carries the header into the object file */
static const char T_pro_VBR_source_sub1_pr_c [] = "MIL_3_Tfile_Hdr_ 60L 30A
opnet 7 3AC6412E 3AC6412E 1 grd0 qwang 0 0 none none 0 0 none 0 0 0 0 0 0";


/* OPNET system definitions */
#include <opnet.h>

#if defined (__cplusplus)
extern "C" {
#endif
FSM_EXT_DECS
#if defined (__cplusplus)
} /* end of 'extern "C"' */
#endif


/* Header Block */

#define FRAME_SEND ((op_intrpt_type()==OPC_INTRPT_SELF)&&(op_intrpt_code()==0))
int data_1[10000],data_1_dis[10000];

/* End of Header Block */


#if !defined (VOSD_NO_FIN)
#undef      BIN
#undef      BOUT
#define     BIN         _fstack_local_info.last_line_passed = __LINE__ -
_block_origin;
#define     BOUT  BIN
#define     BINIT _fstack_local_info.last_line_passed = 0; _block_origin =
__LINE__;
#else
#define     BINIT
#endif /* #if !defined (VOSD_NO_FIN) */



/* State variable definitions */
typedef struct
      {
      /* Internal state tracking for FSM */
      FSM_SYS_STATE
      /* State Variables */
      int                                   frame_counter;
      double                                frame_interval;
      Stathandle                            shandle;
      int                                   payload_len;
      int                                   id;
      int                                   gop;
```

```
/* State variable definitions */
typedef struct
	{
	/* Internal state tracking for FSM */
	FSM_SYS_STATE
	/* State Variables */
	int                                     frame_counter;
	double                                  frame_interval;
	Stathandle                              shandle;
	int                                     payload_len;
	int                                     id;
	int                                     gop;
	double                                  total_gop;
	Stathandle                              yshandle;
	} T_pro_VBR_source_sub1_state;

#define pr_state_ptr                        ((T_pro_VBR_source_sub1_state*)
SimI_Mod_State_Ptr)
#define frame_counter                       pr_state_ptr->frame_counter
#define frame_interval                      pr_state_ptr->frame_interval
#define shandle                             pr_state_ptr->shandle
#define payload_len                         pr_state_ptr->payload_len
#define id                                  pr_state_ptr->id
#define gop                                 pr_state_ptr->gop
#define total_gop                           pr_state_ptr->total_gop
#define yshandle                            pr_state_ptr->yshandle

/* This macro definition will define a local variable called      */
/* "op_sv_ptr" in each function containing a FIN statement. */
/* This variable points to the state variable data structure,     */
/* and can be used from a C debugger to display their values.     */
#undef FIN_PREAMBLE
#define FIN_PREAMBLE    T_pro_VBR_source_sub1_state *op_sv_ptr = pr_state_ptr;


/* Function Block */

enum { _block_origin = __LINE__ };


/* End of Function Block */

#if defined (__cplusplus)
extern "C" {
#endif
	void T_pro_VBR_source_sub1 (void);
	Compcode T_pro_VBR_source_sub1_init (void **);
	void T_pro_VBR_source_sub1_diag (void);
	void T_pro_VBR_source_sub1_terminate (void);
	void T_pro_VBR_source_sub1_svar (void *, const char *, char **);
#if defined (__cplusplus)
} /* end of 'extern "C"' */
#endif
```

```
/* Process model interrupt handling procedure */


void
T_pro_VBR_source_sub1 (void)
      {
      int _block_origin = 0;
      FIN (T_pro_VBR_source_sub1 ());
      if (1)
            {
            packet* pkt_ptr;
            int pkt_len, pkt_num;
            double j;
            List* l_ptr;
            int l_size;
            int i;
            double mean;


            FSM_ENTER (T_pro_VBR_source_sub1)

            FSM_BLOCK_SWITCH
                  {
                  /*---------------------------------------------------------*/
                  /** state (init) enter executives **/
                  FSM_STATE_ENTER_FORCED (0, state0_enter_exec, "init",
"T_pro_VBR_source_sub1 () [init enter execs]")
                        {
                        /* initial */
                        gop=0;
                        total_gop=0;
                        frame_interval=0.033; /* NTSC */
                        frame_counter=1;
                        mean=0;
                        payload_len=1500; /*bytes*/
                        shandle=op_stat_reg("VBR Video Rate (Bytes/Sec)",
OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
                        yshandle=op_stat_reg("GOP Rate (Bytes/Sec)",
OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);

                        /*load video source file"simpsons" to List*/
                        l_ptr=op_prg_gdf_read("simpsons");
                        l_size=op_prg_list_size(l_ptr);

                        /* 9000 is the 5min video */
                        if(l_size>9000)
                          l_size=9000;

                        for (i=0;i<l_size;++i)
                        {
                         data_l[i]=1;
                        }

                        for (i=0;i<l_size;++i)
                        {
                         /* read list to array data */
```

```
                        data_1[i]=(int) atoi(op_prg_list_access(l_ptr, i));
                       }

                       /*** neating ***/
                       for (i=0;i<l_size;++i)
                       {
                        data_1[i]=(data_1[i]/1500)*1500;
                        mean=data_1[i]+mean;
                        data_1_dis[i]=data_1[i];
                       }

                       mean=(mean/l_size)/frame_interval;
                       printf("mean of frame rate=%f (bytes per sec)\n",mean);

                       /* free list */
                       op_prg_list_free(l_ptr);
                       op_prg_mem_free(l_ptr);


                       /* self-interupt for sending packets */
                       op_intrpt_schedule_self(op_sim_time()+frame_interval,0);

                       }


                  /** state (init) exit executives **/
                  FSM_STATE_EXIT_FORCED (0, state0_exit_exec, "init",
"T_pro_VBR_source_sub1 () [init exit execs]")
                       {
                       }


                  /** state (init) transition processing **/
                  FSM_TRANSIT_FORCE (1, state1_enter_exec, ;)
                       /*---------------------------------------------------------
---*/



                  /** state (idle) enter executives **/
                  FSM_STATE_ENTER_UNFORCED (1, state1_enter_exec, "idle",
"T_pro_VBR_source_sub1 () [idle enter execs]")
                       {

                       }


                  /** blocking after enter executives of unforced state. **/
                  FSM_EXIT (3,T_pro_VBR_source_sub1)


                  /** state (idle) exit executives **/
                  FSM_STATE_EXIT_UNFORCED (1, state1_exit_exec, "idle",
"T_pro_VBR_source_sub1 () [idle exit execs]")
                       {
                       }
```

```
                    /** state (idle) transition processing **/
                    FSM_TRANSIT_ONLY ((FRAME_SEND), 2, state2_enter_exec, ;,
"idle")
                         /*---------------------------------------------------
---*/



                    /** state (frame_send) enter executives **/
                    FSM_STATE_ENTER_FORCED (2, state2_enter_exec, "frame_send",
"T_pro_VBR_source_sub1 () [frame_send enter execs]")
                         {
                         /*** interupting for sending each frame in a constant
interval ***/
                         op_intrpt_schedule_self(op_sim_time()+frame_interval,0);

                         /*** fragment ***/
                         pkt_len=data_1[frame_counter]*1; /* getting each frame
from array */
                         pkt_num=pkt_len/payload_len; /* fragment to packets */

                         for (i=1; i<=pkt_num; i++) /* creating pakcets */
                         {
                           pkt_ptr=op_pk_create_fmt("Thesis2001_packet");
                           op_pk_nfd_set (pkt_ptr, "frame_no", frame_counter);
                           id=op_pro_id (op_pro_self ());
                           /*printf("id= %d\n",id);*/
                           op_pk_nfd_set (pkt_ptr,"source_address",id);
                           /*** sent pkg ***/
                           op_pk_send_forced(pkt_ptr,0);
                         }

                         /*****GOP********/
                         gop=gop+1;
                         total_gop=total_gop+pkt_len;

                         if (gop==15)
                         {
                         op_stat_write(yshandle,total_gop/(frame_interval*15));
                         gop=0;
                         total_gop=0;
                         }
                         /************************/

                         op_stat_write(shandle,pkt_len/frame_interval);
                         frame_counter=frame_counter+1;


                         }


                    /** state (frame_send) exit executives **/
                    FSM_STATE_EXIT_FORCED (2, state2_exit_exec, "frame_send",
"T_pro_VBR_source_sub1 () [frame_send exit execs]")
                         {
```

```
                        }


                    /** state (frame_send) transition processing **/
                    FSM_TRANSIT_FORCE (1, state1_enter_exec, ;)
                        /*---------------------------------------------------------
---*/



                    }


              FSM_EXIT (0,T_pro_VBR_source_sub1)
              }
        }


Compcode
T_pro_VBR_source_sub1_init (void ** gen_state_pptr)
        {
        int _block_origin = 0;
        static VosT_Cm_Obtype   obtype = OPC_NIL;

        extern void                     Vos_Vnop (void);

        FIN (T_pro_VBR_source_sub1_init (gen_state_pptr))

        if (obtype == OPC_NIL)
              {
              /* Initialize memory management */
              if (Vos_Catmem_Register ("proc state vars (T_pro_VBR_source_sub1)",
                  sizeof (T_pro_VBR_source_sub1_state), Vos_Vnop, &obtype) ==
VOSC_FAILURE)
                    {
                    FRET (OPC_COMPCODE_FAILURE)
                    }
              }

        *gen_state_pptr = Vos_Catmem_Alloc (obtype, 1);
        if (*gen_state_pptr == OPC_NIL)
              {
              FRET (OPC_COMPCODE_FAILURE)
              }
        else
              {
              /* Initialize FSM handling */
              ((T_pro_VBR_source_sub1_state *)(*gen_state_pptr))->current_block =
0;

              FRET (OPC_COMPCODE_SUCCESS)
              }
        }



void
```

```
T_pro_VBR_source_sub1_diag (void)
	{
	int _block_origin = __LINE__;

	FIN (T_pro_VBR_source_sub1_diag ())

	if (1)
		{
		packet* pkt_ptr;
		int pkt_len, pkt_num;
		double j;
		List* l_ptr;
		int l_size;
		int i;
		double mean;

		/* Diagnostic Block */


		BINIT


		/* End of Diagnostic Block */

		}

	FOUT;
	}



void
T_pro_VBR_source_sub1_terminate (void)
	{
	int _block_origin = __LINE__;

	FIN (T_pro_VBR_source_sub1_terminate (void))

	if (1)
		{
		packet* pkt_ptr;
		int pkt_len, pkt_num;
		double j;
		List* l_ptr;
		int l_size;
		int i;
		double mean;

		/* Termination Block */


		BINIT


		/* End of Termination Block */
```

```
        }
    Vos_Catmem_Dealloc (pr_state_ptr);

    FOUT;
    }


/* Undefine shortcuts to state variables to avoid */
/* syntax error in direct access to fields of */
/* local variable prs_ptr in T_pro_VBR_source_sub1_svar function. */
#undef frame_counter
#undef frame_interval
#undef shandle
#undef payload_len
#undef id
#undef gop
#undef total_gop
#undef yshandle



void
T_pro_VBR_source_sub1_svar (void * gen_ptr, const char * var_name, char **
var_p_ptr)
    {
    T_pro_VBR_source_sub1_state         *prs_ptr;

    FIN (T_pro_VBR_source_sub1_svar (gen_ptr, var_name, var_p_ptr))

    *var_p_ptr = (char *)VOS_NIL;
    prs_ptr = (T_pro_VBR_source_sub1_state *)gen_ptr;

    if (Vos_String_Equal ("frame_counter" , var_name))
        {
        *var_p_ptr = (char *) (&prs_ptr->frame_counter);
        goto func_return;
        }
    if (Vos_String_Equal ("frame_interval" , var_name))
        {
        *var_p_ptr = (char *) (&prs_ptr->frame_interval);
        goto func_return;
        }
    if (Vos_String_Equal ("shandle" , var_name))
        {
        *var_p_ptr = (char *) (&prs_ptr->shandle);
        goto func_return;
        }
    if (Vos_String_Equal ("payload_len" , var_name))
        {
        *var_p_ptr = (char *) (&prs_ptr->payload_len);
        goto func_return;
        }
    if (Vos_String_Equal ("id" , var_name))
        {
        *var_p_ptr = (char *) (&prs_ptr->id);
        goto func_return;
        }
```

```
        if (Vos_String_Equal ("gop" , var_name))
              {
              *var_p_ptr = (char *) (&prs_ptr->gop);
              goto func_return;
              }
        if (Vos_String_Equal ("total_gop" , var_name))
              {
              *var_p_ptr = (char *) (&prs_ptr->total_gop);
              goto func_return;
              }
        if (Vos_String_Equal ("yshandle" , var_name))
              {
              *var_p_ptr = (char *) (&prs_ptr->yshandle);
              goto func_return;
              }

func_return:

        FOUT;
        }
```

2. CODE OF TOKEN BUCKET

```
/* Process model C form file: T_pro_TB_sub1.pr.c */
/* Portions of this file copyright 1992, 1996, 1998 by MIL 3, Inc. */



/* This variable carries the header into the object file */
static const char T_pro_TB_sub1_pr_c [] = "MIL_3_Tfile_Hdr_ 60L 30A opnet 7
3AC658BB 3AC658BB 1 grd0 qwang 0 0 none none 0 0 none 0 0 0 0 0 0";


/* OPNET system definitions */
#include <opnet.h>

#if defined (__cplusplus)
extern "C" {
#endif
FSM_EXT_DECS
#if defined (__cplusplus)
} /* end of 'extern "C"' */
#endif


/* Header Block */

#define ARRIVAL          (op_intrpt_type () == OPC_INTRPT_STRM)
#define TOKEN_COM
((op_intrpt_type()==OPC_INTRPT_SELF)&&(op_intrpt_code()==1))
extern data_1[10000];
extern data_1_dis[10000];
#define FRAME
((op_intrpt_type()==OPC_INTRPT_SELF)&&(op_intrpt_code()==0))
#define STA (op_intrpt_type()==OPC_INTRPT_STAT)
int loss_1;


/* End of Header Block */


#if !defined (VOSD_NO_FIN)
#undef      BIN
#undef      BOUT
#define     BIN          _fstack_local_info.last_line_passed = __LINE__ -
_block_origin;
#define     BOUT  BIN
#define     BINIT _fstack_local_info.last_line_passed = 0; _block_origin =
__LINE__;
#else
#define     BINIT
#endif /* #if !defined (VOSD_NO_FIN) */



/* State variable definitions */
typedef struct
      {
```

```
	/* Internal state tracking for FSM */
	FSM_SYS_STATE
	/* State Variables */
	int                                     i;
	Stathandle                              ishandle;
	double                                  token_rate;
	int                                     token_pool_status;
	int                                     token_pool_size;
	Stathandle                              jshandle;
	int                                     j;
	int                                     total;
	double                                  sta;
	int                                     no;
	Stathandle                              mshandle;
	double                                  frame_interval;
	Stathandle                              tshandle;
	Stathandle                              nshandle;
	Objid                                   my_id;
	double                                  avg_rate;
	double                                  peak_rate;
	double                                  king;
	} T_pro_TB_sub1_state;

#define pr_state_ptr                        ((T_pro_TB_sub1_state*)
SimI_Mod_State_Ptr)
#define i                                   pr_state_ptr->i
#define ishandle                            pr_state_ptr->ishandle
#define token_rate                          pr_state_ptr->token_rate
#define token_pool_status                   pr_state_ptr->token_pool_status
#define token_pool_size                     pr_state_ptr->token_pool_size
#define jshandle                            pr_state_ptr->jshandle
#define j                                   pr_state_ptr->j
#define total                               pr_state_ptr->total
#define sta                                 pr_state_ptr->sta
#define no                                  pr_state_ptr->no
#define mshandle                            pr_state_ptr->mshandle
#define frame_interval                      pr_state_ptr->frame_interval
#define tshandle                            pr_state_ptr->tshandle
#define nshandle                            pr_state_ptr->nshandle
#define my_id                               pr_state_ptr->my_id
#define avg_rate                            pr_state_ptr->avg_rate
#define peak_rate                           pr_state_ptr->peak_rate
#define king                                pr_state_ptr->king

/* This macro definition will define a local variable called      */
/* "op_sv_ptr" in each function containing a FIN statement. */
/* This variable points to the state variable data structure,     */
/* and can be used from a C debugger to display their values.     */
#undef FIN_PREAMBLE
#define FIN_PREAMBLE    T_pro_TB_sub1_state *op_sv_ptr = pr_state_ptr;


/* Function Block */

enum { _block_origin = __LINE__ };
```

```
/* End of Function Block */

#if defined (__cplusplus)
extern "C" {
#endif
      void T_pro_TB_sub1 (void);
      Compcode T_pro_TB_sub1_init (void **);
      void T_pro_TB_sub1_diag (void);
      void T_pro_TB_sub1_terminate (void);
      void T_pro_TB_sub1_svar (void *, const char *, char **);
#if defined (__cplusplus)
} /* end of 'extern "C"' */
#endif



/* Process model interrupt handling procedure */


void
T_pro_TB_sub1 (void)
      {
      int _block_origin = 0;
      FIN (T_pro_TB_sub1 ());
      if (1)
            {
            packet* pktptr;
            int pkt_len,bubu;
            double time,tk,dudu;


            FSM_ENTER (T_pro_TB_sub1)

            FSM_BLOCK_SWITCH
                  {
                  /*---------------------------------------------------------*/
                  /** state (init) enter executives **/
                  FSM_STATE_ENTER_FORCED (0, state0_enter_exec, "init",
"T_pro_TB_sub1 () [init enter execs]")
                        {
                        /* statistic */
                        ishandle=op_stat_reg("Packets Loss",
OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
                        jshandle=op_stat_reg("Token Pool Level (Bytes)",
OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
                        mshandle=op_stat_reg("Received Traffic (Bytes/sec)",
OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
                        tshandle=op_stat_reg("Delay (sec)", OPC_STAT_INDEX_NONE,
OPC_STAT_LOCAL);
                        nshandle=op_stat_reg("Loss_frame", OPC_STAT_INDEX_NONE,
OPC_STAT_LOCAL);

                        /* attributes */
                        /*my_id = op_id_self ();*/
                        /*op_ima_obj_attr_get (my_id, "token rate",
token_rate);*/
```

```
                        /*op_ima_obj_attr_get (my_id, "token pool size",
token_pool_size);*/

                        token_rate=640*1.05;
                        king=1280;
                        avg_rate=640;
                        peak_rate=4320;
                        token_pool_size=650000;

                        /*** Initial ***/
                        no=1;
                        loss_1=0;

                        token_pool_status=token_pool_size;
                        total=0;
                        frame_interval=0.033;
                        sta=op_sim_time();

                        /* set interrupt of token coming */
                        op_intrpt_schedule_self(op_sim_time()+(1/token_rate),1);
                        op_intrpt_schedule_self(1+frame_interval,0);

                        }


                    /** state (init) exit executives **/
                    FSM_STATE_EXIT_FORCED (0, state0_exit_exec, "init",
"T_pro_TB_sub1 () [init exit execs]")
                        {
                        }


                    /** state (init) transition processing **/
                    FSM_TRANSIT_FORCE (1, state1_enter_exec, ;)
                        /*---------------------------------------------------
---*/



                    /** state (idle) enter executives **/
                    FSM_STATE_ENTER_UNFORCED (1, state1_enter_exec, "idle",
"T_pro_TB_sub1 () [idle enter execs]")
                        {
                        }


                    /** blocking after enter executives of unforced state. **/
                    FSM_EXIT (3,T_pro_TB_sub1)


                    /** state (idle) exit executives **/
                    FSM_STATE_EXIT_UNFORCED (1, state1_exit_exec, "idle",
"T_pro_TB_sub1 () [idle exit execs]")
                        {
                        }
```

```
                    /** state (idle) transition processing **/
                    FSM_INIT_COND (ARRIVAL)
                    FSM_TEST_COND (TOKEN_COM)
                    FSM_TEST_COND (FRAME)
                    FSM_TEST_COND (STA)
                    FSM_TEST_LOGIC ("idle")

                    FSM_TRANSIT_SWITCH
                          {
                          FSM_CASE_TRANSIT (0, 2, state2_enter_exec, ;)
                          FSM_CASE_TRANSIT (1, 3, state3_enter_exec, ;)
                          FSM_CASE_TRANSIT (2, 4, state4_enter_exec, ;)
                          FSM_CASE_TRANSIT (3, 5, state5_enter_exec, ;)
                          }
                          /*-------------------------------------------------------
---*/



                    /** state (mark) enter executives **/
                    FSM_STATE_ENTER_FORCED (2, state2_enter_exec, "mark",
"T_pro_TB_sub1 () [mark enter execs]")
                          {
                          pktptr=op_pk_get (op_intrpt_strm ());
                          op_pk_fd_get (pktptr, 1, &time);
                          op_stat_write(tshandle, op_sim_time()-time);

                          pkt_len=1500;
                          total=total+pkt_len;
                          time=op_sim_time();

                          op_stat_write(ishandle, loss_1);
                          op_stat_write(jshandle,token_pool_status);

                          /*** policing ***/
                          if (token_pool_status>=pkt_len)
                          {
                            op_pk_fd_get (pktptr, 0, &j);
                            token_pool_status=token_pool_status-pkt_len;
                            op_pk_send_forced(pktptr,0);
                          }
                          else
                          {
                            loss_1=loss_1+1;
                            op_pk_fd_get (pktptr, 0, &i);
                            data_1_dis[i]=data_1_dis[i]-1500;
                            op_pk_destroy(pktptr);
                          }

                          }


                    /** state (mark) exit executives **/
                    FSM_STATE_EXIT_FORCED (2, state2_exit_exec, "mark",
"T_pro_TB_sub1 () [mark exit execs]")
                          {
                          }
```

```
                    /** state (mark) transition processing **/
                    FSM_TRANSIT_FORCE (1, state1_enter_exec, ;)
                         /*---------------------------------------------------
---*/



                    /** state (token) enter executives **/
                    FSM_STATE_ENTER_FORCED (3, state3_enter_exec, "token",
"T_pro_TB_sub1 () [token enter execs]")
                         {
                         /* set interrupt of token coming */
                         op_intrpt_schedule_self(op_sim_time()+(1/token_rate),1);
                         op_stat_write(jshandle,token_pool_status);

                         /* token pool level increase 1 */
                         token_pool_status=token_pool_status+1000;

                         if (token_pool_status>token_pool_size)
                         {
                           token_pool_status=token_pool_size;
                         }
                         }


                    /** state (token) exit executives **/
                    FSM_STATE_EXIT_FORCED (3, state3_exit_exec, "token",
"T_pro_TB_sub1 () [token exit execs]")
                         {
                         }


                    /** state (token) transition processing **/
                    FSM_TRANSIT_FORCE (1, state1_enter_exec, ;)
                         /*---------------------------------------------------
---*/



                    /** state (frame) enter executives **/
                    FSM_STATE_ENTER_FORCED (4, state4_enter_exec, "frame",
"T_pro_TB_sub1 () [frame enter execs]")
                         {
                         op_intrpt_schedule_self(op_sim_time()+frame_interval,0);
                         op_stat_write(mshandle,data_1_dis[no]/frame_interval);
                         op_stat_write(nshandle,data_1[no]-data_1_dis[no]);
                         no=no+1;
                         }


                    /** state (frame) exit executives **/
                    FSM_STATE_EXIT_FORCED (4, state4_exit_exec, "frame",
"T_pro_TB_sub1 () [frame exit execs]")
                         {
                         }
```

```
                    /** state (frame) transition processing **/
                    FSM_TRANSIT_FORCE (1, state1_enter_exec, ;)
                          /*---------------------------------------------------------
---*/



                    /** state (sta) enter executives **/
                    FSM_STATE_ENTER_FORCED (5, state5_enter_exec, "sta",
"T_pro_TB_sub1 () [sta enter execs]")
                          {
                          bubu=op_stat_local_read(0);
                          token_rate=bubu;
                          }


                    /** state (sta) exit executives **/
                    FSM_STATE_EXIT_FORCED (5, state5_exit_exec, "sta",
"T_pro_TB_sub1 () [sta exit execs]")
                          {
                          }


                    /** state (sta) transition processing **/
                    FSM_TRANSIT_FORCE (1, state1_enter_exec, ;)
                          /*---------------------------------------------------------
---*/



                    }

              FSM_EXIT (0,T_pro_TB_sub1)
              }
        }


Compcode
T_pro_TB_sub1_init (void ** gen_state_pptr)
        {
        int _block_origin = 0;
        static VosT_Cm_Obtype   obtype = OPC_NIL;

        extern void                     Vos_Vnop (void);

        FIN (T_pro_TB_sub1_init (gen_state_pptr))

        if (obtype == OPC_NIL)
              {
              /* Initialize memory management */
              if (Vos_Catmem_Register ("proc state vars (T_pro_TB_sub1)",
                    sizeof (T_pro_TB_sub1_state), Vos_Vnop, &obtype) ==
VOSC_FAILURE)
                    {
```

```
                FRET (OPC_COMPCODE_FAILURE)
                }
        }

    *gen_state_pptr = Vos_Catmem_Alloc (obtype, 1);
    if (*gen_state_pptr == OPC_NIL)
        {
        FRET (OPC_COMPCODE_FAILURE)
        }
    else
        {
        /* Initialize FSM handling */
        ((T_pro_TB_sub1_state *)(*gen_state_pptr))->current_block = 0;

        FRET (OPC_COMPCODE_SUCCESS)
        }
    }


void
T_pro_TB_sub1_diag (void)
    {
    /* No Diagnostic Block */
    }



void
T_pro_TB_sub1_terminate (void)
    {
    int _block_origin = __LINE__;

    FIN (T_pro_TB_sub1_terminate (void))

    if (1)
        {
        packet* pktptr;
        int pkt_len,bubu;
        double time,tk,dudu;

        /* Termination Block */


        BINIT


        /* End of Termination Block */

        }
    Vos_Catmem_Dealloc (pr_state_ptr);

    FOUT;
    }
```

```
/* Undefine shortcuts to state variables to avoid */
/* syntax error in direct access to fields of */
/* local variable prs_ptr in T_pro_TB_sub1_svar function. */
#undef i
#undef ishandle
#undef token_rate
#undef token_pool_status
#undef token_pool_size
#undef jshandle
#undef j
#undef total
#undef sta
#undef no
#undef mshandle
#undef frame_interval
#undef tshandle
#undef nshandle
#undef my_id
#undef avg_rate
#undef peak_rate
#undef king


void
T_pro_TB_sub1_svar (void * gen_ptr, const char * var_name, char ** var_p_ptr)
	{
	T_pro_TB_sub1_state		*prs_ptr;

	FIN (T_pro_TB_sub1_svar (gen_ptr, var_name, var_p_ptr))

	*var_p_ptr = (char *)VOS_NIL;
	prs_ptr = (T_pro_TB_sub1_state *)gen_ptr;

	if (Vos_String_Equal ("i" , var_name))
		{
		*var_p_ptr = (char *) (&prs_ptr->i);
		goto func_return;
		}
	if (Vos_String_Equal ("ishandle" , var_name))
		{
		*var_p_ptr = (char *) (&prs_ptr->ishandle);
		goto func_return;
		}
	if (Vos_String_Equal ("token_rate" , var_name))
		{
		*var_p_ptr = (char *) (&prs_ptr->token_rate);
		goto func_return;
		}
	if (Vos_String_Equal ("token_pool_status" , var_name))
		{
		*var_p_ptr = (char *) (&prs_ptr->token_pool_status);
		goto func_return;
		}
	if (Vos_String_Equal ("token_pool_size" , var_name))
		{
		*var_p_ptr = (char *) (&prs_ptr->token_pool_size);
```

```
        goto func_return;
        }
    if (Vos_String_Equal ("jshandle" , var_name))
        {
        *var_p_ptr = (char *) (&prs_ptr->jshandle);
        goto func_return;
        }
    if (Vos_String_Equal ("j" , var_name))
        {
        *var_p_ptr = (char *) (&prs_ptr->j);
        goto func_return;
        }
    if (Vos_String_Equal ("total" , var_name))
        {
        *var_p_ptr = (char *) (&prs_ptr->total);
        goto func_return;
        }
    if (Vos_String_Equal ("sta" , var_name))
        {
        *var_p_ptr = (char *) (&prs_ptr->sta);
        goto func_return;
        }
    if (Vos_String_Equal ("no" , var_name))
        {
        *var_p_ptr = (char *) (&prs_ptr->no);
        goto func_return;
        }
    if (Vos_String_Equal ("mshandle" , var_name))
        {
        *var_p_ptr = (char *) (&prs_ptr->mshandle);
        goto func_return;
        }
    if (Vos_String_Equal ("frame_interval" , var_name))
        {
        *var_p_ptr = (char *) (&prs_ptr->frame_interval);
        goto func_return;
        }
    if (Vos_String_Equal ("tshandle" , var_name))
        {
        *var_p_ptr = (char *) (&prs_ptr->tshandle);
        goto func_return;
        }
    if (Vos_String_Equal ("nshandle" , var_name))
        {
        *var_p_ptr = (char *) (&prs_ptr->nshandle);
        goto func_return;
        }
    if (Vos_String_Equal ("my_id" , var_name))
        {
        *var_p_ptr = (char *) (&prs_ptr->my_id);
        goto func_return;
        }
    if (Vos_String_Equal ("avg_rate" , var_name))
        {
        *var_p_ptr = (char *) (&prs_ptr->avg_rate);
        goto func_return;
        }
```

```
      if (Vos_String_Equal ("peak_rate" , var_name))
            {
            *var_p_ptr = (char *) (&prs_ptr->peak_rate);
            goto func_return;
            }
      if (Vos_String_Equal ("king" , var_name))
            {
            *var_p_ptr = (char *) (&prs_ptr->king);
            goto func_return;
            }

func_return:

      FOUT;
      }
```

3. CODE OF FEEDBACK CONTROLLER

```
/* Process model C form file: T_pro_VBR_controller.pr.c */
/* Portions of this file copyright 1992, 1996, 1998 by MIL 3, Inc. */



/* This variable carries the header into the object file */
static const char T_pro_VBR_controller_pr_c [] = "MIL_3_Tfile_Hdr_ 60L 30A opnet
7 3AC66365 3AC66365 1 grd0 qwang 0 0 none none 0 0 none 0 0 0 0 0 0";


/* OPNET system definitions */
#include <opnet.h>

#if defined (__cplusplus)
extern "C" {
#endif
FSM_EXT_DECS
#if defined (__cplusplus)
} /* end of 'extern "C"' */
#endif


/* Header Block */

#define STA (op_intrpt_type()==OPC_INTRPT_STAT)

/* End of Header Block */


#if !defined (VOSD_NO_FIN)
#undef      BIN
#undef      BOUT
#define     BIN         _fstack_local_info.last_line_passed = __LINE__ -
_block_origin;
#define     BOUT  BIN
#define     BINIT _fstack_local_info.last_line_passed = 0; _block_origin =
__LINE__;
#else
#define     BINIT
#endif /* #if !defined (VOSD_NO_FIN) */



/* State variable definitions */
typedef struct
      {
      /* Internal state tracking for FSM */
      FSM_SYS_STATE
      /* State Variables */
      Stathandle                          shandle_1;
      Stathandle                          shandle_2;
      Stathandle                          shandle_3;
      int                                 que_len;
      } T_pro_VBR_controller_state;
```

```
#define pr_state_ptr                    ((T_pro_VBR_controller_state*)
SimI_Mod_State_Ptr)
#define shandle_1                       pr_state_ptr->shandle_1
#define shandle_2                       pr_state_ptr->shandle_2
#define shandle_3                       pr_state_ptr->shandle_3
#define que_len                         pr_state_ptr->que_len

/* This macro definition will define a local variable called      */
/* "op_sv_ptr" in each function containing a FIN statement. */
/* This variable points to the state variable data structure,     */
/* and can be used from a C debugger to display their values.     */
#undef FIN_PREAMBLE
#define FIN_PREAMBLE    T_pro_VBR_controller_state *op_sv_ptr = pr_state_ptr;


/* Function Block */

enum { _block_origin = __LINE__ };


/* End of Function Block */

#if defined (__cplusplus)
extern "C" {
#endif
      void T_pro_VBR_controller (void);
      Compcode T_pro_VBR_controller_init (void **);
      void T_pro_VBR_controller_diag (void);
      void T_pro_VBR_controller_terminate (void);
      void T_pro_VBR_controller_svar (void *, const char *, char **);
#if defined (__cplusplus)
} /* end of 'extern "C"' */
#endif



/* Process model interrupt handling procedure */


void
T_pro_VBR_controller (void)
      {
      int _block_origin = 0;
      FIN (T_pro_VBR_controller ());
      if (1)
            {
            double
tr_1,tr_2,tr_3,bubu_1,bubu_2,bubu_3,t_bubu,bw,tutu_1,tutu_2,tutu_3;
            int dudu;


            FSM_ENTER (T_pro_VBR_controller)

            FSM_BLOCK_SWITCH
                  {
                  /*-------------------------------------------------------*/
```

```
                    /** state (init) enter executives **/
                    FSM_STATE_ENTER_FORCED (0, state0_enter_exec, "init",
"T_pro_VBR_controller () [init enter execs]")
                          {
                          /* initial */
                          shandle_1=op_stat_reg("tr_fb_1", OPC_STAT_INDEX_NONE,
OPC_STAT_LOCAL);
                          shandle_2=op_stat_reg("tr_fb_2", OPC_STAT_INDEX_NONE,
OPC_STAT_LOCAL);
                          shandle_3=op_stat_reg("tr_fb_3", OPC_STAT_INDEX_NONE,
OPC_STAT_LOCAL);

                          que_len=2000;
                          }


                    /** state (init) exit executives **/
                    FSM_STATE_EXIT_FORCED (0, state0_exit_exec, "init",
"T_pro_VBR_controller () [init exit execs]")
                          {
                          }


                    /** state (init) transition processing **/
                    FSM_TRANSIT_FORCE (1, state1_enter_exec, ;)
                          /*---------------------------------------------------------
---*/



                    /** state (idle) enter executives **/
                    FSM_STATE_ENTER_UNFORCED (1, state1_enter_exec, "idle",
"T_pro_VBR_controller () [idle enter execs]")
                          {

                          }


                    /** blocking after enter executives of unforced state. **/
                    FSM_EXIT (3,T_pro_VBR_controller)


                    /** state (idle) exit executives **/
                    FSM_STATE_EXIT_UNFORCED (1, state1_exit_exec, "idle",
"T_pro_VBR_controller () [idle exit execs]")
                          {
                          }


                    /** state (idle) transition processing **/
                    FSM_TRANSIT_ONLY ((STA), 2, state2_enter_exec, ;, "idle")
                          /*---------------------------------------------------------
---*/



                    /** state (gop) enter executives **/
```

```
                FSM_STATE_ENTER_FORCED (2, state2_enter_exec, "gop",
"T_pro_VBR_controller () [gop enter execs]")
                        {
                        bubu_1=op_stat_local_read(1); /*frm src_1*/
                        bubu_2=op_stat_local_read(2); /*frm src_2*/
                        bubu_3=op_stat_local_read(3); /*frm src_3*/
                        dudu=op_stat_local_read(0); /*frm queue*/
                        tutu_1=op_stat_local_read(4); /*TB_1*/
                        tutu_2=op_stat_local_read(5); /*TB_2*/
                        tutu_3=op_stat_local_read(6); /*TB_3*/

                        bw=((29990-dudu)*1500/0.5)+1.21*10000000;
                        /*pkt_len:1500,gop_time:0.5,link_sppd:1.79M*/

                        t_bubu=bubu_1+bubu_2+bubu_3;

                        if (((bubu_1/t_bubu)*bw-tutu_1/0.5)/1000>0)
                        {
                          tr_1=((bubu_1/t_bubu)*bw-tutu_1/0.5)/1000;
                        }
                        else
                        {
                          tr_1=1.0;
                        }

                        if (((bubu_2/t_bubu)*bw-tutu_2/0.5)/1000>0)
                        {
                          tr_2=((bubu_2/t_bubu)*bw-tutu_2/0.5)/1000;
                        }
                        else
                        {
                          tr_2=1.0;
                        }

                        if (((bubu_3/t_bubu)*bw-tutu_3/0.5)/1000>0)
                        {
                          tr_3=((bubu_3/t_bubu)*bw-tutu_3/0.5)/1000;
                        }
                        else
                        {
                          tr_3=1.0;
                        }

                        op_stat_write(shandle_1,tr_1);
                        op_stat_write(shandle_2,tr_2);
                        op_stat_write(shandle_3,tr_3);

                        /*printf("t1=%f, t2=%f, t3=%f\n",tr_1,tr_2,tr_3);*/
                        if (dudu>=22000)
                        {
                        printf("dudu=%d\n",dudu);
                        }
                        /*printf("t1=%f, t2=%f, t3=%f\n",tr_1,tr_2,tr_3);
                        printf("tutu1=%f, tutu2=%f,
tutu3=%f\n",tutu_1,tutu_2,tutu_3);*/
                        }
```

```
                    /** state (gop) exit executives **/
                    FSM_STATE_EXIT_FORCED (2, state2_exit_exec, "gop",
"T_pro_VBR_controller () [gop exit execs]")
                          {

                          }


                    /** state (gop) transition processing **/
                    FSM_TRANSIT_FORCE (1, state1_enter_exec, ;)
                          /*---------------------------------------------------------
---*/



                    }


              FSM_EXIT (0,T_pro_VBR_controller)
              }
        }


Compcode
T_pro_VBR_controller_init (void ** gen_state_pptr)
        {
        int _block_origin = 0;
        static VosT_Cm_Obtype   obtype = OPC_NIL;

        extern void                     Vos_Vnop (void);

        FIN (T_pro_VBR_controller_init (gen_state_pptr))

        if (obtype == OPC_NIL)
              {
              /* Initialize memory management */
              if (Vos_Catmem_Register ("proc state vars (T_pro_VBR_controller)",
                    sizeof (T_pro_VBR_controller_state), Vos_Vnop, &obtype) ==
VOSC_FAILURE)
                    {
                    FRET (OPC_COMPCODE_FAILURE)
                    }
              }

        *gen_state_pptr = Vos_Catmem_Alloc (obtype, 1);
        if (*gen_state_pptr == OPC_NIL)
              {
              FRET (OPC_COMPCODE_FAILURE)
              }
        else
              {
              /* Initialize FSM handling */
              ((T_pro_VBR_controller_state *)(*gen_state_pptr))->current_block =
0;

              FRET (OPC_COMPCODE_SUCCESS)
```

```
        }
    }


void
T_pro_VBR_controller_diag (void)
    {
    int _block_origin = __LINE__;

    FIN (T_pro_VBR_controller_diag ())

    if (1)
        {
        double
tr_1,tr_2,tr_3,bubu_1,bubu_2,bubu_3,t_bubu,bw,tutu_1,tutu_2,tutu_3;
        int dudu;

        /* Diagnostic Block */


        BINIT


        /* End of Diagnostic Block */

        }

    FOUT;
    }



void
T_pro_VBR_controller_terminate (void)
    {
    int _block_origin = __LINE__;

    FIN (T_pro_VBR_controller_terminate (void))

    if (1)
        {
        double
tr_1,tr_2,tr_3,bubu_1,bubu_2,bubu_3,t_bubu,bw,tutu_1,tutu_2,tutu_3;
        int dudu;

        /* Termination Block */


        BINIT


        /* End of Termination Block */

        }
    Vos_Catmem_Dealloc (pr_state_ptr);
```

```
	FOUT;
	}


/* Undefine shortcuts to state variables to avoid */
/* syntax error in direct access to fields of */
/* local variable prs_ptr in T_pro_VBR_controller_svar function. */
#undef shandle_1
#undef shandle_2
#undef shandle_3
#undef que_len



void
T_pro_VBR_controller_svar (void * gen_ptr, const char * var_name, char **
var_p_ptr)
	{
	T_pro_VBR_controller_state		*prs_ptr;

	FIN (T_pro_VBR_controller_svar (gen_ptr, var_name, var_p_ptr))

	*var_p_ptr = (char *)VOS_NIL;
	prs_ptr = (T_pro_VBR_controller_state *)gen_ptr;

	if (Vos_String_Equal ("shandle_1" , var_name))
		{
		*var_p_ptr = (char *) (&prs_ptr->shandle_1);
		goto func_return;
		}
	if (Vos_String_Equal ("shandle_2" , var_name))
		{
		*var_p_ptr = (char *) (&prs_ptr->shandle_2);
		goto func_return;
		}
	if (Vos_String_Equal ("shandle_3" , var_name))
		{
		*var_p_ptr = (char *) (&prs_ptr->shandle_3);
		goto func_return;
		}
	if (Vos_String_Equal ("que_len" , var_name))
		{
		*var_p_ptr = (char *) (&prs_ptr->que_len);
		goto func_return;
		}

func_return:

	FOUT;
	}
```