# DESIGN AND IMPLEMENTATION OF AN EFFECIENT TRAFFIC SHAPER USING AN ENHANCED VERSION OF LEAKY BUCKET ALGORITHM AND ITS PERFORMANCE COMPARISON WITH STANDARD LEAKY BUCKET ALGORITHM


**Nikhil Bhargava (141/Coe/98)**
**Manoj Kumar Sah (134/Coe/98)**

**Department of Computer Engineering**
**Netaji Subhas Institute of Technology**
**New Delhi**
**May, 2002**

# Certificate

This is to certify that the project entitled, **"DESIGN AND IMPLEMENTATION OF AN EFFECIENT TRAFFIC SHAPER USING AN ENHANCED VERSION OF LEAKY BUCKET ALGORITHM AND ITS PERFORMANCE COMPARISON WITH STANDARD LEAKY BUCKET "** submitted by **Mr. Nikhil Bhargava (141/COE/98)** and **Mr. Manoj Kumar Sah (134/COE/98)** for the partial fulfillment of Award of "Bachelor of Engineering" in Computers is a bonafide record of the original and authentic work done by the candidates under my guidance. To the best of my knowledge this work has not been submitted for the award of any other degree.

**Ms. Anubha Gupta**
**Assistant Professor,**
**Department of Computer Engineering.**
**Netaji Subhas Institute of Technology**
**Sector-3, Dwarka.**
**New Delhi - 110045**

**Date:**

**Place:**

# Acknowledgement

We acknowledge our thanks to Mrs. Shampa Chakarverty, Head of Department, Computer Engineering, Netaji Subhas Institute of Technology (NSIT) for providing us the golden opportunity to develop our final year project under her aegis. We are highly indebted to our Project Guide, Ms. Anubha Gupta for her guidance and inspiration that she showered on us to make this project a success. We also like to thankful to Mr. S.P Singh, Sr. Lecturer, Electronics and Communication Department, NSIT for helping us to solve many conceptual problems that we faced during the projects. We are also thankful to our department for providing us resources for the development of the project. The Lab assistant of Software Engineering lab, Mr. R.k Gupta deserves a special mention, for their cooperation and assistance in providing us the facilities at the Lab, at all add hours.

We are also highly grateful to Mr. Ashok K. Agrawal, Institute for Advanced Computer Studies and Department of Computer Science, University of Maryland, College Park, Md, USA for solving a conceptual problem that we faced during the coding of the last phase of the project.

We are also grateful to our families, for enduring us throughout the course of this project. Their love, support and understanding worked as antidotes for us and kept us going.

Finally we are grateful to almighty lord Shiva for giving us the strength and patience to complete the project on time.

Nikhil Bhargava

Manoj Kumar Sah

# INDEX

# Index of Figures

# Chapter 1. Introduction

*This Final Year project report documents systematic design, development, implementation of an efficient traffic shaper and includes its comparative study with standard leaky bucket algorithm*

Traffic control is an essential part of today's networks due the increasing demand for large bandwidth and high speed. These include admission control at the connection setup, traffic control at source ends and efficient scheduling schemes at the switches. Most multimedia sources are bursty in nature, which increase congestion the most.

Traffic shapers have been used from the point of view of their effectiveness in smoothing the burstiness. For example, The Leaky Bucket (LB) scheme is a mean rate policer, which smoothes the traffic either using clock ticks or by token generation.

Studies on bursty sources show that burstiness promotes statistical multiplexing at the cost of possible congestion. Smoothing on the other hand helps in providing guarantees at the cost of system utilization. Thus there is a need for flexible scheme, which can provide a reasonable agreement between utilization and performance.

Advances in optical transmission media and high speed switching have paved the way for many exiting multimedia application, such as teleconferencing and real time distributed computing to be supported on computer networks. Most of these new applications, constituted of heterogeneous mix of voice, video and data are characterized by stringent QoS requirements in terms of throughput, delay jitter and low delay guarantees. The heterogeneity of the source calls for effective congestion control scheme to meet the diverse quality of service requirement of each application. These include admission control at connection setup, traffic enforcements and shaping at the edges of the network and multimedia scheduling schemes at the intermediate switches. Latency efforts apparent at the gigabit speed make the conventional feedback technique ineffective. Thus the responsibility of preventing congestion lies with the admission control and traffic enforcement schemes.

## 1.1 Main Goal of the Project

Maximizing bandwidth utilization and performance in the context of multimedia networks are two totally incompatible goals like two shores of the sea. Multimedia sources of today's world are characterized by diverse Quality of Service (QOS) requirements. To satisfy these diverse QOS requirements we need effective traffic control schemes at all levels. These include admission control at source end and efficient scheduling schemes at he switches. We have tried to provide a full proof solution for traffic congestion at the source end.

Since most of the multimedia sources are bursty in nature so some kind of traffic shaper is needed to smoothen the in coming bursty traffic at source end. An interesting fact that has emerged from years of research of multimedia traffic is that burstiness promotes statistical multiplexing at cost of possible congestion. Smoothing on other hand helps in providing performance guarantees at the cost of low system utilization. Thus there is need for flexible scheme, which can provide a good, satisfactory and unbiased tradeoff between system utilization, and performance guarantee is imminent. The most common scheme used for this purpose is Leaky bucket scheme (LB) but this scheme proves to a real bottleneck for real time systems because it introduces a large amount of access delay both at source end and at intermediate routers. Thus there is a need for a policy, which is less strict on short-term burstiness than the LB.

We propose a new traffic shaper, which can adjust the burstiness of the traffic to obtain reasonable bandwidth utilization while maintaining statistical service guarantee.

The aim of our project is to design an efficient traffic shaper for high-speed networks using an enhanced version of leaky bucket algorithm and its comparative study with standard leaky bucket scheme.

## 1.2 Overview of project

Advances in optical transmission media and high speed switching have given way to many new and exciting applications like teleconferencing; video on demand and real time distributed computing. All these applications can't work if they are not given the firm base of fast and efficient computer networks. Most of the new applications consist of a heterogeneous mixture of these 5 things – data, audio, video, image and graphics. Each of these comes in the form of either constant bit rate traffic or variable bit rate traffic and have diverse QOS requirements in term of throughput, jitter delay or mean delay. The heterogeneity of these sources coupled with the aim of high performance and high system utilization makes it absolutely necessary to apply some kind of admission control at connection setup, traffic enforcement and shaping at the edges of the network and various scheduling schemes at intermediate routers and switches.

Admission control is decided by an algorithm, which expects that the user provide an estimate of the traffic parameters, and abides by their negotiated values. In a resource sharing packet network, admission control and scheduling schemes by themselves are not sufficient to provide performance guarantees. This is due to the fact that the users intentionally or otherwise, attempt to exceed the bandwidth allotted to each of them at connection time. This leads to the phenomenon of congestion. Various congestion control schemes have been proposed in the literature like Leaky Bucket (LB), Jumping Window (JW), exponentially weighted Moving average (EWMA) and associated variations. It has been shown that LB and EWMA are most promising candidates in policing bursty traffic.

Traffic shaping, on the other hand, conditions the input stream so that the characteristics of the outgoing traffic are manageable by the scheduling algorithm (at intermediate node or other end) to provide required QOS. Traffic shapers are mainly studied from the point of view of their effectiveness in smoothing the burstiness. The LB is a mean rate policier which smoothes the traffic at token generation rate. Studies on burstiness promote statistical multiplexing at the cost of possible congestion control. Smoothing on the other hand helps in providing guarantees at the cost of system utilization. Normal LB in its attempt to enforce smoothness often (always) introduces large amount access delays, which makes it in capable to handle real time traffic. Thus there is need for a policy, which is less strict on the short-term burstiness while, reverts back to standard LB for long-term burstiness.

We present a new traffic shaper, which can adjust the input traffic to obtain reasonable bandwidth utilization while maintaining statistical service guarantees. It uses a window based shaping policy, which embeds the essence of LB, permits short-term burstiness in a more flexible way and is essentially peak rate enforced.

We carried out the project in three phases. In the first phase we studied the behaviour of the system in the absence of any congestion control scheme. In the next phase we implemented the standard LB and studied the behaviour of

the system with LB congestion control. In the last phase we designed and implemented our new traffic shaper and studied the behaviour of system implementing this new shaper and conclude the phase with comparative study of leaky bucket scheme and new shaper.

## 1.3 Requirements

This project has following Software requirements

- Any POSIX compliant C compiler.
- Any good debugger with stack tracking feature like GDB and LINT
- Windows 9X/Linux platform
- Microsoft Office package (preferably 2000)
- Mathematica
- Adobe Acrobat Reader

My project has no requirements in hardware.

# Chapter 2 System Description

## 2.1 What Is Congestion?

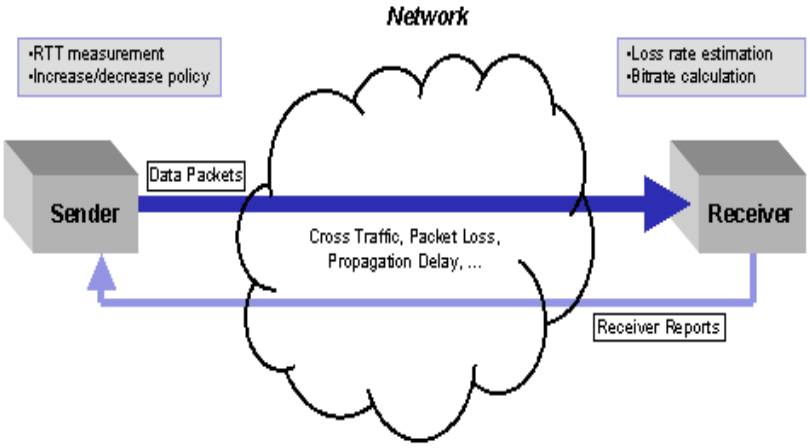



Fig 1 Congestion in network

As seen from figure1, Congestion is abundance of the data or the packets on the network than the provided limits of the bandwidth, which results in packet loss due to excessive delays or retransmission of packets. In other words we can say that when incoming rate of the packets is greater than the rate at which the packets can be transmitted out, congestion is said to occur. For e.g., consider the communication network as shown in the fig 2. Suppose nodes 1, 2 and 5 send packets to node 4 simultaneously and also suppose that the incoming rate of the packets is greater than the rate of servicing. In this case the data buffer in node 4 will build up. If this situation occurs sufficiently long, buffer will eventually become full and start rejecting packets. When the destination detects the missing packets it may ask the sources to retransmit the packets. This will act as a positive feedback to sources and in coming traffic to node 4 will increase many folds. The net result that through put of destination will be low as illustrated in fig 2 (uncontrolled curve).

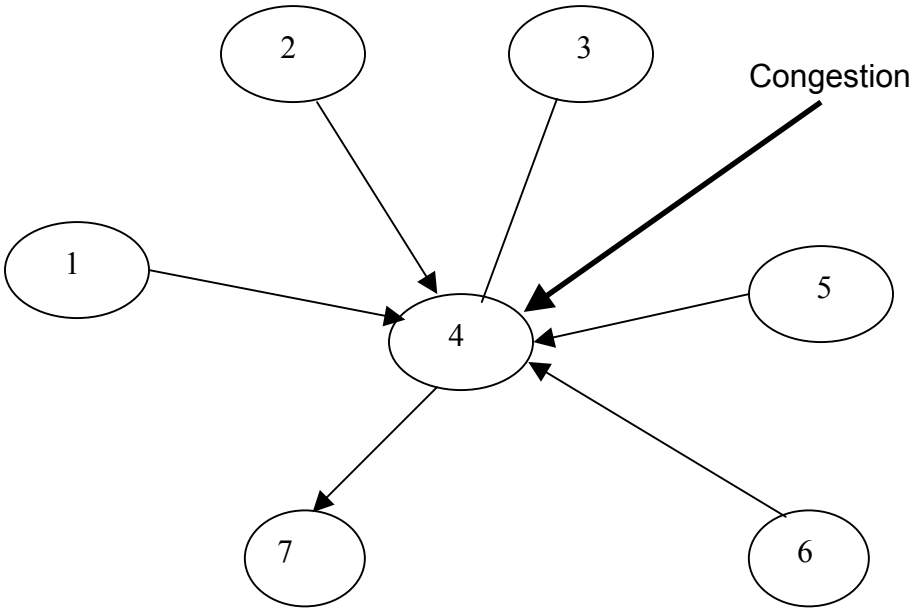



Fig 2. A Congested Switch

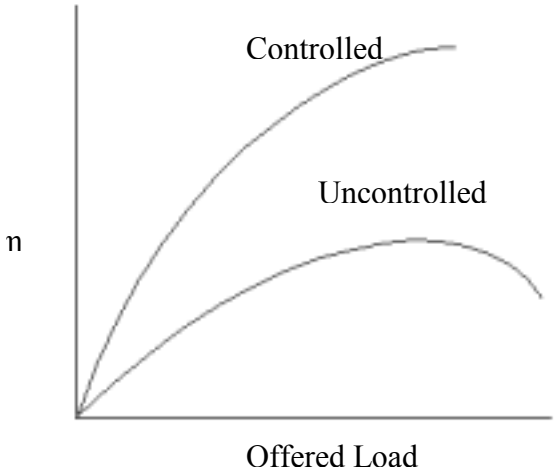



Fig 3. Through put drops when congestion occurs

The purpose of congestion control is to eliminate or reduce congestion. If done properly performance should improve (controlled curve).

Within a network, any shared resource is a potential point at which congestion may occur. If you consider the prototype of a communication network, it consists of a number of nodes (switches, routers...) interconnected by communication links (in this context, often called “trunks”). End-user devices are connected to the nodes through other communication links. there are two places where information flows share

**Network resources:**

1. Links (trunks)

2. Nodes (switches or routers)

Depending on their design, there are different aspects of a node which are shared:

- Memory (packet buffer pools)
- I/O processors (scanners, etc.)
- Internal buses or switches
- Central processors
- Network management processors

Many other things depending on the particular node′s design Whenever there is sharing of resources between data flows, there is usually (notably not always) the temptation for designers to over commit resources. For example, in the traditional packet switch architecture of the 1970s and 1980s a shared memory was used as a “pool” for intermediate storage of data as it passed through the node. This was done because of the immense cost benefit (20 to 1, perhaps 100 to 1) by taking advantage of the statistical characteristics of network traffic. There are actually many aspects to this, for example, the use of a single memory (implying the sharing of access to that memory) instead of multiple small memories and the total size of that memory. Another example is the universal use of shared trunks, the capacity of which is usually significantly less than the total capacity of connected links to end users. Whenever there is a shared resource where there is the possibility of contention for the use of that resource you have to do something to resolve who gets it. The usual “something” is to build a queue in front of the resource. Indeed, a communication network can be seen as a network of queues. The presence of a queue implies a delay (a wait for service). Because queues vary in length, sometimes very quickly, their

presence also implies a variation in delay (jitter). It also implies a cost (for storage to hold the data, for example).
But what do you do when the queue gets too long? You could throw the data away, you could tell the input devices to stop sending input, you could perhaps keep track on the length of the queue and do something to control the input rate so that the queue never became too long. These latter techniques are flow controls.
In the networks of the 1970s and 1980s there were usually flow controls built into the network protocols. This was (in some contexts still is) a very important issue. It was the superiority of the flow controls in IBM SNA that enabled SNA to utilize trunk links to very high utilization while maintaining stable network operation. This meant that users saved cost on trunk connections. It was this aspect that underwrote the enormous success of SNA.

**Queue Behavior**

Figure shows a very famous (but not intuitive) result.

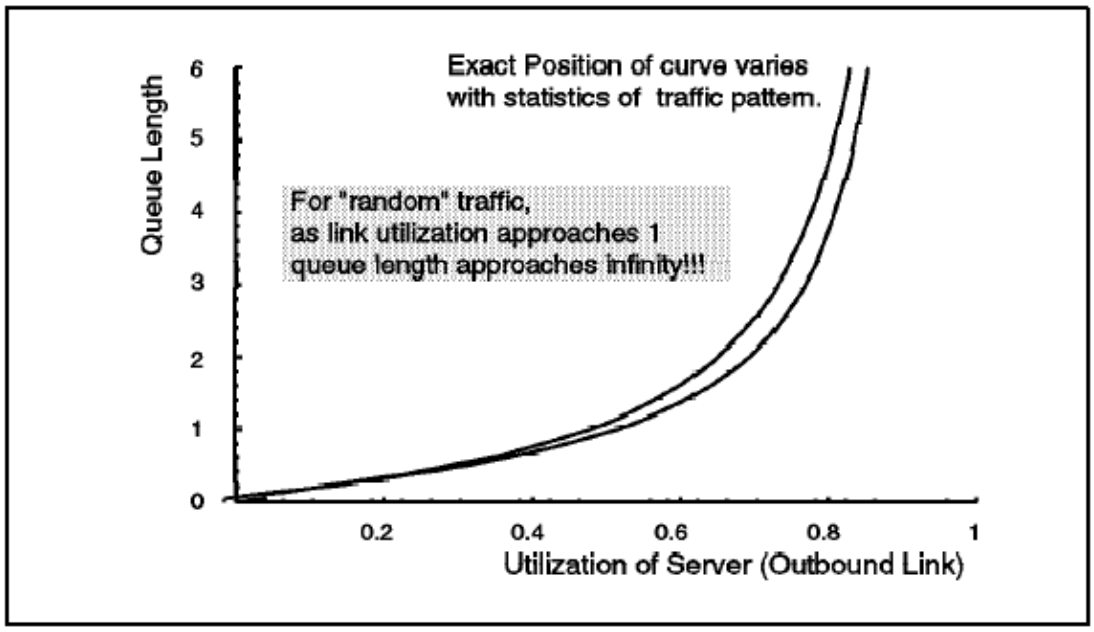



Fig 4. Queue Length versus Resource Utilization

This applies to queuing in general, not only in communications. If you have a shared resource (such as a supermarket checkout) where transactions (customers) arrive randomly (whatever definition of randomness you like to use) you get a queue forming. A critical aspect of this is the rate of arrivals at the shared resource (server) and the rate at which transactions are processed. There is a surprising (and critical) result here: As the average rate of arrivals approaches the average rate of transaction processing the length of the queue

will tend to infinity. This hinges on randomness of arrivals and is influenced by randomness in service times (that is, the number of items in the basket in a supermarket or the length of a block of data in a communication network).

Another way of stating the above is to introduce the concept of utilization of the server. Utilization is just the average arrival rate multiplied by the average time it takes to service a transaction expressed as a proportion of total time. Utilization is usually quoted as the percentage of time that the server is busy. Many books have been written on queuing theory! An important thing to note is that there are many variables here that affect the position of the curve in Figure 90. The precise arrival pattern is very important. If arrivals are at exact intervals and all transactions take exactly the same time to process, then the curve shifts to the right so that there is almost no queue at all until we reach 100% utilization. Different patterns of randomness 54 result in the curve shifting to the left of where it is shown.

The point is that as you increase the loading on any shared facility you reach a point where, if you add any more traffic, you will get severely increased queue length and problems with congestion.

**Effect of Network Protocols**

Figure 91 shows what happens in different kinds of networks when the load offered to the network (potential traffic that end-user devices want to send) is increased beyond the network′s capacity to handle it. The curve on the left is typcal of Ethernet (and ATM with some traffic types). As load is increased, the network handles it fine until a point is reached where the traffic can no longer be handled. Network throughput actually decreases very quickly when this point is reached. In fact, in the case of Ethernet, the network collapses and no data at all gets through. This is because collisions take capacity away from the network, as load increases collisions increase and capacity decreases which itself increases the probability of more collisions!

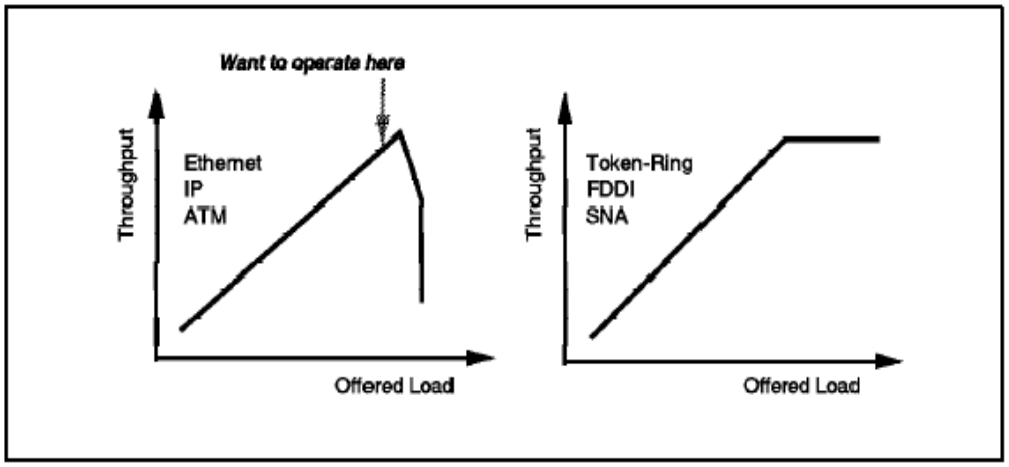



Fig5. ATM Network Throughput under Increasing Load Conditions

It must be pointed out that Ethernet works perfectly well in many situations, mainly as a result of the flow controls imposed by network protocols at higher layers (that is, external to Ethernet itself). The same is true of IP (TCP flow control stabilizes it) and (many would assert) will be true of ATM.

The curve on the right shows what happens in well controlled networks. Once full throughput is reached, the network continues to operate at full capacity. There are queues, of course, but these are mainly in the end-user devices to avoid an overflow of the queues in the network. This comes at the higher cost of token-based access control (in TRN and FDDI) or extensive flow controls and the necessary processing hardware (in SNA). “Ya pays yer money and ya takes yer choice.”

**Deadlocks**

- The first router cannot proceed until the second router does something, and the second router cannot proceed until the first router does something
- Both routers come to a complete halt and stay that way forever

1. Store and Forward Lockup
   - Direct Store and Forward Lockup Types of deadlock
   - Indirect Store and Forward Lockup

**2.** Reassembly Lockup

Direct Store and Forward Lockup
Simplest lockup between two routers

Example:

Suppose router A has five buffers, all of which are queued for output to router B Similarly, router B has five buffers, all of which are queued for output to router A If there is flow control on the link between routers A and B, then neither router can accept any incoming packets from the other. They are both stuck.

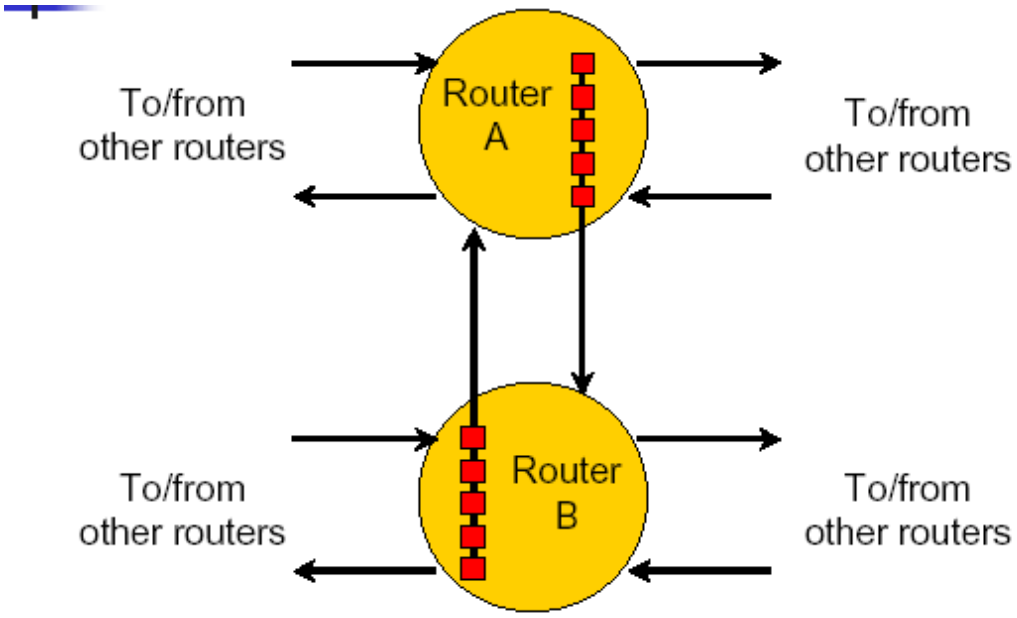



Fig 6. Indirect Store and Forward Lockup

If we view the network as a graph with nodes and edges

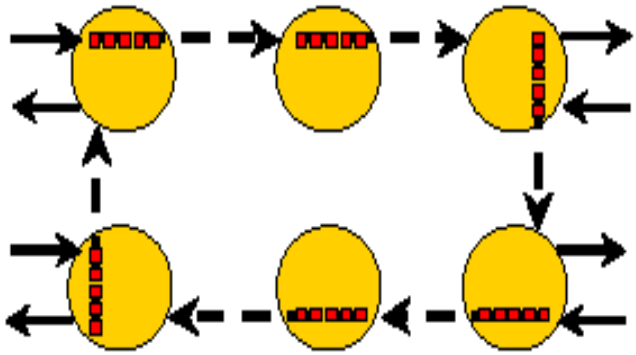


Fig7. Cycles in a Dependency Graph

A deadlock occurs when There is **a cycle** of dependencies in the graph

## Reassembly Lockup

In some network layer implementations, the sending router must split messages into multiple network layer packets. Receiving routers reassemble split

up packets into a single packetIf the receiving router's buffer fills up with incomplete packets, it cannot reassemble any more packets

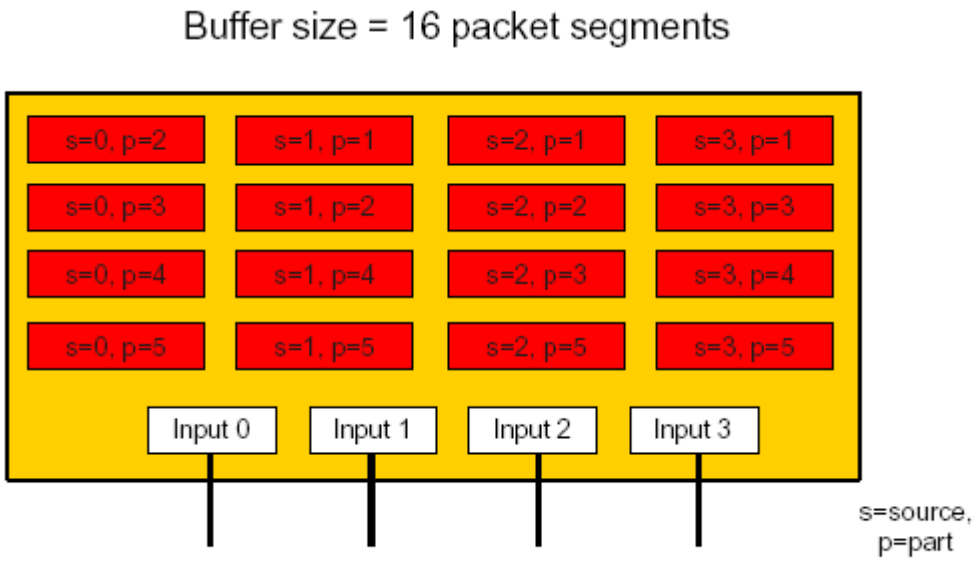



Fig 8. Buffer flow in a congested network

## 2.2 Exploring various causes of Congestion

There are several misunderstandings about the cause and the solutions of congestion control. Congestion is caused by the shortage of buffer space. The problem will be solved when the cost of memory becomes cheap enough to allow very large memory. Larger buffer is useful only for very short-term congestion and will cause undesirable long delays. Suppose the total input rate of a switch is 1Mbps and the capacity of the output link is 0.5Mbps, the buffer will overflow after 16 second with 1Mbyte memory and will also overflow after 1 hour with 225Mbyte memory if the situation persists. Thus larger buffer size can only postpone the discarding of cells but cannot prevent it. The long queue and long delay introduced by large memory is undesirable for some applications.

Congestion is caused by slow links. The problem will be solved when high-speed links become available. It is not always the case; sometimes increases in link bandwidth can aggravate the congestion problem because higher speed links may make the network more unbalanced.

Congestion is caused by slow processors. The problem will be solved when processor speed is improved. This statement can be explained to be wrong similarly to the second one. Faster processors will transmit more data in unit

time. If several nodes begin to transmit to one destination simultaneously at their peak rate, the target will be overwhelmed soon.

## 2.3 What Is Congestion Control?

It is the control on the congestion by providing a smoothly running traffic on an effective bandwidth to improve the performance.

## 2.4 What is expected from Congestion Control?

### Objectives

The objectives of traffic control and congestion control for ATM are: Support a set of QoS parameters and classes for all ATM services and minimize network and end-system complexity while maximizing network utilization.

### Selection Criteria

To design a congestion control scheme is appropriate for ATM network and non-ATM networks as well, the following guidance is of general interest.

### Scalability

The scheme should not be limited to a particular range of speed, distance, number of switches, or number of VCs. The scheme should be applicable for both local area networks (LAN) and wide area networks (WAN).

### Fairness

In a shared environment, the throughput for a source depends upon the demands by other sources. There are several proposed criterion for what is the correct share of bandwidth for a source in a network environment. And

there are ways to evaluate a bandwidth allocation scheme by comparing its results with an optimal result.

**Fairness Criteria**

1. Max-Min
   The available bandwidth is equally shared among connections.

1. MCR plus equal share
   The bandwidth allocation for a connection is its MCR plus equal share of the available bandwidth with used MCR removed.

2. Maximum of MCR or Max-Min share
   The bandwidth allocation for a connection is its MCR or Max-Min share, which ever is larger.
3. Allocation proportional to MCR
   The bandwidth allocation for a connection is weighted proportional to its MCR.
4. Weighted allocation
   The bandwidth allocation for a connection is proportional to its pre-determined weight.

**Fairness Index**

The share of bandwidth for each source should be equal to or converge to the optimal value according to some optimality criterion. We can estimate the fairness of a certain scheme numerically as follows. Suppose a scheme allocates x1, x2, ..., xn, while the optimal allocation is y1, y2, ..., yn. The normalized allocation is zi = xi / yi for each source and the fairness index is defined as following:

Fairness = sum(zi) * sum(zi) / sum(zi * zi)

- Robustness

  The scheme should be insensitive to minor deviations such as slight mistuning of parameters or loss of control messages. It should also isolate misbehaving users and protect other users from them.

- Implement ability

  The scheme should not dictate particular switch architecture. It also should not be too complex both in term of time or space it uses.

## 2.5 Classification of Congestion Control

Based on the place where congestion is controlled, Congestion Control is classified into two broad classes

- Source End Control
- Network Control (Switches and Routers)

Based on the procedure by which the congestion is controlled we can classify Congestion Control algorithms as

- Open loop control algorithms
- Closed loop control algorithms

### 2.5.1 Open Loop Control

Open-loop algorithms prevent congestion from occurring by making sure that the traffic flow generated by the source will not degrade the performance of the network below the specified QoS. If the QoS cannot be guaranteed, the network has to reject the traffic flow. The function that makes the decision to accept or reject the traffic flow is usually called an admission control. Thus open-loop algorithms involve some type of resource reservation.

Open loop congestion control does not rely on feedback information to regulate the traffic flow. Thus this technique assumes that once a source is accepted, its traffic flow will not overload the network.

Open loop congestion control does not rely on feedback information to regulate the traffic flow. Thus this technique assumes that once a source is accepted, its traffic flow will not overload the network.

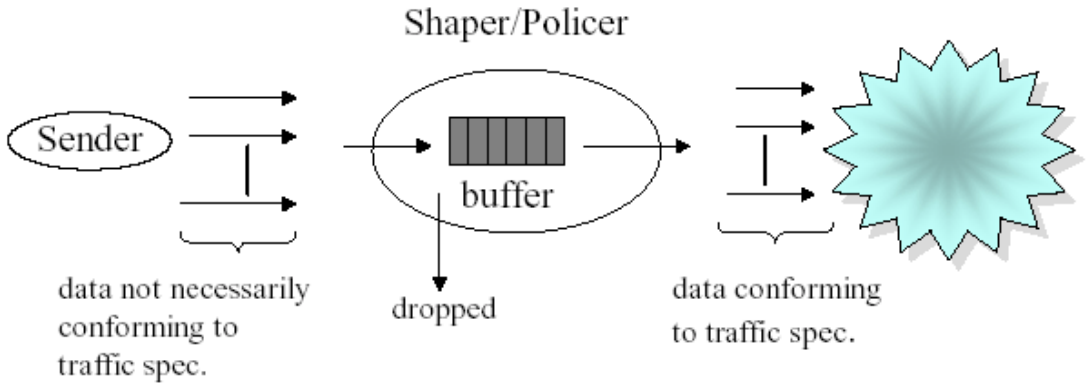
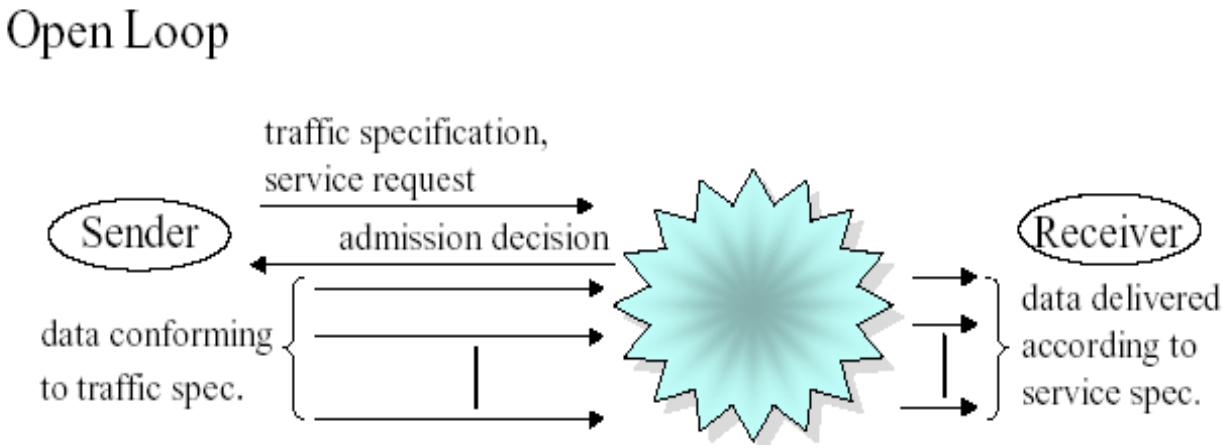



Fig 9. Demonstration of open loop

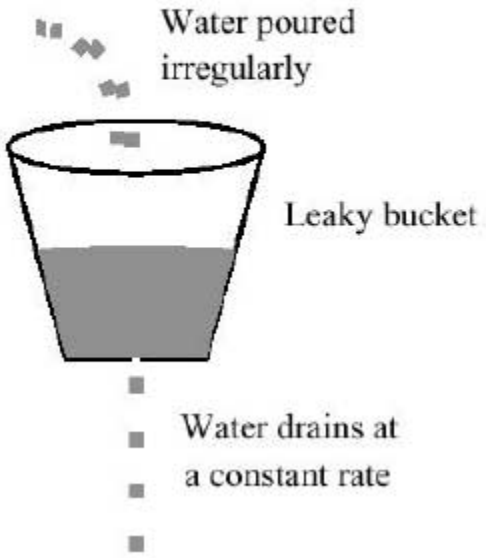



For **delay-sensitive, multi-media traffic** in high-speed networks (e.g. ATM), for which feedback control of congestion not feasible.
May be used in the Internet architecture (traditionally “best-effort”) to provide high quality service (“quality of service” or QoS guarantee).

### 2.5.1.1 Admission control

Admission control is an open loop preventive congestion control scheme. Admission control typically works at the connection level but can also work at the burst level. The analogy of a connection in datagram networks is a flow. At

**Leaky Bucket Policing**




connection level the function is called a connection admission control (CAC). At the burst level, it is called a burst admission control. The main idea of CAC is very simple. When a source requests a connection setup, CAC has to decide whether to accept or reject the connection. If the QOS of all the sources (including the new one) that share the same path is satisfied, the connection is accepted otherwise it is rejected. The QoS can be expressed in terms of maximum delay, loss probability, delay variance etc.,

For determining QOS requirements CAC has to know the traffic flow of each source. Thus each source specifies a set of parameters called traffic parameters called the traffic descriptors. A traffic descriptor may contain peak rate, average rate, maximum burst size, and so on and is supposed to summarize the traffic flow compactly and accurately.
Based on the characteristics of the traffic flow, CAC has to decide how much bandwidth it has to reserve for source. The amount of bandwidth typically lies between peak rate and average rate and is called effective bandwidth of the source.

### 2.5.1.2 Policing

The process of monitoring and enforcing the traffic flow is called traffic policing. When the traffic violets the agreed-upon the contracts, the network may choose to discard or tag the nonconforming traffic. The tagged traffic will be carried by the network but will be given lower priority. If there is any traffic downstream, the tagged traffic is the first one to be lost.

The process of monitoring and enforcing the traffic flow is called traffic policing. When the traffic violets the agreed-upon the contracts, the network may choose to discard or tag the nonconforming traffic. The tagged traffic will be carried by the network but will be given lower priority. If there is any traffic downstream, the tagged traffic is the first one to be lost.

Check if a packet stream obeys its descriptor, and if it violates its descriptor, give penalty!
– Drop packets that violate the descriptor
– Give low priority to them

• Leaky and token buckets are widely used policing techniques
– They can monitor average (and sustainable) rate, peak rate and burst size.

## Flow control

• Sliding Window Flow Control

Let X, t0 and W be the single packet transmission time, the time required for a packet to be acknowledged and the window size (maximum number of outstanding packets) respectively
The throughput is given by □=min (1/X, W/t0) packets/unit time, assuming that the packet loss probability is negligible.
Transmission speed can be indirectly controlled by changing the window size W.

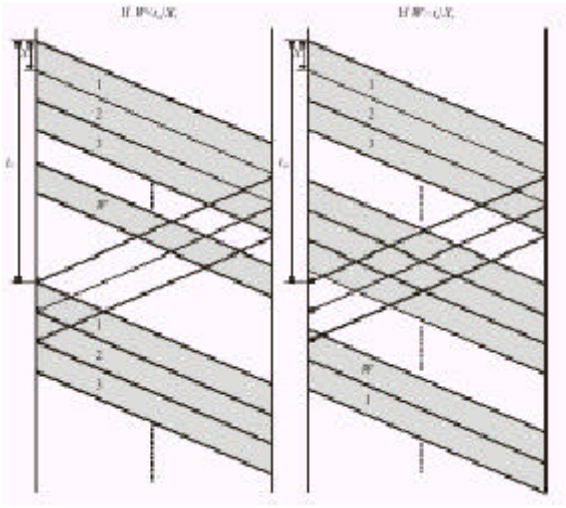


Fig 12. Flow control

**Flow control with explicit feedback**

• A packet may collect the state information along the path it is traveling, and the information can be feed backed to the transmitter.
– ABR congestion control for ATM

• Special cells (packets) called Resource Management (RM) cells collect the network state information, and are sent back to the transmitter

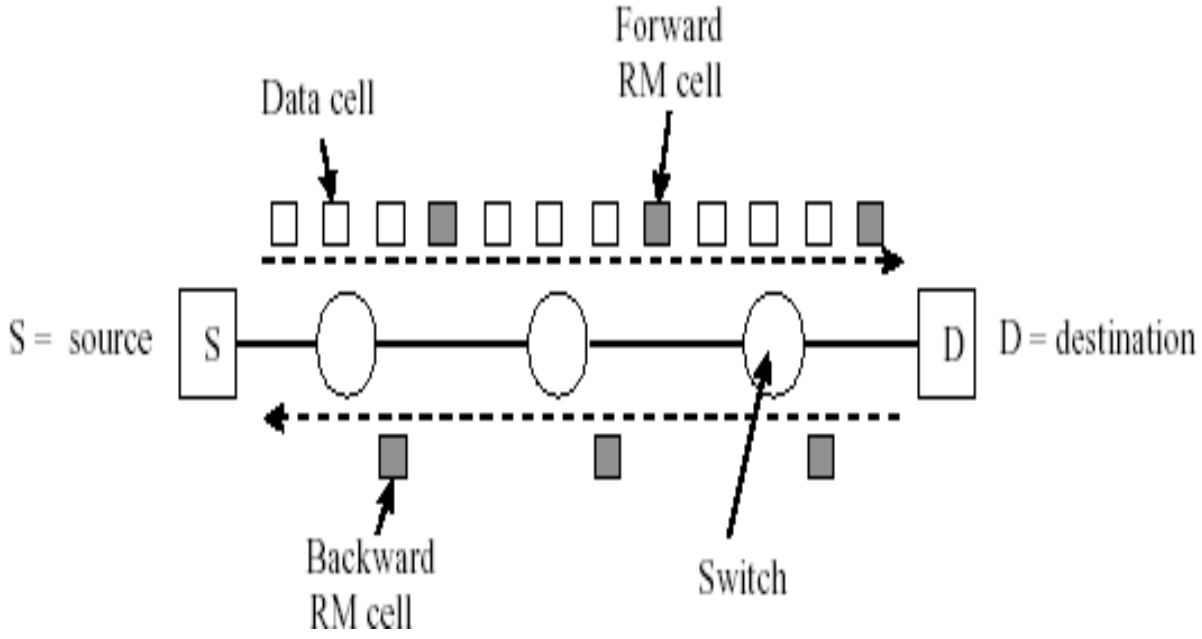



Fig 13. Flow control

### 2.5.2 Closed Loop Controls

Closed loop algorithms, on the other hand, react to congestion when it is already happening or is about to happen, typically by regulating the traffic flow according to the state of the network. These algorithms are called closed loop because the state of the network has to be fed back to the point that regulates the traffic, which is usually the source. Closed loop algorithms typically do not use any reservation.

The Active Congestion Control project is applying Active Networking techniques to feedback congestion control. Feedback congestion control is a very effective system for sharing network bandwidth when the bandwidth delay product of the network is low, but loses its effectiveness in high bandwidth-delay networks. Using Active Networking techniques ACC seeks to increase the range over which feedback is effective. ACC allows internal network nodes to take action immediately in times of congestion, as opposed to endpoint congestion control systems that require all action to be taken at endpoints. It takes time for an endpoint to deduce that there is a problem and to take corrective action. By taking action at the congested node, ACC avoids that delay.

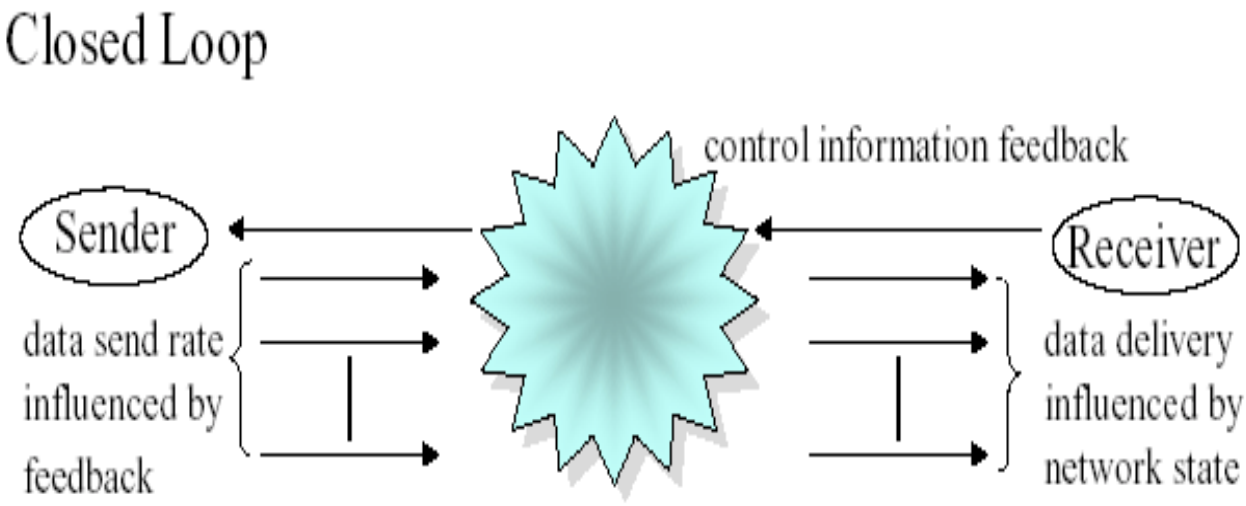



Fig 14. Closed Loop

## Intelligent Load Shedding

Discarding packets does not need to be done randomly

Router should take other information into account

Possibilities:

Total packet dropping

Priority discarding

Age biased discarding

## Total Packet Dropping

When the buffer fills and a packet segment is dropped, drop all the rest of the segments from that packet, since they will be useless anyway Only works with routers that segment and reassemble packets

## Priority Discarding

Sources specify the priority of their packets. When a packet is discarded, the router chooses a low priority packet Requires hosts to participate by labeling their packets with priority levels

## Age Biased Discarding

When the router has to discard a packet, it chooses the oldest one in its buffer. This works well for multimedia traffic, which requires short delays. This may not work so well for data traffic, since more packets will need to be etransmitted

## Random Early Detection

TCP detects packet loss and slows the sending rate accordingly. When the router queues start to fill, randomly drop some packets

## Choke Packets

Each router monitors the utilization of each of its output lines. Associated with each line is a variable u, which reflects the utilization of that line. Whenever u moves above a given threshold value, the output line enters a “warning state” Each newly arriving packet checks if its output line is in the warning state. If so, the router sends a choke packet back to the source .The data packet is tagged (by setting a bit in its header) so that it will not generate any more choke packets at downstream routers. When the source host receives the choke packet, it is required to reduce its traffic generation rate to the specified destination by X% Since other packets aimed at the same destination are probably already on their way to the congested location, the source host should ignore choke packets for that destination for a fixed time interval. After that, it resumes its response to choke packets.

## Choke Packets: Example

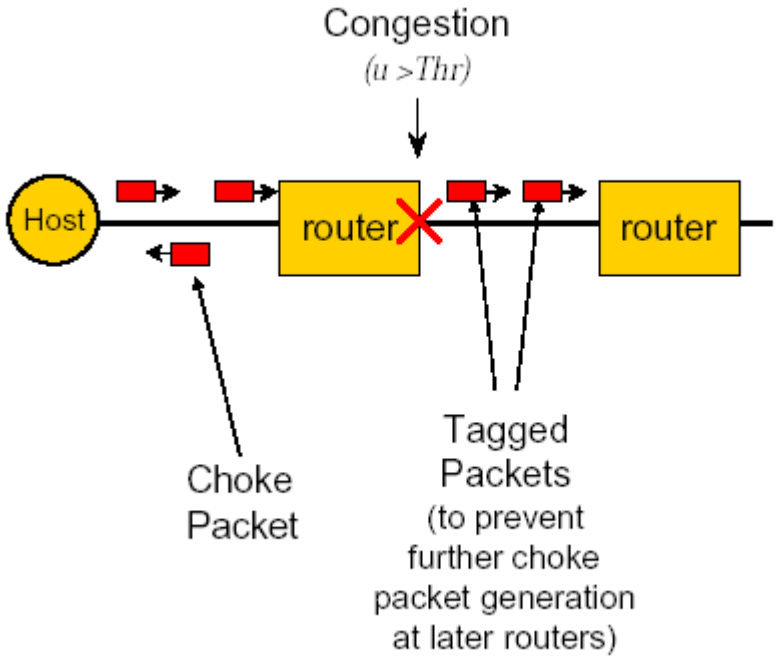



Fig 15. Choke Packets

- **Open-loop versus Closed-loop**

– Open-loop: no feedback from the network or destination
– Closed-loop: explicit or implicit feedback from the network or destination

- **Rate versus Window**

– Rate control: directly controls the transmission rate at
the source
– Window size control: indirect controls the transmission rate by changing the window size (outstanding number of packets or bytes)

## 2.6 Why Do We Need Congestion Control?

The assumption that statistical multiplexing can be used to improve the link utilization is that the users do not take their peak rate values simultaneously. But since the traffic demands are stochastic and cannot be predicted, congestion is unavoidable. Whenever the total input rate is greater than the output link capacity, congestion happens. Under a congestion situation, the queue length may become very large in a short time, resulting in buffer overflow and cell loss. So congestion control is necessary to ensure that users get the negotiated QoS. The final objectives of any sort of traffic control and congestion control are: to support a set of QoS parameters and classes for all network services offered and minimize network and end-system complexity while maximizing network utilization.

## 2.7 Quality of Service (QoS)

A set of parameters is negotiated when a connection is set up on any networks. These parameters are used to measure the Quality of Service (QoS) of a connection and quantify end-to-end network performance at data link layer. The network should guarantee the QoS by meeting certain values of these parameters. These parameters may include mean delay, mean delay variance, jitter delay (in case of video), throughput etc.

The above discussion leads to an important conclusion: Different kinds of network traffic require different service characteristics from the network. These service characteristics52 may be summarized in three critical parameters of which two are illustrated in Figure 87.

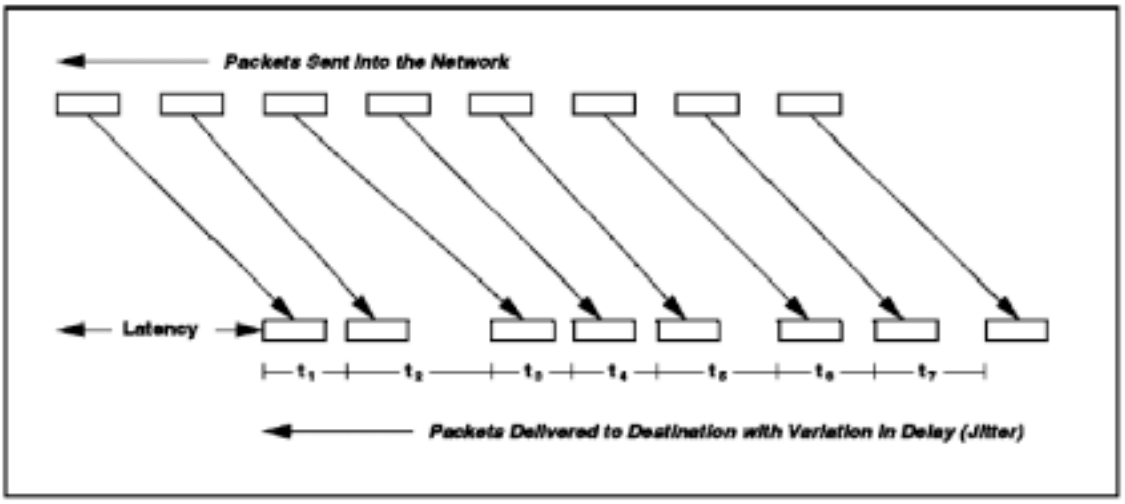



Fig 16. Latency and Jitter

**Latency** is the delay in time between when the stream is transmitted and when it is presented to the end user. This is more than propagation delay because of staging delay within transit nodes, the need for buffering, etc. at the end-user device.

**Jitter** is variation in latency over time. This causes erratic presentation of information to the end user. When you introduce buffering in the receiver to smooth out the presentation, then the presence of the buffers increases the network latency.

**Skew** is the difference in time of presentation to the end user of related things (such as a video of someone speaking and the related sound). This is the critical problem for m **Overrun and Underrun** are perhaps not predominantly network issues. This is where the video or voice signal is generated at a different rate from the rate at which it is played out. In the case of overrun, information is generated faster than it can be displayed and, at some point, information must be discarded. Underrun is where the playout rate is greater than the rate of signal generation and therefore “glitches” will occur when data must be presented but none is there.

In order to avoid these effects you need to provide end-to-end network synchronization. This involves propagating a network clock throughout the ATM network. The importance of each of these factors varies with the application, but skew is both the most important for the multimedia application and the greatest challenge for the network (and incidentally for the workstation itself).

### Interactive Applications

Applications such as videoconferencing (personal or group) have the same requirements as regular voice. That is, a maximum latency of about 150 ms is tolerable. Jitter must be contained to within limits that the system can remove without the user knowing (perhaps 20 ms is tolerable). Skew (between audio and video) should be such that the audio is between 20 ms ahead and 120 ms behind the video.

### One-Way Video Distribution

In this application a delay of several seconds between sender and receiver is quite acceptable in many situations. This largely depends on whether the user expects to watch a two-hour movie or a 20-second animated segment in a training application. The delay really only matters because it is the time between the user requesting the information and when it starts being presented. For a movie, perhaps 30 seconds would be tolerable, but for a short segment, one second is perhaps the limit. Jitter and skew, however, have the same limits as the interactive applications above.

### Audio with Image Applications

These are applications such as illustrated lectures and voice-annotated text where still images are annotated by voice commentary. Depending on the application, latency may need to be less than 500 ms (between the request for the next image and its presentation), but the skew (audio behind the image) could be perhaps as long as a second or so.

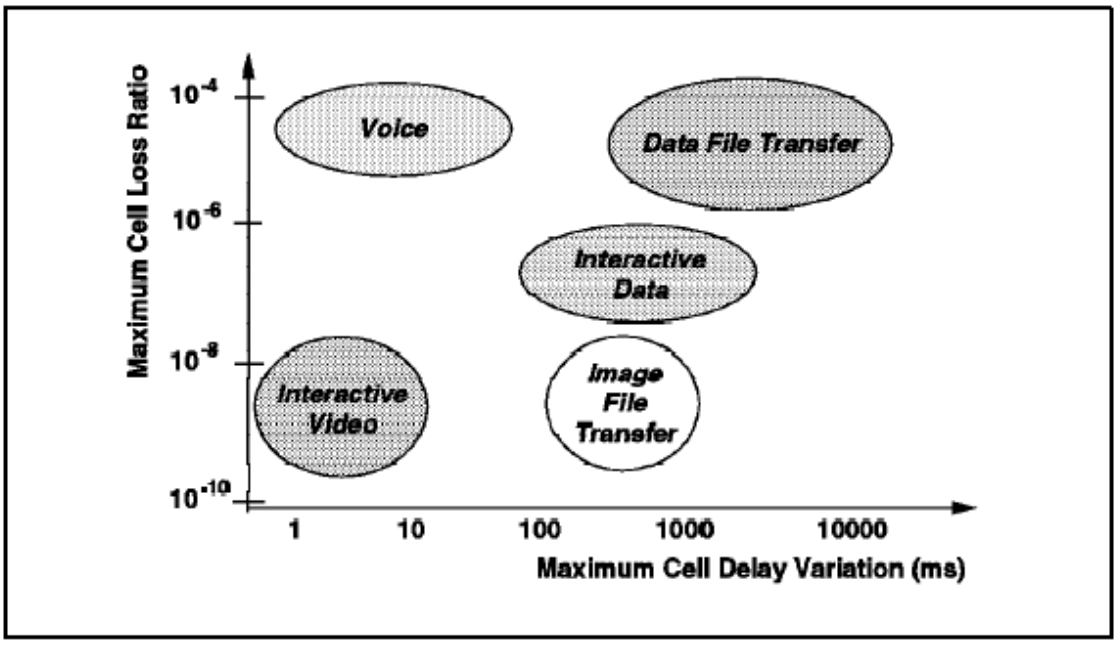



Fig 17. Jitter and Cell-Loss Tolerance of Some Application Types

**You need:**

1. Adequate (high) data rates (to keep latency low and to allow sufficient capacity to service the application)

2. Low latency

3. Very low jitter

4. Very low skew

5. End-to-end control through propagation of a stable clock

**Quality-of-Service Classes**

Quality of service in an ATM network is a concept that attempts to describe the important parameters of the network service provided to a given end user. These parameters include:

- End-to-end delay

- Delay variation (delay jitter)

- Cell loss ratio

This is a very difficult thing to guarantee in a network that handles any kind of bursty traffic. These characteristics vary quite widely with the load on the network. In some situations, tight control of these parameters is a major issue. Voice, video, and CBR traffic need to know these parameters so as to decide on the size of playout buffer queues, etc.

The following QoS classes have been defined in the standards:

**QoS Class 1 (Service Class A Performance Requirements)**

This QoS class applies to circuit emulation and constant bit rate traffic (CBR video and voice for example).53 These are to be set such that the result should provide performance comparable to current digital private lines.

**QoS Class 2 (Service Class B Performance Requirements)**

This is not yet fully defined but should provide suitable conditions for packetized video and audio in teleconferencing and multimedia applications.

**QoS Class 3 (Service Class C Performance Requirements)**

This is intended for interoperation of connection-oriented data services such as frame relay.

**QoS Class 4 (Service Class D Performance Requirements)**

This is intended for connectionless services such as LAN emulation, IP, or SMDS services.

A particular QoS class has specified performance parameters attached to it, and it may have two different cell loss ratios. A different cell loss ratio is appropriate for cells with CLP=0 to that for those cells with CLP=1. There is also a QoS class with no specified performance parameters (unspecified QoS class). Each connection may have its own unique QoS class attached to it.

Practical networks may support one or many specified QoS classes, as well as traffic with unspecified QoS.

**Traffic Management**

Service Categories

The ATM Forum has specified five “service categories” in relation to traffic management in an ATM network.

These categories are:

**Constant Bit Rate (CBR)**

CBR traffic includes anything where a continuous stream of bits at a predefined constant rate is transported through the network. This might be voice (compressed or not), circuit emulation (say the transport, unchanged, of a T1 or E1 circuit), or some kind of video. Typically you need both short transit delay and very low jitter in this service class.

**Real-Time Variable Bit Rate (rt-VBR)**

This is like CBR in the sense that we still want low transit delay but the traffic will vary in its data rate. We still require a guaranteed delivery service. The data here might be compressed video, compressed voice with silence suppression, or HDLC link emulation with idle removal.

**Non-Real-Time Variable Bit Rate (nrt-VBR)**

This is again a guaranteed delivery service where transit delay and jitter are perhaps less important than in the rt-VBR case. An example here might be MPEG-2 encoded video distribution. In this case, the information may be being retrieved from a disk and be one-way TV distribution. A network transit delay of even a few seconds is not a problem here. But we do want guaranteed service because the loss of a cell in compressed video has quite a severe effect on the quality of the connection.

**Unspecified Bit Rate**

The UBR service is for “best effort” delivery of data. It is also a way of allowing for proprietary internal network controls. A switch using its own (non-standard) internal flow controls should offer the service as UBR class. You send data on a UBR connection into the network and if there is any congestion in any resource, then the network will throw your data away. In many cases, with appropriate end-to-end error recovery protocols this may be quite acceptable. This should be workable for many if not most traditional data applications such as LAN emulation and IP transport.

**Available Bit Rate (ABR)**

The concept of ABR is to offer a guaranteed delivery service (with minimal cell loss) to users who can tolerate a widely varying throughput rate. The idea is to use whatever bandwidth is available in the running network after other traffic

utilizing guaranteed bandwidth services has been serviced. One statement [2] of the primary goal of the ABR service is for “the economical support of applications with vague requirements for throughputs and delays”.
In an operational network, there may be bandwidth “allocated” to a particular user but in fact going unused at this particular instant in time. Either by providing feedback from the network to the sender or by monitoring the network′s behavior, the ABR service can change the bit rate of the connection dynamically as network conditions change. The end-user system must be able to obey the ABR protocol and to modify its sending rate accordingly. Many people believe that ABR service requires the use of complex flow and congestion controls within the network. Others disagree very strongly.

## 2.8 Smoothness of a General Stream

A generalized stream of is defined to be ($n_1$,$T_1$:$n_2$,$T_2$;….$n_k$,$T_k$) smooth if, over any time period of duration $T_1$, number of packets <= $n_1$, over any time period of duration $T_2$number of packets <= $n_2$, over any time period of duration $T_k$, number of packets <= $n_k$, where k is denoting the no. of windows for characterizing the smoothness of the stream.

## 2.9 What is traffic shaping?

### Traffic Characteristics

In ATM we wish to integrate many kinds of network traffic onto the same network and share the network′s facilities between them. Each type of network traffic has its own peculiar characteristics and, therefore, needs to be treated differently from the others.

The traffic types may be summarized as follows:

- Traditional data traffic
- Voice and high-quality sound
- Full-motion video and interactive multimedia

Traditional data networks were built to handle both interactive and batch data but were not built to handle image, voice, or video traffic. The new types of traffic put a completely new set of requirements onto the network.

### Throughput Demand

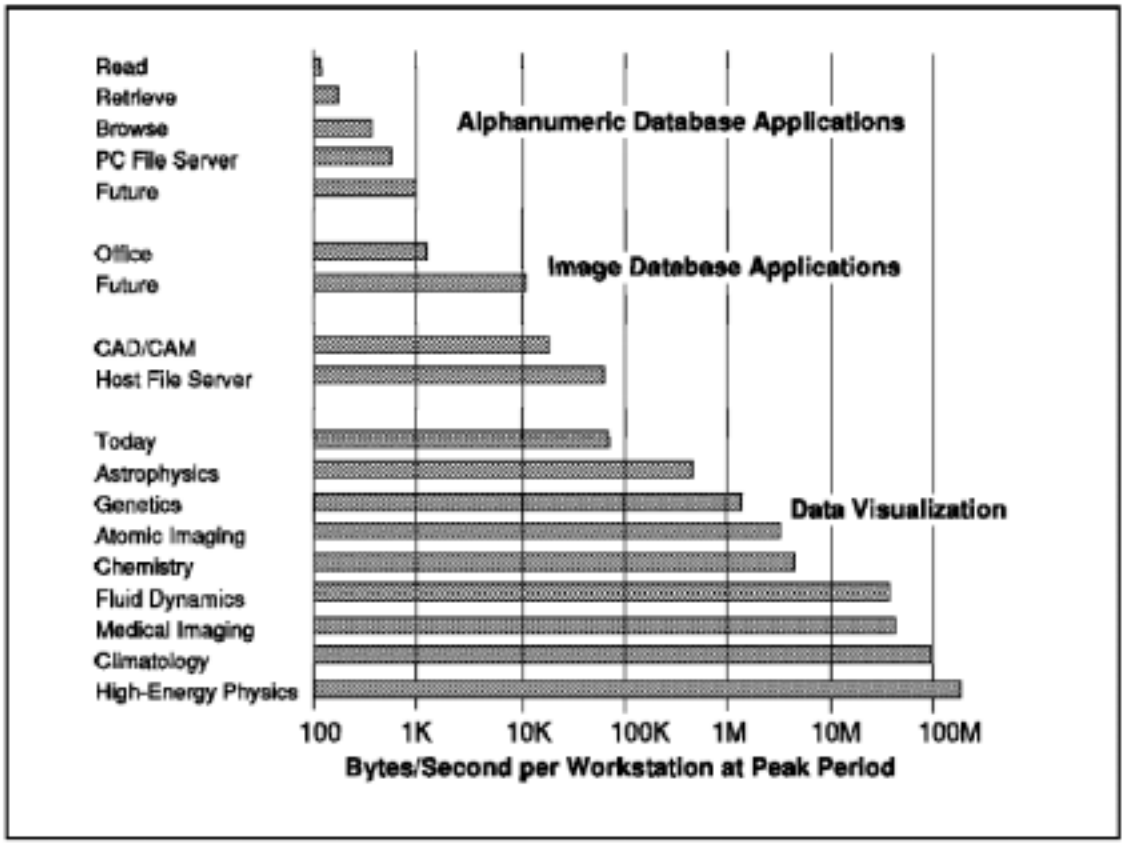



Fig 18. Application Throughput Requirements

One of the most important factors in considering traffic is the amount of throughput required. Some of the computer applications for high-speed communications can be seen easily from Figure 82. (Note that here the x-axis is using a logarithmic scale.)

Traffic shaping is a mechanism that forces the traffic to conform to a certain specified behavior. Usually, the specified behavior is a worst case or a worst case plus average case (i.e. at worst, this application will generate 100 Mbits/s of data for a maximum burst of 2 seconds and its average over any 10 second interval will be no more than 50 Mbits/s). It is about regulating the average rate (and burstiness) of data transmission. Traffic shaping reduces congestion and thus helps the carrier live up to its promise. Such agreements are not so important for file transfers but are of great importance for real-time data, such as audio and video connection, which do not tolerate congestion well.

By traffic shaping, we could achieve better network efficiency while meeting the QoS objectives (such as the smallest cell-loss, etc.). Meanwhile, traffic shaping ensures conformance at a subsequent interface. So it could reduce congestion by forcing the packets to be transmitted at a more predictable rate.

## Desirable Properties of a Traffic Shaper

The traffic envelope it enforces on the following property

- It should be simple to implement and easy to police
- It should be able to capture a wide range of traffic characteristics: difficult
- Example
- Traffic envelope captures the characteristics of the original source as close as possible

– Peak rate approximation of the source: no delay in the shaper but network underutilization

– Average rate approximation of the source: high network utilization but higher shaper delay

u Traffic envelopes enforced by a single LB, MW, JW are too simple for an accurate characterization of bursty sources. -> can do better multiple shapers to shape a traffic source

## Composite Shapers

- Shapers: enforce a specific rate constraint on a source; a declared peak or average rate

- Most applications: generate bursty traffic

– Enforcing average rate -> higher delay in shaper buffer

  – peak rate enforcement -> over allocation of system resource

- To solve the problem: multiple shapers to enforce multiple rate constraints

– enforce a traffic envelope that is close to the original shape of the traffic

– simple to specify and monitor

- Example:

– dual moving window: w1 = 4$\tau$, m1=|I| + |P| +2|B|, w2 = $\tau$, m2=|I|

» first shaper: enforces the longer term average

» second shaper: controls the short term peak rate

  – dual moving window: w1 = 4$\tau$, m1=|I| + |P| +2|B|, w2 = 2$\tau$, m2=|I| + |B|, w3 = $\tau$, m3=|I|

**Composite Leaky Bucket**

– Worst case behavior of a Leaky Bucket: traffic envelope that starts with a burst that is
equal to the bucket size, followed by a straight line of slope equal to the rate of token
generation.

Example:

» LB4: redundant component

» essential set: (b1, t1), (b2, t2), … .(bn, tn), bi>bj, ti>tj, for I>j

– n component composite LB
» Bk = ∞ k=0
» Bk = ⌊ (bktk - bk+1*tk+1) / (tk - tk+1) ⌋ k = 1, 2, … .. n
» Bk = 0 k = n+1
» a(I) = ∑(I-bk+1)tk[U(I-Bk) - U(I-Bk-1)], I=1, 1, … . ∞
» CI = (1/tk) * tk + bk

**Composite Moving window**

Example:
» MW1 = (w1, m1), MW2 = (w2, m2), MW3=(w3,m3) where w1 = 3 x w2, w2 = 4 x w3,
m1 = 2x m2, m2 = 2 x m3
» Moving window MW2 determines the burst size distribution within w1
– n component composite MW
» (wk, mk), k=1,… n where wi > wj, mi > mj and mi/wi < mj/wj, for 1 < I < j < n
» a(I) = ∑( ⌊ i / mk ⌋ - ⌊ i / mk-1 ⌋ * (mk-1 / mk))wk, I=1, 1, … . ∞

**Composite Jumping window**

– Example:
» JW1 = (w1, m1), JW2 = (w2, m2), JW3=(w3,m3) where w1 = 3 x w2, w2 = 4 x w3,
m1 = 2x m2, m2 = 2 x m3
» In a single moving window shaper, two full-size bursts are always separated by at least one window length. In a jumping window shaper, two bursts can appear next to each other.

- component composite MW

» (wk, mk), k=1,… n where wi > wj, mi > mj and mi/wi < mj/wj, for 1 < I < j < n
» a(I) = ∑( ⌊ (i / mk)+1 ⌋ - ⌊( i / mk-1)+1 ⌋ * (mk-1 / mk))wk, 0 < I < m1
» a(I) = ∑( ⌊ i / mk⌋ - ⌊ i / mk-1 ⌋ * (mk-1 / mk))wk, m1 < I < ∞

## Shaping and BW Allocation

- Larger token arrival rate reduces the access delay at the policer:

– but needs a larger bandwidth allocation
– (λp / r) < λt < λbw < λp

- λo = ∑λbw(I) for m streams multiplexed to the same output;

– large statistical multiplexing gain is possible only if λt is near the average arrival rate λa = (λp / r)

- Small λt means larger access delay and/or violation probability incurred by the source

- Trade-off between the access delay introduced by the policer and the network delay; (lenient enforcement policy increase the delay at the switching node)
- Effect of input rate control

– Total delay = access delay + network delay. The policer transfers the network delay on to the input side, thereby avoiding overflow losses/delays within the network. Unless the source has a large buffer and can tolerate excess delay (many RT application not), the
input rate control as performed by the LB can hardly improve the network performance
– Stringent input rate control may increase the user end to end delay
– The minimum total average delay is achieved when no traffic enforcement is invoked.
Network bandwidth is greater than the source rate: smoothed by statistical mixing.

Nevertheless, input policer is needed to check excessive burstiness & rate violation

- Reducing the access delay; more short term burstiness subject to

**– Max. burst size should be bounded and burst arrival must be peak rate enforced**
– Number of arrivals over a larger time duration to be bounded at the average policing rate

- LB: token buffer b . OA: average policing rate
- EWMA: dynamic response, implementation complexity

• Similar to policing, but done at user's side

• Not to violate the traffic descriptor agreed upon
– Pass the packet stream through a traffic shaper before they are actually transmitted to the network (or the policing unit)

• Implementations of traffic shaping
– Leaky Bucket Traffic Shaper

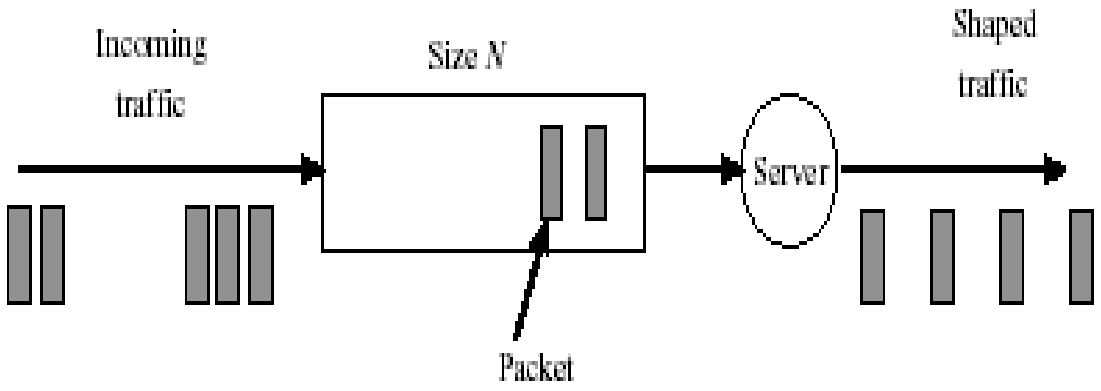
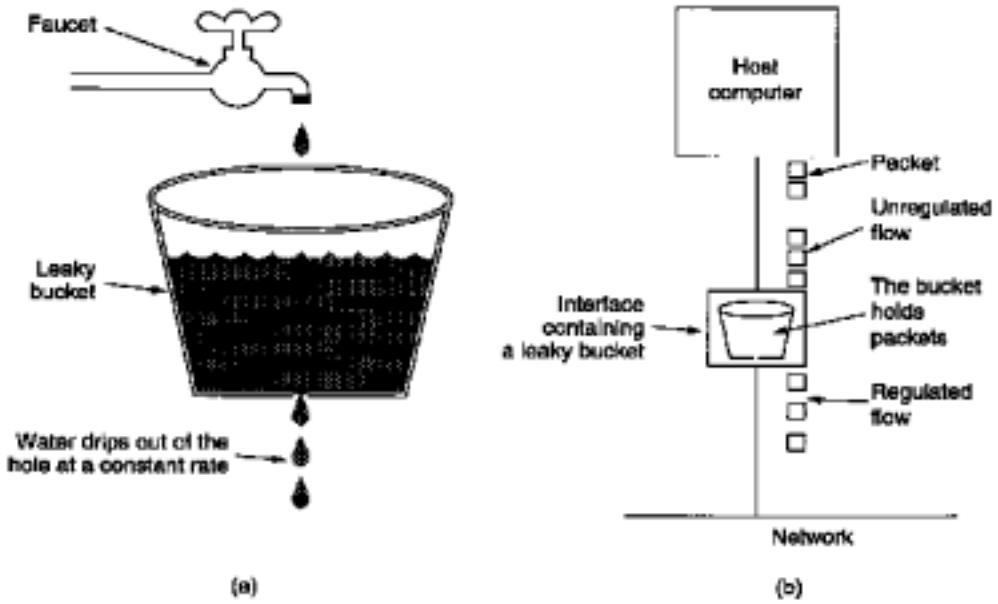



Fig 19. Congestion diagram for slots

Then, what is traffic shaping? Traffic shaping is a mechanism that forces the traffic to conform to a certain specified behavior. Usually, the specified behavior is a worst case or a worst case plus average case (i.e. at worst, this application will generate 100 Mbits/s of data for a maximum burst of 2 seconds and its average over any 10 second interval will be no more than 50 Mbits/s). It is about regulating the average rate (and burstiness) of data transmission. Traffic shaping reduces congestion and thus helps the carrier live up to its promise. Such agreements are not so important for file transfers but are of great importance for real-time data, such as audio and video connection, which do not tolerate congestion well.

## 2.10 Leaky Bucket Algorithm

Most implementations of traffic policing use the leaky bucket algorithm. To understand how a leaky bucket can be used as a policing device, imagine the traffic flows to a policing device as water being poured into a bucket that has a hole at the bottom. The bucket has a certain depth and leaks at the constant rate when it is not empty. A new container (that is, packet) of water is said to be conforming if the bucket does not overflow when the water is poured in the bucket the bucket will spill over if the amount of water in the container is too large or if the bucket is nearly full from prior containers. The bucket depth is used to absorb the irregularities in the water flow. If we expect the flow to be very smooth, then the bucket can be made very shallow. If the flow is bursty, the bucket should be deeper. The drain rate corresponds to the traffic rate that we want to police.




(a) A leaky bucket with water. (b) A leaky bucket with packets.

Fig 20. Leaky Bucket

Basically leaky bucket can be divided in two parts: -

- First is the host is allowed to put one packet per clock tick onto the network.
- The second is the leaky bucket holds tokens generated by a clock at the rate of one token every delta t seconds. For a
  Packet to be transmitted, it must capture and destroy one token. This is mostly used to allow saving upto maximum size of bucket, n. This means that burst of upto n packets can be sent at once, allowing some burstiness in the output stream and giving fast response to sudden burst of data.

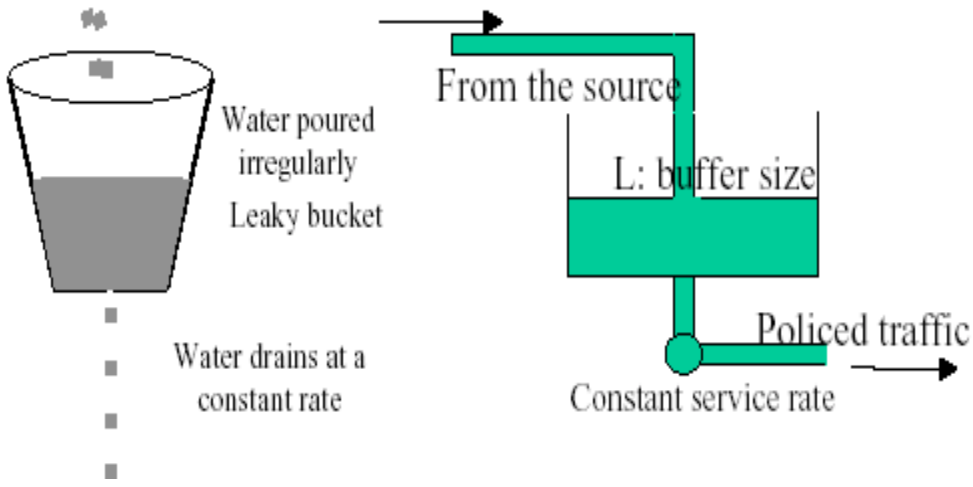



Fig21. Demonstrations of flow in LB

• If packets arrive faster than the rate □̣for a long period, the buffer will overflow and packets causing the overflow will be penalized

• A traffic source can transmit packets at an average rate not larger than □̣and burst transmission of packets is allowed only temporarily
– □̣enforces the long-term transmission rate
– L determines the maximum size of the burst packet arrivals.

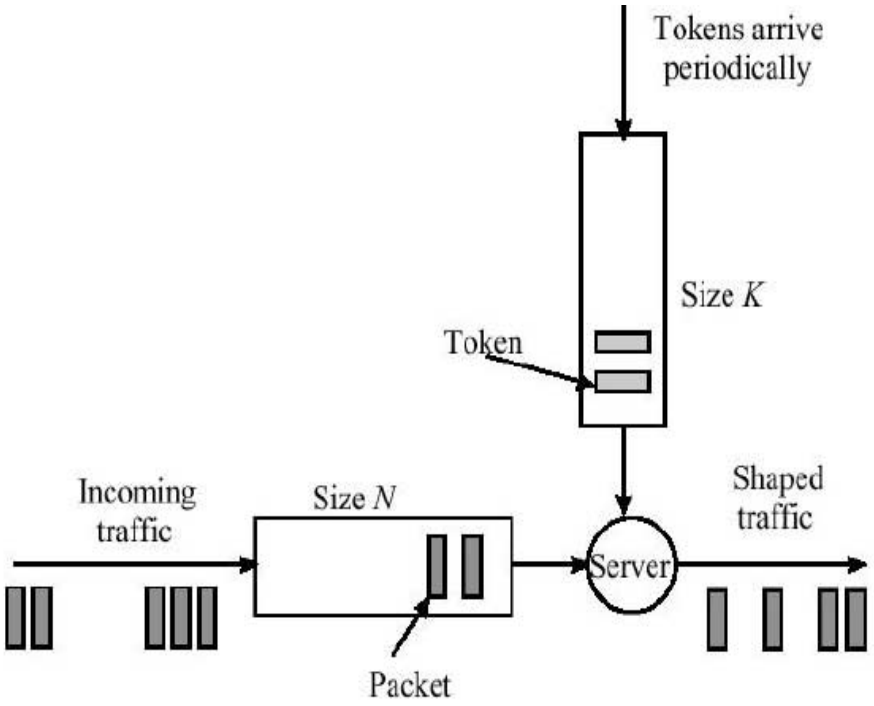



Fig 22. (Leaky) Token Bucket Traffic Shaper

Let b = token bucket size in bytes, r = token rate in bytes/s
Then in any *time interval* of length T, the maximum number of bytes out of the token bucket shaper is **(b + Tr)** bytes. Token-bucket shaped traffic will undergo **no drops and a maximum delay of no more than b/R (end-to-end)** if network nodes transmit at a rate of R>r bytes/s and have at least b bytes of buffer.
If network nodes use **weighted fair queuing** then R is the guaranteed rate that the flow gets at a node. In practice, if the WFQ is implemented on max. size packets of size M bytes, additional max. delays of M/R may occur at each node. Trans. delays will also accumulate over a multi-hop network.
A node can check if traffic conforms to a leaky token bucket flow by

passing it through a leaky token bucket with the same parameters for its token bucket operation (r and b). If the flow is conforming, it undergoes 0 delay.

The following algorithm will summarize the LB. At the arrival of the first packet, the content of the bucket X is set to 0 and the last conforming time (LCT) is set to the arrival time of the 1st packet. The depth of the bucket is L+I, where L depends on the maximum burst size. At the arrival of $k_{th}$ packet, the auxiliary variable X` records the difference between bucket counter at arrival of LCT and the inter arrival time between LCT and the $k_{th}$ packet. If auxiliary variable is greater than L, the packet is non conforming else packet is confirming. The bucket content and arrival time of the packet are then updated.

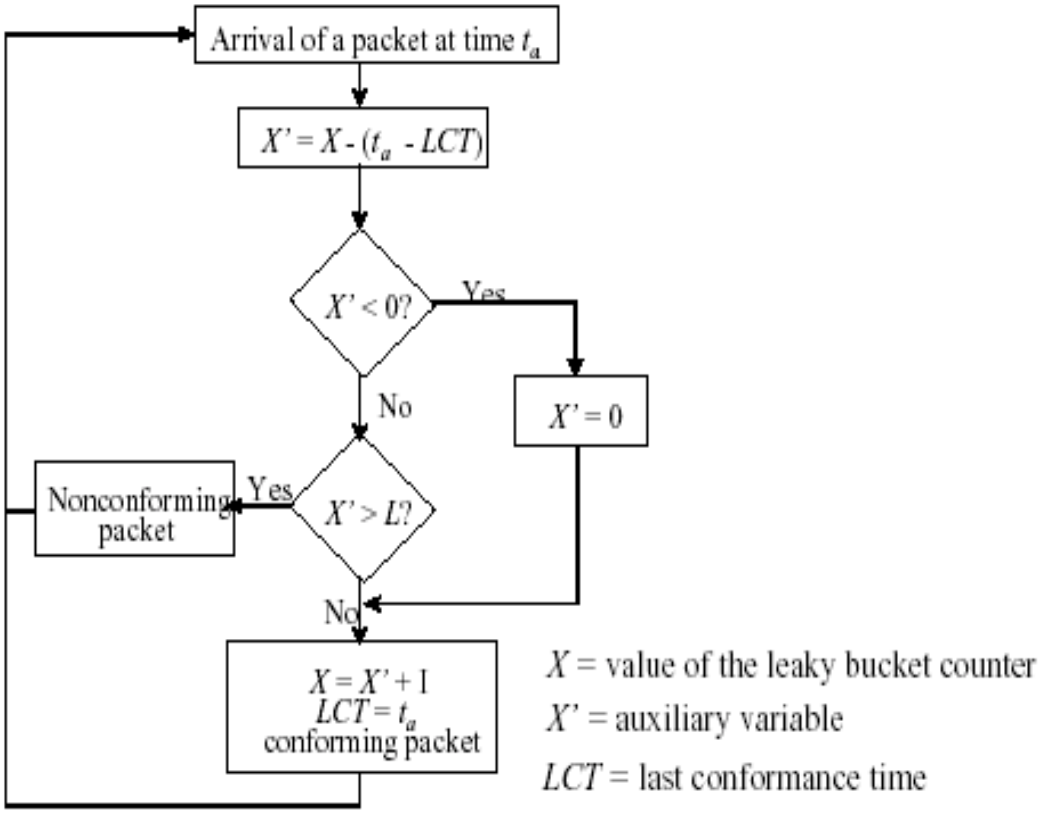



Fig23. General Algorithm of Leaky Bucket

As an example of a leaky bucket, imagine that a computer can produce data at 25 MB/sec (200 Mbps) and that the network also runs at this speed. However, the router can handle this data rate only for short intervals. For long intervals, they work best at rates not exceeding 2 MB/sec. Now suppose data comes in 1-million-byte bursts, one 40-msec burst every second. To reduce the average rate to 2 MB/sec, we could use a leaky bucket with $\rho$ = 2 MB/sec and a capacity, C, of 1 MB. This means that bursts of up to 1 MB can be handled without data loss, and that such bursts are spread out over 500 msec, no matter how fast they come in.

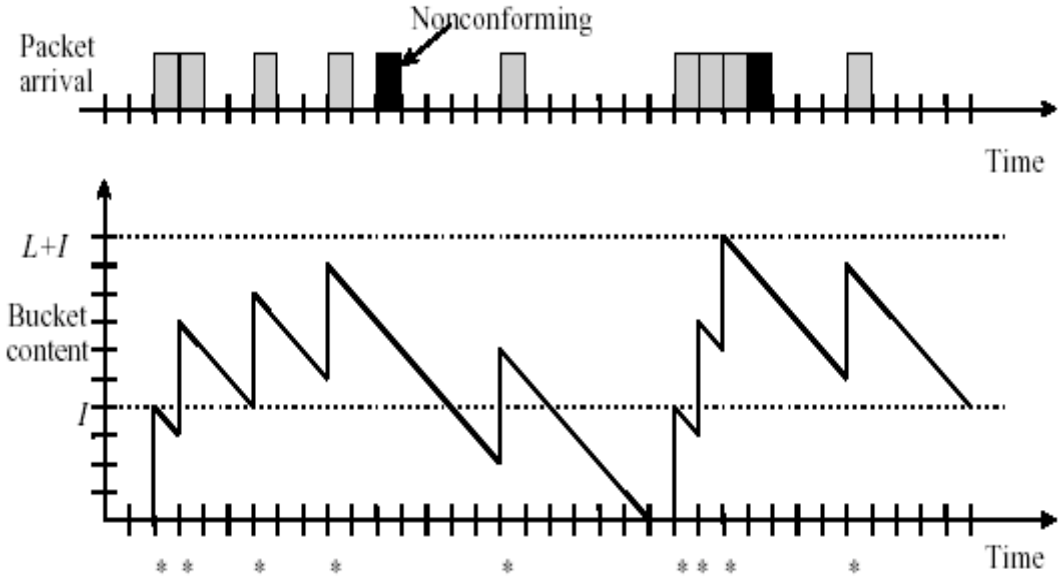



Fig 24. Behavior of leaky bucket

## 2.11 Exponentially Weighted Moving Average Scheme

EWMA is windows based scheme where the maximum number of cells permitted within a fixed time window is limited. If we consider connection time to consist of consequent windows of same size, the maximum number of cells accepted in the $i_{th}$ window Ni is a function of the mean number of the cells per window N and an exponentially weighted sum of the cells accepted in the preceding windows is

$$N_i = ((N - (1-\lambda)(\lambda N_i\text{-}1 + \ldots\ldots + \lambda_{i-1}N_i)) - \lambda S_o) / (1-\lambda)$$

Where $S_o$ is the initial value for EWMA.

A no zero value of $\lambda$ permits more burstiness. Thus larger value of $\lambda$ increases reaction time and thus the dynamic behavior of EWMA is the worst.

# Chapter 3. Window Based Traffic Shaper (Functional description)

## 3.1 Leaky Bucket Scheme

In the generalized scheme as shown n figure, tokens are generated at a fixed rate as long as the token buffer of size b is not full. When a packet size arrives from the source, it is released into the network if and only there is at least 1 token in the token buffer. This scheme enforces token arrival rate $\lambda_l$.

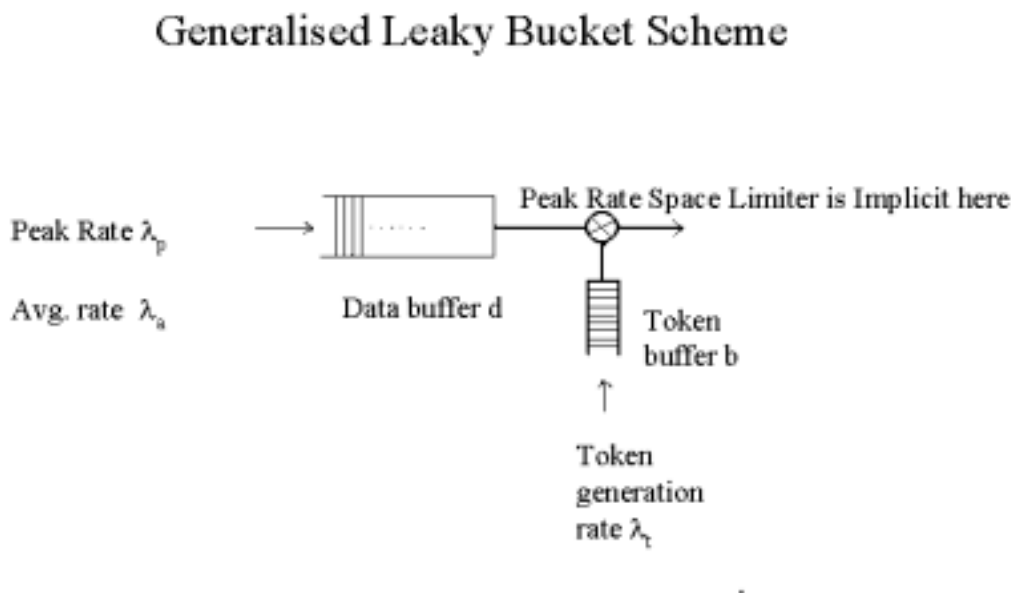



Fig 25. Functional view of Leaky Bucket

Clearly token generation rate should lie between avg. arrival rate and peak rate. An arriving packet finding the input buffer full is said to be violating packet and can be dropped. A space limiter is embedded into LB. When a burst of packets arrive at the source even if the token buffer is not empty, these packets are not transmitted immediately but are delayed by InterCellTime $\tau$ ( $\tau = 1/\lambda_p$). For the LB defined here, maximum burst size at output is $b` = b/(1- \lambda_l /\lambda_p)$.

## 3.2 Shift Register Traffic Shaper (SRTS)

One serious drawback in the leaky bucket algorithm is that number of packets in it over any time duration T starting from 0 is bounded by $\lambda_t$ token generation rate. So for both long burstiness as well as short burstiness it delays the traffic to its token generation rate and thus introduceses an appreciable amount of access delay at the node. This large amount of access delay is unacceptable for Real time traffic as well as Multimedia traffic. So we try to design a traffic shaper, which will have following features:

- It should permit short-term burstiness but bounds the long term burstiness.
- It should be able to incorporate variable burstiness up to a certain level.
- It is peak rate enforced
- It is a window-based shaper with two (initially) windows. More the no. of windows more will be flexibility.
- It is designed using a Shift register and two counters and hence can easily be implemented in hardware.

## 3.3 Description of new scheme (SRTS)

SRTS make use of the temporal profile (history) of packet stream admitted by the shaper over past N time slots (each slot = $\tau=1/\lambda_p$ ). This history is maintained by a Shift Register with 1 bit corresponding to every packet. The shift register is shifted to 1 bit every time slot $\tau$. The entry into the register is made as per following:

- A 1 is shifted when $f_d = 1$ and $f_a = 1$ .
- A 0 is shifted otherwise

Where,

$f_d$ = 1 if Data buffer is not empty and 0 otherwise and

$f_a$ denote the admit function defined as $f_a = (n(T_1)<n_1)$ & $(n(T_2)<n_2)$ .. depending on no. of windows

Thus the bit contents of the shift register at any instant give a snapshot of the packets send. To determine the no. of packets send in any time duration, a counter is used. It is incremented when a “1” is enters the shift register and decrements when a “1” is leaves the right edge of shift register.

Figure drawn below describes an enforcement scheme using two windows. This scheme generates an $(n_1,T_1;n_2,T_2)$ smooth traffic, which means

that over any period of time duration $T_1$, the number of packets $n(T_1) <= T_1$ , and over any period of time duration $T_2$, the number of packets $n(T_2) <= T_2$. Further flexibility is possible in moulding the burstiness using appropriate number of windows.

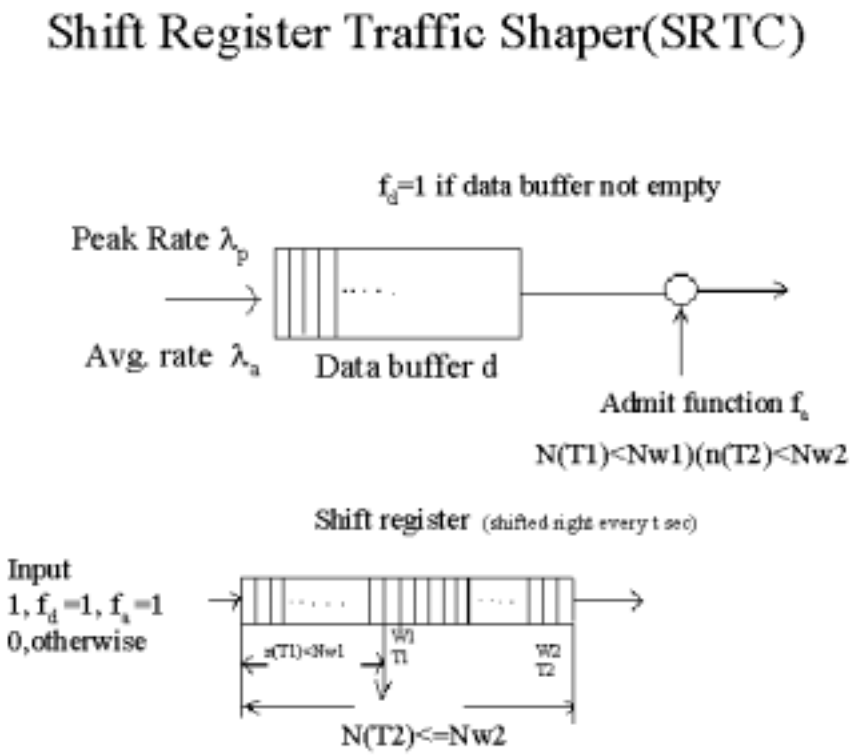



Fig 26. SRTS with 2 windows

One limitation that arises in the above arrangement is due to discretization of the time slots of $\tau$. A slot is termed active if a cell is transmitted into that slot and idle, otherwise. Since packet arrival need not synchronize with the packet transmission, the cell arriving in between the slot will have to wait till its end.

This limitation is removed by using soft discretization. If a slot arrives in an ideal slot , say after $\tau$` elapses out of $\tau$., idle slot is frozen and an active slot is

generated immediately. At the termination of this active slot if either the data is absent or the admit function is false, the residual slot of $\tau$-$\tau$` is completed.

This is illustrated as a Finite State Machine with two states Idle and Active in the following diagram.

The 3 window models also shown in the next figure.

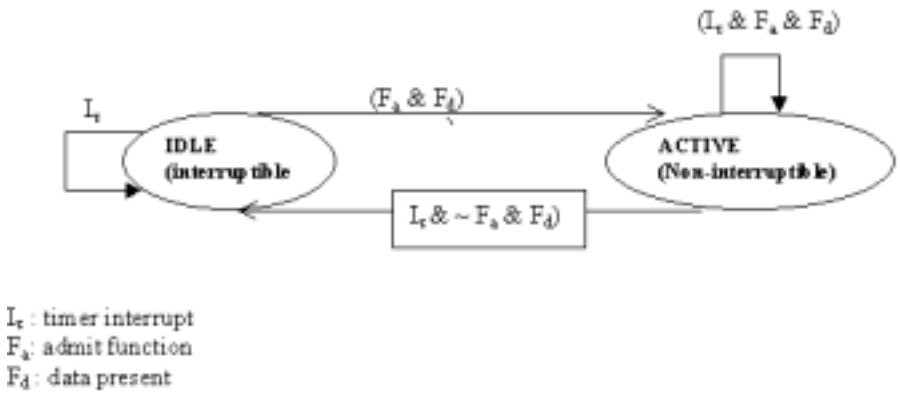
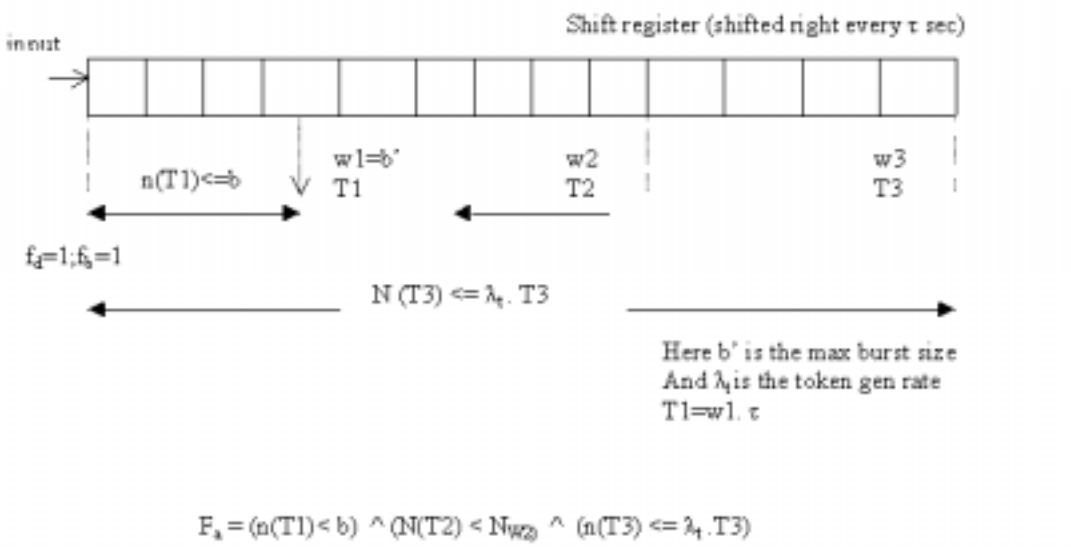



Fig 27. FSM for discretization of slots




$$F_a = (n(T1) < b) \ \wedge (N(T2) < N_{W2}) \ \wedge \ (n(T3) <= \lambda_t . T3)$$

Fig 28. Three window SRTS

# Chapter 4 Functional Details and Source Code

## Source model used for traffic Shaping

The source model used for measuring performance is the ON-OFF bursty model. The ON-OFF model is characterized by interspersed ON and OFF periods each exponentially distributed with mean $T_{ON}$ and $T_{OFF}$ respectively. During an On period, the packets are periodically transmitted with mean $T_{ON}$ at the rate of $\lambda_p$. The $_{average}$ rate $\lambda_a$ is

$$\lambda_a = \lambda_p . T_{ON} /( T_{ON} + T_{OFF})$$

And the burstiness is $R= ( T_{ON} + T_{OFF}) /T_{ON}$. The effective bandwidth requirement for this source $\lambda_{eff}$ is such that $\lambda_a <= \lambda_{eff} <= \lambda p$.

## Process Model used

Source is characterized by a peak rate $\lambda_p$ average rate $\lambda_a$ rate and mean ON duration $T_{ON}$. Packets are modeled on Poison distribution with controlling parameter as $\lambda_p$ for incoming traffic and $\lambda_s$ for outgoing traffic.

## 4.1 Source Code of module 1

```
////////////////////////////////////////////////// MODULE 1 ////////////////////////////////////////////////////////

/*****************************************************************************************/
/*********************************  Date : 12-4-2002 *********************************/
/**********************  Author's name: Nikhil Bhargava ***************************/
/*****************************************************************************************/

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

/*
   Simulation of simple network data layer showing
   congestion. We assume ideal channel with no loss during
   transmission. Probability of loss is 0.0. No Congestion
   control what so ever is applied. The sole aim of this
   program is to show the behaviour of packets lost under no
   congestion control scheme at the data link layer.
*/
```

```
// Header files included

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <dos.h>

#define TRUE 1
#define FALSE 0

#define BLACK -100.00
#define PAC_MAX 100           // Max number of packets
#define INPUT_QUEUE_SIZE 30        // Max data buffer size
#define LAMBDA_P 100               // Peak rate of traffic
#define LAMBDA_S 40           // Service rate of traffic
#define RANDOM_MAX 100

/*
  QOS chosen here to be ratio of packets rejected to
   packets produced. Weight of 2 out of     10 means 20 %
*/

#define QOS 0.2
#define PROCESS_DELAY 40      // 20 millisecond delay for
                              rejected packets

struct packet
{
     int packet_no;
     float time_reject;
     int reject_flag;
     int delay_flag;
     struct time time_stamp;
     struct packet *next;
};
struct packet *root,*rej;
struct packet *del;

int source_queue=0;

int packet_produced=0;
int packet_rejected=0;
int packet_send=0;

struct bt
```

```
{
     float birthtime;
     int head_no;
}birth[PAC_MAX];

struct dt
{
     float deathtime;
     int head_no;
}death[PAC_MAX];

float clk_time_gen=0.0;
float clk_time_send=0.0;

// function for generating the packet
     void gen_packet(void);

// function for transmitting the packet at destination end
     void send_packet(void);

// This function checks the performance of the system in
terms of congestion
     int performance_check(void);

void free_list();

float myrand();

void record_result(void);

int packets_match(int i,int o);

void main()
{
 struct packet *p=NULL;
 struct packet *r=NULL;

 int i=0;

/* Generate the root packet for reference of generated
packet linklist */

 p=(struct packet *)calloc(1,sizeof(struct packet));
 if(p==NULL)
 {
     printf("\n\aNOT ENOUGH MEMORY AVAILABLE!\n");
     getche();
```

```
    exit(0);
 }
 else
 {
    gettime(&(p->time_stamp));
    p->time_reject=BLACK;
    p->reject_flag=FALSE;
    p->delay_flag=FALSE;
    p->packet_no=0;
    p->next=NULL;
    root=p;
    p=p->next;
 }

/* Generate the root for rejected packet linklist */

 r=(struct packet *)calloc(1,sizeof(struct packet));
 if(r==NULL)
 {
    printf("\n\aNOT ENOUGH MEMORY AVAILABLE!\n");
    getche();
    free_list();
    exit(0);
 }
 else
 {
    gettime(&(r->time_stamp));
    r->time_reject=BLACK;
    r->reject_flag=TRUE;
    r->delay_flag=FALSE;
    r->packet_no=0;
    rej=r;
    r->next=NULL;
    r=r->next;
 }

 gen_packet();      // start with generation process
 while(performance_check()==TRUE)
 {
    if(clk_time_gen<clk_time_send)     gen_packet();
    else send_packet();
    //tck[i]=clk_time_gen<clk_time_send?clk_time_gen:clk_t
ime_send;
    //i++;
    if(packet_produced==PAC_MAX) break;
 }
 record_result();
```

```
 free_list();
}

void free_list()
{
 struct packet *dummy,*a;

/* For the list of generated packets */

 dummy=root->next;
 while (dummy->next!=NULL)
 {
     root->next=dummy->next;
     a=dummy;
     dummy=dummy->next;
     free(a);
 }
 if(dummy->next==NULL)
 {
     root->next=NULL;
     free(dummy);
     free(root);
 }

/* For the discarded packets list */

 dummy=rej->next;
 while (dummy->next!=NULL)
 {
     rej->next=dummy->next;
     a=dummy;
     dummy=dummy->next;
     free(a);
 }
 if(dummy->next==NULL)
 {
     rej->next=NULL;
     free(dummy);
     free(rej);
 }

}

void gen_packet()
{
 struct packet *p,*r,*d,*t;
 float x=0.0;
```

```
 static int num=1;

 p=root;
 r=rej;

 while (p->next!=NULL) p=p->next;
 //if(p->next==NULL) p=p->next;

 while (r->next!=NULL) r=r->next;
 //if(r->next==NULL) r=r->next;

/*
   Check the data buffer first
*/

 If(source_queue>=INPUT_QUEUE_SIZE)
 {
  /* The packet which is going to be produced will be
rejected */

     t=(struct packet *)calloc(1,sizeof(struct packet));
     if(t==NULL)
     {
          printf("\n\aNOT ENOUGH MEMORY AVAILABLE!\n");
          getche();
          free_list();
          exit(0);
     }
     else
     {
          gettime(&(t->time_stamp));
          t->reject_flag=TRUE;
          t->delay_flag=FALSE;
          t->time_reject=clk_time_gen+PROCESS_DELAY; // x/2
is arbit amount of time
          t->packet_no=num;
          num++;              // Increase the num for
header

          t->next=NULL;
          r->next=t;
          r=r->next;
          r->next=NULL;

          birth[packet_produced].birthtime=clk_time_gen;
          birth[packet_produced].head_no=r->packet_no;
```

```
          packet_rejected++;
          packet_produced++;

          /*
               Calculate time for the next packet to send
          */

          x=(float)myrand();
          x=1.00-x;
          x=1000*(log(1.0/x)/LAMBDA_P);
          clk_time_gen+=x;
     }
 }

 else          // Queue is not full
 {

     /* Check whether any rejected packet is scheduled to
come in the queue
        prior to this new packet.
     */

     /*
        If yes than enter the First rejected process and
continue to do this
        till spawn time of rejected process is < clk_spawn.
Then generate this
        packet and put it in rejected queue.
     */

     /*
        Else generate a new packet and store it in ready
queue
     */

     if(p!=root)
     {
          p=root->next;
          while(p->next!=NULL) p=p->next;
     }
     r=rej->next;

     if ((r!=NULL) && (r->time_reject<clk_time_gen))
     {
          while((r!=NULL) && (r->time_reject<clk_time_gen))
// scope of a min function
```

```
          {
               rej->next=r->next;
               r->next=NULL;

               p->next=r;
               p=p->next;
               p->next=NULL;       // Last node should be
NULL
               p->time_reject=0;
               p->reject_flag=FALSE;

               source_queue++;
               r=rej->next;
          }


          /* The packet which is going to be produced will
be rejected */

          while (r->next!=NULL) r=r->next;

          t=(struct packet *)calloc(1,sizeof(struct
packet));
          if(t==NULL)
          {
               printf("\n\aNOT ENOUGH MEMORY
AVAILABLE!\n");
               getche();
               free_list();
               exit(0);
          }
          else
          {
               gettime(&(t->time_stamp));
               t->reject_flag=TRUE;
               t->delay_flag=FALSE;
               t->time_reject=clk_time_gen+PROCESS_DELAY;
               t->packet_no=num;
               num++;              // Increase the num for
header
               t->next=NULL;
               r->next=t;
               r=r->next;
               r->next=NULL;

     birth[packet_produced].birthtime=clk_time_gen;
               birth[packet_produced].head_no=r->packet_no;
```

```
                packet_rejected++;
                packet_produced++;

                /*
                     Calculate time for the next packet to
send
                */

                x=(float)myrand();
                x=1.00-x;
                x=1000*(log(1.0/x)/LAMBDA_P);
                clk_time_gen+=x;
           }
     }
     else
     {
           /*
              Generate new packet and store it ready queue
           */

           t=(struct packet *)calloc(1,sizeof(struct
packet));

           if(t==NULL)
           {
                printf("\n\aNOT ENOUGH MEMORY
AVAILABLE!\n");
                getche();
                free_list();
                exit(0);
           }
           else  // t is not null
           {
                gettime(&(t->time_stamp));
                t->reject_flag=FALSE;
                t->delay_flag=FALSE;
                t->time_reject=BLACK;
                t->packet_no=num;
                num++;              // Increase the header
number
                p->next=t;
                p=p->next;
                p->next=NULL;

     birth[packet_produced].birthtime=clk_time_gen;
                birth[packet_produced].head_no=p->packet_no;
```

```
                packet_produced++;
                source_queue++;

                /*
                   Calculate time for the next packet to
send
                */

                x=(float)myrand();
                x=1.00-x;
                x=log(1.0/x);
                x=(1000*x)/LAMBDA_P;
                clk_time_gen+=x;
           }
      }
 }
}

void send_packet()
{
 struct packet *mov=NULL;
 float y=0.0;
 struct time t;

 randomize();
 y=myrand();
 y=1.00-y;
 y=1000*(log(1.0/y)/LAMBDA_S);
 clk_time_send+=y;

 //gettime(&t);
 mov=root->next;
 death[packet_send].deathtime=clk_time_send;
 death[packet_send].head_no=mov->packet_no;
 packet_send++;
 source_queue--;

 /* record the difference of time
 packetdelay[delay_index]=(t.ti_hour-mov-
>time_stamp.ti_hour)/3600+(t.ti_min-mov-
>time_stamp.ti_min)/60+(t.ti_sec-mov-
>time_stamp.ti_sec)+(t.ti_hund-mov-
>time_stamp.ti_hund)/1000;
 delay_index++;*/

 root->next=mov->next;
 mov->next=NULL;
```

```
 free(mov);
}

int performance_check()
{
 float n=0.0;
 n=(float)(packet_rejected)/packet_produced;
 if(n>QOS) return(FALSE);   // QOS ratio of packet are
rejected then performance is below par;
 else return(TRUE);
}

// Now only thing to be resolved is time calculation

void record_result()
{
 int j=0,i=0,o=0;
 float m=0.0;

 clrscr();
 //getche();
 for ( ;j<packet_send;j++)
 {
     i=o;
     while(packets_match(i,o)!=TRUE)
     {
          o++;
     }
     if(packets_match(i,o)==TRUE)
     {
          m+=(death[o].deathtime-birth[i].birthtime);
          i++;
          o++;
     }
 }
 printf("\nPACKETS PRODUCED = %d",packet_produced);
 printf("\nPACKETS SEND = %d",packet_send);
 printf("\nPACKETS REJECTED = %d",packet_rejected);
 printf("\nMAXIMUM CAPACITY OF DATA BUFFER =
%d",INPUT_QUEUE_SIZE);
 printf("\nmean delay=%f",m/packet_produced);
 getche();
}

int packets_match(int i,int o)
{
  if (birth[i].head_no==death[o].head_no) return (TRUE);
```

```
  else return(FALSE);
}

float myrand()
{
     float a;
     randomize();
     a=random(RANDOM_MAX);
     a=a/(float)(RANDOM_MAX);
     if(a<.3 && a>0.7) a=fabs(a-0.5);
     return(a);
     //return(0.5);
}
```

## 4.2 Source Code for module 2

```
////// definition file for module 2 ////////////////////

/* Parameters used in the Simulation of std Leaky bucket */

/* Peak rate of packet generation */
     #define LAMBDA_P 100

/* Service rate of node */
     #define LAMBDA_S 40

/* Token generation rate of node */
     #define LAMBDA_L 40

/* Maximum Buffer size of the node */
     #define DATA_BUFFER_MAX  40

/* Maximum Size of Token Buffer */
     #define TOKEN_BUFFER_MAX 18


#define INTERCELL_ARRIVAL_TIME 10  // it is = 1/LAMBDA_P
#define T_ON  100   // 20 here means 20 milliseconds
#define T_OFF 800

 /* System Parameters */

#define TRUE 1
#define FALSE 0
#define RANDOM_MAX 10
#define PACKET_LIMIT 1000
#define LIMIT 300

/*

   long int packet_produce;      // counter for the no. of
                                    packets produce
   long int packet_send;      // counter for the no. of
                              packets send
   int source_queue=0;

*/

float clk_spawn=00.00;
float clk_send=00.00;
```

```
int token[LIMIT];

////////////////////////// MODULE2 ///////////////////////
/*******************************************************/
/*******************************************************/
/***************** Date : 12-4-2002 ********************/
/***************** Author's name: Nikhil Bhargava *********/
/*******************************************************/

////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
/*
  Static Simulation of standard Leaky Bucket Congestion
  Control algorithm
*/
////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////


/*
  The aim of this program is Simulation of Leaky Bucket
  Algorithm (which works at Data Link Layer) which
  basically manages Flow Control (it is an open loop
  Congestion Control technique)
*/

////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////

// Include header files

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <dos.h>
#include
"C:\mydocu~1\\study_~1\\my_fin~1\\my_code\\defleak2.h"



/*
```

```
   We assume ideal channel whose maximum capacity is equal
   to peak rate of packet generation at source end
*/

/*
   We are using ON-OFF Bursty model with An ON period which
   is exponenetially distributed over a truly generated
   random variable with mean as Ton and during which the
   packets are produced i.e bursty as well as streamy data
    is generated and transmitted out
*/

/*
   While in the OFF period,there is transmission of
   packets.
*/




/*
   We are modelling time of generation of packets as
   poisson's distributed function on a truly generated
   random variable according to equation :
    t=(1/Lambda)ln(1/(1-truly generated random variable by
   rand variable))
*/

/*
   All parameters of the standard leaky bucket have been
   defined in another file
*/

 int source_queue=0;
 long int packet_produce=0;   // counter for the no. of
packets produce
 long int packet_send=0; // counter for the no. of packets
send

/*
   Declarations of functions used in this program
*/

float myrand();
void result();
float minimum(float a, float b);
```

```
void send_packet(int i,int count);
void produce_packet();

void main()
{
 int i,j,k;

 float a=0.0;
 float t_on[LIMIT],t_off[LIMIT];

 /*
  generate array for t_on statically (this time not
  dynamically)
 */

 clrscr();
 for(i=0;i<LIMIT;i++)
 {
     a=myrand();
     if(i==0)   t_on[i] = T_ON * 2.303* log10(1.0/(1.0-a));
     else
     t_on[i] = (T_ON * 2.303 * log10(1.0/(1.0-a)))+t_on[i-
1];
 }

 /*
   generate array for t_off statically (this time not
  dynamically)
 */

 for(i=0;i<LIMIT;i++)
 {
     a=myrand();
     if(i==0)   t_off[i] = T_OFF * 2.303 * log10(1.0/(1.0-
a));
     else
     t_off[i] = (T_OFF * 2.303 * log10(1.0/(1.0-
a)))+t_off[i-1];
 }
// getche();
 /*
    generate array for token buffer
 */

 for(i=0,k=j=0 ;i<LIMIT ;i++)
 {
     if(j==0)
```

```
     {
          token [i] = (int)((t_on[j] * LAMBDA_L)/1000);
          j++;
     }

     if(k==0 && i==1)
     {
          token[i]=(int)(((t_on[j] - t_on[j-1] + t_off[k])*
LAMBDA_L)/1000);
          j++;
          k++;
     }

     if(i>1)
     {
          token[i]=(int)((t_on[j] - t_on[j-1] + t_off[k] -
t_off[k-1]) * (LAMBDA_L/1000.0));
          j++;
          k++;
     }
     if (token[i]>TOKEN_BUFFER_MAX)
token[i]=TOKEN_BUFFER_MAX;
 }
 //getche();
 i=j=0;

 /*
    generate packets and send packets in the ON period only
 */

 // produce the first packet
 produce_packet();

 while(packet_produce<PACKET_LIMIT)
 {
     /*  ON period begins  */

     if(i>0 && i<LIMIT)
     {
          while(packet_produce!=PACKET_LIMIT &&
minimum(clk_spawn,clk_send)<(t_on[i]+t_off[i-1]))
          {
               if(clk_spawn<=clk_send &&
clk_spawn<=(t_on[i]+t_off[i-1])) produce_packet();
               if(clk_send<clk_spawn &&
clk_send<=(t_on[i]+t_off[i-1])) send_packet(i,TRUE);
          }
```

```
          if(packet_produce==PACKET_LIMIT) break;
     }

     if(i==0)
     {
          while(minimum(clk_spawn,clk_send)<t_on[i])
          {
               if(clk_spawn<=clk_send &&
clk_spawn<(t_on[i])) produce_packet();
               if(clk_send<=clk_spawn &&
clk_send<(t_on[i]))send_packet(i,TRUE);
          }
     }

     /*  OFF period begins  */

     if(i==0)
     {
          while(source_queue!=0 && token[i]>0 &&
clk_send<(t_on[i]+t_off[i])) send_packet(i,FALSE);
          clk_spawn+=t_off[0];
          clk_send=clk_spawn;
     }
     else
     {
          while(source_queue!=0 && token[i]>0 &&
clk_send<(t_on[i]+t_off[i])) send_packet(i,FALSE);
          clk_spawn+=t_off[i+1]-t_off[i];
          clk_send=clk_spawn;
     }

     i++;
 }
 //getche();

 // send the last paacket
 //send_packet(i,TRUE);
 if(packet_produce!=packet_send)
 {

     {
          //getche();
          clk_spawn+=1000;
          while(packet_send!=packet_produce)
send_packet(i,TRUE);
          i++;
     }
```

```
 }

 clrscr();
 result();
 getche();
}

float myrand()
{
     float a;
     randomize();
     a=random(RANDOM_MAX);
     a=a/(float)(RANDOM_MAX);
     if(a<.1 && a>0.9) a=fabs(a-(RANDOM_MAX/2));
     return(0.5);
}

static int violation=0;

float a=0.0,k,produce[PACKET_LIMIT];
float send[PACKET_LIMIT];

void produce_packet()
{
     //getche();
     if (source_queue<DATA_BUFFER_MAX)
     {
          /*
              produce the packets and store it in data
buffer
          */

          produce[packet_produce]=clk_spawn;

          packet_produce++;
          source_queue++;

          a=myrand();
          //printf("%f\t",clk_spawn);
          //getche();
          k=1000*(1.0/LAMBDA_P);
          k=k*2.303 *log10(1.0/(1.0-a));
          clk_spawn+=k;
          //clk_spawn+=INTERCELL_ARRIVAL_TIME;
     }

     /*
```

```
         packet is nonconforming so reject it and increase
the count of
         rejected packets by 1
     */
     else
     {
          violation++;
          printf("\n%d",violation);
          if(clk_spawn<=clk_send) clk_spawn=clk_send+10.00;
     }
}

int MAX_BURST=TOKEN_BUFFER_MAX/(1-LAMBDA_L/LAMBDA_P);

void send_packet(int i,int count)
{
 int outburst=0;    // used for checking the burst of out
going packets
 float a=0.0;
 //getche();

 if (count==FALSE)
 {
     while(source_queue>0 && outburst<MAX_BURST &&
token[i]>0)
     {
          send[packet_send]=clk_send;
          packet_send++;
          source_queue--;
          token[i]--;
          outburst++;

          if(outburst==0)
          {
               a=myrand();

     clk_send+=1000*(1.0/LAMBDA_S)*2.303*log10(1.00/(1.0-
a));
          }

          if(outburst>1 && source_queue>0)  clk_send+=
INTERCELL_ARRIVAL_TIME;
     }
 }
 else
 {
     if(i>0 && token[i-1]>0)
```

```
      {
          /*
             this is to add unused tokens of the just
preceding interval to
             the token count of the current interval. Care
should be taken to
             ensure that the token count doesn't increase
BUFFER_MAX
          */
          token[i]+=token[i-1];
          token[i-1]=0;
          if (token[i]>TOKEN_BUFFER_MAX)
token[i]=TOKEN_BUFFER_MAX;
      }

      /*if(source_queue>0 && outburst<MAX_BURST &&
token[i]>0)
      {
          a=myrand();

clk_send+=1000*(1.0/LAMBDA_S)*2.303*log10(1.00/(1.0-a));
      }*/

      if(source_queue==0 && clk_send<clk_spawn)
      {
          clk_send=clk_spawn+1;
      }

      if(source_queue>0 && clk_send<clk_spawn &&
token[i]==0)
      {
          clk_send=clk_spawn+1;
      }

      while(source_queue>0 && outburst<MAX_BURST &&
token[i]>0 && clk_send<=clk_spawn)
      {
          send[packet_send]=clk_send;
          packet_send++;
          source_queue--;
          token[i]--;

          if(outburst==0)
          {
               a=myrand();
```

```
     clk_send+=1000*(1.0/LAMBDA_S)*2.303*log10(1.00/(1.0-
a));
          }

          outburst++;
          if(outburst>1 && source_queue>0)  clk_send+=
INTERCELL_ARRIVAL_TIME;
      }
 }
}

void result()
{
 int i=0;
 int BURST=((T_ON+T_OFF)/T_ON);    // weight of 2 out of 10
meaning 20%
 float mf=0.00;

 //getche();
 for(;i<PACKET_LIMIT;i++)
 {
     mf=mf+send[i]-produce[i];
 }
 printf("\n violation
probability=%d,%f",violation,violation/(float)PACKET_LIMIT)
;
 printf("\n mean delay = %f",mf/PACKET_LIMIT);
}

float minimum(float a, float b)
{
  if ((a-b)<0.00) return(a);
  else return(b);
}
```

## 4.3 Source Code for Module 3

```
////////////////////////////////////////////////// Module 3 ////////////////////////////////////////////////////
/*********************************************************/
/****************** Date : 12-4-2002 *********************/
/************ Author's name: Nikhil Bhargava *************/
/*********************************************************/

////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////

/*
   This is the code for a traffic shaper for congestion control and traffic  shaping at data
   link layer. It is based on shift register scheme and is essentially peak rate enforced. It
   accommodates both short term burstiness  and variable burstiness with bounds but in
   the long run switch to standard Leaky Bucket scheme.
*/

/*
   It uses three windows scheme; more the number of windows the      more
   flexible our traffic  shaper will be.
*/

/*
   It divides the shift register into N slots for noting the history of packets send
   and uses  counter to store the packets in each window.
/*

/*
   It uses soft discretization of time slots into active and
   idle transitions
*/

////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////

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


/*
   Library specific header files included to make the code
   backward compatible
*/

#include <stdio.h>
#include <stdlib.h>
```

```
#include <conio.h>
#include <math.h>

/*
   Parameters used in the Simulation of SRTS
*/


/* Peak rate of packet generation */
     #define LAMBDA_P 100

/* Effective bandwidth of node */
     #define LAMBDA_E 40

/* Maximum Buffer size of the node */
     #define DATA_BUFFER_MAX  12

/* Maximum Size of Token Buffer */
     #define TOKEN_BUFFER_MAX 18

/* Time between two succesive cells departure */
     #define INTERCELL_TIME 10     // it is = 1/LAMBDA_P

/* Mean of exponentially distributed ON Time Period */
     #define T_ON  200             // 20 here means 20
milliseconds

/* Mean of exponentially distributed OFF Time Period */
     #define T_OFF 1000

/* QOS is chosen to be the no. of Rejected packets */
     #define QOS 0.2
/* Maximum limit of Shift register for noting temporal
history */
     #define N 450

     #define W1 30
     #define W2 75
     #define W3 450

     #define A 1    // active time slot

/*
It indicates that an ideal slot has been interrupted and is
completed at the end
*/
     #define H -1
```

```
/*
   Active slot
*/
     #define I 0




/*
   System Parameters
*/

#define TRUE 1
#define FALSE 0
#define RANDOM_MAX 10
#define PACKET_LIMIT 4000
#define TIME_LIMIT 300

struct shift_register
{
     int status;
     int value;
     float start_time;
     float end_time;
};
struct shift_register reg[N];


long int packet_produce=0;
long int packet_send=0;
float clk_spawn=0.0;
float clk_send=0.0;

void produce(void);
void consume(void);
float myrand();
void gen_register();
void modify_register(int );
int data();
int admit();
void record_result();
float minimum(float , float );
```

```
void shift();

void main(void)
{
 int i,j;
 float n,a=0.0;
 float on[TIME_LIMIT],off[TIME_LIMIT];

 /*
    generate array for t_on statically (this time not
dynamically)
 */

 gen_register();
 //getche();

 clrscr();
 for(i=0;i<TIME_LIMIT;i++)
 {
     a=myrand();
     if(i==0)   on[i] = T_ON * log(1.0/(1.0-a));
     else    on[i] = (T_ON * log(1.0/(1.0-a)))+on[i-1];
 }

 //getche();
 /*
     generate array for t_off statically (this time not
dynamically)
 */

 for(i=0;i<TIME_LIMIT;i++)
 {
     a=myrand();
     if(i==0)   off[i] = T_OFF * log(1.0/(1.0-a));
     else      off[i] = (T_OFF * log(1.0/(1.0-a)))+off[i-
1];
 }
 //getche();

 i=0;
 while(packet_produce<PACKET_LIMIT || n>QOS)
 {
     /*  ON period begins  */
     //getche();
     if(i==0)
     {
          while(minimum(clk_spawn,clk_send)<on[0])
```

```
          {
               if(clk_spawn<=clk_send && clk_spawn<(on[0]))
produce();
               if(clk_send<=clk_spawn && clk_send<(on[0]))
consume();
          }
     }

     if(i>0 && i<TIME_LIMIT)
     {
          while(packet_produce!=PACKET_LIMIT &&
minimum(clk_spawn,clk_send)<(on[i]+off[i-1]))
          {
               if(clk_spawn<=clk_send &&
clk_spawn<=(on[i]+off[i-1])) produce();
               if(clk_send<clk_spawn &&
clk_send<=(on[i]+off[i-1])) consume();
          }
          if(packet_produce==PACKET_LIMIT) break;
     }

     /*  OFF period begins  */

     //getche();
     clk_spawn+=off[0];
     if(i==0)
     {
          while(data()==TRUE && clk_send<(on[0]+off[0]))
consume();
          //clk_spawn+=off[0];
          clk_send=clk_spawn;
     }
     else   // i is not equal to zero
     {
          while(data()==TRUE && clk_send<(on[i]+off[i]))
consume();
          //clk_spawn+=off[i+1]-off[i];
          clk_send=clk_spawn;
     }

     i++;
 }

 if (packet_produce!=packet_send)
 {
     clk_spawn+=100;
     while(data()==TRUE) consume();
```

```
 }

 clrscr();
 //getche();
 record_result();
 getche();
}

float myrand()
{
     float a;
     /*randomize();
     a=random(RANDOM_MAX);
     a=a/(float)(RANDOM_MAX);
     if(a<.1 && a>0.9) a=fabs(a-(RANDOM_MAX/2));*/
     return(0.5);
}


static int violation=0;
int source_queue=0;

float prod[PACKET_LIMIT];
float send[PACKET_LIMIT];

void produce()
{
     float a=0.0,k;
     //getche();

     if (source_queue<DATA_BUFFER_MAX)
     {
          /*
              produce the packets and store it in data
buffer
          */

          prod[packet_produce]=clk_spawn;

          packet_produce++;
          source_queue++;

          a=myrand();
          //printf("%f\t",clk_spawn);
          //getche();
          k=1000*(1.0/LAMBDA_P);
          k=k*log(1.0/(1.0-a));
```

```
          clk_spawn+=k;
          //clk_spawn+=INTERCELL_ARRIVAL_TIME;
     }

     /*
         packet is nonconforming so reject it and increase
the count of
         rejected packets by 1
     */
     else
     {
          violation++;
          printf("\n%d",violation);
          while(clk_spawn<=clk_send)
          {
               a=myrand();
               k=1000*(1.0/LAMBDA_P);
               k=k*log(1.0/(1.0-a));
               clk_spawn+=k;
          }
     }
}

void gen_register()
{
 int i=0;
 float a=0.0;
 for ( ;i<N;i++)
 {
     reg[i].value=0;
     reg[i].status=I;
     reg[i].start_time=a;
     a=a+INTERCELL_TIME;
     reg[i].end_time=a;
 }
}

// nw1 refers to maximum number of packets in the 1st window

int nw1=TOKEN_BUFFER_MAX /(1.0 -
((float)LAMBDA_E/LAMBDA_P));

// nw1 refers to maximum number of packets in the 2nd window

int nw2=TOKEN_BUFFER_MAX /(1.0 -((float)LAMBDA_E/LAMBDA_P))
+ LAMBDA_E*(W2-W1)*INTERCELL_TIME/1000;
```

```
// nw3 refers to maximum number of packets in the3rd window

int nw3=(LAMBDA_E*W3)/1000 *INTERCELL_TIME;

void consume()
{

 float last,a=0.0;
 int count=0;

 //if(data()!=TRUE && admit()!=TRUE) modify_register(0);

 while(data()==TRUE && admit()==TRUE && clk_send<clk_spawn)
 {
     send[packet_send]=clk_send;
     packet_send++;
     a=myrand();
     if(count<=1)
     {
          last=1000*(1.0/LAMBDA_E)*log(1.00/(1.0-a));
          clk_send+=last;
     }
     modify_register((int)(last));
     source_queue--;
     count++;

 }
}

int counter1=0;
int counter2=0;
int counter3=0;

void modify_register(int a)
{
 int j,i=0;

 i=a%INTERCELL_TIME;
 if (i>=0) i=a/INTERCELL_TIME;
 for(j=0;j<i;j++)
 {
     if (reg[W3-1].value==1) counter3--;
     if (reg[W2-1].value==1)
     {
          counter3++;
          counter2--;
     }
```

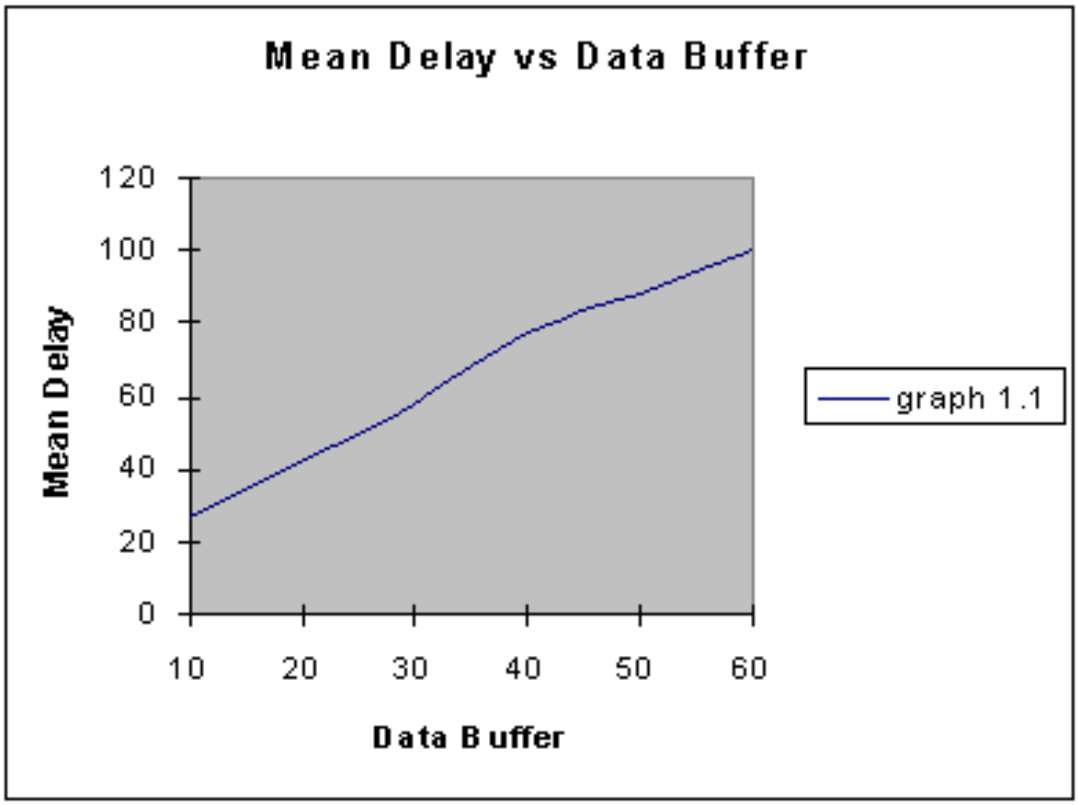
```
     if (reg[W1-1].value==1)
     {
          counter1--;
          counter2++;
     }
     //getche();
     shift();
     reg[0].value=0;
     reg[0].status=I;
 }

 if (reg[W3-1].value==1) counter3--;
 if (reg[W2-1].value==1)
 {
     counter3++;
     counter2++;
 }
 if (reg[W1-1].value==1)
 {
     counter1--;
     counter2++;
 }

 counter1++;
 shift();
 reg[0].value=1;
 reg[0].status=A;
 //printf("\ncounter1 %d,counter2 %d,counter3
%d\n",counter1,counter2,counter3);
 //getche();
}
 //getche();

void shift()
{
 int i;
 i=W3-1;
 while (i>0)
 {
     reg[i].value=reg[i-1].value;
     if (reg[i].value==1) reg[i].status=A;
     if (reg[i].value==0) reg[i].status=I;
     i--;
 }
 //getche();
}
```

```
int data()
{
 if (source_queue>0) return(TRUE);
 else return(FALSE);
}

int admit()
{
 int n1=0,n2=0,n3=0;
 //int i=0;
 /*while (i<W1)
 {
     if(reg[i].value==1) n1++;
     i++;
 }
 //getche();
 n2=n1;
 while(i<W2)
 {
     if(reg[i].value==1) n2++;
     i++;
 }
 n3=n2;
 //getche();
 while(i<W3)
 {
     if(reg[i].value==1) n3++;
     i++;
 }

 //getche();*/
 n1=counter1;
 n2=counter2;
 n3=counter3;
 //if(n1<nw1 && n2<nw2 && n3<nw3) return (TRUE);
 //else return(FALSE);
 return(TRUE);
}

void record_result()
{
 int i=0;
 int BURST=((T_ON+T_OFF)/T_ON);    // weight of 2 out of 10
meaning 20%
 float mf=0.00;

 //getche();
```

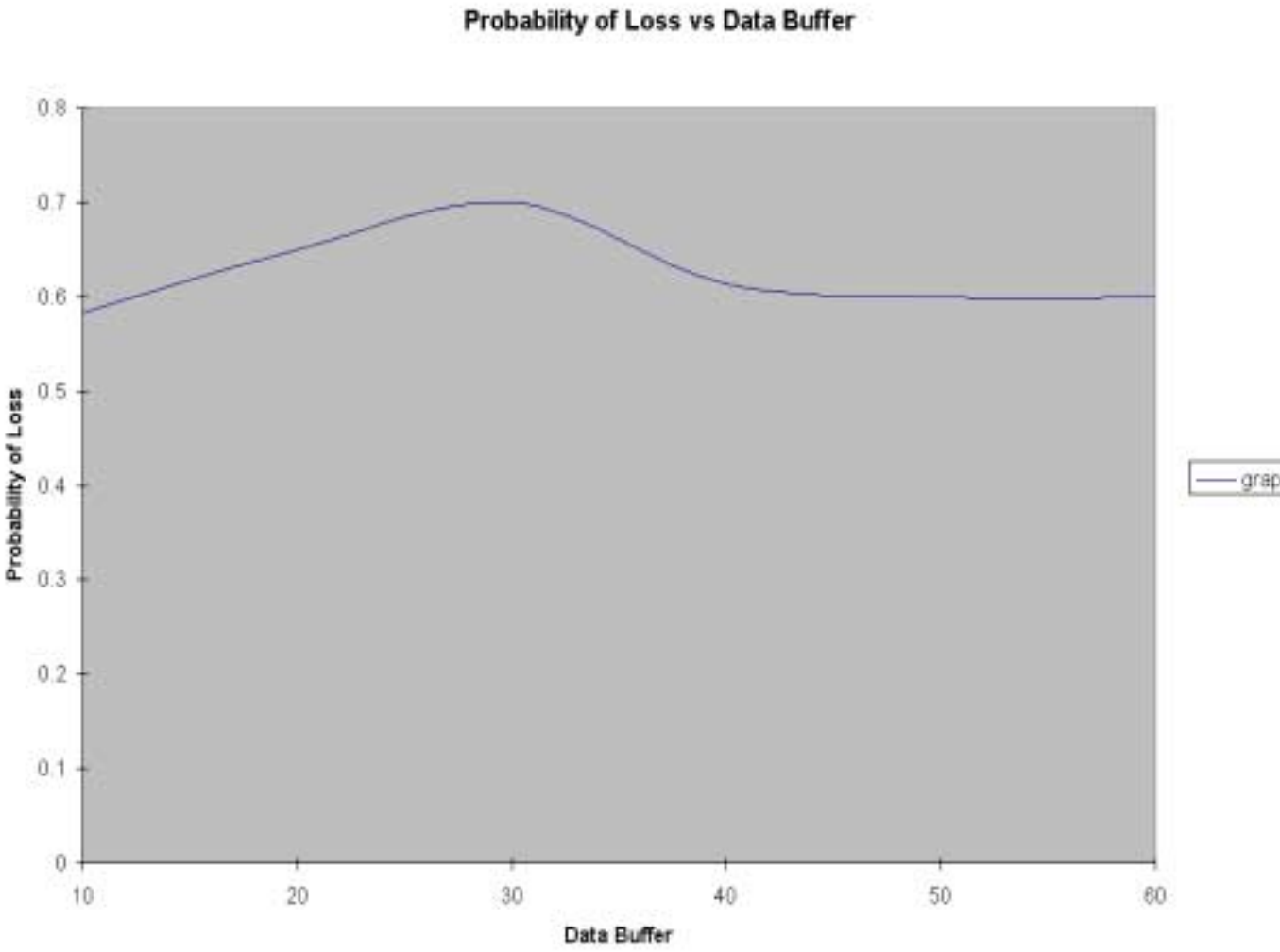
```
 for(;i<PACKET_LIMIT;i++)
 {
     mf=mf+send[i]-prod[i];
 }
 printf("\n violation
probability=%d,%f",violation,violation/(float)PACKET_LIMIT)
;
 printf("\n mean delay = %f",mf/PACKET_LIMIT);
 printf("\n BURST = %d",BURST);
}

float minimum(float a, float b)
{
  if ((a-b)<0.00) return(a);
  else return(b);
}
```

# Chapter 5 Results and Inferences

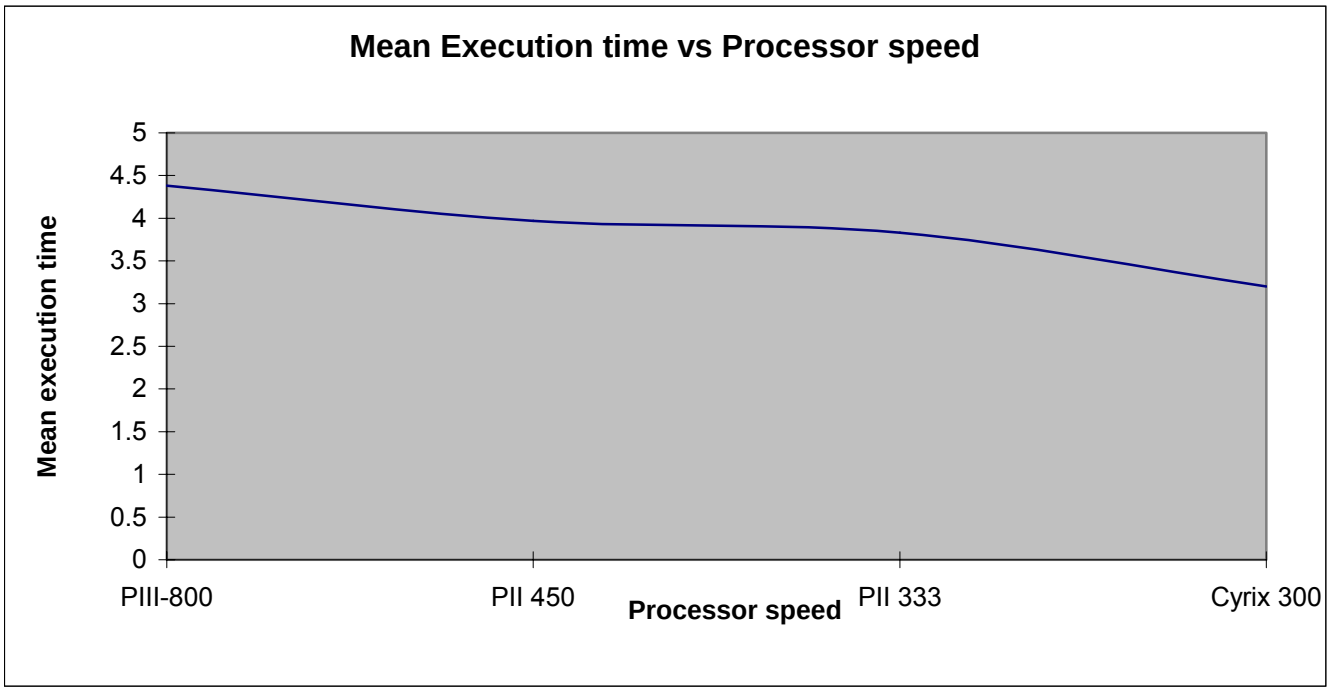
## 5.1 Simulation of a system without any congestion control scheme




Graph 1(a) Mean Delay vs Data Buffer for No Congestion Control




Graph 1(b) Probability of Loss vs Data Buffer for No Congestion Control




Graph 1(c) Mean Execution Time vs Processor Speed

Parameters used in the simulation code are as follows :

$\lambda_p$ = Peak rate of incoming packet=100
$\lambda_s$ = Service rate of outgoing packets=40
d= Data buffer capacity
QoS= Ratio of packets rejected over packets spawned=0.6

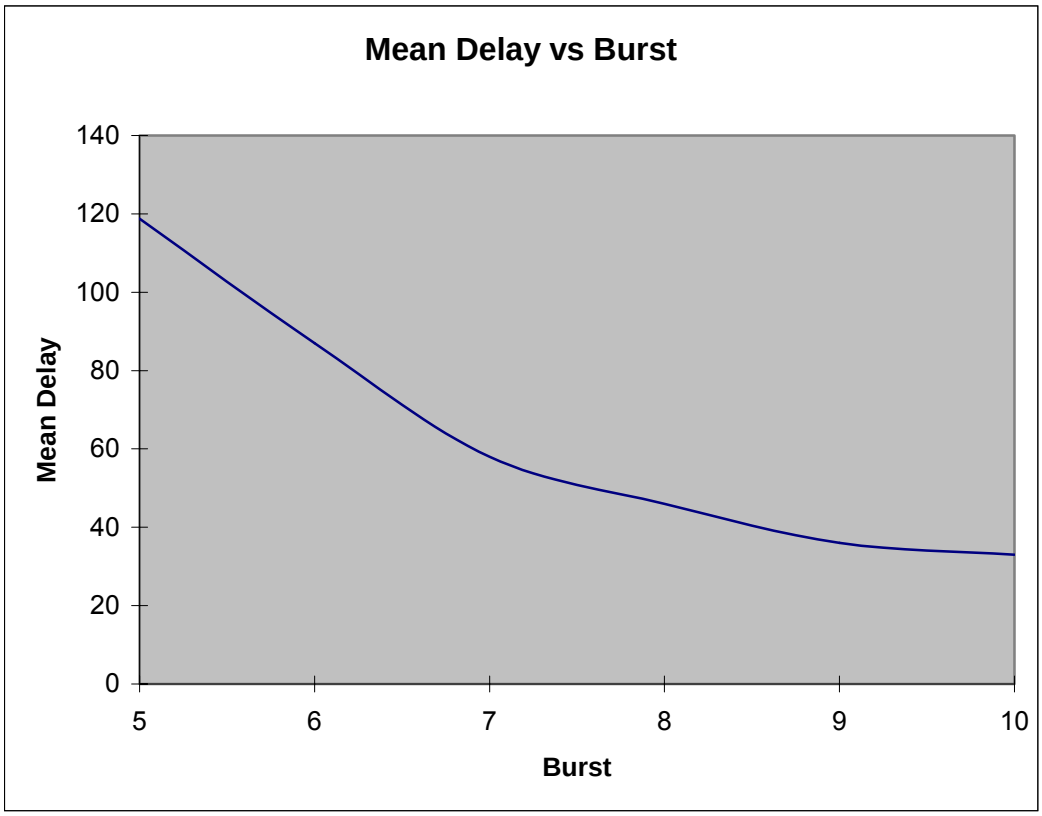
## Inferences from Results

- Looking at the graph 1(a), we find that as size of data buffer increases mean dealay also increases. This is obvious because greater is the capacity of Data buffer, more no. of packets will be admitted, more time they will spend in the queue and hence greter mean delay.

- From analysis of 1(b), we can infer that loss of probability(Non conforming packets) first increase to a maximum than becomes constant at a value. This can be explained in the light of the fact that the moment buffer is full , the node will reject all incoming packets at rate slower than their arrival and it stops accepting packets when QOS value has reached.

- From 1©, we can conclude that as node has faster processing speed (High end Processor and large RAM), onset of Congestion will be late. This is obvious because faster the processor, lesser will be the processing time and hence congestion will occur late.

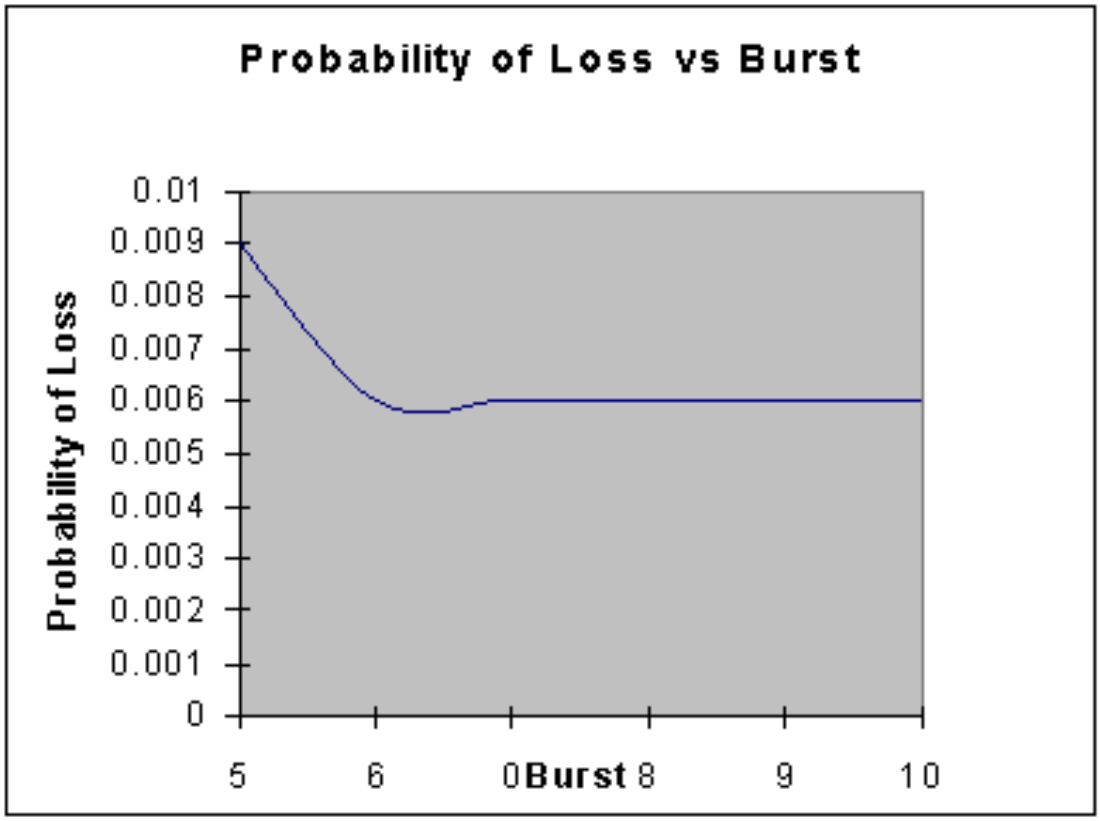
## 5.2 Simulation of a system with LBP




Graph 2(a) Mean delay vs Burst for LBP

Parameters used in this sinulation are

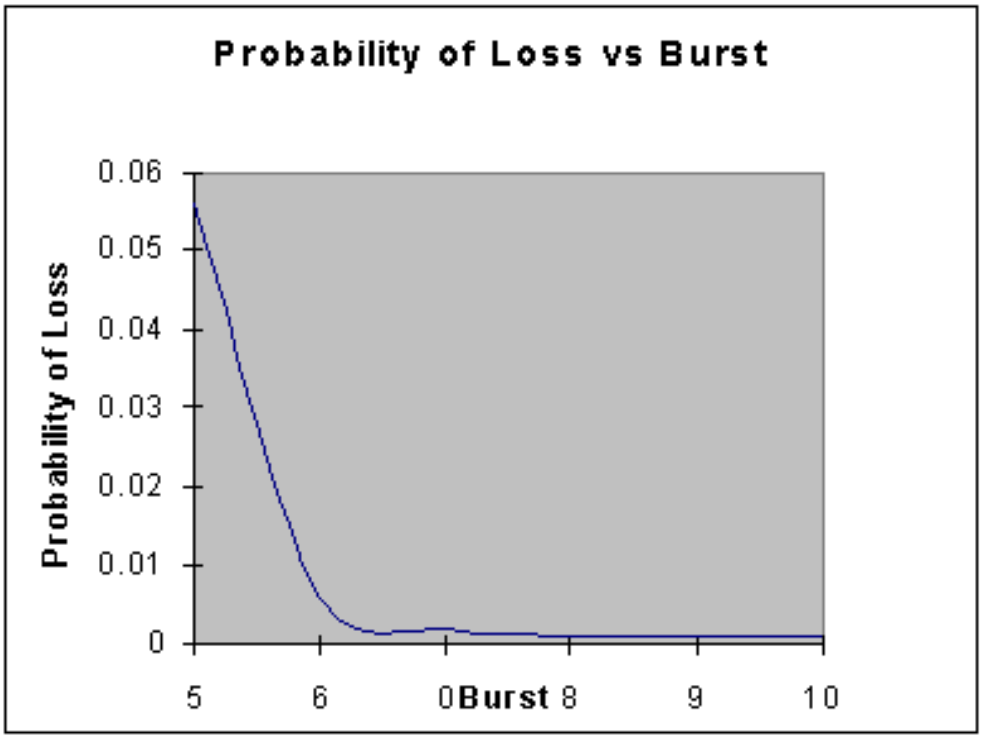
$\lambda_p$ = Peak rate of incoming packet=100
$\lambda_l$ = Service rate of outgoing
$\lambda_s$ = Service rate of outgoing packets=40
d= Data buffer capacity (very large)
QoS= Ratio of packets rejected over packets spawned=0.6
$T_{ON}$ varies from 200 to 88.
$T_{OFF}$ = 800




Graph 2(b) Loss probability Vs burst for LBP

Parameters used in this sinulation are

$\lambda_p$ = Peak rate of incoming packet=100
$\lambda_s$ = Token generation rate =40
$\lambda_l$ = Service rate of outgoing packets =40
d= Data buffer capacity =18
QoS= Ratio of packets rejected over packets spawned=0.6
$T_{ON}$ = 200
$T_{OFF}$ varies from 200 to 88.




Graph 2(c) Probability of loss vs Burst for LBP

Parameters used in this sinulation are

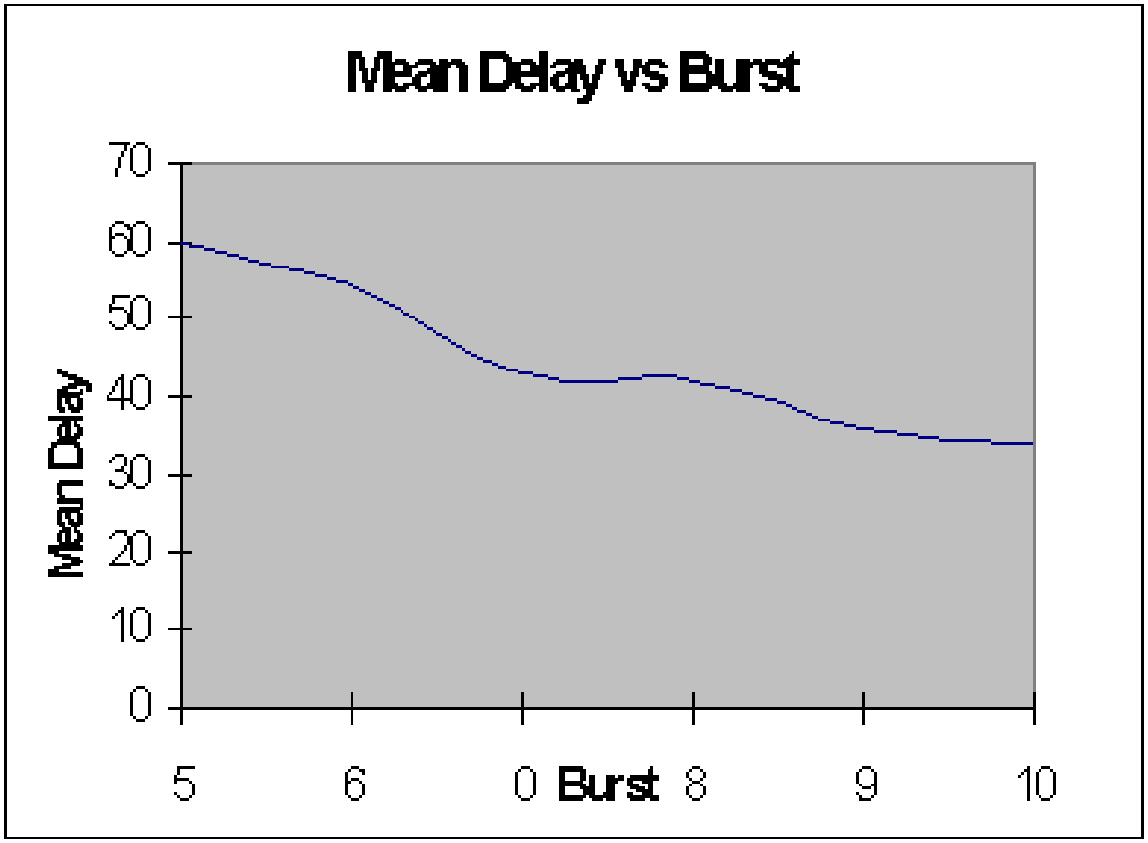
$\lambda_p$ = Peak rate of incoming packet=100
$\lambda_s$ = token generation rate =40
$\lambda_l$ = Service rate of outgoing=40
d= Data buffer capacity =18
QoS= Ratio of packets rejected over packets spawned=0.6
$T_{ON}$ varies from 200 to 88.
$T_{OFF}$ = 800

**Inferences from the graphs**

- As seen from graph 2(a) , mean delay decreases with increase in burstiness. This is because as burtiness increases less number of packets will remain in the queue a hence less delay.

- From graph 2(b) and 2(c) ,probability of loss increases with burstiness . this is because the stored packets are transmitted in the prolonged OFF of period and hence data buffer becomes empty .

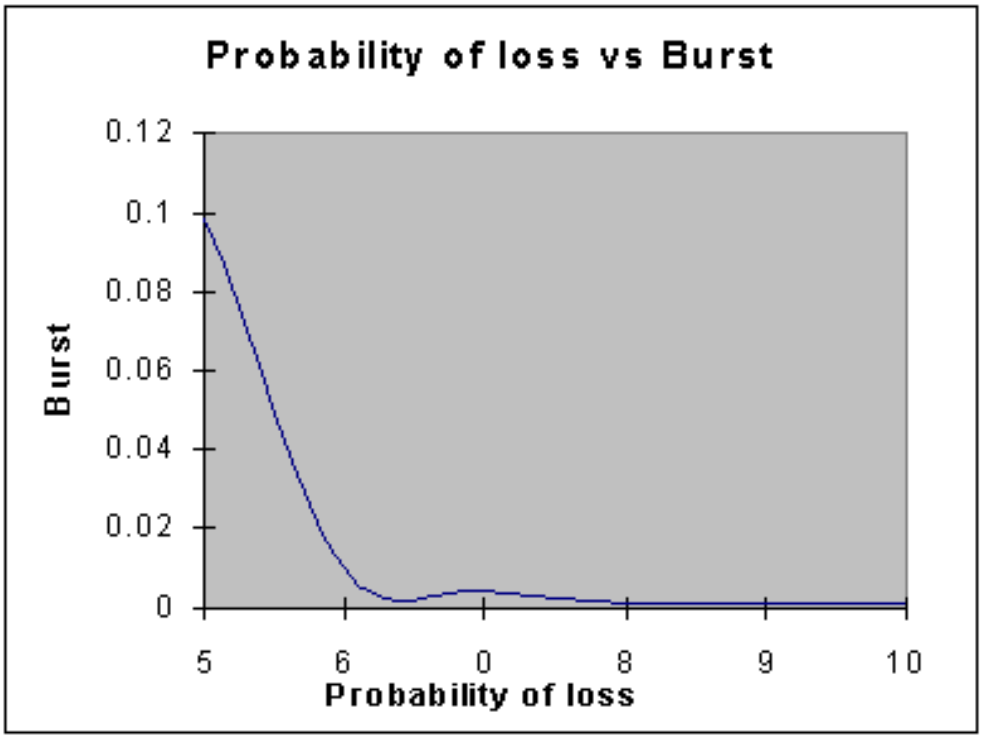
## 5.3 Simulation of a system with SRTS




Graph 2(a) Mean delay vs Burst for LBP

Parameters used in this sinulation are

$\lambda_p$ = Peak rate of incoming packet=100
$\lambda_l$ = Service rate of outgoing=40
d= Data buffer capacity (very large)
QoS= Ratio of packets rejected over packets spawned=0.2
$T_{ON}$ varies from 200 to 88.
$T_{OFF}$ = 800




Graph 2(c) Probability of loss vs Burst for SRTS

Parameters used in this sinulation are

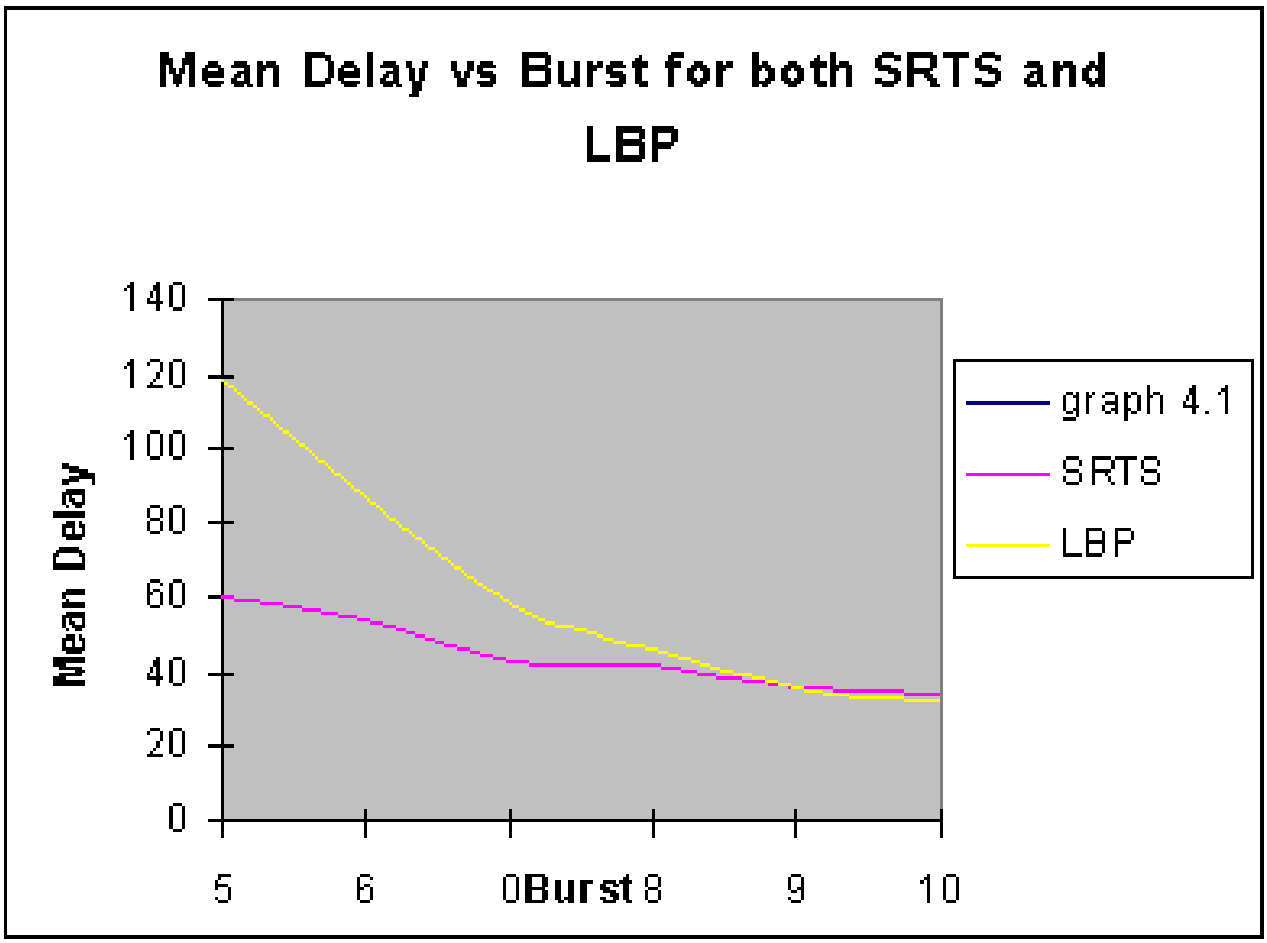
$\lambda_p$ = Peak rate of incoming packet=100
$\lambda_s$ = Token Generation rate =40
d= Data buffer capacity =18
QoS= Ratio of packets rejected over packets spawned=0.6
$T_{ON}$ varies from 200 to 88.
$T_{OFF}$ = 800

**Inferences from the graphs**

- As seen from graph 3(a) , mean delay decreases with increase in burstiness. This is because as burtiness increases less number of packets will remain in the queue a hence less delay.

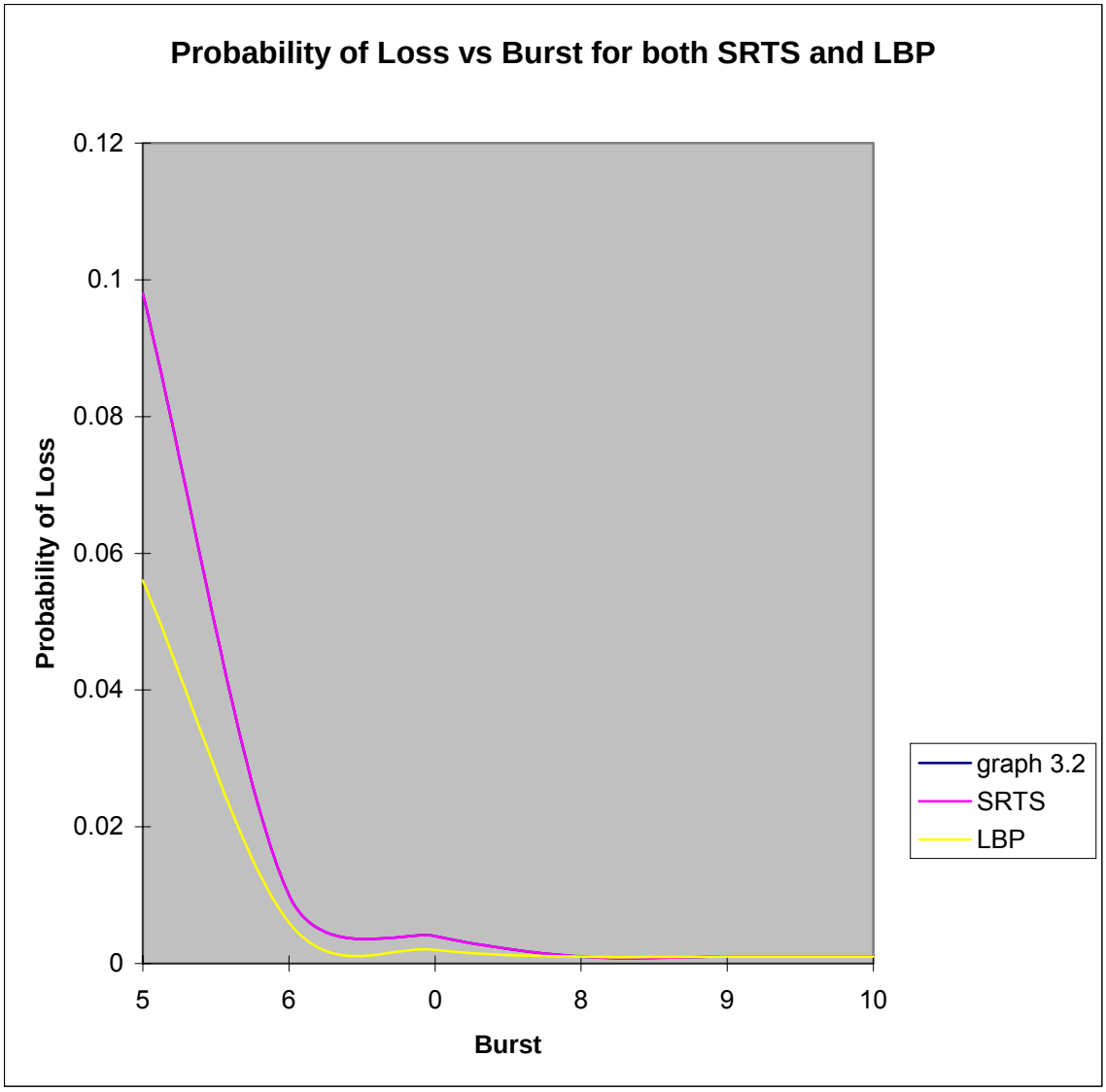
- From graph 3(b) , Probability of loss increases with burstiness . this is because the stored packets are transmitted in the prolonged OFF of period and hence data buffer becomes empty .

## 5.4 Comparitive study between LBP and SRTS




Graph 4(a) Mean delay vs Burst for LBP and SRTS

Parameters are same as in 1 and 2




Graph 4(b) Probability of loss vs Burst for SRTS and LBP

**Inferences from the graphs**

- As seen from 4(a), SRTS behaves very well for short term burstiness but for long term burstiness it reverts backt to LBP behaviour.
- Probability of Loss is greater for SRTS than LBP . This is so because SRTS uses Discretization of slots and it performs admission control at the input end.

# Chapter 6 Conclusion

In this project, we proposed a flexible traffic shaper and compared its performance with a LBP. The motivation for the new scheme is derived from the output characteristics exhibited by the LBP. Two main goals were set. One is to provide an adjustable burstiness feature so that higher bandwidth utilization along with reasonable guarantees can be obtained. The second was to reduce the access delays for real-time traffic by being more liberal in permitting short-term burstiness. The window based shaping policy adopted in the SRTS scheme can be used to achieve both the goals.

By adopting a more liberal, yet bounded attitude over short durations, SRTS reduces the access delays for time critical traffic. For providing the desired utilization and guarantees, a traffic shaper must work in unison with the buffer management and scheduling schemes at the switches. A composite study involving the shaper and the scheduler is necessary to see the effect of SRTS shaping on end –to-end performance. Such a study will constitute our future research.

# Chapter 7 Future Scope

After the comparative study between standard LBP and our new Traffic Shaper, our new Traffic Shaper clearly wins the race when it comes to providing a solution to the problem of incorporating short-term burstinss. Thus it has proved to be the best scheme for traffic control at Data Link Layer, at source end. Hence our next step will be to incorporate it in some kernel.

We have already started working in this realm. We have chosen freely distributed Linux Kernel (version - 2.4.2) to incorporate our module in it and see the effects. We are initially using TCP/IP implemented network (see \usr\src\network\tcp.c line no. 1 – 50 for more details). Normally the TCP/IP protocol suite doesn't give any protocol at data link layer, it simply uses protocol defined by the underlying network. This Linux kernel uses Sliding Window Protocol. We are trying to put our congestion control scheme in the flow control mechanism of Sliding Window Protocol.

So we are trying to incorporate our module with this module and recompile the kernel after that and run or module.

The following flow diagram depicts the basic kernel architecture.

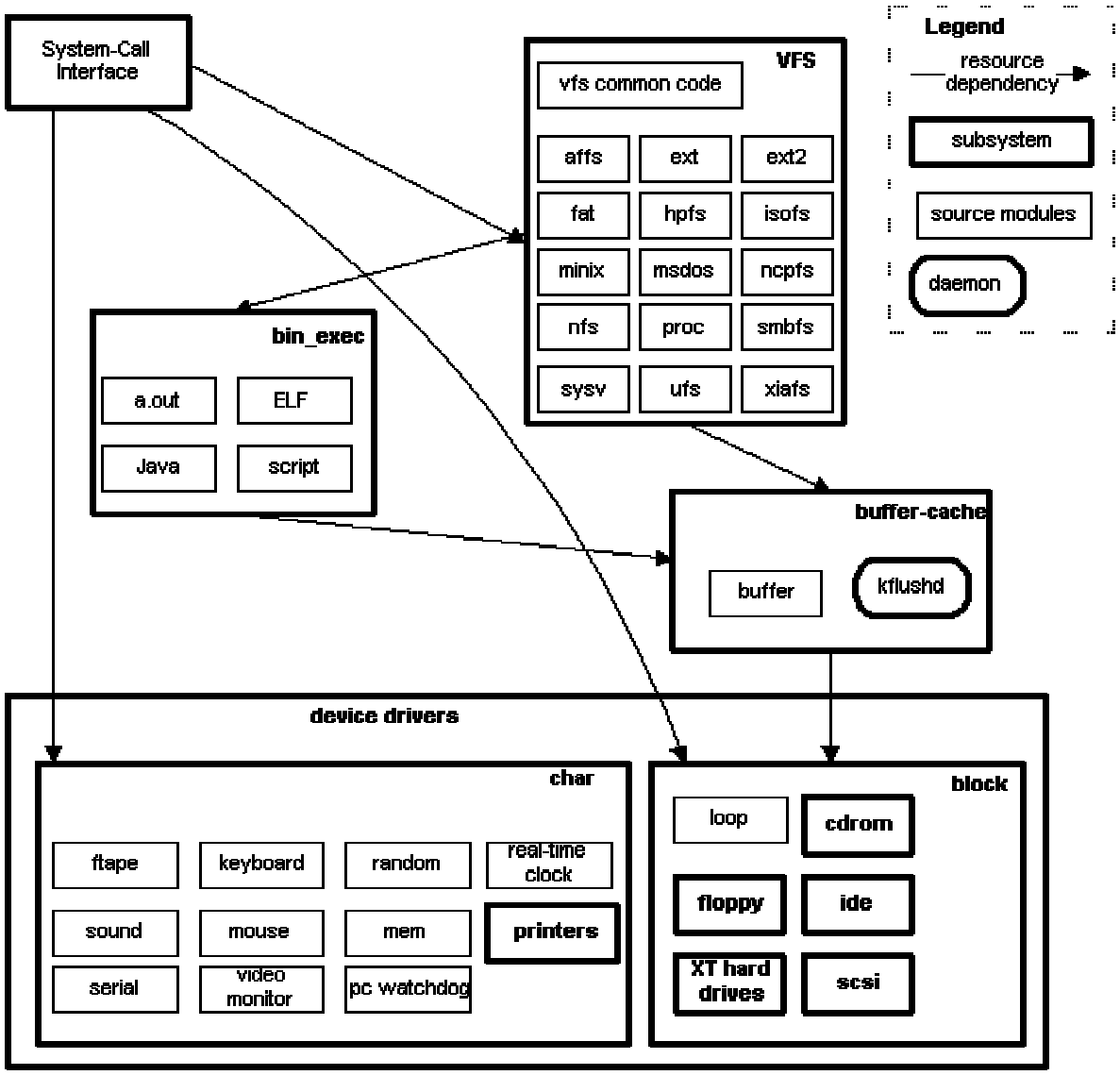



Fig 29. Structure of kernel

The following list highlights some of the header files and parameters useful for coding the module

- *# include <linux.h/netdevice.h>*

  This header hosts the definitions of struct device and includes a few other headers that are needed by the network drivers.

- *# include <linux/if.h>*

Included by *netdevice.h*, this file declares the interface flags (IFF-macros) and struct ifmap, which has a major role in the *ioctl* implementation for network drivers.

- *# include <linux.h/if_ether.h>*
- *ETH_ALEN*
- *ETH_P_IP*
- *Struct ethdr;*
- *Struct enet_statistics;*

Included by netdevice.h, if_ether.h defines all the ETH_macros used to represent octet lengths (like the address length) and network protocols (like IP). It also defines the structures ethdr and enet_statistics.

# Chapter 8 Glossary

- LBP leaky bucket scheme with peak rate policer.
- QoS Quality of Service.
- MW Moving window
- JW Jumping window scheme
- EWMA Exponentially weighted moving average
- CBR Constant bit rate
- VBR Variable bit rate
- SRTS Shift Register Traffic Shaper
- $\lambda_p$ Peak rate of input traffic
- $\lambda_a$ Average rate of input data
- $\lambda_t$ Rate of token Generation
- $\lambda_{eff}$ Effective bandwidth for guarantying QoS
- $\lambda_s$ Service rate of traffic
- BW Bandwidth

# Chapter 9 Bibliography

[1] Alberto Leon -Garcia and Indra Widjaja, Communication Networks: Fundamental Concepts and Key Architectures, Tata McGraw-Hill Edition 2000, (pages-516-534).

[2] S. Radhakrishnan, S.V. Raghavan, Ashok K. Agrawala : A flexible traffic shaper for high speed networks: design and comparitive study with leaky bucket, Computer Networks and ISDN Systems 28(1996) 453-469.

[3] Andrew S. Tanenbaum, Computer Networks, Recent Edition (pgs 376-380)

[4] S.P Singh, Sujata Sengar, S.L Maskara : An Overview of Congestion Control Techniques in ATM Networks and Some Performance Results, IETE Technical Review, Vol. 17, No. 3 May-June 2000, pp 87-103

[5] SIMON S. LAM, Senior member, IEEE, and Y.C Luke Lien,
Student member IEEE : Congestion Control Of Packet Communication Networks By Input Buffer Limits – A simulation Study.

[6] D.W. Davies, National Physical Laboratory, U.K : THE CONTROL OF CONGESTION IN PACKET SWITCHING NETWORKS

[7] Web reference

- http://www.mhhe.com/leon-garcia
- http://kabru.eecs.umich.edu/qos_network/diffserv/DiffServ_papers/papers/Sahu99-tcp-tb.pdf.
- http://www.halcyon.com/ast/tcpatmcc.htm
- http://www.cis.ohio-state.edu/~jain/cis788-95/atm_cong/
- http://casal.upc.es/~ieee/proceed/melich/melich.html
- http://www.ics.uci.edu/~duke/Abstracts/netmag91.html
- http://www.eeng.dcu.ie/~murphyj/the/the/node75.html

# Appendix

## A. Commonly used Probability Mass Functions

1. Bernoulli distribution:

   A typical Bernoulli random variable has one of the two values 0 or 1. In other words its sample space Ω={0,1}

   $P_X(0)=p$ for x=1
   $P_X(1)=q$ for x=0
   $P_X(x)=0$ for x!={0,1}

2. Binomial distribution

   A typical binomial distributed random variable is characterized by three parameters i.e b(k,n,p)
   K= number of success
   n= number of trials
   p=probability of success

   $P_X(x=k)={}^nC_k\, p^k\, (1-p)^{n-k}$
   $F_X(x)=\sum_{x_i<x} P_X(x_i)\, u(x-x_i)$

3. Poisson distribution

   This is one of the most common distributions used in to model various things. It is characterized by a controlling parameter a (a>0)

   $P_X(x=k)= (e^{-a} \cdot a^k) / k!$

   $F_X(x)=\sum_{x_i<x} P_X(x_i)\, u(x-x_i)$

## B. Transformations

Transformations are basically used when we want to transform a random distributed variable into some other form of distributed random variable. As an example I am showing transformation of a uniformly distributed random variable to an exponentially distributive random variable.

Let $F_X(x) = Y = \text{rand}()$ where,

rand() is the function which generates a truly distributive random variable.

Now,

$$F_X(x) = Y = \text{rand}()$$

Or, $1 - e^{(-x/\mu)} = Y$

Or, $-x/\mu = \ln(1-y)$

Or, $x = -\mu \ln(1-y)$

Or, $x = \mu \ln(1/1-\text{rand}())$

Now x is a truly exponentially distributed random variable.

## C. Important series Expansions

1. $e^x = 1 + x + x^2/2! + x^3/3! + \ldots\ldots\ldots$
2. $\log x = x - x^2/2! + x^3/3! + x^4/4! + \ldots\ldots$
3. $\sin x = x + x^3/3! + x^5/5! + \ldots\ldots..$
4. $\cos x = 1 + x^2/2! + x^4/4! + \ldots\ldots\ldots$
5. $\tan x = x + x^3/3 + 2x^5/15 + \ldots\ldots..$
6. $(1+x)^n = 1 + nx + n(n-1)/2! * x^2 + \ldots\ldots\ldots$