1997

Traffic management methodologies for ATM networks : a new approach

Asha G. Dinesh
San Jose State University

INFORMATION TO USERS

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

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

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

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

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

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

TRAFFIC MANAGEMENT METHODOLOGIES FOR ATM NETWORKS: A NEW APPROACH

A Thesis

Presented to

The Faculty of the

Department of Mathematics and Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Asha G. Dinesh

May 1997

APPROVED FOR THE DEPARTMENT OF

MATHEMATICS AND COMPUTER SCIENCE

Dr. Melody Moh

Dr. Mario Albarran

Dr. Jon Pearce

APPROVED FOR THE UNIVERSITY

Serena M. Stanford

ABSTRACT

TRAFFIC MANAGEMENT METHODOLOGIES FOR ATM NETWORKS: A NEW APPROACH

by Asha Dinesh

ATM Networks are high speed networks with guaranteed quality of service. The main cause of congestion in ATM networks is over utilization of the physical bandwidth. Unlike constant bit rate traffic, the bandwidth reserved by variable bit rate [VBR] traffic is not fully utilized at all instances. Hence, this unused bandwidth is allocated to available bit rate traffic [ABR]. As the bandwidth used by VBR traffic changes, available bandwidth for ABR traffic varies. In other words, available bandwidth for ABR traffic is inversely proportional to the bandwidth used by the VBR traffic.

To manage ATM networks efficiently, two new protocols are presented. A burst level admission control mechanism, Modified Fast Reservation Protocol, to avoid congestion by reserving bandwidth for long bursts of ABR traffic and a rate based congestion control algorithm, Explicit Allowed Rate Algorithm, based on the relationship between VBR and ABR traffic. Both these algorithms significantly improve the network throughput with minimal overhead on the switch.

TABLE OF CONTENTS

1. INTRODUCTION

Transfer mode is a technique for transmitting, multiplexing and switching information in a communication network [CCITT]. In other words, it is the mapping of network user information onto the physical network. Asynchronous transfer mode (ATM) is a means of transferring data in a BISDN (Broadband Integrated Services Digital Network) network. An ideal BISDN network [3,7] is envisioned to have the following features:

- Support different traffic types with guaranteed quality of service.
- Support high link speeds of 155 to 622 Mbits/sec.
- Efficient bandwidth utilization, i.e., usage of the physical bandwidth of the link must be close to 100% at all instances. In other words, provide bandwidth-on-demand to all users.
- All data must be formatted into short, fixed length packet with a header containing routing information.
- Simple switching technique for transferring data
- Minimum buffer size.
- Least additional overhead in end systems.

ATM networks support high speed links and different traffic types with guaranteed quality of service. For efficient bandwidth utilization, it uses the concept of bandwidth-on-demand. They are connection oriented, using fixed length packets for data transfer. Each packet (cell) is 53 bytes long, consisting of 48 bytes data and 5 bytes header. The header provides the cell priority, type and routing information [virtual path identifier and virtual channel identifier used locally in each node for identifying the next node].

ATM is a universal media for data transfer. ATM supports bandwidth on demand to all users by using the method of statistical multiplexing i.e., the statistical mean rate is taken into account while admitting a connection rather than its peak rate. The source requests the network for a connection by providing the required peak rate, the statistical mean rate and the minimum rate. The network accepts a source if it can support the requirements of that traffic type. An overview of the characteristics and requirements of different traffic types supported by ATM networks are described next in this section.

1.1 Constant Bit Rate (CBR)

This traffic type transmits at a constant bit rate. It is very delay sensitive. The time between two cell transmissions must be within the

maximum cell transfer delay (CDT) specified by the source. It is required to keep the cell transfer delay between two consecutive cell transfers within the specified cell delay variation (CDV). Low cell loss is acceptable. E.g. voice data transfer

1.2 Real Time Variable Bit Rate (RTVBR)

The required rate for this traffic type varies. It is also delay sensitive. The network must be able to support the peak rate for the maximum burst length specified by the source at the time of admission. When the source is not utilizing the peak bandwidth, the network can allocate that bandwidth to other traffic types. E.g. interactive compressed video

1.3 Non Real Time Variable Bit Rate (NRTVBR)

This traffic type also requires variable bit rate. Unlike CBR and RTVBR, NRTVBR is not delay sensitive. Hence, cell transfer delay and cell transfer variation is not an issue for this traffic type. E.g. Compressed Video, Transaction Processing.

1.4 Available Bit Rate (ABR)

The available bandwidth at any instance can be used by this traffic type. Hence, if a VBR source is not using the reserved peak bandwidth, then the network allocates the unused bandwidth to the ABR sources. The

ABR sources are controlled by rate feedback from the network. Cell loss must be at a minimum, but it is not sensitive to cell transfer delay or cell transfer variation. E.g. File Transfer, RPC.

1.5 Unspecified Bit Rate (UBR)

This traffic type is not sensitive to delay or cell loss. It uses the left over bandwidth if any. E.g. News Feed, Network Information.

Table 1.1: **Traffic Characteristics**

Traffic Type	**Bandwidth Usage**	**Cell Transfer Delay**	**Cell Delay Variation**	**Cell Loss Acceptance**
CBR	Constant cell transmission. Always uses Peak rate.	Very Sensitive	Very Sensitive	Moderately Sensitive
RTVBR	Variable cell transmission. Varies between Peak rate and Minimum rate.	Very Sensitive	Sensitive thorough the burst	Moderately Sensitive
NRTVBR	Variable cell transmission rate. Varies between the Peak rate and Minimum rate.	Sensitive	N/A	Sensitive
ABR	Variable cell transmission rate, according to the current network load.	N/A	N/A	Not Acceptable
UBR	Variable cell transmission rate, according to the left over network capacity.	N/A	N/A	N/A

The table above summarizes the requirements and characteristics of different traffic types[20,21,22,23].

Traffic management of the above types with guaranteed quality of service and efficient bandwidth utilization is a challenge to ATM networks. The average bit rate used by VBR class traffic is usually much lesser than its reserved peak rate. For better bandwidth utilization, the network takes advantage of the variable rate of VBR class and allocates the unused bandwidth to ABR class. Hence, serving all users according to the bandwidth required at that instance [bandwidth-on-demand]. The allocation of bandwidth according to the used statistical rate is called statistical multiplexing.

Statistical Multiplexing [bandwidth-on-demand] increases the bandwidth utilization but tends to cause congestion in the network. In other words, if VBR users increase their transmission rate, the physical link's bandwidth will be over utilized. This will eventually lead to buffer overflow and cell loss. In ATM networks, if a cell is lost, the source retransmits the whole packet. Under heavy load conditions, this will drastically decrease the network throughput. Hence, appropriate traffic management is critical in ATM networks.

In summary, it is seen that congestion in ATM networks is mainly caused by the ABR traffic class. This is because, the ABR class users do

not reserve the bandwidth at the time of admission. Hence, to decrease the chances of congestion, it is better that ABR traffic class reserves bandwidth for long bursts. In this thesis, a burst level admission control protocol for ABR class users with long bursts and a simple rate based traffic management algorithm for ATM networks, called Explicit Allowed Rate Algorithm [EARA], is presented. EARA manages ABR traffic class efficiently with/without the modified fast reservation protocol. EARA

- Supports all traffic types with the required quality of service
- Utilizes the bandwidth efficiently
- Transmits ABR traffic at the fastest possible rate
- Avoids congestion, hence requiring less buffer space

This thesis is organized as follows. Chapter 2 describes some existing congestion control mechanisms for ATM networks. The Modified Fast Reservation Protocol (MFRP)[25], a burst level admission control protocol, and its performance analysis is presented in chapter 3. Explicit Allowed Rate Algorithm (EARA), the new proposed traffic management scheme for ATM networks is described in chapter 4. Chapter 5 compares and analyses the performance of Explicit Allowed Rate Algorithm (EARA) with Proportional Rate Control Algorithm (PRCA)[8,11,22]and Explicit Rate Indication Congestion Avoidance (ERICA)[21] and chapter 6 concludes the thesis.

2. CONGESTION CONTROL SCHEMES

ATM network users can be divided broadly into two groups, namely, closed-loop and open-loop[8,11]. Open-loop users reserve the required bandwidth at call-setup. The network cannot change the reserved rate throughout the entire message transfer. CBR, RTVBR and NRTVBR traffic types belong to this category. Closed-loop users transmit cells by using the feedback information from the network. In other words, the network informs these users about the availability of bandwidth as well as congestion and the users change their transmission rates accordingly. This category consists of ABR and UBR traffic types.

When the network gets congested, it is required to inform the ABR source about congestion as soon as possible. This will eliminate cell-loss and hence packet retransmission. An overview of the two major approaches for managing ABR traffic efficiently, namely, credit-based and rate-based[8,11,20] are described in this section.

2.1 Credit Based

This scheme consists of per-link, per-vc, window flow control. Each link has a sender and a receiver node [can be a switch]. Each node maintains a separate queue for each VC and determines the number [credit] of cells that the sender can transmit on that VC without congestion and cell-loss. Though this scheme manages the traffic efficiently, it is very costly for the vendor.

2.2 Rate Based

In this scheme, the network controls the ABR traffic transmission rate according to the load. It uses resource management [RM] cells to change ABR traffic transmission rate. An efficient rate-based algorithm must support the following features:

- Efficient bandwidth utilization
- No/minimal cell-loss
- Minimal buffer size
- Simple switch algorithm [less overhead]
- Fair bandwidth allocation among ABR traffic class.

A number of rate-based algorithms have been proposed for ATM networks[8,11,20,21,22]. Some of which are:

2.2.1 **DECnet Protocol**[22]- *Ramakrishnan and Jain*

It uses ANSI Frame Relay standards. At fixed intervals, the window size is either increased by a fixed amount or decreased by an amount proportional to the current window size. The packet header contains a bit to indicate congestion and congestion is signaled in the forward direction. It uses end-to-end feedback loop.

2.2.2 **Backward Explicit Congestion Notification**[11,20,22]- *Newmann*

In this scheme, the switch sends an resource management [RM] cell to the source when it detects congestion [Negative Feedback]. If the source receives an RM cell from the switch, it decreases its transmission rate by an amount proportional to its current rate, otherwise, it increases its rate by a fixed amount.

2.2.3 **Explicit Forward Congestion Indication [EFCI] Scheme** [8, 11, 22] - *Hluchyi and Yin*

This scheme uses a positive feedback mechanism. A single bit in the header of the ATM data cell is used to indicate congestion. Destination checks this bit periodically and sends an RM cell to the source if the congestion bit is not set. If the source receives an RM cell, it will increase its rate by a fixed amount; otherwise, it will decrease its rate by an amount proportional to its current rate.

2.2.4 Modified EFCI Scheme[11,22]- *ATM Forum RBFC (Rate Based Flow Control) Group*

This scheme signals in the forward direction and sends negative feedback to the source. A new concept of segmentation o control-loop - a network can be divided into two or more segments by introducing intermediate network that act as virtual destination and virtual source. The virtual destination has to send an RM cell to the source if congestion bit is set - was introduced.

2.2.5 Proportional Rate Control Algorithm [PRCA] [8,11,22]- *Barnhart*

It uses positive bi-directional feedback mechanism. It supports segmentation of control-loop. Destination sends a RM cell periodically. Upon receipt of an RM cell, the source will check the EFCI bit and increase or decrease its rate accordingly.

2.2.6 Explicit Rate Feedback Scheme[22]- *Adams, Charnt, Jain, Lyles and Roberts*

The concept of calculating the explicit rate for each ABR source was introduced in this scheme. The source sends an RM cell periodically with the current transmission rate. The switch calculates its fair share according to the current rate. Any switch along the path - from source to destination - that does not have the capacity to support the specified explicit rate can reduce it. It uses end-to-end feedback loop.

2.2.7 Enhanced Proportional Rate Control Algorithm [EPRCA] [8,11, 20,22]- *L.Roberts*

This scheme is a combination of PRCA and Explicit Rate Feedback schemes. This increases the flexibility of the switch as it can control congestion through EFCI (forward) or explicit rate (backward) or both. Explicit rate is interpreted as a dynamic upper bound on the rate calculated by PRCA. The switch keeps track of every VC [Per-VC accounting] usage, to ensure that it is still active. It also selectively signals congestion to sources with large ACR [allowed cell rate].

2.2.8 Adaptive Proportional Rate Control [APRC] [11]- *K.Y.Siu and H.T.Tzeng*

This scheme detects congestion on switch depending on the change in the queue length rather than comparing the queue length to the threshold value.

2.2.9 Adaptive Proportional Rate Control 2 [APRC2] [11]- *K.Y.Siu and H.T.Tzeng*

This scheme is the same as APRC, but it shortens the ramp-up time. It is done by calculating the mean of current cell rate rather than dividing available bit rate among all active users.

2.2.10 Enhanced Proportional Rate Control Algorithm+ [EPRCA+]

[11]- *R. Jain, S.Kalyanaram and R. Viswanathan*

Congestion is detected by traffic load at the switch. It has an interval timer and the count of number of packets received. It signals the source by using backward RM cells with explicit rate.

2.2.11 Enhanced Proportional Rate Control Algorithm++ [EPRCA++]

[11]- *R. Jain, S.Kalyanaram and R. Viswanathan*

This scheme uses positive feedback mechanism - source decreases its rate if an RM cell is not received. It uses a counter at the source for forward RM cells instead of an interval timer.

2.2.12 Explicit Rate Indication Congestion Avoidance [ERICA]

[20,21]- *Raj Jain, Shiv Kalyanaraman and Ram Viswanathan*

This scheme avoids congestion by monitoring the load on the switch, according to the averaging interval, which is determined by the link capacity. The ABR source sends RM cells periodically [determined by the network]. The switch sends its load level information back to the source, which in turn computes and changes its transmission rate accordingly. The target bandwidth utilization is about 90% of the physical bandwidth.

2.3 Burst Level Admission Control Algorithms

To reduce the severity of congestion, a number of bandwidth reservation schemes for ABR traffic class has also been proposed. Fast

reservation protocol (FRP)[10,18] was the first burst level admission protocol proposed. An overview of FRP and two major variations of FRP -- Adaptive fast reservation protocol (AFRP)[4,9,29] and Fast reservation protocol with Immediate transmission (FRP/IT)[9,29] -- are presented next.

2.3.1 Fast Reservation Protocol (FRP)

FRP was the first burst level admission protocol proposed. It works as follows:

The source sends a cell requesting bandwidth equivalent to the peak rate for each burst. If the network can support the requested bandwidth along all the links from source to destination, it sends an ACK, otherwise it sends a NACK. If the source receives an ACK, it starts transmitting the data at peak rate. Once the transmission is completed, it sends a cell to release the bandwidth it was using. If the source receives a NACK, it backs off for a random time and re-attempts to reserve the peak bandwidth at a later time.

2.3.2 Adaptive Fast Reservation Protocol (AFRP)

AFRP uses the concept of bandwidth negotiation and random back-off. It is as follows: Initially the source requests the peak rate as the required bandwidth. It the source receives an NACK, it will back off for a period of time and request again for a reduced rate which is:

*Mean Rate <= New Rate = Old Rate * Decrement Factor*

If the source receives an ACK, it will transmit this burst at the rate accepted by the network, but will increment its asking rate for the next burst request as follows:

Mean Rate <= New Rate = Old Rate +
*(Old Rate * Increment Factor) <= Peak Rate*

2.3.3 Fast Reservation Protocol/Immediate Transmission (FRP/IT)

FRP/IT was proposed around the same time as FRP. The concept used is very similar to FRP. The protocol description is as follows:

The source requests the network to allocate the required peak rate. It assumes that the network will be able to support the requested rate and starts transmitting immediately. It keeps a copy of the data transmitted until it receives an ACK from the network. If the source receives an ACK, it continues its transmission and releases the bandwidth at the end of this burst transmission. It discards the copy of transmitted data. If it receives a NACK, it stops transmitting and retries later with the same peak rate.

Table 2.1: **Comparison of major burst level control schemes**

FRP	AFRP	FRP/IT
Always requests for peak rate	Negotiates the rate with the network	Always request for peak rate
High admission delay	Lesser admission delay	High admission delay
Less transmission delay	Transmission delay may be more than FRP and FRP/IT	Least transmission delay
No overhead	The source must keep track of the current requested rate and depending on whether it receives a ACK or NACK, it must increase or decrease its rate for the next request	The source needs to keep a copy of the data transmitted before it receives an ACK from the network.

3. MODIFIED FAST RESERVATION PROTOCOL [25]

There are three major approaches to bandwidth management of bursty traffic: the peak rate allocation, the minimum throughput allocation and the negotiated bandwidth allocation. In this new burst level admission control scheme, the concept of bandwidth negotiation is used. MFRP deals with three different rates. They are:

- **Requested Rate:**

This is the desired rate at which the source would like to transmit a particular burst. It is chosen to be in-between the Peak Rate and the Mean Rate.

- **Minimum Rate:**

It is the Mean or Average Rate required for transmitting a particular burst.

- **Allowed Rate:**

This is the rate the network has allocated to the source for transmitting a particular burst. The allowed rate ranges between:

Requested Rate >= Allowed Rate >= Minimum Rate.

Modified Fast Reservation Protocol is as follows: At admission request, the source sends the request cell with the desired rate and the minimum acceptable rate. The desired rate is chosen to be in-between the peak rate and the minimum rate, to reduce the overall blocking probability. The network, at each node, from source to destination reserves a rate (Allowed Rate) along the link, such that,

Min Rate <= *Allowed Rate* <= *Requested Rate*

If the source receives an ACK then it will transmit at the Allowed Rate. After transmitting the burst, it will release the reserved bandwidth and wait until the next burst is ready to be transmitted. If the source receives an NACK, it backs off for a period of time and re-attempts.

- Pseudo Code for manipulation at each node of the network:

```
Forever
{
  Wait for a Request
  If (Minimum Rate > Available Rate)
  {
       Allowed Rate = 0;
       Send NACK (Allowed Rate);
  }
  else
  {
      If (Requested Rate <= Available Rate)
      {
          Allowed Rate = Requested Rate;
          Available Rate = Available Rate - Allowed Rate;
          Send ACK (Allowed Rate);
      }
      else
      {
```

```
            Allowed Rate = Available Rate;
            Available Rate = 0;
            Send ACK (Allowed Rate);
        }
    }
}
```

- **Pseudo Code for Source:**

```
If (a burst is ready)
{
    Choose Requested Rate such that
    Minimum Rate < Requested Rate < Peak Rate
    Forever
    {
        Send_Request (Required Rate, Minimum Rate);
        Wait for Acknowledgment;
        If (ACK) is received
        {
            Transmit at Allowed Rate;
            Release the reserved bandwidth;
            Wait for next burst;
            Choose Requested Rate such that
            Minimum Rate < Requested Rate < Peak Rate
        }
        If (NACK) is received
        {
            Back off for a random period of time
        }
    }
}
```

3.1 Modified Fast Reservation Protocol/Immediate Transmission (MFRP/IT):

MFRP/IT is very similar to MFRP except that the source assumes that the Requested Rate will be granted and starts transmitting before receiving an ACK from the network. It keeps a copy of the data it is transmitting as a back up. If the source receives an ACK from the network,

it will continue transmitting. The rate of transmission will now be changed to allowed rate. It also discards the backup copy of the transmitted data. If it receives a NACK, it will stop transmitting. It will back-off and retry at a later time as in MFRP.

- **Pseudo Code (Source)**

```
If (a burst is ready)
{
    Choose Requested Rate such that
       Minimum Rate < Requested Rate < Peak Rate
    Forever
    {
       Send_Request (Requested Rate, Minimum Rate);
       Start transmitting data at Requested Rate;
       Keep a copy of the transmitted data as a backup;
       If (ACK) is received
       {
          Change transmission rate to Allowed Rate;
          Continue Transmission;
          Release bandwidth at the end of this burst transmission;
          Discard backup copy of transmitted data;
          Wait for next burst;
          Choose Requested Rate such that
             Minimum Rate < Requested Rate < Peak Rate
       }
       If (NACK) is received
       {
          Stop transmission;
          Back off for a random period of time;
       }
    }
}
```

3.2 Advantages of MFRP and MFRP/IT

- MFRP (MFRP/IT) Server more fairly accepts sources with different peak rate as compared with FRP, FRP/IT and AFRP. Since the Requested Rate is less than the Peak Rate and the Allowed Rate can be as low as the Minimum Rate.
- Utilizes the physical bandwidth more effectively as more sources are accepted, hence it can carry higher load.
- The admission delay is very low.
- The blocking probability is also low since, the source is ready to accept the minimum rate as the allowed rate unlike other schemes.

3.3 Disadvantages of MFRP and MFRP/IT

- The source must send the minimum acceptable rate along with the desired (requested) rate and the network must send the rate that it can support (Allowed Rate) with acknowledgment, though no extra hardware is required. It only requires more software overhead.
- The transmission delay is more than other schemes but the overall delay i.e., admission delay + transmission delay is much lesser than other schemes.

3.4 Performance Evaluation of MFRP and MFRP/IT:

The performance of MFRP and MFRP/IT is compared with the existing protocols, namely FRP, FRP/IT and AFRP via simulation. Both single rate and multi rate over single stage and multi stage ATM LANs and WANs are used in this simulation. This model can be extended to any depth and the results obtained here are valid for any general configuration.

The performance of the different protocols are compared based on blocking probability, end-to-end delay and carried load.

Blocking probability is the probability of a source being rejected by the network, as the requested bandwidth is more than the available bandwidth. It calculates the ratio of blocked requests and total requests.

End-to-end delay (overall delay) is the sum of admission delay and transmission delay. Admission delay is the time taken for a source to be accepted by the network. It includes the back-off delay. Transmission delay is the time from the start of acceptance by the network till the end of bandwidth release to the network.

Carried Load (Network Throughput) is the data rate at which the network has actually transmitted.

All protocols are simulated under two configurations (single stage and multi stage). Under each configuration, single rate (all sources transmit at the same rate) and multi-rates (each source can transmit at a different rate) are used. The simulation under each condition is run for 10 seconds and an average of 3 runs is used for evaluation. Three different traffic types are used for simulating each configuration.

The characteristics of the different traffic types are as follows:

Table 3.1: **Simulated Traffic Characteristics**

Traffic Type	**Mean Rate (Mbps)**	**Peak Rate (Mbps)**
Type 0	10	35
Type 1	15	55
Type 2	20	85

All configuration parameters used in the simulation are listed in the next table.

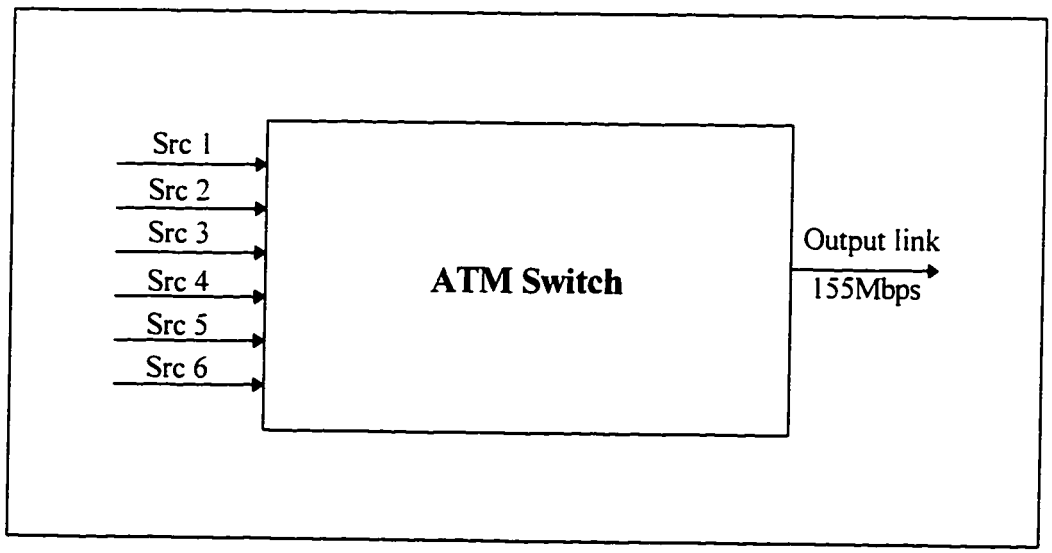
Table 3.2: **Configuration Parameters**

Parameter Description	**Parameter Value**
Network	Single-stage and Multi-stage
Traffic	Single-rate and Multi-rate
Source to switch propagation time	0.01 milliseconds
Switch to switch propagation time (WAN)	5 milliseconds
Switch to switch propagation time (LAN)	0.1 milliseconds
Arrival process Random	Burst Length
5 Mbits	Back-off Period
3 * (Mean burst transmission time) Switch transmission link speed	155 Mbps
Traffic Load	90 Mbps - 180 Mbps
Requested Rate (MFRP) 1.25 * Mean Rate Increment Factor (AFRP) 0.125	Decrement Factor (AFRP) 0.5

3.4.1 Single Stage Configuration:

In this configuration six sources share a common ATM switch with an output link capacity of 155 Mbps (figure 1). FRP and AFRP are compared with MFRP and FRP/IT is compared with MFRP/IT.

Figure 3.1: **Single Stage Configuration**



3.4.1.1 Single-Rate Simulation:

All sources carry either type 0, type 1 or type 2. Both FRPs and FRP/ITs are simulated. For simulation results of FRPs please refer Figure 2. Both LAN and WAN environments have the same results as the source to switch distance is the same for both the cases and the cells pass through only one switch.

3.4.1.2 Multi-Rate Simulation:

Source 1 and source 4 carry type 0 traffic, source 2 and source 5 carry type 1 traffic and source 3 and source 6 carry type 2 traffic. The single stage multi rate simulation results of FRPs and FRP/ITs are shown in figures 3.3, 3.4, 3.5 and 3.6.

3.4.2 Evaluation:

Both MFRP and MFRP/IT out perform FRP, AFRP and FRP/IT, under single as well as multi rate, in both LAN and WAN environments. Simulation results show that with MFRP, there is a significant increase in the network throughput and decrease in the overall delay and blocking probability. In multi-rate simulation, the network throughput is reduced and the blocking probability as well as the overall delay is increased, as compared to single-rate simulation. In a multi-rate configuration, there are three different traffic types competing for the same limited physical link bandwidth. It can be seen from the results that, type 0 is favored the most, followed by type 1 and type 2. This is because the network tends to accepts VCs asking for lower request rate more faster (dependent on available bandwidth) than that of higher request rate. As the blocking probability increases, the overall delay increases and the throughput decreases. MFRP and MFRP/IT have relatively smaller blocking probability than FRP, AFRP and FRP/IT since the request rate is chosen to be in between the minimum and peak rates. The smaller blocking probability in MFRP and MFRP/IT leads to smaller overall delay and higher network throughput. Simulation results also show that FRP/IT performs better than FRP and similarly, MFRP/IT performs better than MFRP. This shows that MFRP/IT holds the original design goal of FRP/IT.

Figure 3.2: **Single Stage Single Rate [FRPs]**

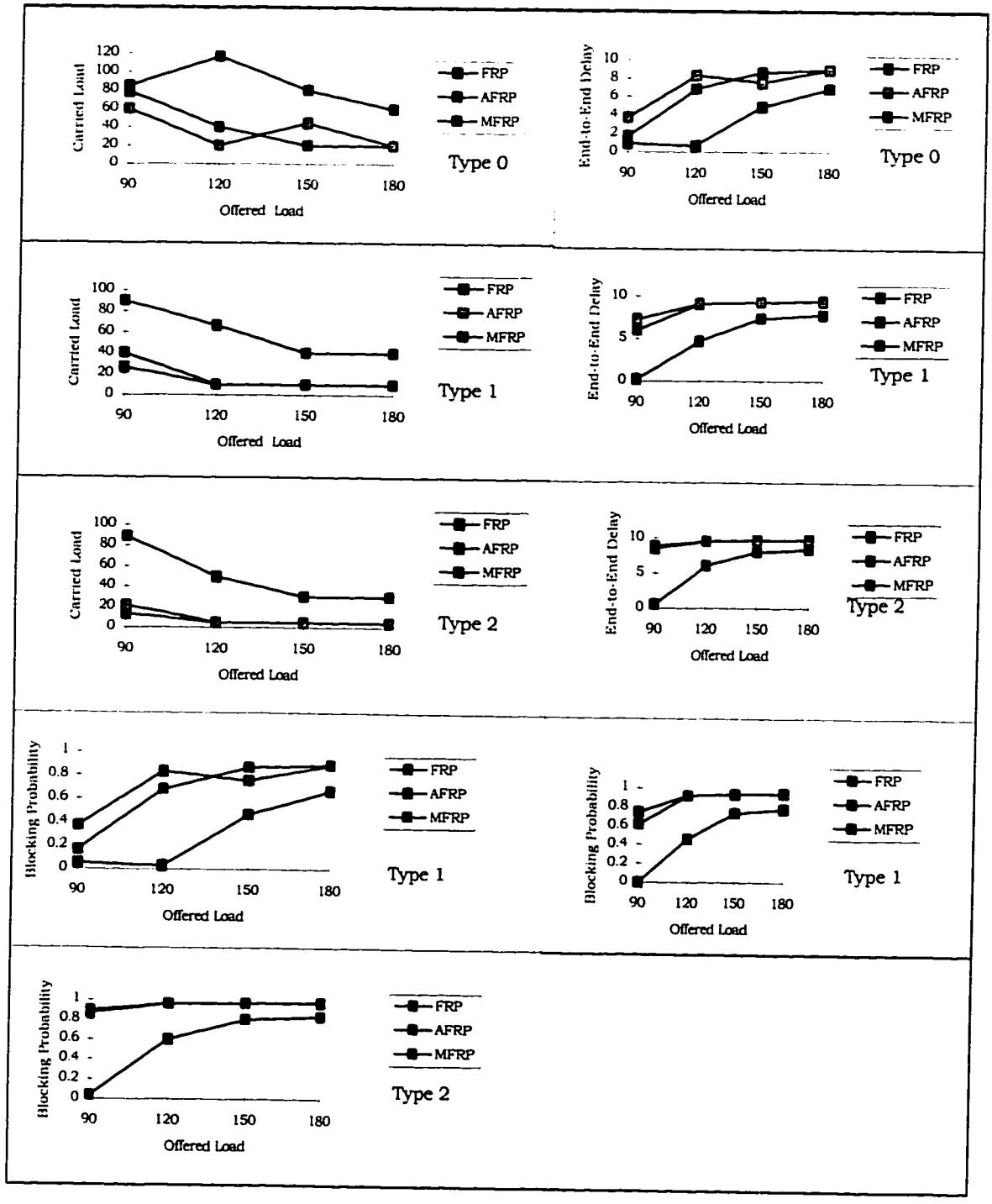


Figure 3.3: **Single Stage Multi Rate [FRPs] - LAN**

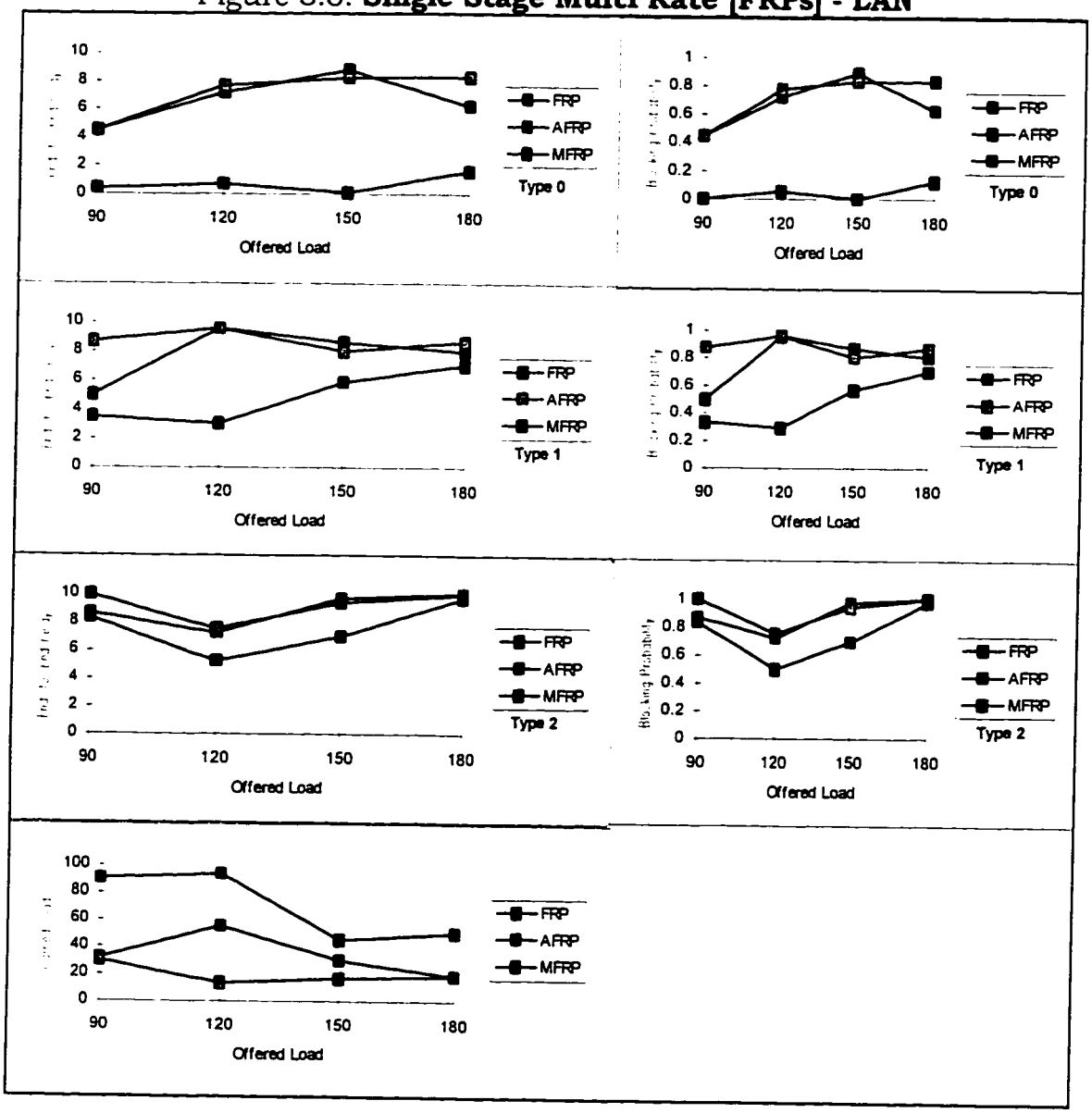


Figure 3.4: **Single Stage Multi Rate [FRPs] - WAN**

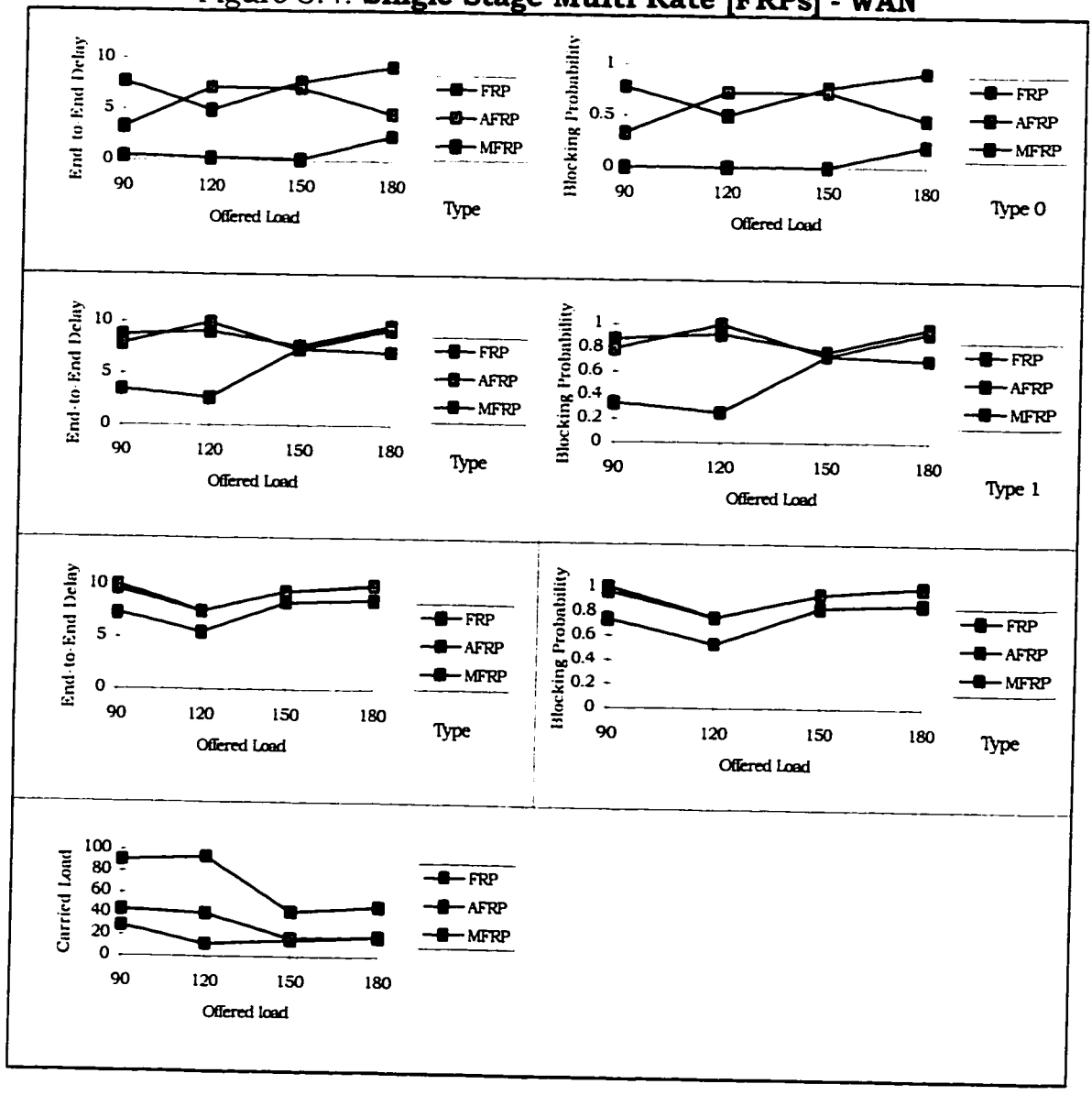


Figure 3.5: **Single Stage Multi Rate [FRP/ITs] - LAN**

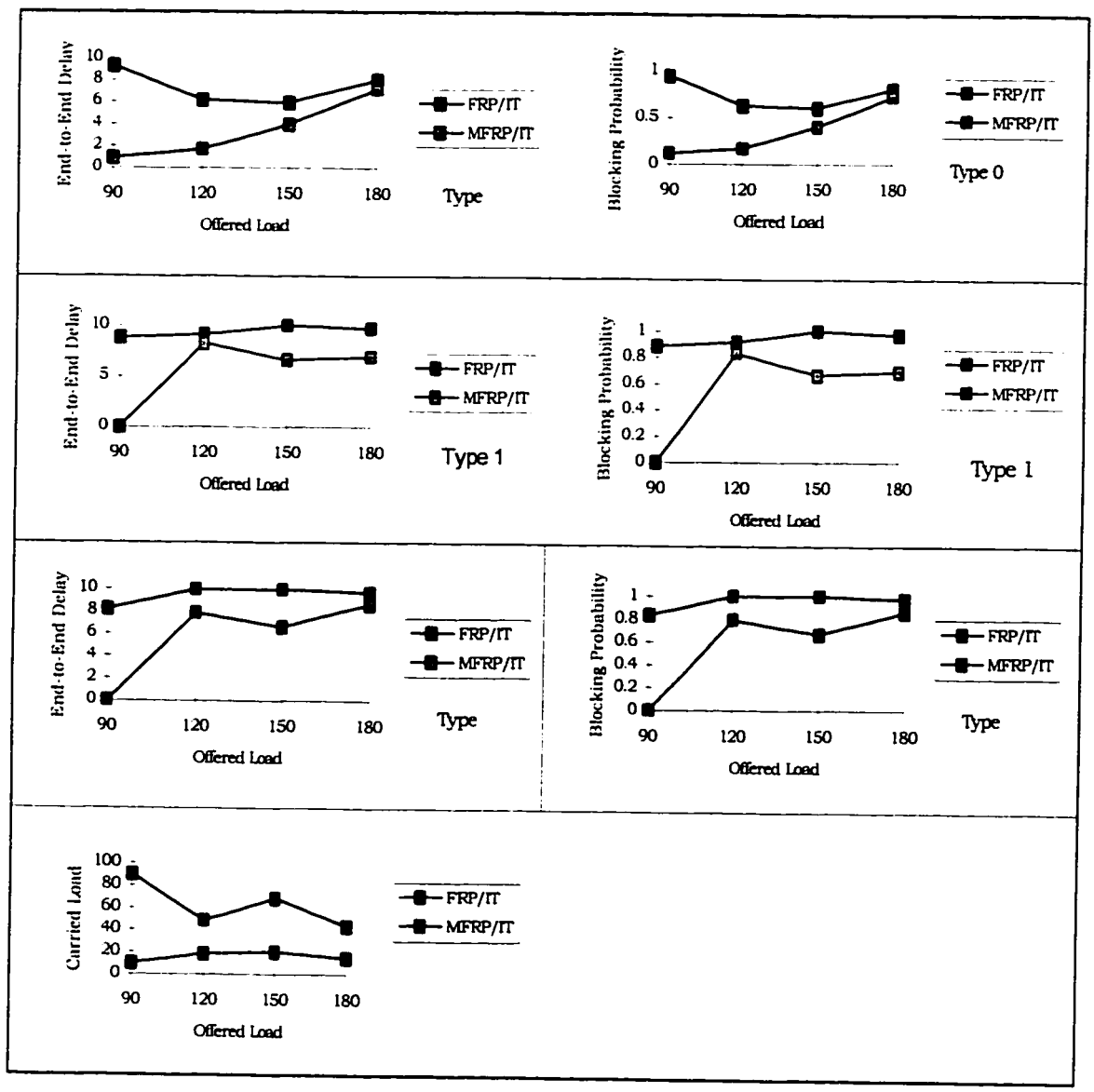
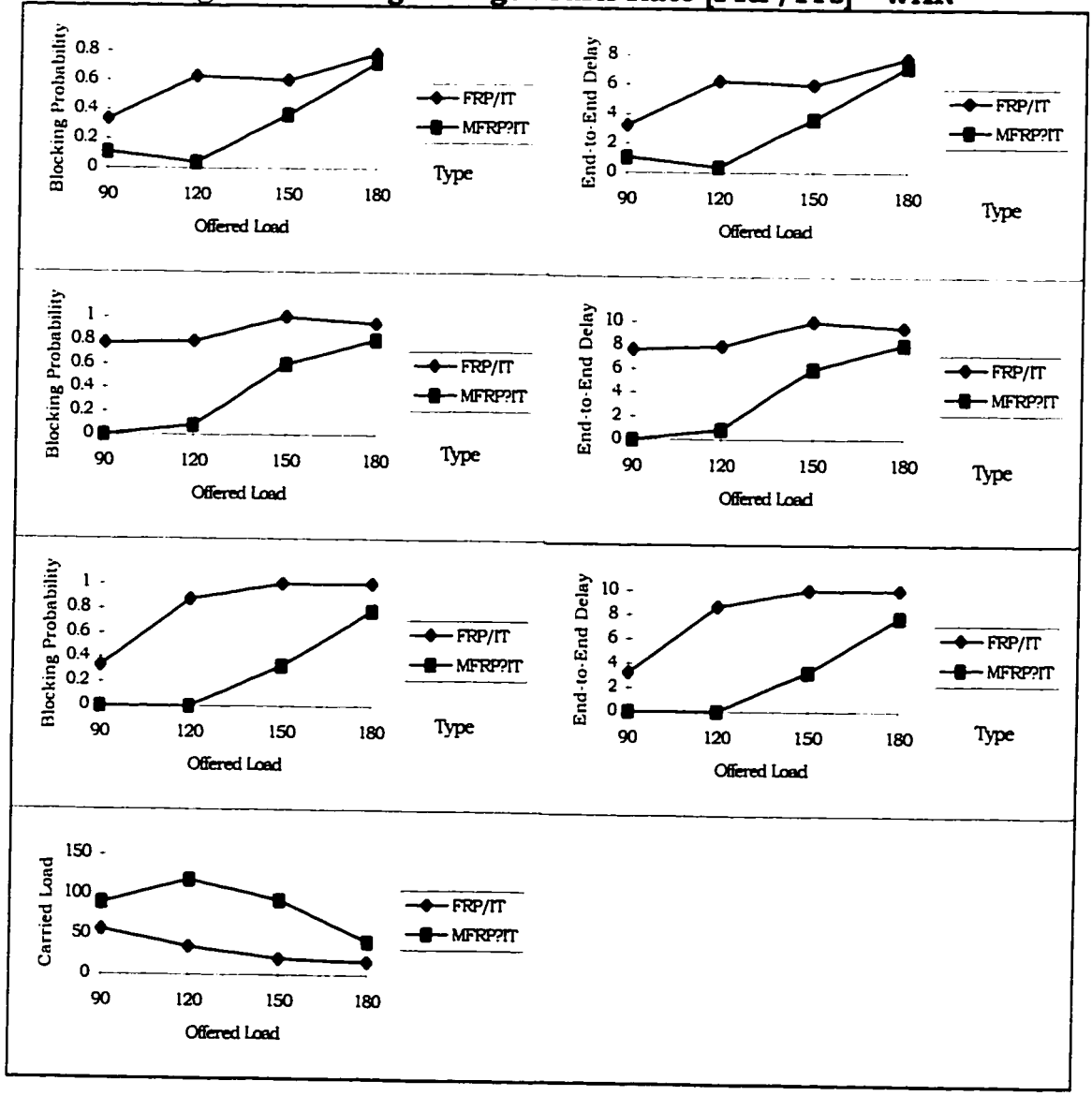


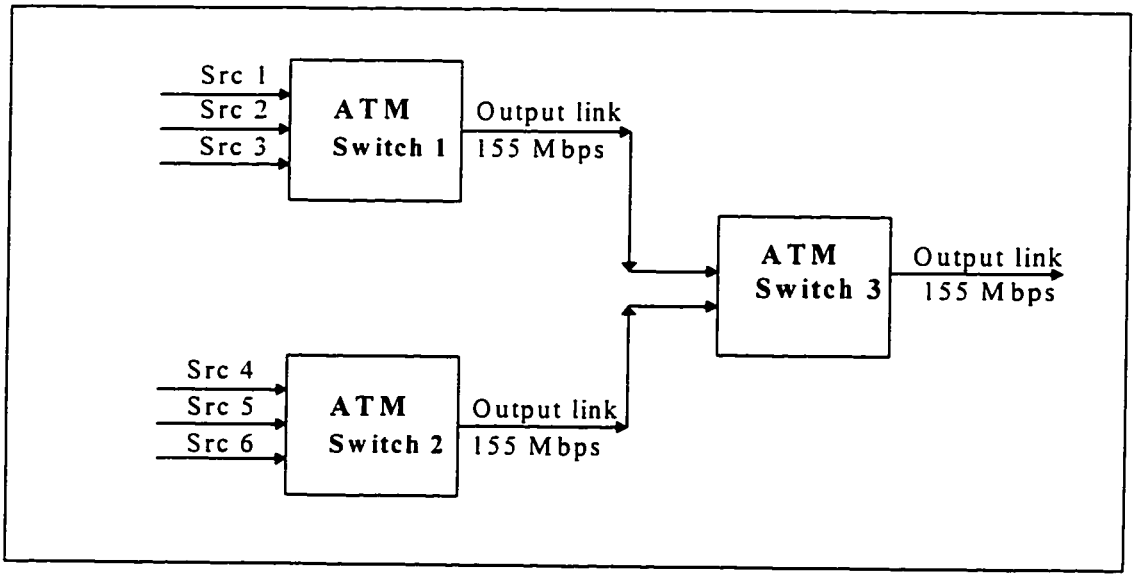
Figure 3.6: **Single Stage Multi Rate [FRP/ITs] - WAN**



3.5 Multi Stage Configuration:

In this configuration, six sources share two local ATM switches with output link capacity of 155 Mbps. These two switches are connected to an other ATM switch, whose output link capacity is 155 Mbps.

Figure 3.7: **Multi Stage Configuration**



3.5.1 Single-Rate Simulation:

All input sources on both ATM switches carry either type 0, type 1 or type 2, which in turn is transmitted to the third ATM switch. Both FRPs and FRP/ITs are simulated, in LAN as well as WAN environments. The simulation results of multi stage single rate FRPs are shown in figures 3.8 and 3.9.

3.5.2 Multi-Rate Simulation:

Source 1 on switch 1 and source 4 on switch 2 carry type 0 traffic, source 2 on switch 1 and source 5 on switch 2 carry traffic type 1 and

source 3 on switch 1 and source 6 on switch 2 carry type 2 traffic. The output from these two switches are transmitted to the third switch that transmits to the destination. For simulation results of FRPs and FRP/ITs, please refer figures 3.10, 3.11, 3.12 and 3.13.

3.5.3 Evaluation:

Under both single and multi rate, in LAN as well as WAN environments, MFRP and MFRP/IT outranks FRP, AFRP and FRP/IT. Though, the data passes through more number of switches in a multi stage network, the characteristics of MFRP are similar under both single and multi stages. In multi-stage configuration, both FRP and AFRP suffer from high blocking probability and low throughput. The network throughput has diminished in this configuration as the blocking probability (back-off delay) dominates the overall performance. Though the performance of MFRP and MFRP/IT is not the same as in single-stage configuration, it continues to perform better than FRP and AFRP due to its flexible bandwidth acceptance capacity.

From the simulation results, it is clear that the overall delay in all configurations is directly proportional to the blocking probability. As the blocking probability is much lesser in MFRP and MFRP/IT, its performance is improved significantly as compared to FRP, AFRP and FRP/IT.

Figure 3.8: **Multi Stage Single Rate [FRPs] - LAN**

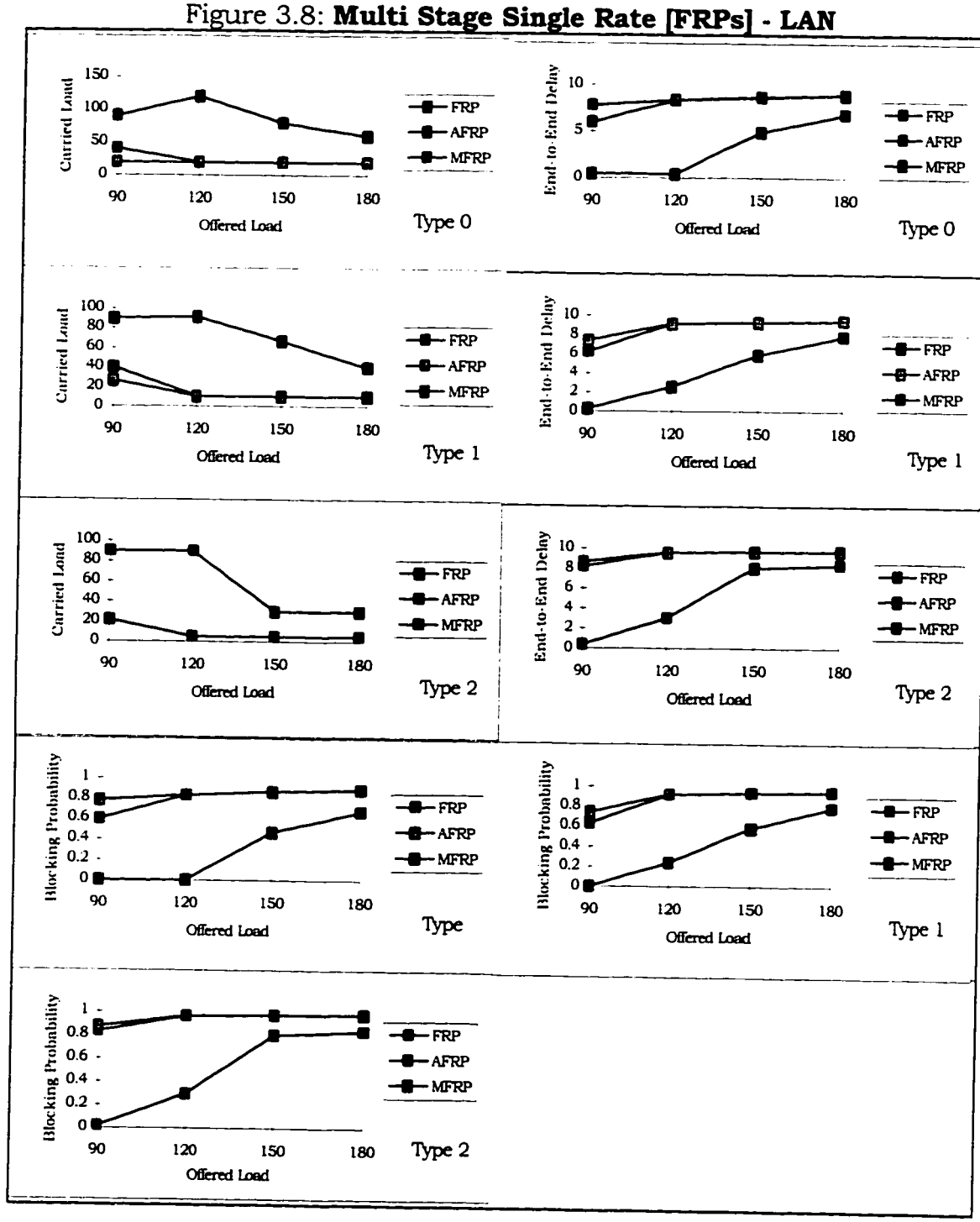


Figure 3.9: **Multi Stage Single Rate [FRPs] - WAN**

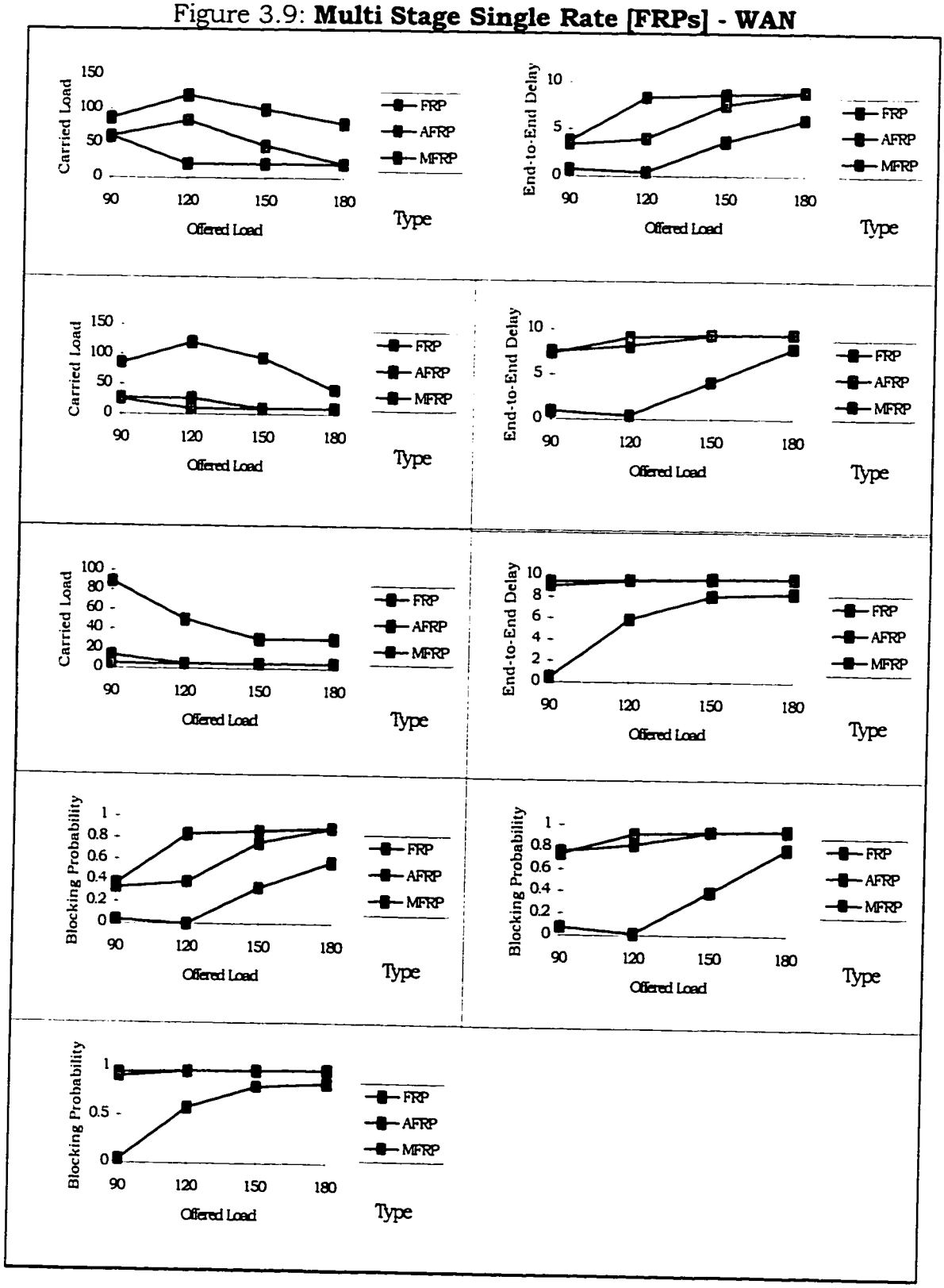


Figure 3.10: **Multi Stage Multi Rate [FRPs] - LAN**

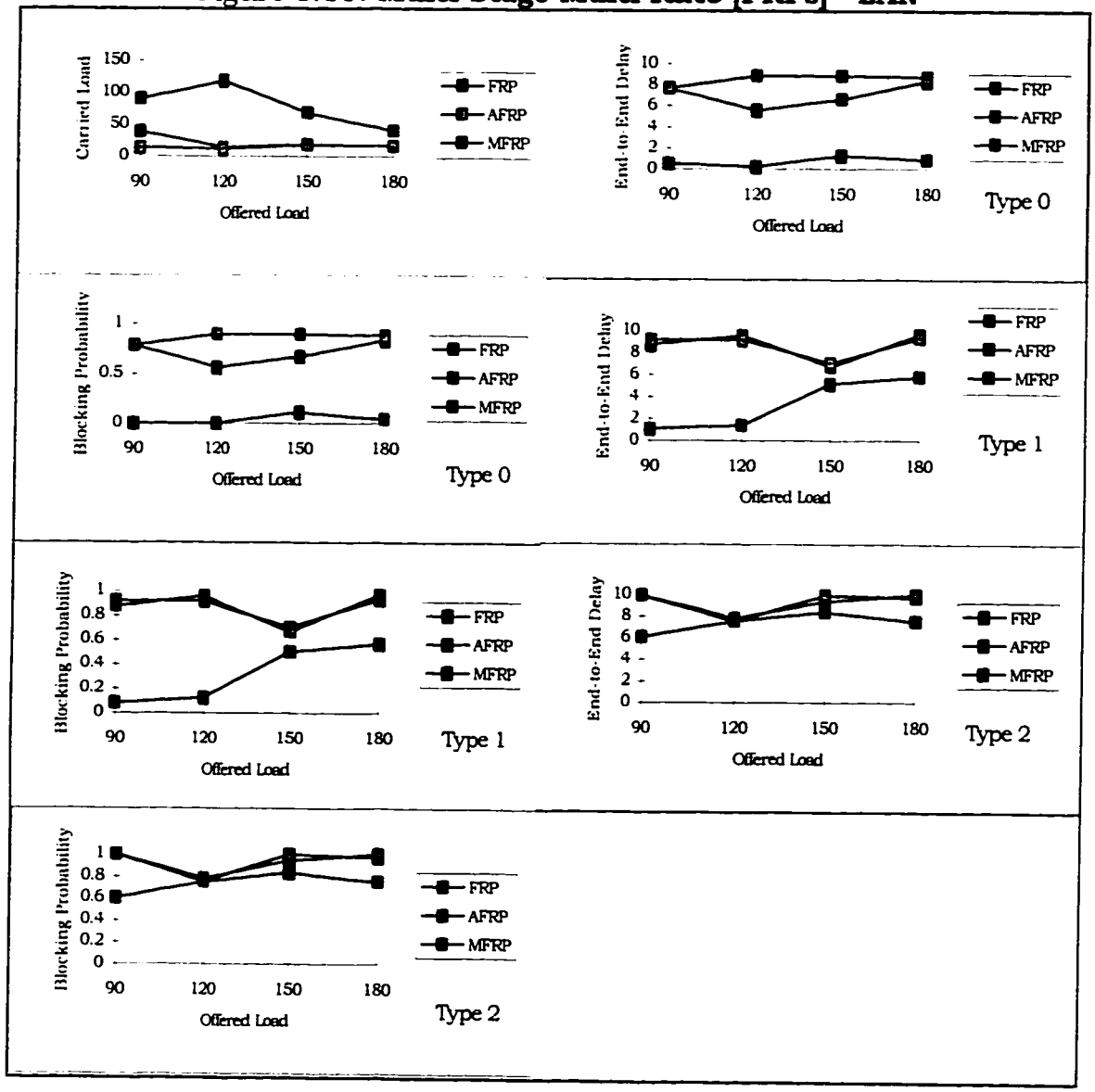


Figure 3.11: **Multi Stage Multi Rate [FRPs] - WAN**

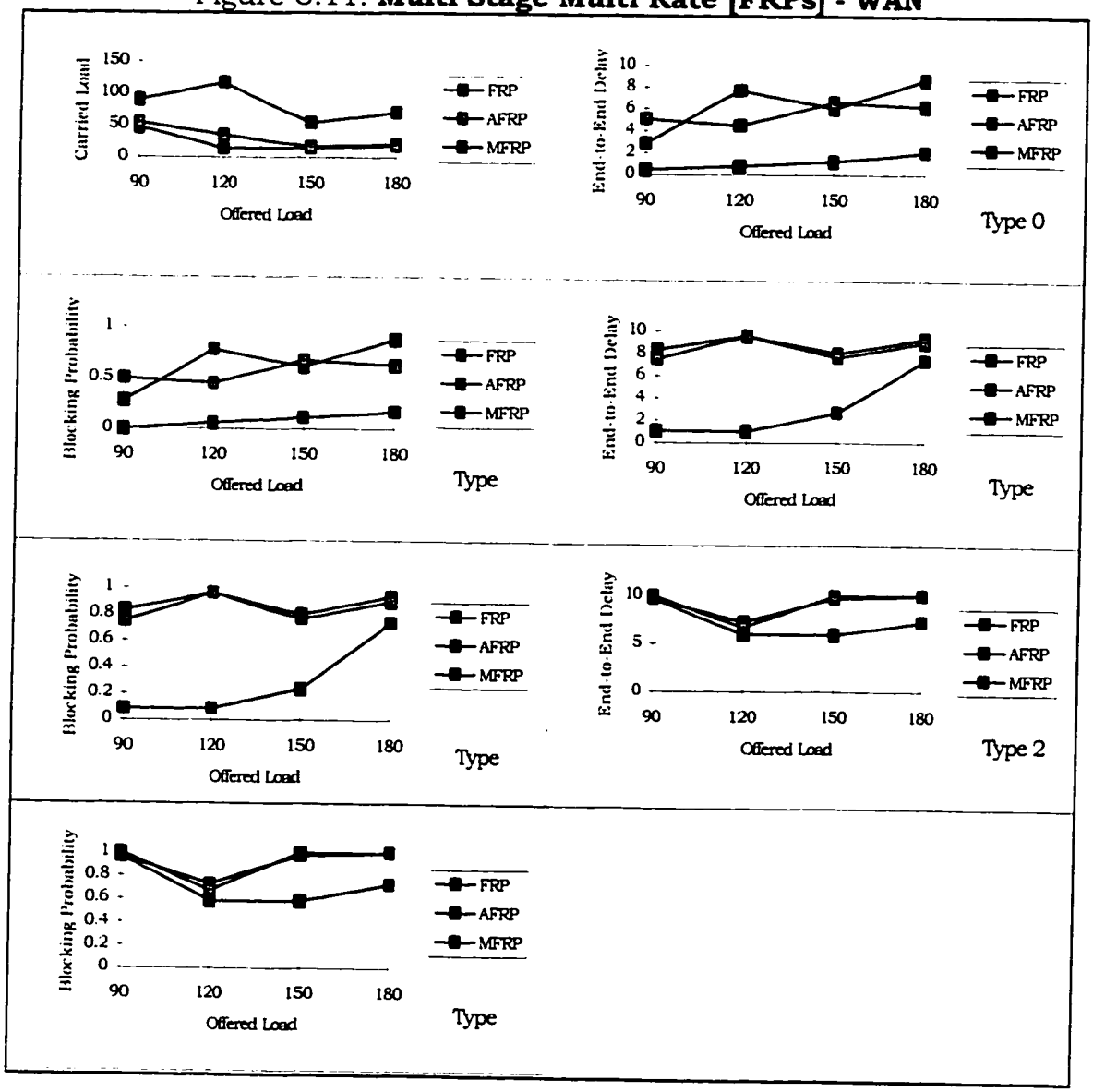


Figure 3.12: **Multi Stage Multi Rate [FRP/ITs] - LAN**

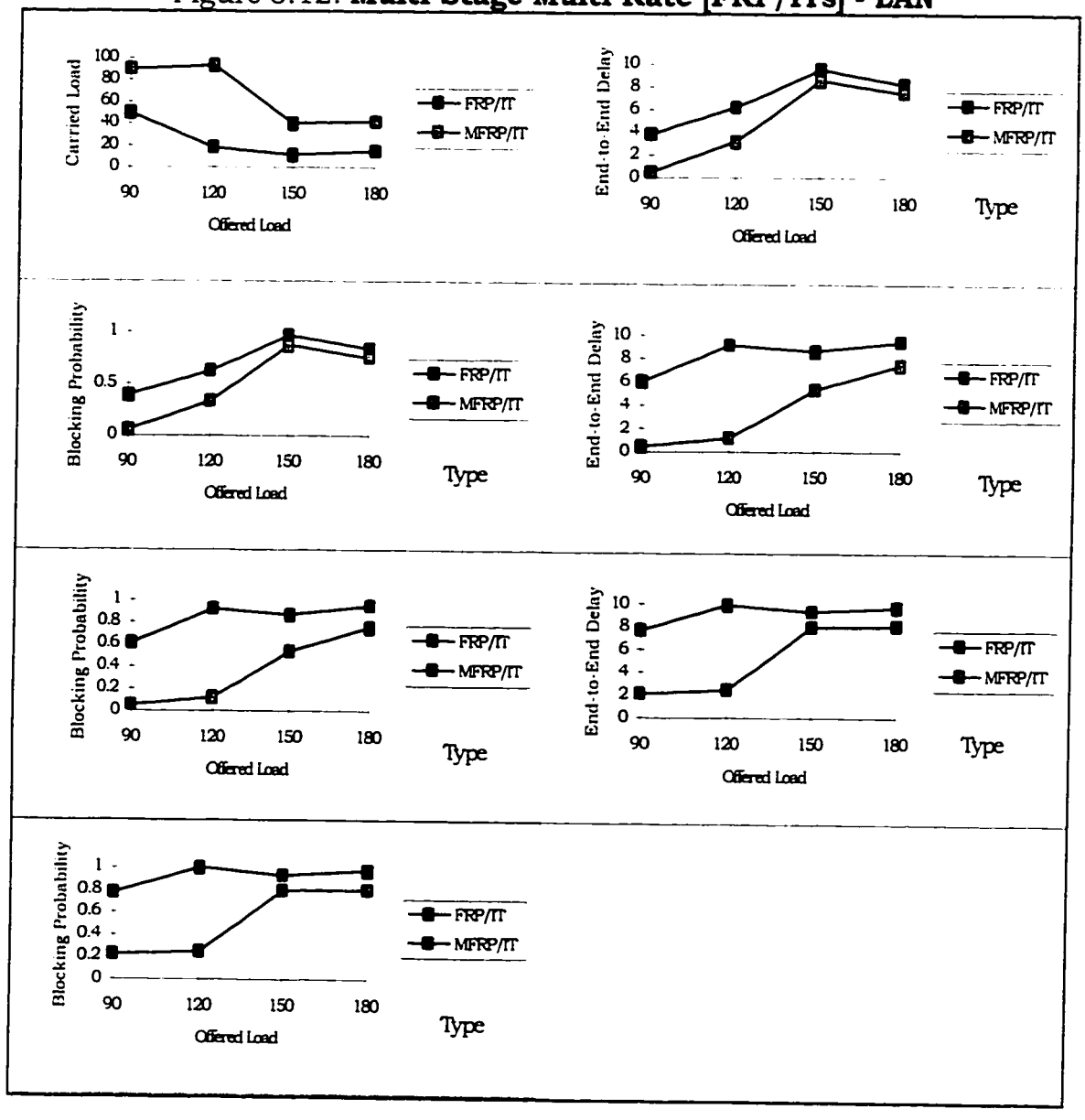
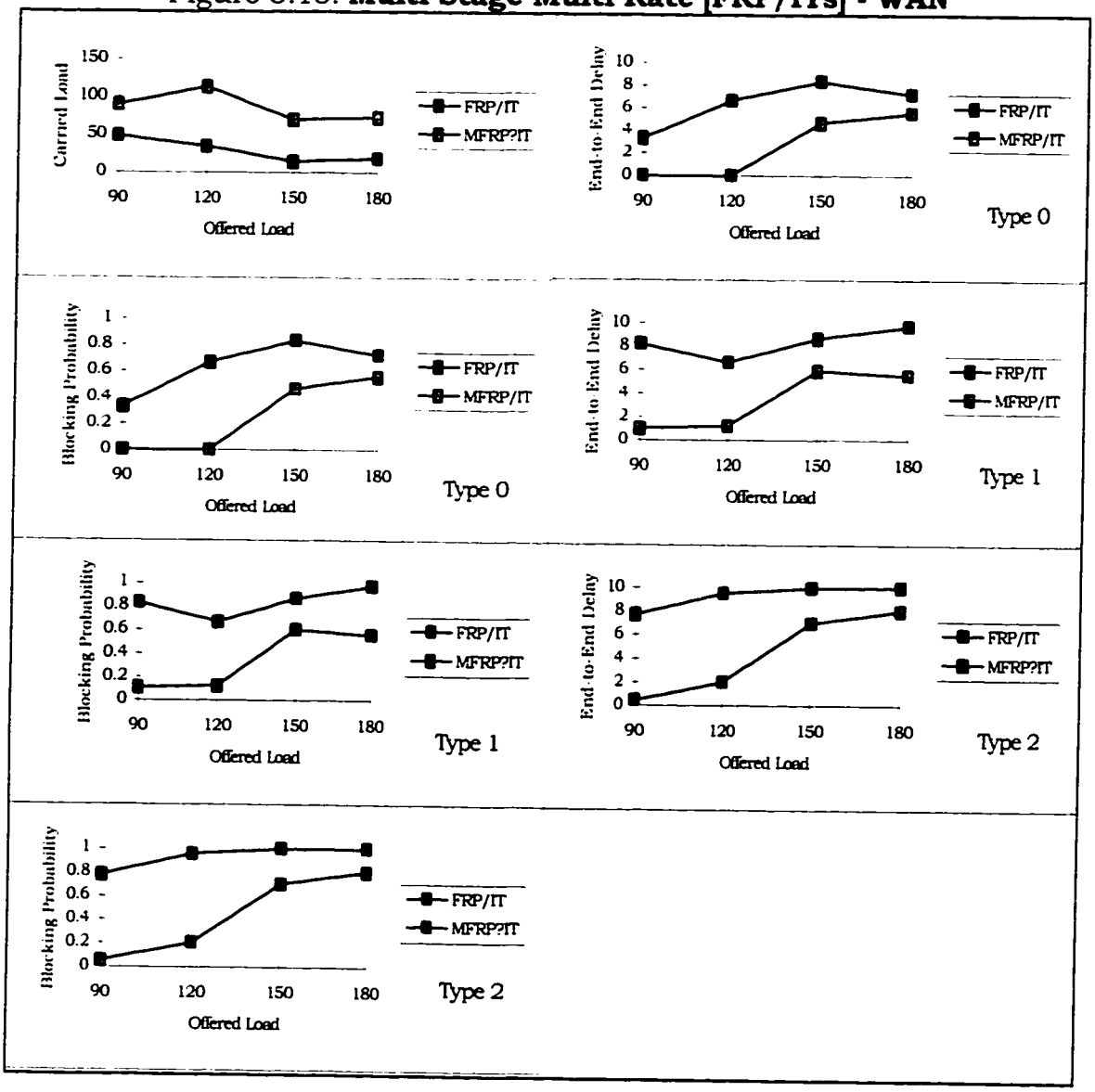


Figure 3.13: **Multi Stage Multi Rate [FRP/ITs] - WAN**



3.6 Performance Analysis of MFRP and MFRP/IT:

3.6.1 Blocking Probability characteristics:

FRP has the highest blocking probability, since the Requested Rate is always the Peak Rate. If a source is blocked, AFRP reduces the Requested Rate for the following admission request. This increases the chance of being accepted by the network, hence the blocking probability is lower than FRP. In MFRP, the Requested Rate is chosen to be in between the Peak Rate and the Mean Rate, rather than the Peak Rate. It also sends the minimum acceptable rate to the network, so that the network, depending on the available bandwidth, can allocate the highest possible rate. This decreases the blocking probability as the source is rejected only if the network cannot allocate the minimum required rate.

3.6.2 Throughput characteristics:

With MFRP, the carried load is tripled when compared to FRP and AFRP. By choosing the Request Rate to be in between the Peak Rate and the Mean Rate, MFRP allows more sources to be admitted into the network. This in turn increases the network throughput.

3.6.3 Delay characteristics:

MFRP has the least admission delay, followed by AFRP and FRP, while the transmission delay is the least in FRP, followed by AFRP and

MFRP. This is due to the fact that in FRP, all sources transmit at their Peak Rate, thus transmitting faster. As the Requested Rate is always the Peak Rate, the overall admission delay is the highest in FRP. AFRP negotiates the bandwidth with the network, thus reducing the admission delay and increasing the transmission delay. MFRP Requests for a rate in between the Peak Rate and the Mean Rate. It also notifies the network the minimum acceptable rate. This decreases the admission delay to a great extent but increases the transmission delay. Though the transmission delay is the highest in MFRP, it has the least overall delay, because the admission delay counterbalances the transmission delay.

3.6.4 Source acceptance characteristics:

Sources with lower peak rate have a higher probability of being accepted by the network than other sources. In other words, sources seeking higher peak rate have higher blocking probability. This unfair acceptance by the network can be controlled by the protocol to an extent.

FRP is the most unfair protocol with respect to source acceptance, since the Requested Rate is always the Peak Rate. AFRP is fairer than FRP, though initially, the Requested Rate is the Peak Rate, since, if a source gets blocked, it reduces the Requested Rate by the Decrement Factor. Thus, increasing the chance of being accepted by the network in the subsequent call request. MFRP is the fairest protocol as it chooses the

Requested Rate to be in between the Peak Rate and the Mean Rate, increasing the chance of being accepted earlier. This in turn allows more sources to be accepted by the network.

The source acceptance characteristics of FRPs and FRP/ITs are clear from the simulation results. Please refer figures 14 and 15 for the "source acceptance fairness" comparison of FRP/ITs in multi rate configurations under both LAN and WAN environments.

Figure 3.14: **Fairness Evaluation [FRP/ITs] - LAN**

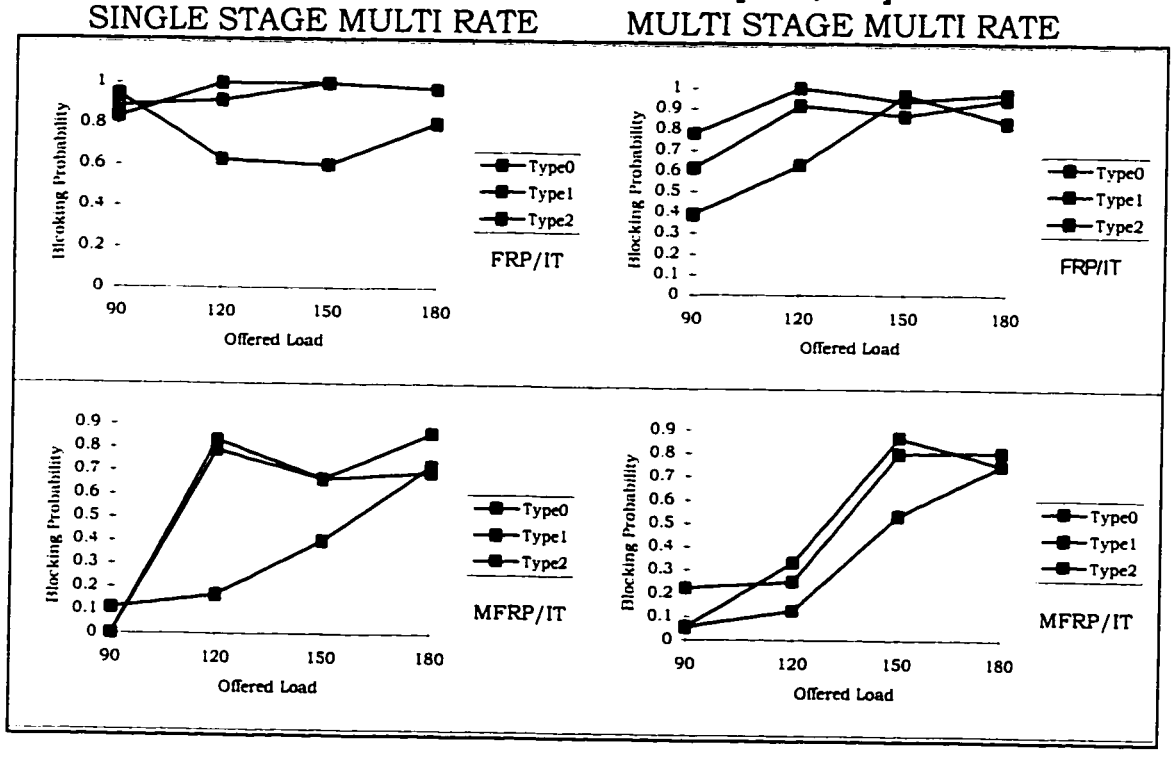


Figure 3.15: **Fairness Evaluation [FRP/ITs] - WAN**

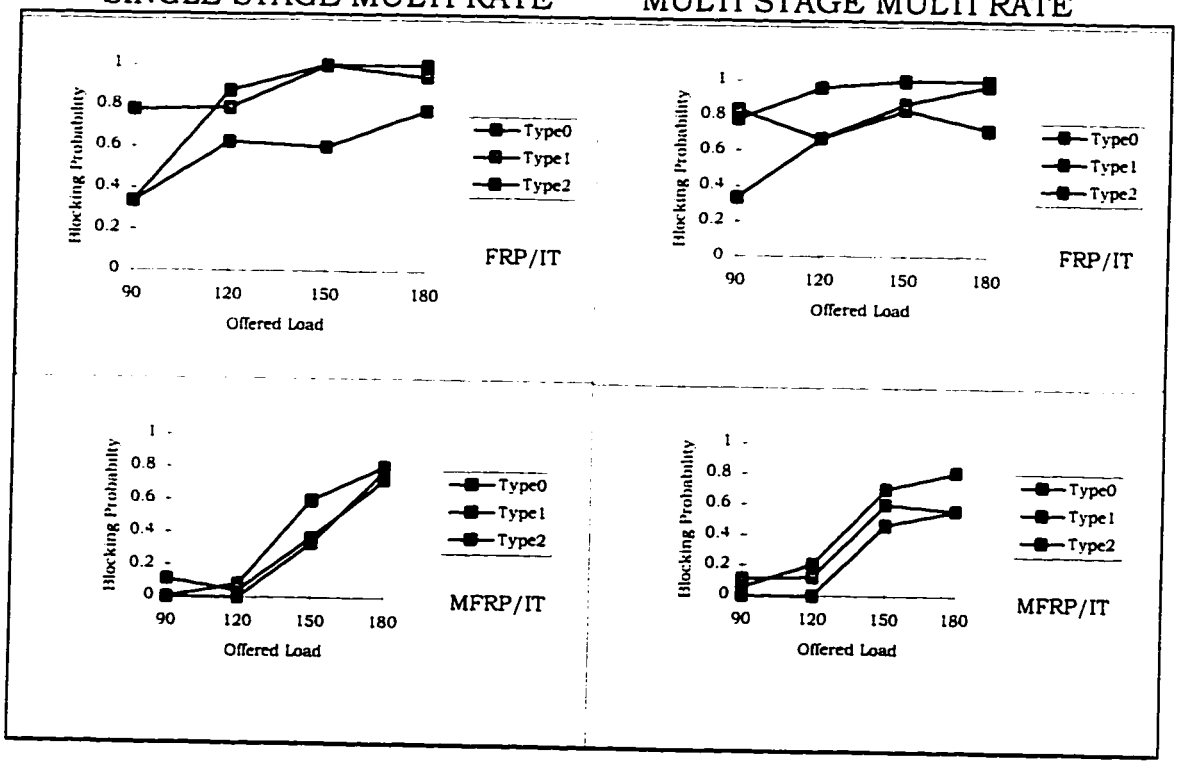


Table 3.3: **Summary of the five FRP Protocols**

	FRP	**AFRP**	**MFRP**	**FRP/IT**	**MFRP/IT**
Mechanism	Request peak rate	Back-and-forth rate negotiation	One time rate negotiation	Request peak rate	One time rate negotiation
Blocking rate	High	Moderate	Low	High	Low
Transmission time	Short	Moderate	Varies (depends on allowed rate)	Short	Varies (depends on allowed rate)
Extra Overhead	None	Source keeps track of current requested rate and increases or decreases its rate depending on ACK or NACK received from switch	One extra parameter at the request cell and the ACK	Source must keep a copy of its data trans. before it receives a ACK from network	Those of MFRP and FRP/IT

4. EXPLICIT ALLOWED RATE ALGORITHM

With statistical multiplexing, it is seen that the bandwidth used by the VBR traffic type has a direct impact on ABR transmission rate. Based on this observation, EARA explicitly specifies the ABR transmission rate. In other words, the switch finds out the available bandwidth and explicitly informs the ABR source its allowed transmission rate by using RM cells. As per the ATM Forum Specifications[23], ABR sources send an RM cell to the switch indicating the current rate that it is using [CCR], for every 32 [Nrm-1] in-rate BRM and data cells transmitted. The fair share of each VC or allowed explicit rate is calculated on the current rate used by that VC.

4.1 Call Admission

A source is admitted into the network if it can support all its requirements. When a CBR or VBR source is admitted, if the used bandwidth becomes more than the total bandwidth, then the allowed explicit rate for ABR sources are calculated [please refer section 4.3 for details on Fair Share calculation] and if there is any change in the allowed rate, then an RM cell will be generated.

4.1.1 Pseudo code for admission:

- AB - Available Bandwidth on switch
- UB - Used Bandwidth on switch
- TB - Total Bandwidth on switch
- ACR - Allowed Cell Rate
- ER - Explicit Rate

CBR:

```
if ((AB < Constant Rate) || (TB < (ΣCBR[Constant Rate] +
                                   ΣVBR[Peak Rate] + ΣABR[Minimum Rate])))
    Send NACK
else
{
    AB = AB - Constant Rate
    UB = UB + Constant Rate
    Send ACK
    if (UB > TB)
        Calculate the allowed ER for all ABR VCs and Send RM cells
        in backward directions.
}
```

RTVBR & NRTVBR:

```
if ((AB < Mean Rate) || (TB < (ΣCBR[Constant Rate] +
                               ΣVBR[Peak Rate] + ΣABR[Minimum Rate])))
    Send NACK
else
{
    AB = AB - Mean Rate
    UB = UB + Initial Rate
    Send ACK
    if (UB > TB)
        Calculate the allowed ER for all ABR VCs and Send RM cells
        in backward directions
}
```

ABR:

```
if ((AB < Minimum Rate) | | (TB < (ΣCBR[Constant Rate] +
                                 ΣVBR[Peak Rate] + ΣABR[Minimum Rate])))
    Send NACK
else
{
    AB = AB-Minimum Rate
    Set ACR = Initial Request Rate
    Send ACK
    Calculate the allowed ER for all ABR VCs, including the one
    starting.
    if (new allowed ER > current ACR)
        Send RM cell in the forward direction
    if (new allowed ER < current ACR)
        Send RM cell in the backward direction.
Note: RM cells are not generated if new allowed ER is equal to current
ACR.
}
```

4.2 Flow Control

The source and the destination behavior for EARA are as specified in the ATM Forum Traffic Management Specifications. Given below are the highlights of the source and destination algorithms with detailed EARA switch algorithm.

4.2.1 Source Algorithm

ABR source starts transmitting at the negotiated initial rate. For every Nrm-1 data cells transmitted, it generates an RM cell indicating the current cell rate. The RM cell will go to the destination and return back to the source with the allowed rate explicitly specified by the switch(es). Each ABR source keeps the time-stamp of the last RM cell it received. This time-

stamp indicates the time of RM cell creation. When an ABR source receives an RM cell, it checks the stored time-stamp against the one in the RM cell. If the time-stamp on the RM cell is greater than the stored time-stamp, then it changes its allowed rate [ACR] and keeps the new time-stamp. Otherwise, it discards the RM cell and continues transmitting at the current ACR.

Pseudo code [source]

```
Initialize:
        data_cells = 0;  //# of data cells transmitted-
                            counter to send RM cells
        time-stamp = start-time;

if (time to transmit next cell)
{
        if (data_cells == Nrm-1)
                Generate and send a FRM cell
                Reset data_cells to 0
        else
                Generate and send Data cell
                Increment data_cells
}

if (BRM cell received)
{
        if (time-stamp < time-stamp on BRM cell)
                Update allowed cell rate [ACR] to ER on RM cell
                Reset time-stamp to the time-stamp on RM cell
        else
                Discard BRM cell
}
```

4.2.2 Switch Algorithm

To avoid congestion and delay, the switch needs to keep track of the current rate used by CBR and VBR traffic class together and ABR class. In order to achieve it, the following parameters are required:

Table 4.1: **Switch Parameter List**

Parameter	Description
Rate Monitor Interval	Time interval to check current load
Total Cells	# of cells that the switch can process in a given interval of time
Reserved Cells	# of CBR and VBR cells processed in a given interval of time
ABR Cells	# of ABR cells that can be processed in a given interval of time

The switch monitors its load periodically, according to the rate_monitor_interval. This interval is changed if needed, while admitting a VBR source. It depends on the peak rate of all admitted VBR VCs. Rate-monitor interval can be defined as the time required for the VBR VC (which has the highest peak rate among all VBR VCs currently in the network) to transmit a cell at its peak rate. Initially, this interval is set to a very high value. When a VBR source is admitted, the new rate_interval is calculated. If the new rate_interval is less than the current rate_interval, then the current rate_interval is changed to the new rate_interval. The new rate_interval is the time required to send a cell at peak rate. When the

rate_interval changes, the total number of cells the switch can process in that interval of time must be updated.

When an RM cell is received from the source, the *current cell rate* used by the source is noted on the switch. For fair allowed explicit rate calculation, this used rate is taken into account, rather than the allowed rate [use-it-or-lose-it].

In order to find out the available bandwidth for ABR traffic class dynamically, the switch must keep a count of the total number of cells it got in an interval from CBR and VBR sources [reserved_cells]. The number of ABR cells that the switch can handle without causing congestion and delay as well as utilize the bandwidth efficiently, is:

new ABR_cells = total_cells - [reserved_cells + remaining_cells]

where the remaining_cells is the number of cells currently in queue (CBR, RTVBR, NRTVBR, ABR & RM), if any, to be processed.

If the newly calculated number of ABR cells is different from the current number of ABR cells the switch can handle, then the fair share of each ABR VC is calculated as:

Fair Share = new_ABR_cells/cur_ABR_cells

For each ABR VC,

*new allowed_rate = Fair Share * current_cell_rate*

If the new allowed_rate is less than the current allowed_rate, it implies that the switch bandwidth is over-utilized and there is a potential for congestion. Hence RM cells with explicit rate are sent in backward direction to the source. If the new allowed rate is more than the current allowed rate, it implies that the switch bandwidth is under-utilized. Hence RM cells are sent in the forward direction to the destination, which in turn reverse the direction and sends it back to the source. This is done in order to ensure that all switches in the path from source to destination can support the explicit rate set in the RM cell. If any switch on the path cannot support this rate, the RM cell is dropped by that switch.

Pseudo code [switch]

```
Initialize:
        rate_monitor_interval=highest peak_rate among all active VBR VCs
        total_cells = total bandwidth * rate_monitor_interval [units of time]
        reserved_cells = 0;
        time_counter = 0;
        cur_ABR_cells = 0;
        remaining_cells = 0;

if (received data cell)
        Increment remaining_cells
        if (traffic type is CBR or RTVBR or NRTVBR)
                Increment reserved_cell

if (transmitted data cell)
        Decrement remaining_cells

if (received RM cell)
        Update current cell rate of that ABR source to the CCR in RM cell
        if (ACR of that ABR source < ER in RM cell)
```

```
            Forward to next node
        else
            Drop that RM cell

if (time_counter == rate_monitor_interval)
        if ((reserved_cells + cur_ABR_cells) != total_cells)
        {
            new_ABR_cells = total_cells - [reserved_cells+remaining_cells]
            Fair_Share = new_ABR_cells/cur_ABR_cells
            For each ABR VC,
                new_allowed_rate = Fair_Share*current_cell_rate
                if ((new_allowed_rate < current_allowed_rate)
                    Generate and send RM cell in the backward
                    direction to the source
                else
                    Generate and send RM cell in the forward
                    direction.
        }
        Reset reserved_cells to 0
```

4.2.3 Destination Algorithm

If an RM cell is received, the destination changes its direction and sends it back to the source.

Pseudo code [destination]

```
if (received RM cell)
        Reverse the direction by setting DIR to 1
        Send it back to the source
```

4.3 Advantages

- The queue length at any time will be a minimum and it can grow dynamically as needed.
- The maximum queue size needed at any instance is,

$$\Sigma((\text{Peak-Rate})\text{VBR} * (\text{Max.Burst-Size})\text{VBR}) - \Sigma((\text{Mean-Rate})\text{VBR} * (\text{Avg.Burst-Size})\text{VBR}$$

- ABR users are granted the maximum possible rate at any instance. Hence, they can transmit faster.
- RM cells are generated only when required by the network. It is used for both decreasing as well as increasing the bandwidth allocated to ABR VCs.
- Only one rate monitor is required for keeping a count of the number of cells received from CBR and VBR class in a time interval.
- Congestion detection depends mainly on the current VBR traffic transmission rate.
- If a potential for congestion is detected, then RM cells are sent to the source directly, so that the source can decrease the rate quickly. Also, as the switch is not under congested state, when RM cells are generated, the chances of the RM cells not reaching the source is decreased significantly. In other words, RM cells are generated as a preventive congestion mechanism rather than using RM cells for notifying the ER, when the switch is congested.
- By setting the allowed cell rate explicitly, policing will be more robust.
- No major hardware requirements.

4.4 Disadvantages:

- Initiating RM cells to all ABR users, when used bandwidth is more or less than the total physical bandwidth.

5. PERFORMANCE ANALYSIS

Explicit Allowed Rate Algorithm (EARA) is compared with Proportional Rate Control Algorithm (PRCA)[8,11,22] and Explicit Rate Indication Congestion Avoidance (ERICA)[21] algorithm. All the three algorithms are simulated in two different configurations. Given below is a overview of the algorithms used in the simulation and an analysis of its results under both configurations.

5.1 Proportional Rate Control Algorithm [PRCA]

5.1.1 Source Algorithm

- ABR source starts transmission at initial rate.
- When the source receives an RM cell, it checks the CI bit. When this bit is set, it indicates that a switch along the path to the destination is congested. Hence, according to the status of the CI bit, the source changes its transmission rate by multipicative decrease or increase of its current rate.

5.1.2 Switch Algorithm

- The switch keeps track of the total number of cells in the queue, including CBR, RTVBR, NRTVBR, ABR and RM.

- A switch is said to be congested if the number of cells in the queue exceeds the set threshold.
- It sets the CI bit of the data cell, if it is congested.

5.1.3 Destination Algorithm

- Destination sends RM cells periodically to the source.
- If the last data cell it received had the CI bit set, then it will set the CI bit in the RM cell, indicating congestion to the source.

5.2 Explicit Rate Indication Congestion Avoidance [ERICA]

5.2.1 Source Algorithm

- ABR source starts its transmission at the negotiated initial rate.
- It sends an RM cell to the switch periodically, according to the averaging interval determined by the network.
- The source keeps track of the time-stamp of the last RM cell. Initially, it is set to the start time.
- When the source receives the RM cell, it checks the stored time-stamp value and checks it against the time-stamp of the received RM cell. If the time-stamp on the RM cell received is more than the time-stamp on the source, then it adjusts its rate according to the indicated load level and updates its time-stamp. Keeping track of the time-stamp is

required as the switch sends a copy of RM cell backwards, when the switch is congested.

5.2.2 Switch Algorithm

- The switch monitors its load periodically (according to averaging interval), trying to maintain the input load close to the output load.
- When a switch receives an RM cell, it updates the current cell rate used by that ABR source.
- For each averaging-interval period, the switch calculated the fair-share of each ABR VC. Fair-share of a VC is the available bandwidth divided by the total number of ABR sources.
- It then checks the load level, which is the ratio of the input load to the output load.
- If the load level is more than 1, it indicates that there is a potential for congestion. Hence, it makes a copy of the RM cell and sends in the backward direction to inform the source about congestion at its earliest. .
- The available physical bandwidth for ABR sources is calculated by assuming that CBR and VBR sources will be transmitting at the same rate for the next interval.
- A VC's share is the maximum of the fair-share and the current rate divided by the load level.

5.2.3 Destination Algorithm

- When the destination receives an RM cell, it reverses the direction and sends it back to the source.

5.3 Explicit Allowed Rate Algorithm [EARA]

5.3.1 Source Algorithm

- ABR source starts transmitting at the negotiated initial rate.
- It sends an RM cell periodically, indicating the current rate used.
- When it receives an RM cell, it checks the stored time-stamp against the time-stamp on the RM cell. If the time-stamp of the received RM cell is greater than the stored time-stamp, it will reset its transmission rate according to the explicit rate specified by the network and update its time-stamp.

5.3.2 Switch Algorithm

- The switch monitors its load according to the rate-monitor interval. If the total bandwidth used in this interval is not equal to the total physical bandwidth, then fair share of each ABR source is calculated and informed, as needed.
- The switch calculates the fair share of each ABR source, by assuming that the CBR and VBR sources will be transmitting the same number of

cells in the next interval. It also takes into account, the cells that are currently in the queue and needs to be processed in the next interval.

- When the switch receives an RM cell, it updates the current rate used by that VC. If the specified ER is less than the allowed rate for that VC, then it transmits to the next node. Otherwise, it is dropped.

5.3.3 Destination Algorithm

- When the destination receives an RM cell, it reverses the direction and sends it back to the source.

5.4 Congestion Configuration

In this configuration, all VCs pass through two switches[17,23,26]. A total of 46 VCs are simulated. The link between switch 1 and switch 2 gets congested when all VCs are active.

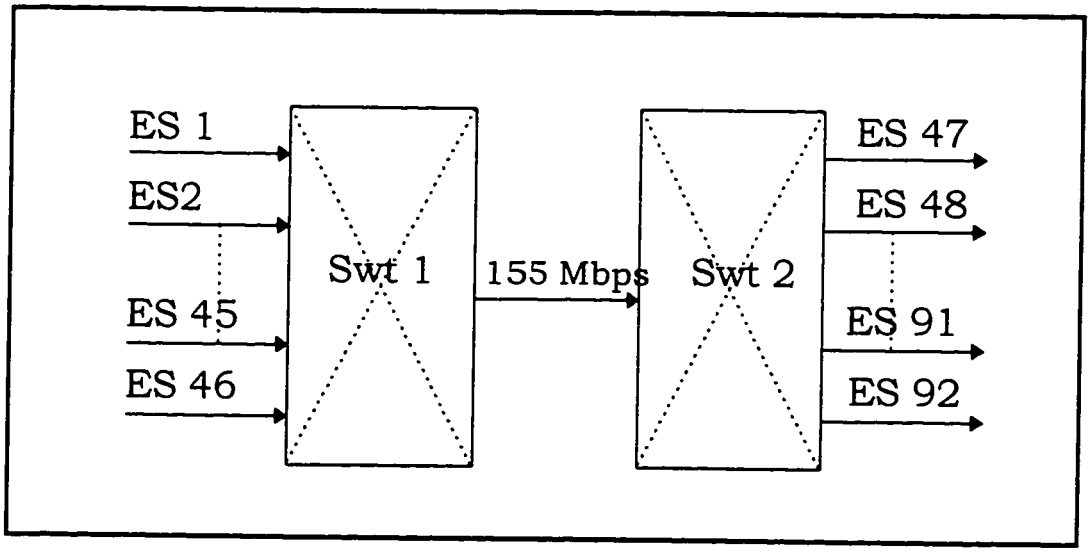


Figure 5.1: **Congestion Configuration**

5.4.1 Generation of VBR Traffic

Typically, VBR applications have varying amount of data transmitted in a continuous manner. Real time VBR traffic is simulated by generating 30 frames per second. In other words, a frame is generated every 33 milliseconds. The number of cells in each frame varies. Hence, having variable bit rate. The number of cells in the n^{th} frame $\lambda(n)$ is determined by the Auto Regressive model [AR(2)][6,12], which is,

$$\lambda(n) = a\lambda(n-1) + b\omega(n),$$

where a and b are constants and $\omega(n)$ is a Guassian random variable with a mean m. The mean $E(\lambda)$ and the auto-covariance of the bit rate C(n) are equal to:

$$E(\lambda) = bm/(1-a);$$

$$C(n)= b^2a^n/(1-a^2);$$

From these two equations, the values of a and b are determined.

Non real time VBR traffic is generated by alternating busy and idle periods. Busy periods are for a constant time (16 milliseconds). The number of frames generated per busy period is either 0 or 1. The number of cells in a frame is determined by AR(2). Idle periods are generated by using exponential distribution with Standard Deviation, B = 1/16 and Mean U, a random number between 0 and 1. The above traffic generated results in the bit rate approximately given in reference [19].

5.4.2 Input Parameters

Table 5.1 **Input Parameters**

Type	Peak Rate (Kbps)	Mean Rate (Kbps)	Init. Rate (Kbps)	Msg-Length	# of VCs
CBR	2000	2000	2000	50ms	4
RTVBR	8280	6784	7400	50ms	10
NRTVBR	7310	5088	6000	250000Kbits	12
ABR	5173	0	259	200000Kbits	20

5.4.3 Configuration parameters

Table 5.2 **Configuration Parameters**

Parameter Description	Parameter Value
Source to switch propagation time	0.01 milliseconds
Switch to switch propagation time (LAN)	0.1 millisecond
Switch to switch propagation time (WAN)	5 millisecond
Arrival process	Poisson
Switch transmission link speed	155 Mbps
Decrement rate [PRCA]	1/16 * current rate
Increment rate [PRCA]	1/32 * current rate
Averaging interval [ERICA]	300 microseconds

5.4.4 Simulation Results and Analysis

The simulated algorithms are compared by evaluating the total time taken to complete message transmission, the buffer size required at any

instance and bandwidth usage. In this section the results for the WAN and LAN environment in congestion configuration are illustrated.

5.4.4.1 Message Transfer Time

5.4.4.1.1 Description

Message transfer time is the period a VC is actively transferring data. A VC is active, from the time it is admitted into the network until the source completes transmitting all messages and releases the bandwidth to the network. VCs are admitted into the network at poisson arrival rate. This in turn, determines the start time for each VC. The total time taken by each VC depends on its transmission rate and congestion at each link in its path to the destination. Hence, the total time taken differs according to the congestion control mechanism used. The total time taken by each VC under simulation of different algorithms are shown in fig. 5.2 and 5.3.

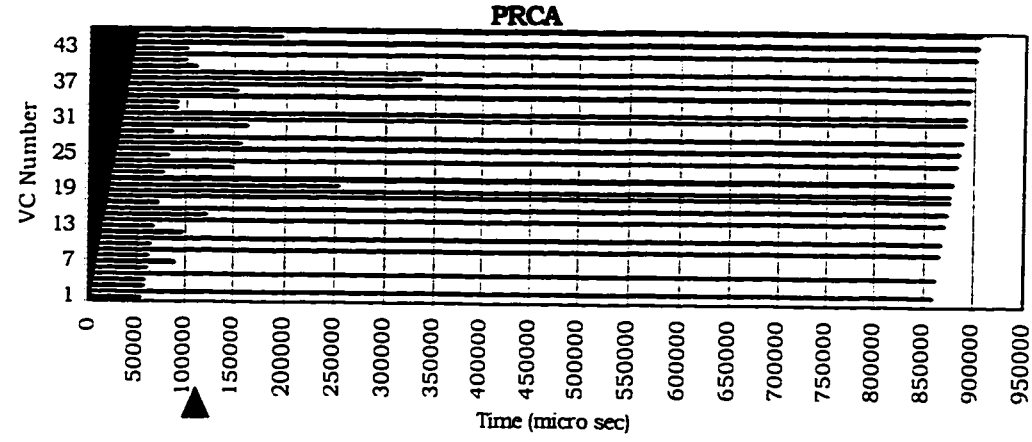
5.4.4.1.2 Results

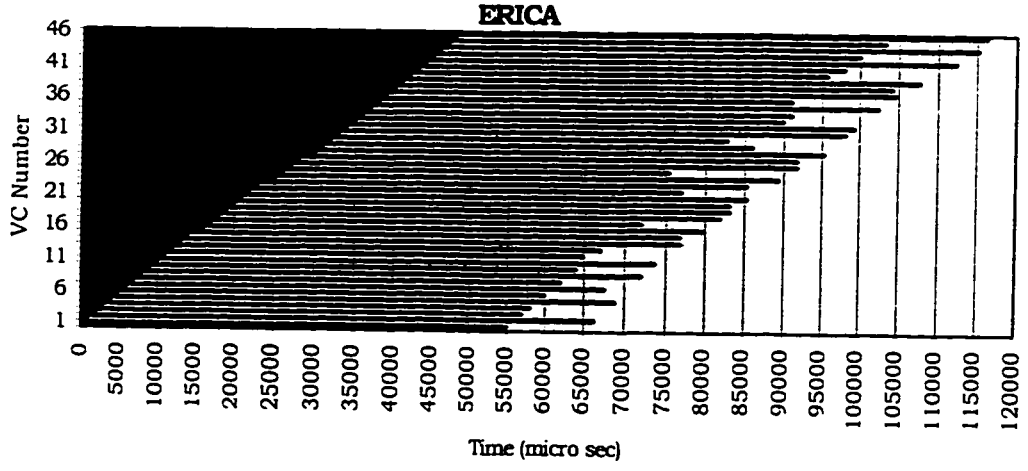
The total time take for message transfer by each ABR source and the average time taken by ABR sources to completely transmit the message in both LAN and WAN environments, under PRCA, ERICA and EARA algorithms are shown in the table next.

Table 5.3: **Total Time for complete Message Transfer by ABR Sources**

	WAN			**LAN**		
VC Number	**PRCA**	**ERICA**	**EARA**	**PRCA**	**ERICA**	**EARA**
1	857661	65310	68688	852761	60410	63788
4	857661	64911	69207	852761	60011	64307
8	857661	64072	69253	852761	59172	64353
10	857661	63685	70392	852761	58785	65492
13	857661	63841	69352	852761	58941	64452
15	857661	63868	69346	852761	58968	64446
17	857661	63901	69261	852761	59001	64361
18	857661	64125	68962	852761	59225	64062
20	857661	64317	69578	852761	59417	64678
23	857661	64425	69446	852761	59525	64546
25	857661	64783	69527	852761	59883	64627
27	857661	65397	69321	852761	60497	64421
30	857661	65248	68563	852761	60348	63663
31	857661	65399	68703	852761	60499	63803
34	857661	65413	67906	852761	60513	63006
36	857661	65620	68441	852761	60720	63541
38	857661	66676	67443	852761	61776	62543
41	857660	68377	65631	852760	63477	60731
43	857661	69178	64089	852761	64278	59189
45	857661	68348	62294	852761	63448	57394
Average	**857661**	**65345**	**68270**	**852761**	**60445**	**63370**

Figure 5.2: **WAN - *Total Time for Message Transmission***





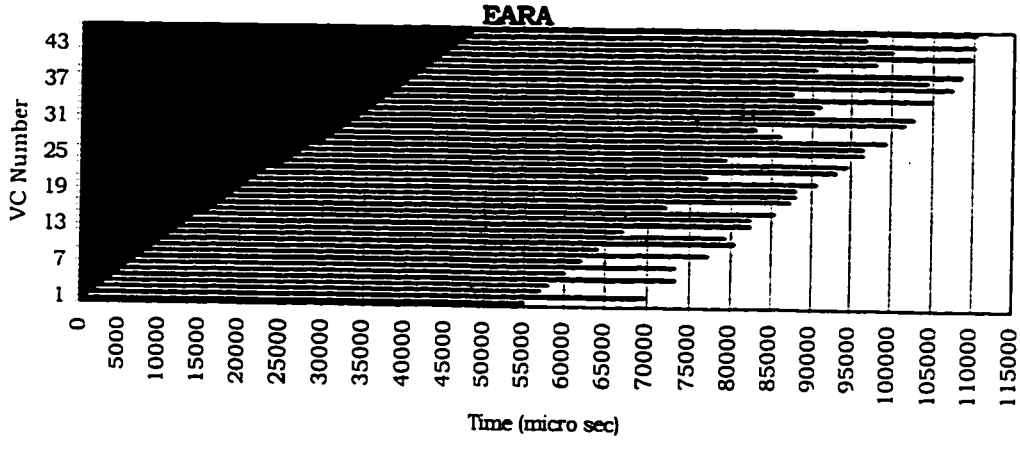
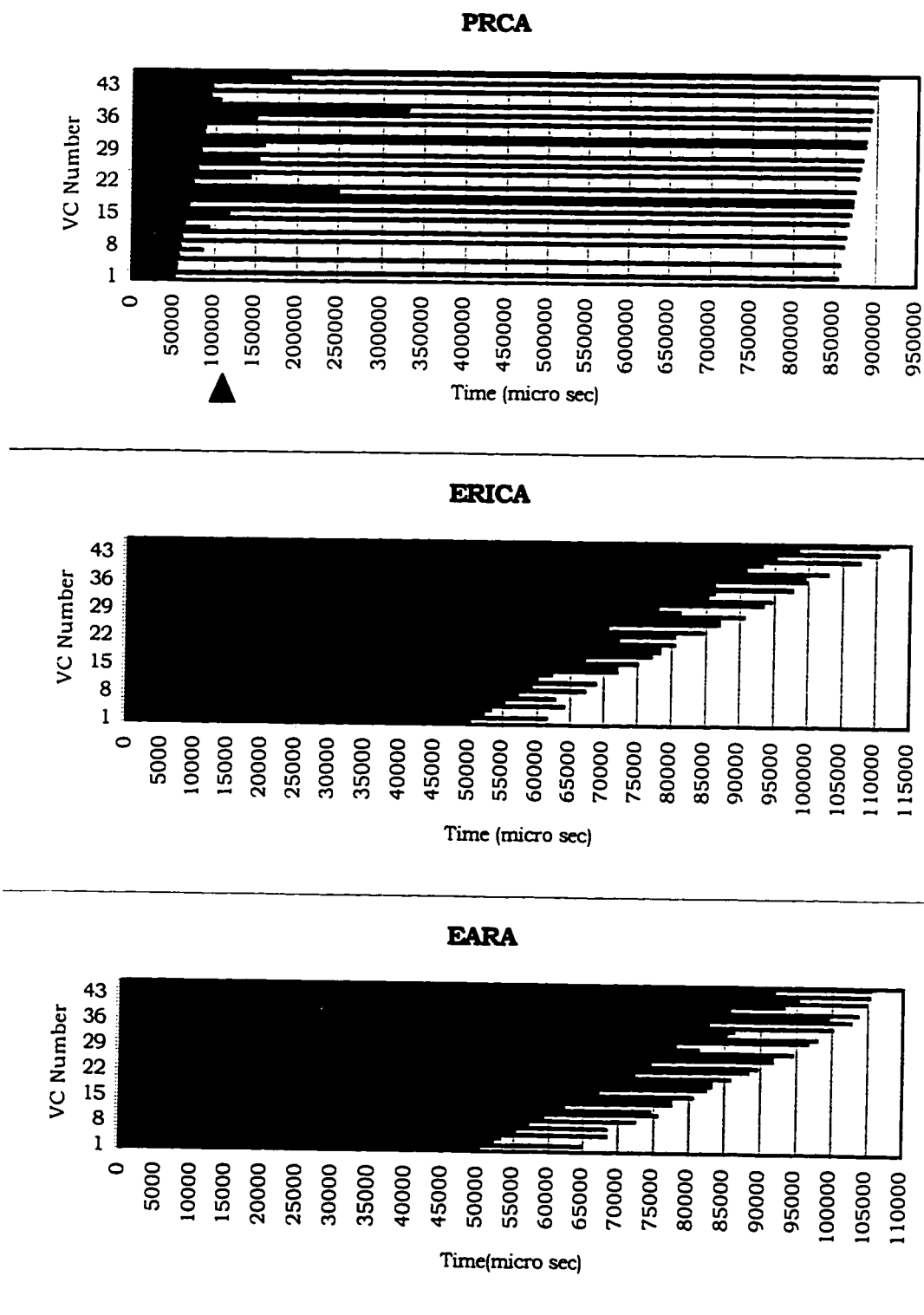


Figure 5.3: **LAN - *Total Time for Message Transmission***



5.4.4.1.3 Analysis

From the results shown in the figure 5.2 and 5.3, it can be seen that the total time taken for transferring all messages using EARA and ERICA is approximately 1/8th the times required using PRCA. This is because PRCA starts off at a low initial rate and keeps increasing its rate very slowly. It has no information about the network load and hence cannot make use of all the available bandwidth. Thus, this algorithm takes the longest time to complete transmission.

In the table above, we see that the time to complete message transmission by ABR sources with EARA increases and then decreases, according to their start time. This is because, the switch allocates a lower transmission rate when its input load is higher than the output load. This helps to keep the queue size and cell loss at a minimum. As the congestion diminishes, EARA allows ABR sources to transmit at higher rates. Hence, ABR sources complete transmitting faster when there is no congestion.

ABR sources seem to complete transmitting with ERICA during congestion as in the simulation the buffer size is infinite. If the buffer size is limited, then there would be cell loss, which leads to retransmission of the lost packets. Thus, increasing the time to complete transmission tremendously. Hence we can conclude that under practical situations, EARA transmits faster than ERICA.

5.4.4.2 Buffer Size

5.4.4.2.1 Description

The total number of cells, including both Data and RM cells, that the switch needs to handle at any given instance determines the required queue size. The purpose of figures 5.4 and 5.5 is to illustrate the total buffer size required in a switch, which depends on the algorithm used. For PRCA, the buffer size used is negligible, as this algorithm does not fully use the available bandwidth in the network. Under ERICA, switch 2 the required buffer space is insignificant. Hence, this case is not shown.

5.4.4.2.2 Results

The average number of cells in queue under each traffic type for ERICA and EARA during congestion, is shown in the following table.

Table 5.4: **Average number of cells in queue**

Algorithm	CBR	RTVBR	NRTVBR	ABR	RM	TOTAL
ERICA [Swt 1]	0.03	0.04	0.43	185.5	0.31	186.3
EARA [Swt 1]	0.22	0.25	5.23	11.50	3.74	24.66
EARA [Swt 2]	0.16	0.17	3.80	9.16	4.15	21.59

The average number of cells in both LAN and WAN environments are the same, as the only difference is the increase in message transfer delay for both Data and RM cells. Figures 5.4 and 5.5 show the total number of cells, including both Data and RM cells with respect to the time. Cells accumulated by different traffic types are not shown in the figure as they overlap each other and hence not very visible.

Figure 5.4: **WAN - Total buffer space required at an instance**

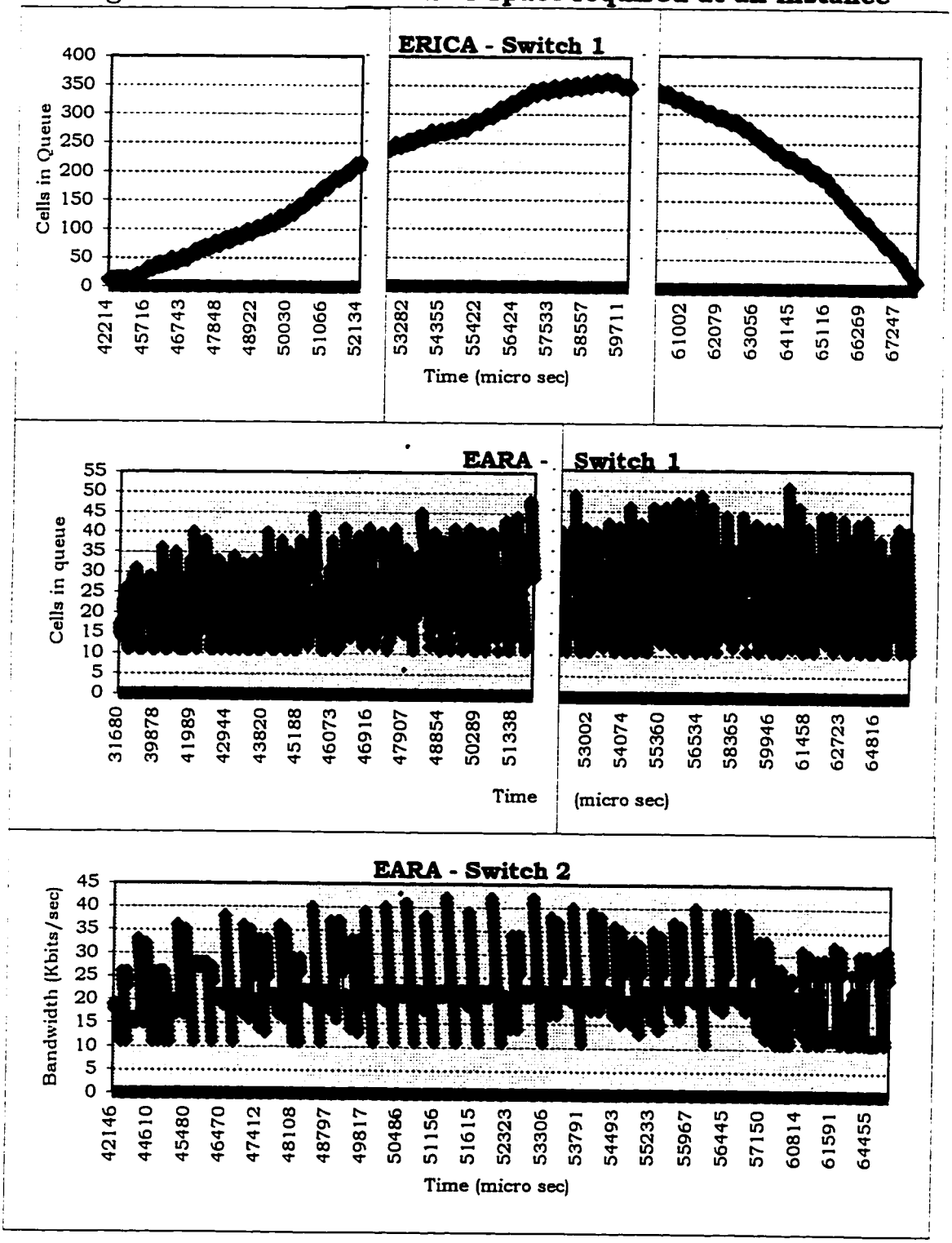
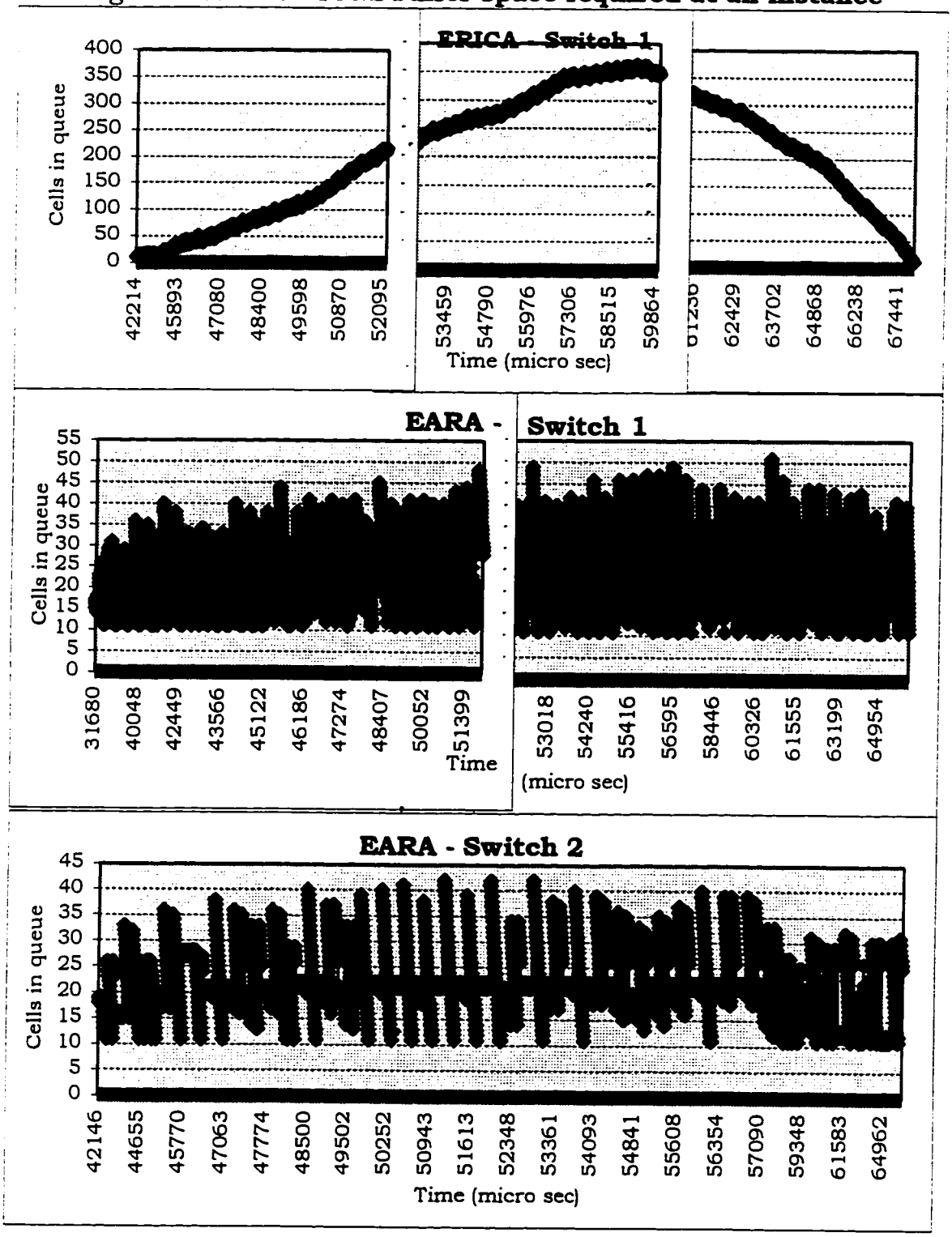


Figure 5.5: **LAN - Total buffer space required at an instance**



5.4.4.2.3 Analysis

Under ERICA, only the first switch gets congested, whereas under EARA, both switches seem to be congested. The data in the table and the graphs illustrate that switch 1 would need lesser number of buffers while using EARA than ERICA. Whereas, switch 2 needs more buffers with EARA than with ERICA. But it is important to notice that with EARA, the number of buffers required in a switch is only about $1/8^{th}$ of that required by ERICA. Also, the number of buffers required on both switches is almost the same. In other words, the total number of cells accumulated at any instance is much lower in EARA than ERICA. Hence, it can be concluded that with unlike ERICA, EARA the buffer requirements are less and almost same on all switches in the path.

With EARA, the switch 2 seems congested because, when switch 2 realizes that the VBR traffic sources are increasing their rate of transmission, it calculates the allowed rate for ABR sources and transmits RM cells in the backward direction. When this switch completes processing all the cells it has in its queue, it recalculates the allowed rates and transmits RM cells in the forward direction. Hence, the number of RM cells processed by this switch is much higher. When these RM cells reach the switch 1, it just drops it as the allowed rate for these ABR sources on switch 1 is lower. This scenario is seen because, the switch sends RM cells

only in the backward direction when the input load exceeds the output load. Though this might seem to impose some overhead on switch 2, it is better to utilize the output link capacity and to make sure that the congestion on switch 1 is controlled on time. Also, it is important to understand that switch 2 keeps track of the input rate of the ABR sources, i.e., the allowed rate is calculated on the current rate used by the source. Hence, if a switch is congested on one path, the bandwidth is allocated to other VCs that do not have congestion.

The buffer size used for this simulation is infinite. Hence, there is no cell loss. But, in practical conditions, the buffer size is limited. In other words, there will be buffer overflow and hence, cell loss. The buffer size required by ERICA is much larger than EARA on switch 1. If the buffer size was limited, then ABR sources would have to retransmit the lost packets, which would both increase the congestion on the switch as well as the total time required for complete message transmission. In summary, it is seen that a switch using EARA needs a much lesser buffer space as compared to a switch using ERICA.

5.4.4.3 Bandwidth Usage

5.4.4.3.1 Description

A switch is said to over utilize the bandwidth when the bandwidth used by the VCs is exceeding the physical link's bandwidth. Under this

situation, the cells start accumulating, eventually causing congestion and cell loss. Hence, it is required to avoid this scenario. When a switch under utilizing its bandwidth, the bandwidth is wasted and hence not acceptable. In an ideal network, the total physical bandwidth must be fully utilized at all instances. In other words, there should be no source requesting bandwidth to transmit messages, when the link's bandwidth is under utilized.

In this section, the simulated algorithms are compared and analyzed with respect to the bandwidth utilization. Bandwidth utilization on each switch is shown by the total physical bandwidth of the link, the available bandwidth and the used bandwidth. Available bandwidth is the bandwidth that is left over for admitting more VCs. Used bandwidth shown in the figures are the bandwidth used at an instance. In other words, it is the rate used by CBR and VBR VCs and the the allowed rate of the ABR VCs. The allowed rate for ABR VCs depend on the variable bit rate of VBR users and the remaining number of cells in the queue.

5.4.4.3.2 Results

Figures 5.6, 5.7, 5.8 and 5.9 show the total bandwidth, available bandwidth and the used [CBR and VBR]/allocated[ABR] bandwidth on the switch under both LAN and WAN environments. The bandwidth used by the extra cells in the queue is not shown in these figures.

Figure 5.6: **WAN - Bandwidth Usage** - *(Switch 1)*

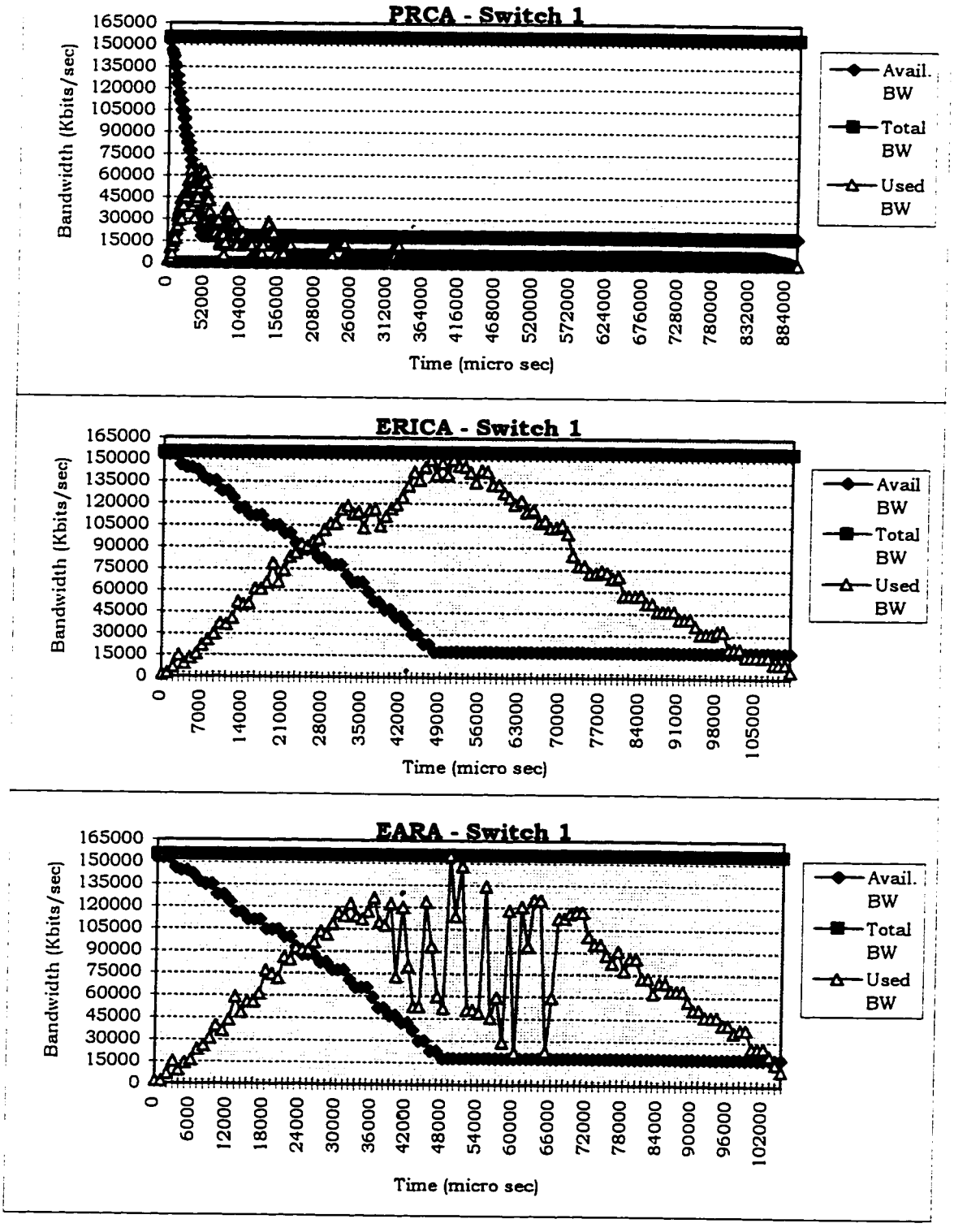


Figure 5.7: **WAN - Bandwidth Usage** - *(Switch 2)*

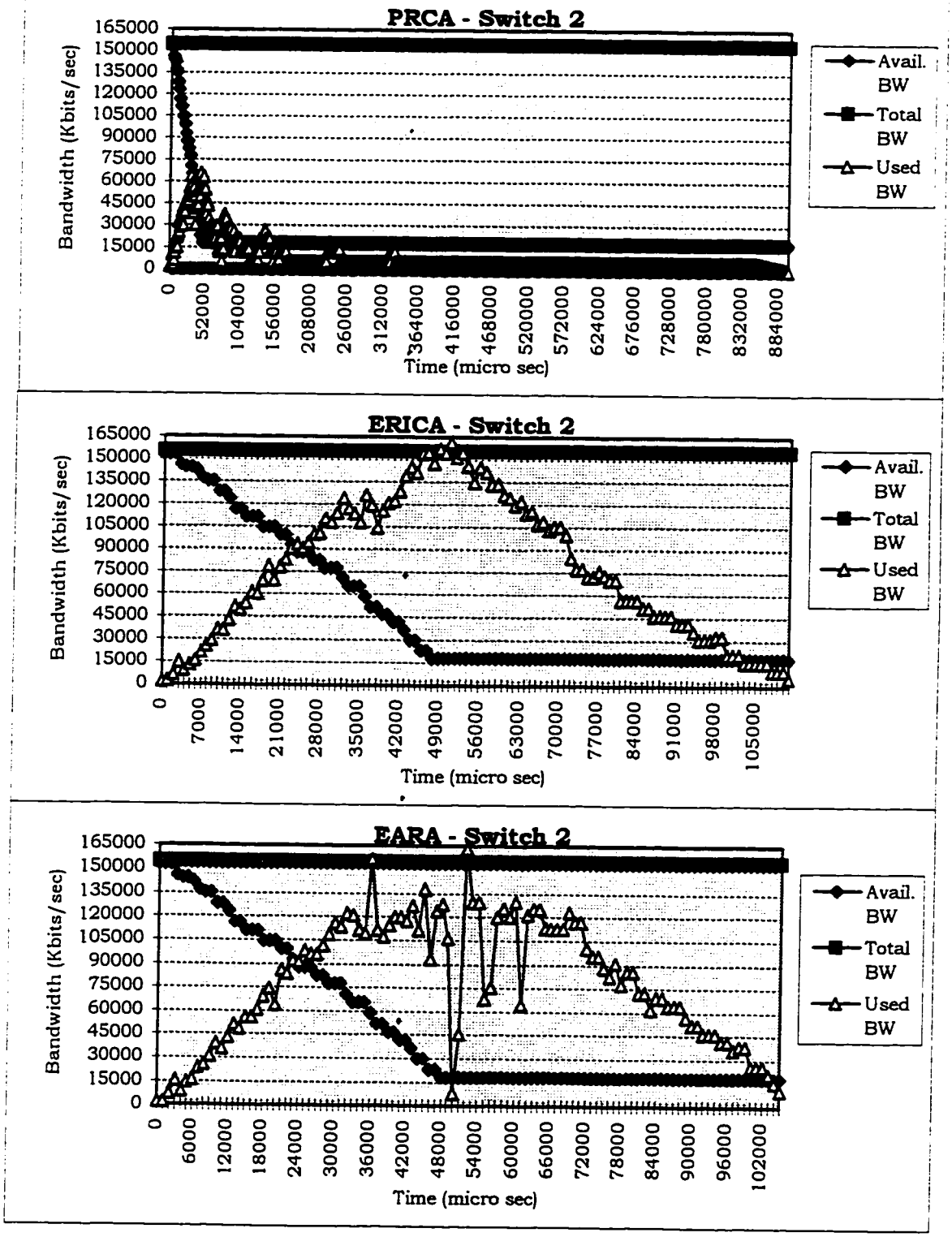


Figure 5.8: **LAN - Bandwidth Usage** - *(Switch 1)*

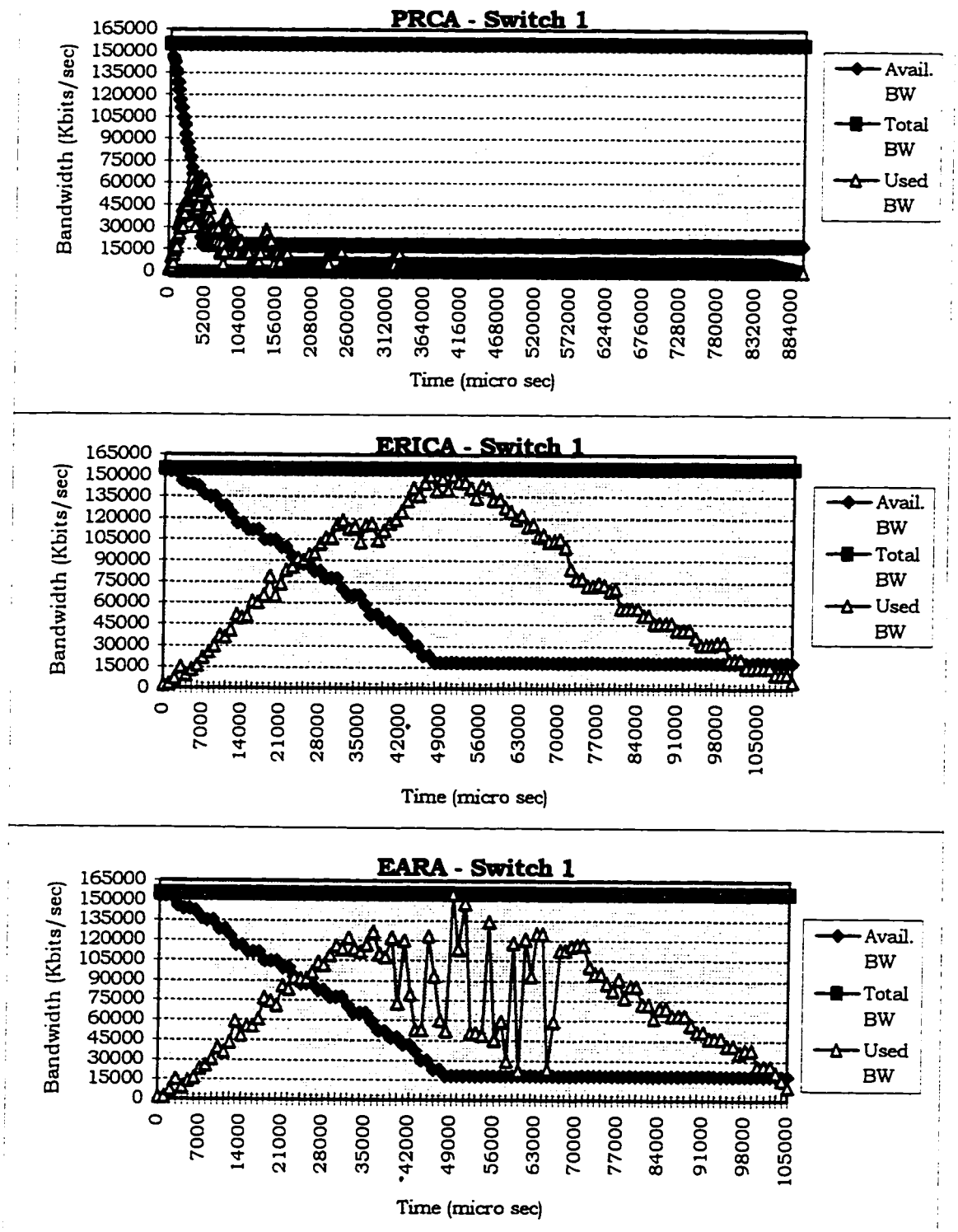
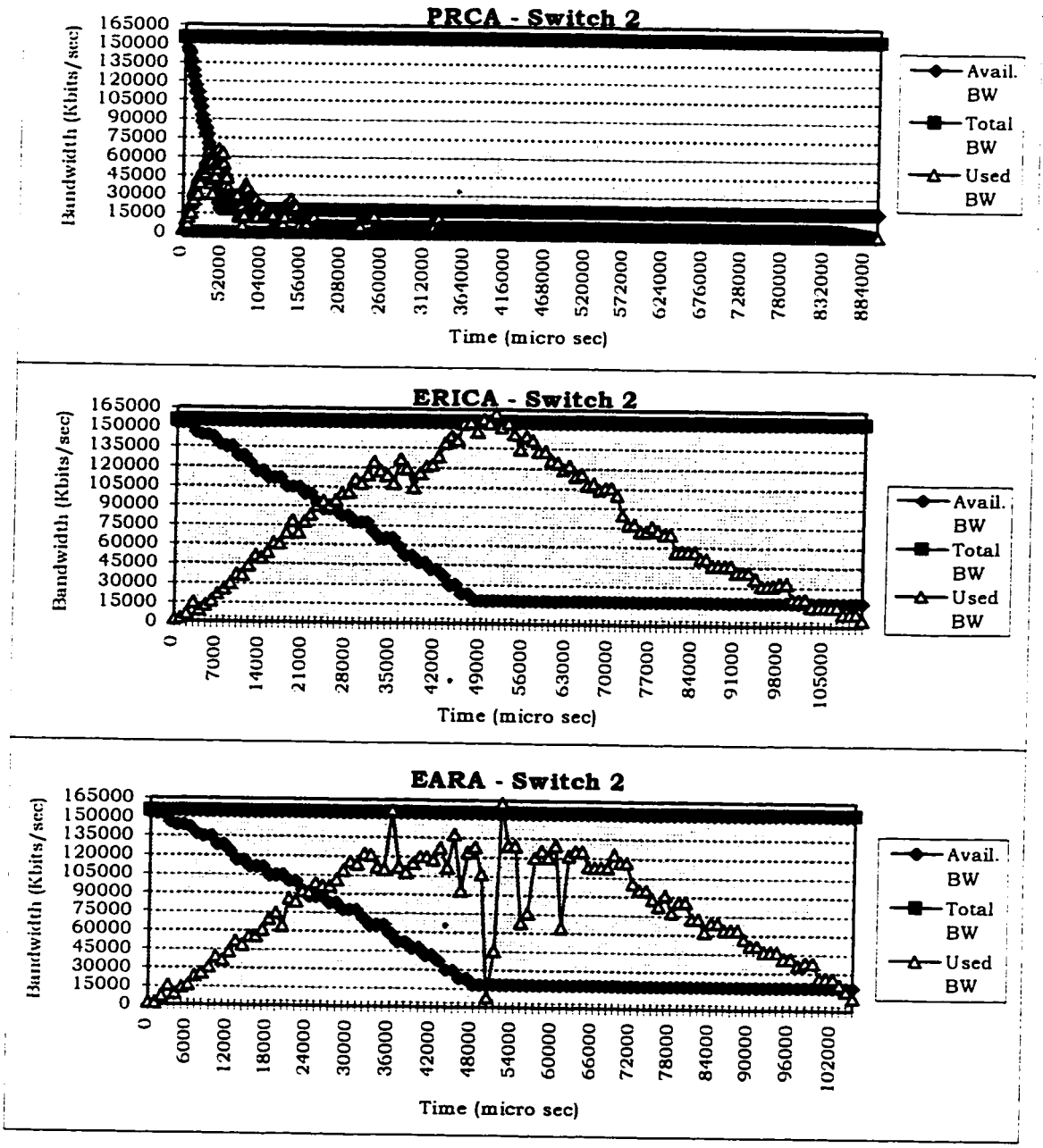


Figure 5.9: **LAN - Bandwidth Usage** - *(Switch 2)*



5.4.4.3.3 Analysis

From figures 5.6, 5.7, 5.8 and 5.9, it can be seen that PRCA does not utilize its bandwidth efficiently. Unlike PRCA, both ERICA and EARA try to utilize their bandwidth to the full extent at all times. This

try to utilize their bandwidth to the full extent at all times. This maximum utilization of bandwidth speeds up the ABR message transfer, though it creates some overhead in the switch of calculating the fair explicit rate [ER].

ERICA seems to allow its sources to transmit at high rates at all instances. It is seen from the results that the dip in the transmission rates during congestion is very small. This is because, ERICA does not consider the number of cells remaining in the queue, when calculates the fair share of the ABR sources. In other words, it over utilizes the bandwidth, though the target bandwidth utilization is 90% of the physical bandwidth. This can eventually lead to buffer overflow and hence, cell loss. In an ATM network, when cell loss is detected, the source must retransmit the whole packet. This will increase congestion on a switch and message transfer delay. Hence it will increase the total time required for complete message transfer.

EARA utilizes its bandwidth more efficiently than ERICA even though, in EARA the switch has to send RM cells to all ABR users. RM cells seem to increase the queue size when the input rates are exceeding the output rates. But, as the switch detects this in a very early stage, it avoids congestion by sending RM cells. Though RM cells utilize some of the

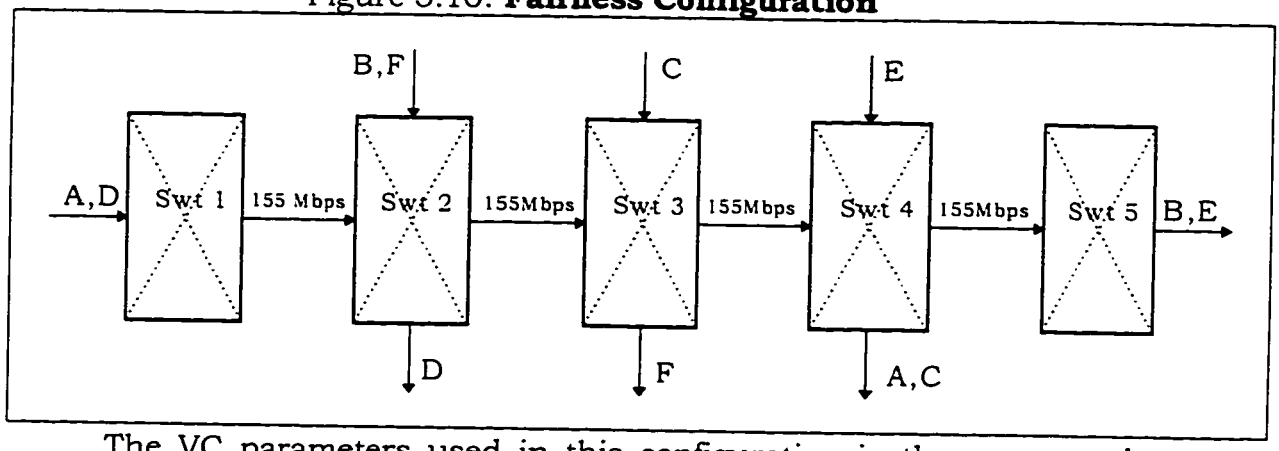
switch's bandwidth, it ultimately increases the overall bandwidth utilization rather than decreasing.

In figure 5.6, 5.7, 5.8 and 5.9, it is seen that in EARA the bandwidth drop down after it exceeds the total physical bandwidth. This is because, when calculating the allowed explicit rate for ABR users, the bandwidth required for transmitting the cells already in the queue is taken into account. The bandwidth used by the remaining cells in the queue [not shown in the figures] decreases the bandwidth allocated for ABR sources. This helps the switch to transmit the cells already in the queue, thus keeping the queue size is at a minimum at all instances. As congestion diminishes, the allowed bandwidth for ABR sources increases, thus allowing the sources to transmit at higher rates. Hence, it can be concluded that EARA efficiently utilize the bandwidth at all instances with early congestion detection and hence its avoidance.

5.5 Fairness Configuration

There are five switches in this configuration and the number of VC passing through each switch is different. An algorithm is said to be fair if it treats all sources equally at all instances. This configuration is used to evaluate the fairness of an algorithm[23].

Figure 5.10: **Fairness Configuration**



The VC parameters used in this configuration is the same as the congestion configuration, but VCs are grouped into different categories depending on the path it uses. Group A and B pass through four switches and they have the same number of VC passing through them. Similarly, all other groups pass through two switches and have equal load.

Given below is the list of VC in each group and the number of switches it passes through before reaching its destination.

Table 5.5: **VC Configuration**

Group	**# of CBR VCs**	**# of RTVBR VCs**	**# of NRTVBR VCs**	**# of ABR VCs**	**# of Switches**	**Switch Numbers**
A	0	2	2	4	4	1,2,3,4
B	0	2	2	4	4	2,3,4,5
C	1	2	2	3	2	3,4
D	1	2	2	3	2	1,2
E	1	2	2	3	2	4,5
F	1	2	2	3	2	2,3

5.5.1 Simulation Results and Analysis

Fairness of the simulated algorithms in both LAN and WAN environments, are evaluated by looking at the average time taken for message transfer in each group as well as the bandwidth used in each switch. In this configuration, only ERICA and EARA's results are shown and analyzed as PRCA's performance does not play a significant roll.

5.5.1.1 Total Time

Group A and Group B pass through the same number of switches and have the same number and types of VCs. Hence the average time taken for message transfer by ABR VCs must be almost the same under both groups.

Given below is the total time taken by each VC in group A and B.

Table 5.6: **Comparison of Groups A & B by total time of msg. transfer**

	WAN				LAN			
	ERICA		EARA		ERICA		EARA	
	A	B	A	B	A	B	A	B
VC1	93612	93605	73257	73264	59302	59336	44600	44598
VC2	94390	94590	73281	73277	59350	59647	44633	44717
VC3	98904	97810	73271	73279	59691	59652	44618	44624
VC4	106049	104995	73304	73327	59907	59930	44668	44671
Avg.	98238.75	97750	73278.25	73286.75	59562.5	59641.25	44629.75	44652.5
Diff.	**488**		**8.5**		**78.75**		**22.75**	

Groups C, D, E and F have the same inputs and configuration. Hence the time taken by ABR VCs of each group must be almost the same.

Table 5.7: **Comparison of Groups C,D,E&F by total time of msg. trnfr.**

WAN								
	ERICA				EARA			
	C	D	E	F	C	D	E	F
VC 1	63618	63739	63846	63865	53816	53788	53807	53793
VC 2	64398	64318	64381	64455	53858	53842	53841	53827
VC 3	64496	64487	64380	64318	53860	53880	53874	53873
Avg.	64170.67	64161.33	64202.33	64212.67	53844.67	53836.67	53840.67	53831
Diff.	51.34				13.67			

Table 5.8: **Comparison of Groups C,D,E&F by total time of msg. trnfr.**

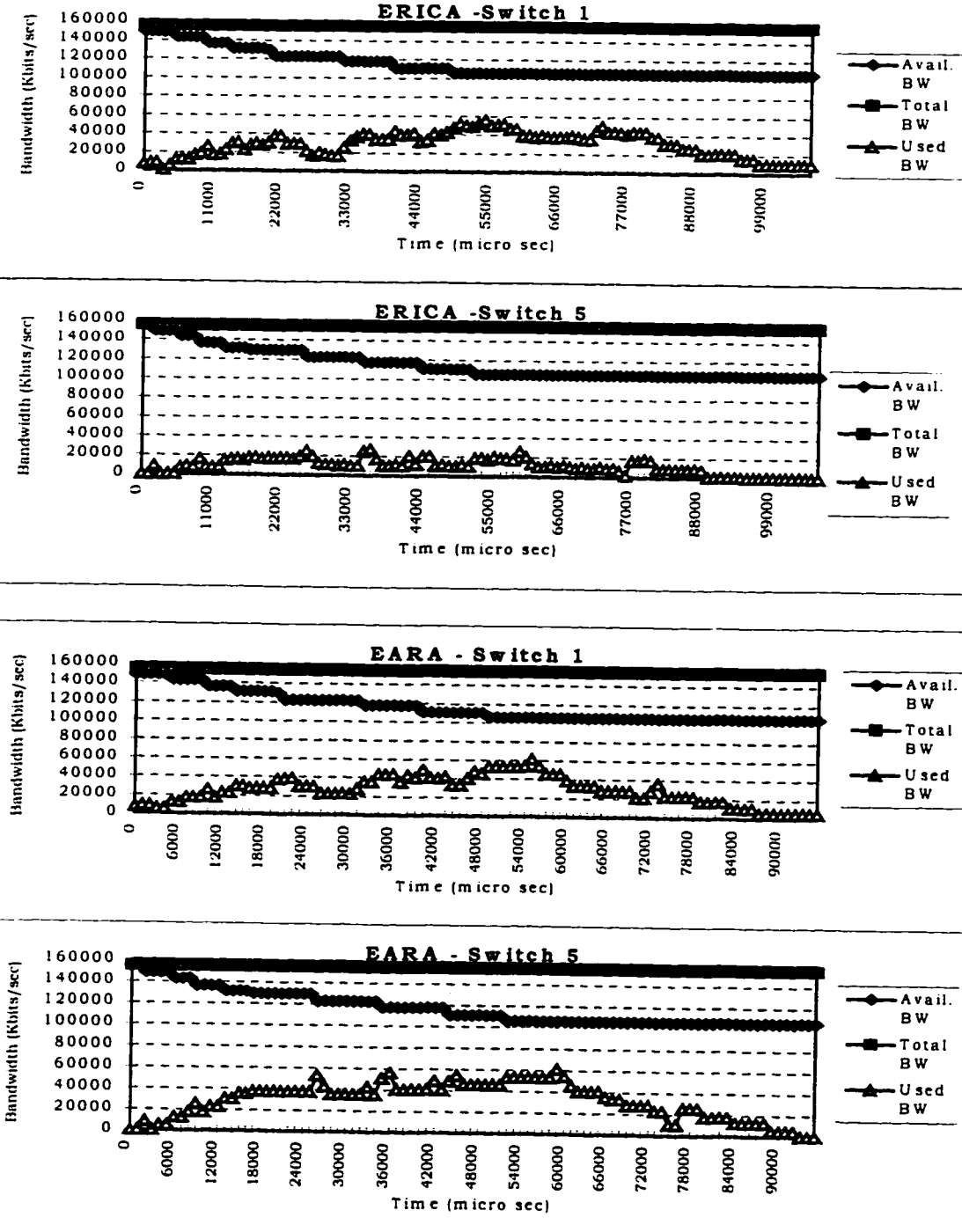
LAN								
	ERICA				EARA			
	C	D	E	F	C	D	E	F
VC 1	59162	59098	59199	59176	44264	44235	44256	44244
VC 2	59629	59391	59395	59542	44295	44317	44288	44275
VC 3	59818	59800	59872	59831	44292	44326	44324	44290
Avg.	59536.33	59429.67	59488.67	59516.33	44283.67	44292.67	44289.33	44269.67
Diff.	106.66				23			

From the above tables, it can be seen that the difference in the average time taken by similar groups using EARA is lesser than that of ERICA. Hence, it can be concluded that EARA is fairer than ERICA.

5.5.1.2 Bandwidth Usage

The number of groups (VCs) passing through switch 1 and switch 5 are same. Similarly, the number of VCs passing through switch 2, 3 & 4 are same. Hence the bandwidth used in each of these switch groups must be almost the same. Given below are the results of simulation.

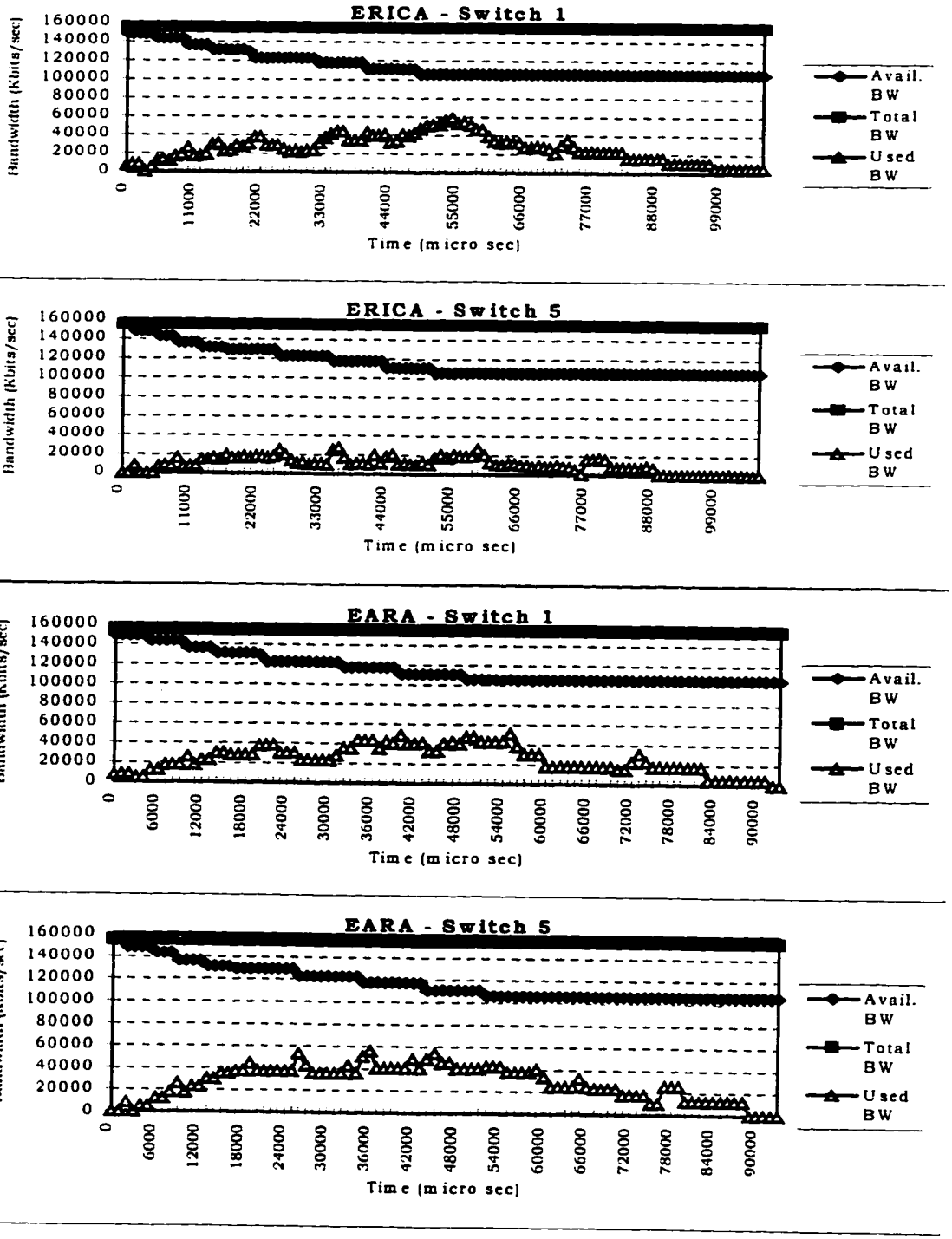
Figure 5.11: **Bandwidth Usage of Switch 1 & 5- WAN**



From the figures, it can be seen that the bandwidth used in switches 1 and 5 are almost equal using EARA as compared to ERICA in

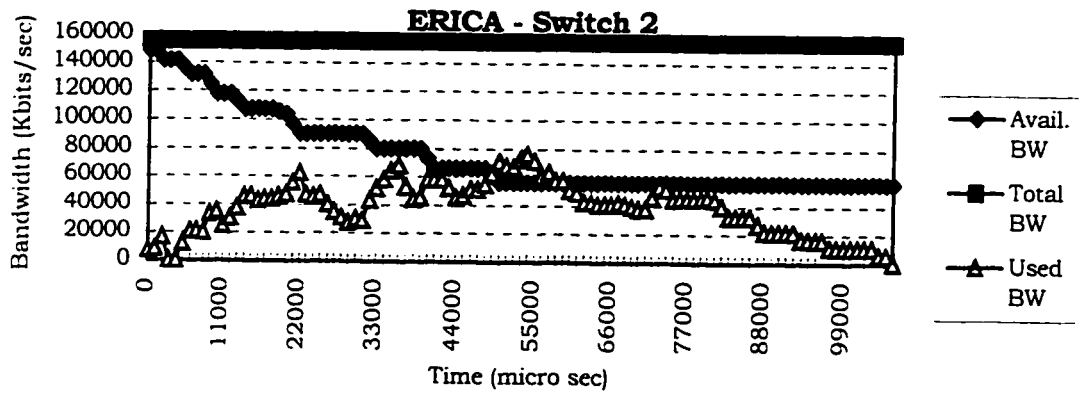
the WAN environment.

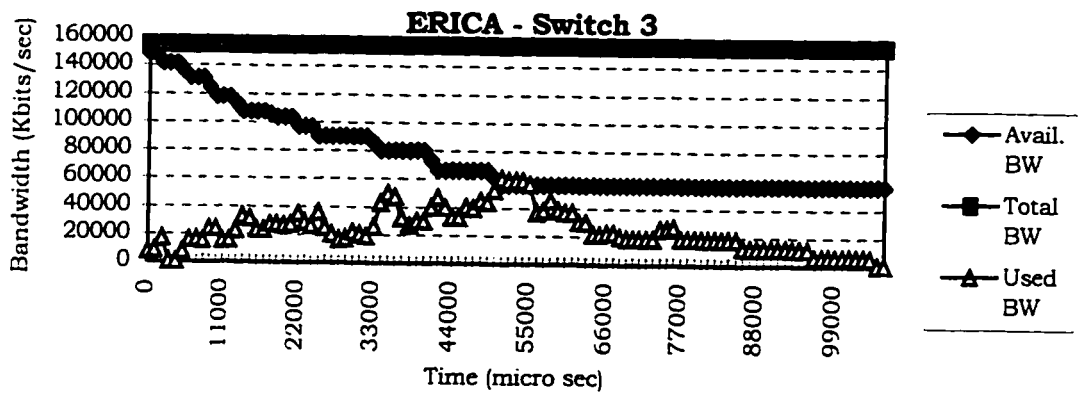
Figure 5.12: **Bandwidth Usage of Switch 1 & 5 - LAN**



The results in the LAN environment is similar to that of the WAN environment.

Figure 5.13a: **ERICA - Bandwidth Usage of Switch 2,3 & 4 - WAN**





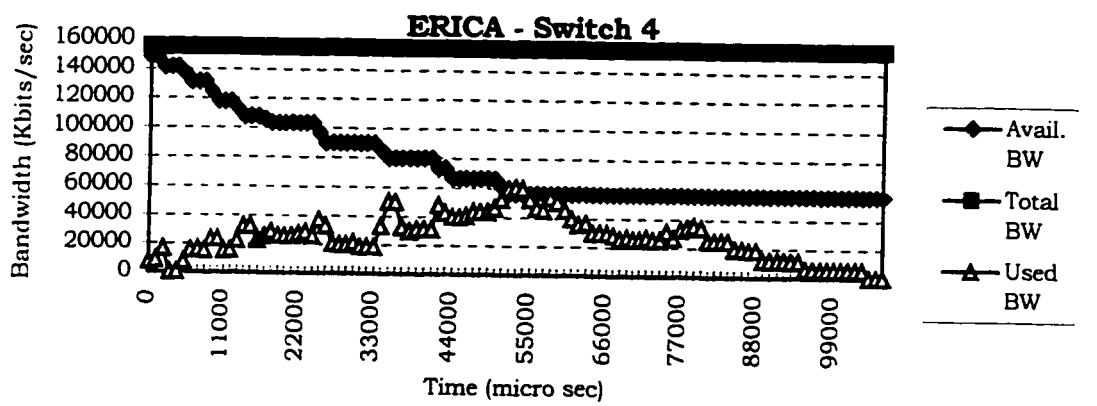
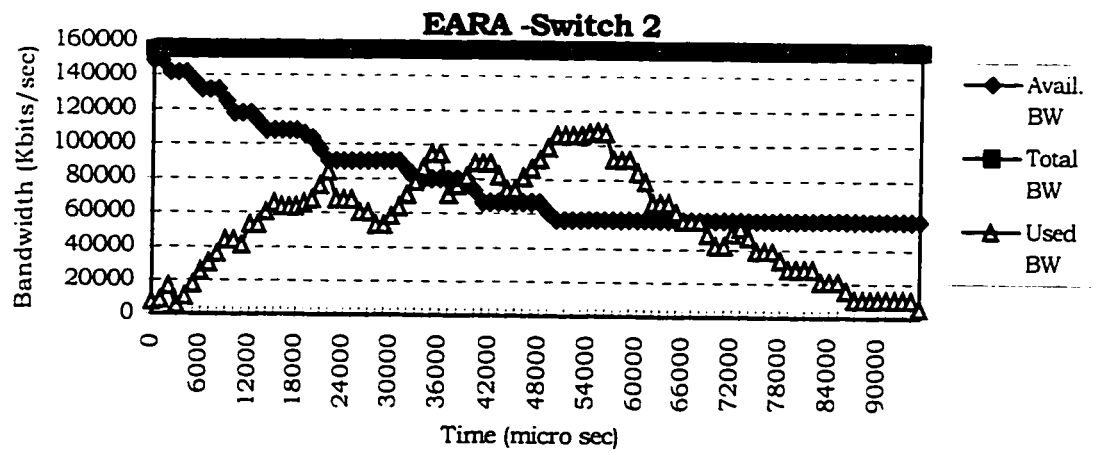
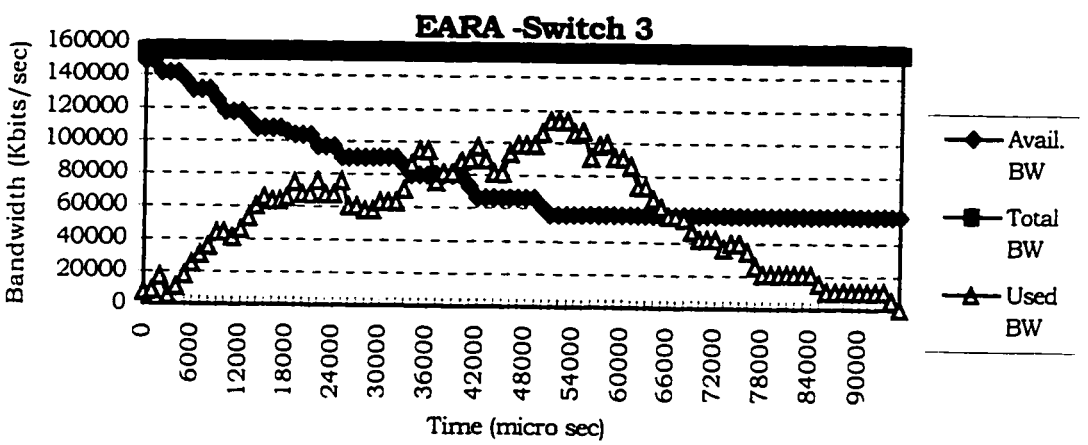


Figure 5.13b: **EARA - Bandwidth Usage of Switch 2,3 & 4 - WAN**





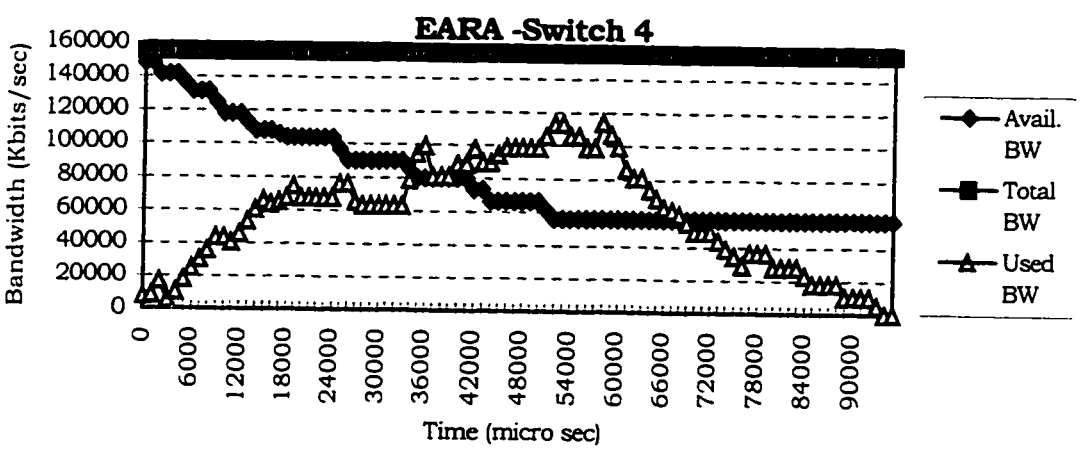


Figure 5.14a: **ERICA - Bandwidth Usage of Switch 2,3 & 4 - LAN**

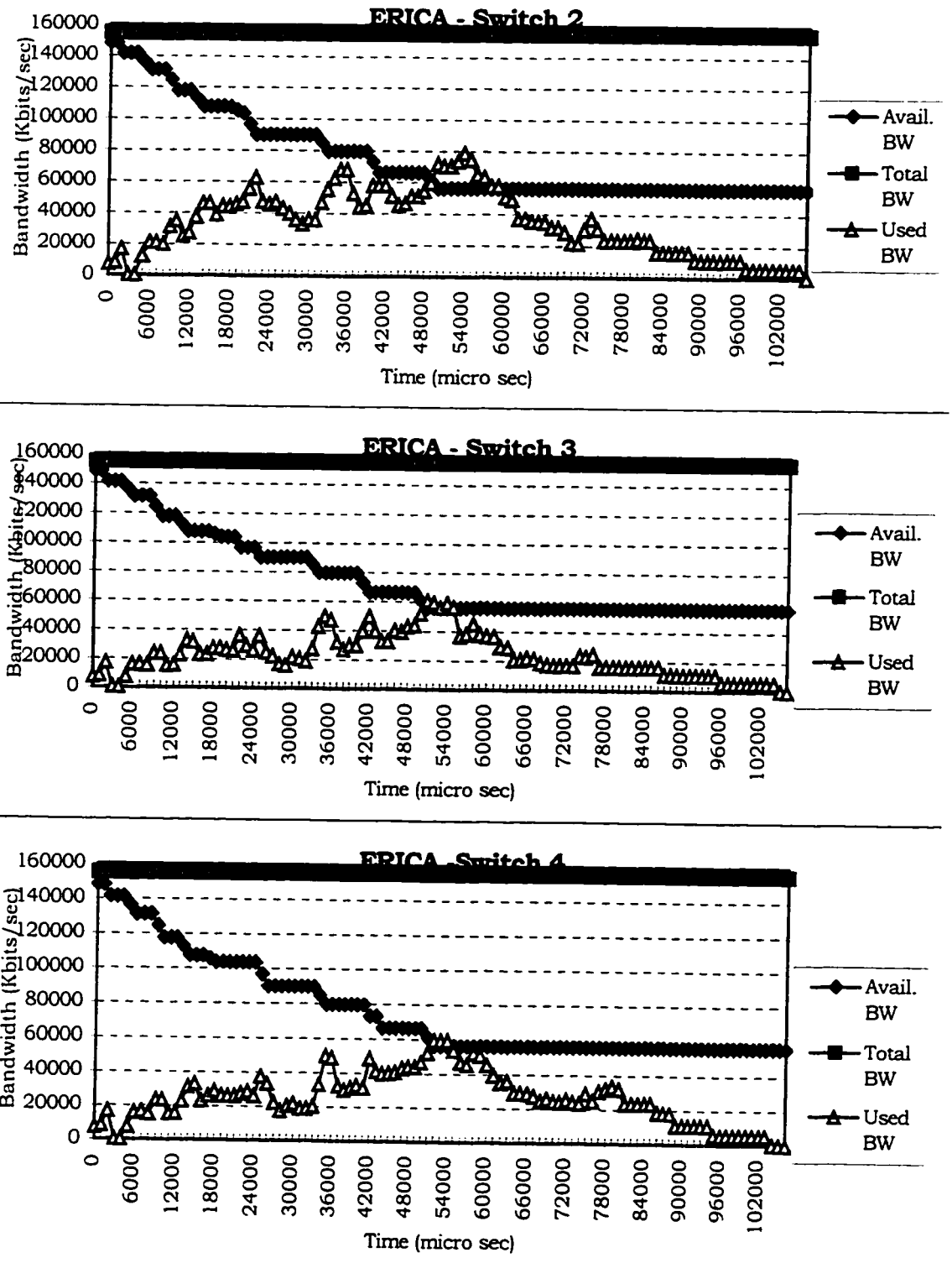
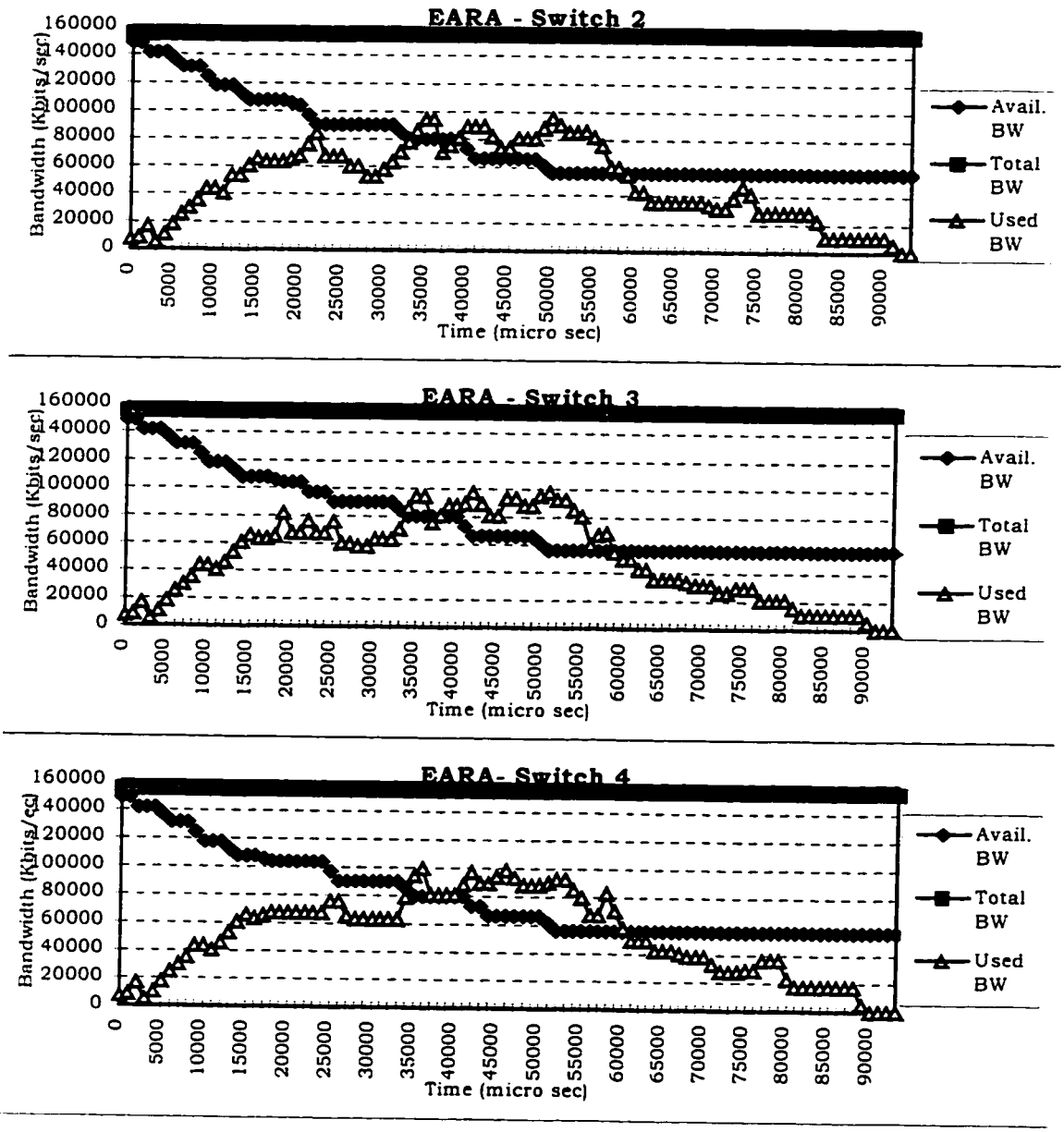


Figure 5.14b: **EARA - Bandwidth Usage of Switch 2,3 & 4 - LAN**



The simulation results shown next in both LAN and WAN environments , clearly illustrates that, unlike ERICA, the bandwidth used in switches 2, 3 and 4 are almost the same in EARA. Thus, it can be clearly seen that EARA is fairer than ERICA.

6. CONCLUSION

In this thesis, two new protocols have been introduced, namely, Modified Fast Reservation Protocol [MFRP] and Explicit Allowed Rate Algorithm [EARA]. MFRP is a burst level admission control mechanism and EARA is an algorithm for controlling congestion in ATM networks.

MFRP and MFRP/IT are improved burst level admission control protocols for ATM networks. It uses a simple mechanism of choosing the Request Rate in between the Peak Rate and the Mean Rate. This allows more sources to be admitted into the network, thus increasing the network throughput. This also leads to fairer acceptance of sources with different Peak Rates. Along with the Request Rate, MFRP also sends the minimum acceptable rate to the network at call request. This allows the network to allocate the maximum possible rate in between the Minimum Rate and the Request Rate. Thereby, significantly reducing the blocking probability and the overall delay.

EARA detects congestion at a very early stage, hence making it possible to avoid congestion. From the results, it is seen that BRM cells need to be transmitted when there are data cells waiting for transmission. As the number of ABR users increase, there is a potential that the cell

delay for real time traffic becomes unacceptable. Therefore, it is suggested that BRM cells are processed after transmitting real time data cells. This will ensure that the real time traffic will have the least cell delay and also the input-output load will be balanced soon. In summary, it is seen that EARA has the following characteristics:

- Detects congestion at a very early stage and hence avoids it.
- Switch needs minimal buffer space.
- Bandwidth is fully utilized at all instances.
- ABR traffic sources can transmit at the highest possible rate that the link to the destination can support. In other words, as expected, it takes a longer time when congestion is detected in the network. Otherwise, it completes transmission in minimum time.

Further study open on this subject are,

- Simulation of EARA with non-persistent ABR traffic and its comparison with other algorithms.
- Add a factor δ in EARA algorithm, to decrease RM cell generation, by allowing the number of remaining cells to be within a range. In other words, while calculating the number of ABR cells that can be handled by the switch without congestion will be given by,

  $$new\ ABR\_cells = total\_cells - [reserved\_cells + \delta * remaining\_cells],$$

where $\delta \leq 1$.

- To decrease RM cell traffic, for each ABR VC, the switch can keep the time-stamp of the last RM cell processed, instead of the source.

REFERENCES

1) A.Iwata, N. Mori, C. Ikeda, H.Suzuki and M.Ott, "ATM connection and traffic management schemes for multimedia inter-networking", Communications of the ACM, Vol.38, No.2, pp.73-89, February, 1995.
2) Aleksandar Kolarov and G. Ramamurthy, "End-to-end Adaptive Rate Based Congestion Control Scheme for ABR Service in Wide Area ATM Networks".
3) Anthony S. Acampora, "An Introduction to broadband networks", Plenum Press, New York.
4) C.Ikeda and H. Suzuki, "Adaptive congestion control schemes for ATM LANs", Proceedings of Infocom `94, Toronto, pp. 820-838, June 1994.
5) CCITT, CCITT Recommendations, I series (B-ISDN), July 1992.
6) Chikara Ohta, Hideki Tode, Miki Yamamoto, Hiromi Okada and Yoshikazu Tezuka, "Peak rate regulation scheme for ATM networks and its performace", Infocom `93.
7) David E. McDysan and Darren L. Spohn, "ATM theory and application, McGraw-Hill, Inc.
8) Flavio Bonomi and Kerry W, Fendick, "The rate-based control framework for the available bit rate ATM service", IEEE Network, March/April 1995.
9) G.M.Bernstein and D.H. Nguyen, "Blocking reduction in fast reservation protocols", Proceedings of Infocom `94, Toronto, pp.1208-1215, June 1994.
10) H.Suzuki and F.A. Tobagi, "Fast bandwidth reservation scheme with multi-link and multi-path routing in ATM networks", Proceedings of Infocom `92, Florence, Italy, pp. 2233-2240, May 1992.
11) Hiroyuki Ohsaki, Masayuki Murata, Hiroshi Suzuki, Chinatsu Ikeda and Hideo Miyahara, "Rate-based congestion control for ATM networks", Computer Communications Review.
12) Jaime Jungok Bae and Tatsuya Suda, "Survey of Traffic Control Schemes and Protocols in ATM Network, "Proceedings of IEEE, Volume 79, Number 2, February 1991, pg[170-189].
13) K.K.Ramakrishnan and Peter Newman, "Integration of rate and credit schemes for ATM flow control", IEEE Networks, March/April 1995.

14) Kai-Yeung Siu and Hong-Yi Tzeng, "Intelligent congestion control for ABR service in ATM networks", Computer Communication Review, October, 1995.
15) Kim and P.Wang, "ATM networks: Goals and challenges", Communications of the ACM, Vol. 38, No. 2, pp. 39-44, Feb 1995.
16) Kompella and I. Widjaja, "Burst-level admission control protocols with multirate traffic and arbitrary network topology", Proceedings of IC3N '95, Las Vegas, Sept. 1995.
17) Madhavi Hegde and W. Melody Moh, "Effect of bursty source traffic on rate-based ABR congestion control schemes".
18) P.E.Boyer and D.P. Tranchier, "A reservation principle with applications to the ATM traffic control", Computer Networks and ISDN Systems, 24 (1992), pp.321-334.
19) Raif O.Onvural, "Asynchronous Transfer Mode Networks: Performance Issues", Artech House.
20) Raj Jain, "Congestion control and traffic management in ATM networks: Recent advances and a survey", August 3, 1995.
21) Raj Jain, ShivKalyanaraman and Ram Viswanathan, "The OSU scheme for congestion avoidance in ATM networks using explicit rate indication".
22) Robert Walthall, "Using rate based flow control to manage available bit rate traffic in asynchronous transfer mode networks", October, 1995.
23) The ATM Forum, "ATM user-network interface specification", Version 4.0, lpq Hall, 1993.
24) Turner, "Managing bandwidth in ATM networks with bursty traffic", IEEE Network Magazine, pp. 50-58, Sept. 1992.
25) W.Melody Moh, Usha Rajagopal and Asha Dinesh, "Improved burst-level admission control schemes for ATM networks", Proceedings of the Fifth International Conference on Computer Communications and Networks.
26) W.Melody Moh and Madhavi Hegde, "Evaluation of ABR congestion control protocols for ATM LAN and WAN".
27) White Paper, "ATM switch traffic management essentials", Integrated Telecom Techonlogy.
28) Y.Chang, N.Golmie and David Su, "Study of interoperability between EFCI and ER switch mechanisms for ABR traffic in an ATM network".
29) Y.Z.Cho and A. Leon Garcia, "Performance of burst-level bandwidth reservation in ATM LANs", Proceedings of Infocom '94, Toronto, pp. 812-820, June 1994.

30) Yazid, H.T. Mouftah and T.Yang, "Fast reservation protocol and statistical multiplexing: A comparative study", Proceedings of ICC '94, New Orleans, pp. 733-737, May 1994.
31) Yoon Chang, Nada Golmie and David Su, "A rate based control switch design for ABR service in an ATM network".

APPENDIX: Source Code

```
/*****************************************************************************
 File Name: List.h

 Description: Definition of data structures to store the VC information and
              the class VC
*****************************************************************************/
struct ER_Data
{
long      ER_rate;   // N_TCR - New Transmitted Cell Rate
double    wait_time; //Delay before it really reaches the source
double    ER_Chng;   //time stamp of the last time the ER was changed
struct    ER_Data *next;
};

struct vc_node
{
long      peak_rate; //peak rate at which the source can transmit
long      mean_rate; //mean rate at which the source transmits
long      init_rate; //start rate of the source transmission
long      cur_rate; //OCR-Offered Cell Rate = TCR-Transmitted Cell Rate
long      Msg_Len; //total length of the message to be transmitted
long      Cells2Trns; //equivalent number of cells to be transmitted
int       CellsSent; //number of cells transmitted
int       Interval_timer; // 300 micro sec
int       tglRM;
int       num_sws; //number od switches
int       traffic_type; //1-CBR; 2-RTVBR; 3-NRTVBR; 4-ABR
int       vc_num; //Virtual Circuit Number
int       from; //Source
int       to; //Destination
int       active; //0-not started; 1-transmistting; 2-Done
int       CI; //Congestion Indicator
int       Busy; //Used for NRTVBR
int       CellsInNxtFrm; //Used for VBR
int       CellsInCurFrm; //Used for VBR
int       CellsInLstFrm; //Used for VBR
int       swt_nums[10]; //Swithces in the path
double    StartTime; //Starting time of transmission
```

```
double   NxtCellTime; //Time at which next cell must be transmitted
double   EndTime; //Time at which transmission was complete
double   LstRMtime; //Time last RM cell was processed
double   RMwaittime; //Delay
double   rem_cells; //# of cells yet to be transmitted
double   transmit; //used by VBR
struct   ER_Data *ERdat; //next explicit rate
struct   vc_node *next_vc; //next VC
};

class vc
{
public:
    struct vc_node *vc_head; // head of the VC list
    vc() {vc_head=0;} // Constructor
    ~vc(){ } //Destructor
    int init_vclist(); //initialize the list
    int print_vclist(); //print the list
    int clean_vclist(); //free the allocated memory
};
/****************************** end of list.h ******************************/
```

```
/*****************************************************************************
 File: source.h

 Description: Definition of the class source and some of the
              constant values used in the simulation.
*****************************************************************************/
//constants used for generating RT and NRT VBR traffic
#define  RTDELAY    33
#define  NRTDELAY   16
#define  a0         2.462
#define  a1         1.2068
#define  a2         0.2257
#define  Mean       0
#define  StdDev     12.67

class source
{
  public:
    int Avg_Int;  //ERICA - interval
    source() {Avg_Int = 300;} //constructor
    ~source() { } //destructor

    //check, format and send data/rm cells at appropriate time
    int send_cells(struct vc_node *vhead,struct sw_node *shead,
                   double time,int ACR,FILE *cell_loss,FILE *abrvc);
    //finds the exponential
    double expo(double rate);
    //clean up by freeing the allocated memory
    int All_done(struct vc_node *vhead,struct sw_node *shead);

  private:
    //Generate VBR traffic
    double Gen_VBR(struct vc_node *curvc);
    double getEn();
    //generate random number
    double frand();
    //prioritize the array according to the time and explicit rate
    void SortArray(struct vc_node *vcnxt);
};
/***************************** end of source.h ******************************/
```

```
/*****************************************************************************
 File: switch.h

 Description: Definition of class switch and structures for holding the
      information on the switch, Data and RM cells.
*****************************************************************************/
struct RMcells
{
  double  timestamp; //time this cell was created
  double  prop_time; //delay in the network
  double  Ld_Adj_Fctr; //Load Adjustment Factor
  long    cur_rate; //OCR - Offered Cell Rate
  long    ER_rate; //TCR -Transmitted Cell Rate
  int     vc_num; //virtual circuit number
  int     num_sws; //number of switches it passes through
  int     swt_nums[10]; //swithes in the path
  int     CI; //congestion indication
  int     DIR; //0-to destination; 1-to source
  struct RMcells  *next_cell;
};

struct cells
{
  double  timestamp; //time this cell was created
  double  prop_time; //delay in the network
  long    num_cells; //number of cells to be transmitted
  int     vc_num; //virtual circuit number
  int     num_sws; //number of switches it passes through
  int     swt_nums[10]; //swithes in the path
  int     last_cell; //transmission complete-release bandwith
  int     CI; //congestion indication
  struct cells  *next_cell;
};

struct vc_info
{
  long   cur_num_cells; //# of cells processed in this interval
  long   used_cur_rate; //current used rate
  long   allowed_cur_rate; //allowed current rate
  long   timer; //timer to check the interval
  long   interval; //period to wait
```

```
    int     vc_num; //virtual cirtuit number
    int     type; //traffic type
    struct vc_info  *next_vclist;
};

struct sw_node
{
    double  TCR; //target cell rate
    double  LL; //load level
    double  Cellspermicsec; //cells per micro second
    double  Ready; //ready to transmit a cell on the link
    double  timer; //timer for next cell
    double  Ready_time; //time elapsed for next transmission
    long    TB; //total bandwidth
    long    UB; //used bandwidth
    long    AB; //available bandwidth
    long    FS; //fair share
    long    interval; //time to wait
    int     sw_num; //switch number
    int     TCC;
    int     qued_cells; //# of cells queued
    int     rem_cells; //# of cells remaining in the switch
    struct  vc_info *sw_vclist; //list of VCs on the switch
    struct  cells *Chead; //Data cells of CBR VCs
    struct  cells *Rhead; //Data cells of RTVBR VCs
    struct  cells *Nhead; //Data cells of NRTVBR VCs
    struct  cells *Ahead; //Data cells of ABR VCs
    struct  RMcells *BRMhead; //Backward RM cells queue
    struct  RMcells *FRMhead; //Forward RM cells queue
    struct  sw_node *next_sw;
};

class swtch
{
  public:
    int       debug,TOT_SWT,TOT_ABR;
    double  SRC_SWT, SWT_SWT,ULB,LLB;
    struct sw_node  *sw_head;

    swtch() {  //constructor - initialize all fields
      sw_head = 0;
```

```
    debug = 0;
    TOT_SWT = 5;
    TOT_ABR = 20;
    SRC_SWT=10.0;
    SWT_SWT=1000.0;
    ULB = 1.1;
    LLB = 0.9;
  }

  ~swtch() {} //destructor
  int init_swlist(); //initaliaze the switch list
  int print_swlist(); //print the switch list
  int clean_swlist(); //free the memory allocated for the switch
  //call set up
  int call_setup(struct vc_node *vhead,int ACR,double time);
  // switch all cells in the queue according to the timer and bandwidth
  int swt_all(FILE *qcount1, FILE *qcount2, FILE *qcount3,
              FILE *qcount4, FILE *qcount5, FILE *bwidth1,
              FILE *bwidth2, FILE *bwidth3, FILE *bwidth4,
              FILE *bwidth5,vc_node *vhead,double time,
              int acr);
private:
  //transmit cells to the next node
  int switch_cells(struct sw_node *swcur, double time,
                   struct vc_node *vhead, int ACR);
  //switch backward RM cells
  int switch_BRMcells(struct sw_node *swcur,struct vc_node *vhead,
                      double time,int ACR);
  //switch forward RM cells
  int switch_FRMcells(struct sw_node *swcur,struct vc_node *vhead,
                      double time,int ACR);
  //log the number of cells in each queue
  void QCount(struct sw_node *swcur,FILE *qcount1,FILE *qcount2,
              FILE *qcount3,FILE *qcount4,FILE *qcount5,
              double time);
  //unique to EARA
  int EARA(struct sw_node *cur_sw,struct vc_node *vhead,double time);
  int ERICA(struct sw_node *cur_sw); //unique to ERICA
  //check the load level
  int chk_sw_load(struct sw_node *swcur,struct vc_node *vhead,
                  double time,int ACR);
```

```
    // monitor the rate used
    int monitor_rate(struct sw_node *swcur,int ACR);
    //log the bandwdith used
    void log_BW(struct sw_node *swcur,double time,FILE *bwidth1,
                FILE *bwidth2,FILE *bwidth3,FILE *bwidth4,
                FILE *bwidth5, int ACR);
    long findER(struct sw_node *cur_sw,long rate); // find explicit rate
    int swap_cells(struct sw_node *swcur,struct cells *head,int type,
                 double time,struct vc_node *vhead,int ACR);
    int add_vc(struct vc_node *vccur,int ACR); // add an other VC
    //add backward RM cell
    int Add_BRMcell(struct RMcells *newRM, int sw_num);
    //add forward RM cell
    int Add_FRMcell(struct RMcells *newRM, int sw_num);
    //add RM cell
    int Add_RM(struct RMcells *newRM, int sw_num);
};
/****************************** end of switch.h *******************************/
```

```
/*****************************************************************************
 File: callsetup.cpp

 Description: This file contains the code for admitting a VC. According to
              the traffic type, different rules are excuted.
*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "list.h"
#include "switch.h"

/****
Description: This routine is called from the main routine. VC arrival is
             Piossion. It sets up a VC account in the switches along the
             path from source to destination. To achive this it undergoes
             two passes. If the first pass is successfull, then it means
             that the VC can be admitted. In the second pass, the required
             rates are allocated and the bandwidth allowed for ABR VCs
             are adjusted if necessary.
****/
int swtch::call_setup(vc_node *vhead,int ACR,double time)
{
   struct vc_node    *vcnxt;
   struct sw_node    *swnxt;
   int     i,ret=0;
   long   tmp_interval;

   if (debug == 1)
      printf("In callsetup\n");

   vcnxt = vhead;
   // parse the VC list to find the next VC to be added.
   while (vcnxt != 0)
   {
      if (vcnxt->active != 0)
         vcnxt = vcnxt->next_vc;
      else
         break;
   }
```

```
if (vcnxt == 0)  // all VCs are active or completed transmission
   return(0);
// set the start time of message transmission
if (vcnxt->StartTime == 0)
   vcnxt->StartTime = time;

// First Pass
int success = 1;
for (i=0;i<vcnxt->num_sws;i++) // for each switch in the path
{                              // from source to destination
   if (success == 0)  //can't support - reject
     break;

   swnxt = sw_head; //find the switch in the switch list
   while ((swnxt != 0) && (swnxt->sw_num != vcnxt->swt_nums[i]))
      swnxt = swnxt->next_sw;

   if (swnxt == 0) // did not find the specified switch
   {
      printf("ERR: Switch numbers not in sink. Module callsetup\n");
      exit(1);
   }

   switch(vcnxt->traffic_type)
   {
      case 1: //CBR - check to see if the constant/peak rate
         if (vcnxt->peak_rate > swnxt->AB) //can be supported.
            success = 0; //if not, reject
         break;
      case 2: //RTVBR
      case 3:    //NRTVBR
      case 4: //ABR
         if (vcnxt->mean_rate > swnxt->AB) // check if mean rate
            success = 0; // can be supported, else reject.
         break;// Note: must check the sum of peak rates of all
      default:  //CBR and VBR VCs before accepting.
         printf("NO SUCH TYPE! VC_NUM: %d\n",vcnxt->vc_num);
   }
}
if (success == 0) // Do not have enough bandwidth to support
   return(1);    // reject the VC
```

```
else
{               // accept the VC
   vcnxt->active = 1;       // Set VC to active
   ret = add_vc(vcnxt,ACR);// Add the VC information to each switch
   if (ret != 0)   // Could not add successfully,
      return(-1); // reject it.
}

// Second Pass
for (i=0;i<vcnxt->num_sws;i++) // for each switch in the path,
{
   tmp_interval = 0; //initialize local interval to check if the
                     //switch monitor interval must be updated

   swnxt = sw_head;  //Find each switch
   while ((swnxt != 0) && (swnxt->sw_num != vcnxt->swt_nums[i]))
      swnxt = swnxt->next_sw;

   if (swnxt == 0) // Did not find the switch - internal error
   {
      printf("ERR: Switch numbers not in sink. Module callsetup\n");
      exit(1);
   }

  // update the information in switch
  switch(vcnxt->traffic_type)
  {
     case 1: //CBR - decrement the available rate and
             // increment the used rate
        swnxt->AB = swnxt->AB - vcnxt->peak_rate;
        swnxt->UB = swnxt->UB + vcnxt->peak_rate;
        //if EARA, update the allowed bandwidth if neccessary
        if ((swnxt->UB > swnxt->TB) && (ACR == 0))
        {
           ret = EARA(swnxt,vhead,time);
           if (ret != 0) // error encountered
              return(-1);// reject
        }
        break;
     case 2: //RTVBR
     case 3:   //NRTVBR
```

```
    // decrement the available bandwidth by mean rate
    // and increment the used bandwidth by initial rate
    swnxt->AB = swnxt->AB - vcnxt->mean_rate;
    swnxt->UB = swnxt->UB + vcnxt->init_rate;
    //if EARA, update the allowed bandwidth if necessary
    if ((swnxt->UB > swnxt->TB) && (ACR == 0))
    {
       ret = EARA(swnxt,vhead,time);
       if (ret != 0)  // error encountered
          return(-1); // reject
    }
    if (ACR == 0)  //if EARA, update the monitor interval
    {                  //if needed
       tmp_interval = 424000/vcnxt->mean_rate;
       if (swnxt->interval > tmp_interval)
          swnxt->interval = tmp_interval;
    }
    break;
case 4: //ABR
    //derement the available rate.
    //check the used rate and accordingly
    //allocate the bandwidth for that VC
    swnxt->AB = swnxt->AB - vcnxt->mean_rate;
    //increment the used rate by initial rate
    if ((ACR != 0) ||
       ((swnxt->TB - swnxt->UB) < vcnxt->peak_rate))
       swnxt->UB = swnxt->UB+vcnxt->init_rate;
    else
    {
       swnxt->UB = swnxt->UB + vcnxt->peak_rate;
       vcnxt->cur_rate = vcnxt->peak_rate;
    }
    //if EARA, update the allowed bandwidth if necessary
    if ((ACR == 0) &&
       ((swnxt->TB - swnxt->UB) < vcnxt->peak_rate))
    {
       ret = EARA(swnxt,vhead,time);
       if (ret != 0)  // error encountered
          return(-1); // reject
       }
    break;
```

```
            default:
                printf("NO SUCH TYPE! VC_NUM: %d\n",vcnxt->vc_num);

        }
    }

    if (debug == 1)
    {
        swnxt = sw_head;
        while (swnxt != 0)
        {
            printf("SW_NUM: %d TB: %ld UB: %ld AB: %ld\n",
                    swnxt->sw_num,swnxt->TB,swnxt->UB,swnxt->AB);
            swnxt = swnxt->next_sw;
        }
        printf("Out of callsetup.\n");
    }
    return(0); //successfully admitted the VC
}

/****
 Description: This routine adds the given VC info to all switches in the
              path from source to destination.
****/
int swtch::add_vc(struct vc_node *vccur,int ACR)
{
    struct vc_info *svlist,*svnxt;
    int i;
    struct sw_node *swcur;

    for (i=0;i<vccur->num_sws;i++) //for each switch in the path
    {
        swcur = sw_head;   // find the switch
        while ((swcur != 0) && (swcur->sw_num != vccur->swt_nums[i]))
            swcur = swcur->next_sw;
        if (swcur == 0) // could not find the switch - internal error
        {
            printf("Error in switch numbers. VC: %d\n",vccur->vc_num);
            exit(1);  // abort the program
        }
```

```
        // allocate space for storing the newly admitted VC
        svlist = new(vc_info);
        if (svlist == 0) // out of memory
        {
           printf("Out of memory in add_vc. SW_NUM: %d VC_NUM: %d\n",
                   swcur->sw_num,vccur->vc_num);
           exit(1); // abort the program
        }
        //successfully allocated the space for this VC.
        //initialize it with the given values
        svlist->vc_num = vccur->vc_num;
        svlist->cur_num_cells = 0;
        svlist->type = vccur->traffic_type;
        svlist->inact = 0;
        svlist->timer = 0;
        if (ACR == 0)
        {
           svlist->allowed_cur_rate = vccur->peak_rate;
           svlist->interval = 424000/vccur->peak_rate;
           svlist->used_cur_rate = vccur->peak_rate;
        }
        else
        {
           svlist->allowed_cur_rate = vccur->init_rate;
           svlist->interval = 300; //300 micro sec;
           svlist->used_cur_rate = vccur->init_rate;
        }
        svlist->next_vclist = 0;
        //insert the VC info into the switches VC list
        if (swcur->sw_vclist == 0)
           swcur->sw_vclist = svlist; //first element in the list
        else
        {
           svnxt = swcur->sw_vclist; // add to the end of the list
           while (svnxt->next_vclist != 0)
              svnxt = svnxt->next_vclist;
           svnxt->next_vclist = svlist;
        }
     }
     return(0); //successfully added
}
```

```
int swtch::EARA(struct sw_node *cur_sw,struct vc_node *vhead,
                double time)
{
   int l,i,sw_num,ret;
   struct vc_info *svlist;
   struct RMcells nxtrm;
   struct sw_node *nxtsw;
   struct vc_node *vcnxt;
   struct ER_Data *tmpERdat,*newERdat;
   double   FairShare = 0;
   long   TotABR = 0; // used for fair allocation of bandwidth
   long   CurAB = 0;  // currently, available bandwidth for ABR VCs
   long   SureUB = 0; // bandwidth currently used by CBR and VBR VCs
   long    cur_rate = 0;

   if (debug == 1)
   {
      printf("IN EACR.\n");
      printf("EARA-UB: %ld AB: %ld\n",cur_sw->UB,cur_sw->AB);
   }
   svlist = cur_sw->sw_vclist;
   while (svlist != 0)
   {
      if (svlist->type == 4)
      {
         if (svlist->allowed_cur_rate == 0)
         {
            vcnxt = vhead;
            while ((vcnxt->vc_num != svlist->vc_num) && (vcnxt != 0))
               vcnxt = vcnxt->next_vc;
            if (vcnxt == 0)
            {
               printf("No such VC. EARA \n");
               return(1);
            }
            // this is done in order to allocate bandwidth fairly
            TotABR = TotABR + vcnxt->init_rate;
         }
```

```
        else
        {
                // persistent traffic - look at allowed rate as the
                // used rate cannot be different
                TotABR = TotABR + svlist->used_cur_rate;
        }
    }
    else  // bandwidth currently used by CBR and VBR VCs
       SureUB = SureUB + svlist->used_cur_rate;
    svlist = svlist->next_vclist;
}
// Total available rate for ABR VCs
CurAB = cur_sw->TB - (SureUB +
                (long)((cur_sw->rem_cells*424000)/cur_sw->interval));

if (TotABR > 0) // if any ABR VCs are present
{  //calculate the fairshare
   FairShare = (double)CurAB/(double)TotABR;

   svlist = cur_sw->sw_vclist;
   while (svlist!= 0) // scan the VC list
   {
      if (svlist->type == 4) // if ABR VC
      {
         vcnxt = vhead; // find the VC
         while ((vcnxt->vc_num != svlist->vc_num) && (vcnxt != 0))
            vcnxt = vcnxt->next_vc;

         if (vcnxt == 0) // VC not found - internal error
         {
            printf("No such VC. EARA \n");
            return(1);
         }
         //if cuurent allowed rate is zero, then allocate
         //rate according to the initial rate
         //Note: since only persistent traffic is used, the
         //used rate is never less than the allocated rate
         if (svlist->allowed_cur_rate == 0)
            cur_rate = (long)(FairShare* (double)vcnxt->init_rate);
         else // set rate according to the used rate
            cur_rate = (long)(FairShare*(double)svlist->used_cur_rate);
```

```
// make sure that the allowed rate does not exceed the
// peak rate
if (cur_rate > vcnxt->peak_rate)
   cur_rate = vcnxt->peak_rate;
if (cur_rate < 0)
   cur_rate = 0;
if (cur_rate != svlist->allowed_cur_rate) //Send RM Cell
{
   //set all the fields for an RM cell
   nxtrm.timestamp = time;
   nxtrm.vc_num = svlist->vc_num;
   nxtrm.num_sws = vcnxt->num_sws;
   nxtrm.cur_rate = svlist->allowed_cur_rate;
   nxtrm.ER_rate = cur_rate;
   nxtrm.Ld_Adj_Fctr = 0;
   nxtrm.CI = 0;
   nxtrm.prop_time = 0;
   nxtrm.swt_nums[0] = vcnxt->from;
   //set all the switch numbrs in the path
   for (i=0;i< vcnxt->num_sws;i++)
      nxtrm.swt_nums[i+1] = vcnxt->swt_nums[i];
   // set destination
   nxtrm.swt_nums[i+1] = vcnxt->to;
   //send RM cell in backward direction to the source
   if (cur_rate < svlist->allowed_cur_rate)
   {
      nxtrm.DIR = 1;
      ret = Add_BRMcell(&nxtrm,cur_sw->sw_num);
   }
   else
   {// send RM cell in the forward direction to notify all switches
      nxtrm.DIR = 0;
      ret = Add_FRMcell(&nxtrm,cur_sw->sw_num);
   }
   if (ret != 0) //error processing RM cell - internal error
      return(-1);
   //set the allowed rate
   svlist->allowed_cur_rate = cur_rate;
   //update its interval
   if (cur_rate != 0)
      svlist->interval = 424000/cur_rate;
```

```
                else
                   svlist->interval = 424000/vcnxt->init_rate;

                svlist->timer = 0;   // reset the timer
                //set the used rate
                svlist->used_cur_rate = svlist->allowed_cur_rate;
             }
          }
          svlist = svlist->next_vclist; //process next VC
       }
    }
    if (debug == 1)
       printf("EARA-UB: %ld AB: %ld\n",cur_sw->UB,cur_sw->AB);
    return (0); // successfully balanced the load
}

/****
 Description: This routine adds an RM cell to the end of backward RM list
****/
int swtch::Add_BRMcell(struct RMcells *newRM, int sw_num)
{
   struct sw_node   *swcur;
   struct RMcells   *rmcur,*rmprv;
   int i;

   //find the given switch
   swcur = sw_head;
   while ((swcur != 0) && (swcur->sw_num != sw_num))
      swcur = swcur->next_sw;

   //scan the list
   rmcur = swcur->BRMhead;
   rmprv = swcur->BRMhead;

   while ((rmcur != 0) &&
          (rmcur->vc_num != newRM->vc_num))
   {
      rmprv = rmcur;
      rmcur = rmcur->next_cell;
   }
```

```
//if a rm cell for the same vc is found
if ((rmcur != 0) && (rmcur->vc_num == newRM->vc_num))
{
   if (newRM->DIR == rmcur->DIR)
   { // going in the same direction
      if (newRM->ER_rate < rmcur->ER_rate)
      { //update the allowed rate
         rmcur->ER_rate = newRM->ER_rate;
         rmcur->CI = newRM->CI;
      }
   }
   else
   {
      if (newRM->DIR == 1)
      { //update the timestamp and the rate
         rmcur->timestamp = newRM->timestamp;
         rmcur->prop_time = newRM->prop_time;
         rmcur->cur_rate = newRM->cur_rate;
         rmcur->ER_rate = newRM->ER_rate;
         rmcur->DIR = 1;
         rmcur->CI = newRM->CI;
      }
      else
      { //update rate only
         if (newRM->ER_rate < rmcur->ER_rate)
            rmcur->ER_rate = newRM->ER_rate;
      }
   }
}
else
{ // an RM cell for this VC does not exist. Create a ne one.
   rmcur = new(RMcells);
   if (rmcur == 0)
   {
      printf("Out Of Memory. - Drop_RMcells\n");
      exit(1);
   }
   else
   { //allocated successfully. Initialize it.
      rmcur->timestamp = newRM->timestamp;
      rmcur->vc_num = newRM->vc_num;
```

```
            rmcur->num_sws = newRM->num_sws;
            for (i=0;i<newRM->num_sws+2;i++)
               rmcur->swt_nums[i] = newRM->swt_nums[i];
            rmcur->cur_rate = newRM->cur_rate;
            rmcur->ER_rate = newRM->ER_rate;
            rmcur->Ld_Adj_Fctr = 0;
            rmcur->prop_time = newRM->prop_time;
            rmcur->DIR = newRM->DIR;
            rmcur->CI = newRM->CI;
            rmcur->next_cell = 0;

            //insert the new RM cell into the list
            if (swcur->BRMhead == 0)
               swcur->BRMhead = rmcur;
            else
            {
               rmprv = swcur->BRMhead;
               while (rmprv->next_cell != 0)
                  rmprv = rmprv->next_cell;
               rmprv->next_cell = rmcur;
            }
         }
      }
      return(0); //added successfully
}

/****
 Description: This routine adds an RM cell to the end of forward RM list
****/
int swtch::Add_FRMcell(struct RMcells *newRM, int sw_num)
{
   struct sw_node *swcur;
   struct RMcells *rmcur,*rmprv;
   int i;

   //find the given switch
   swcur = sw_head;
   while ((swcur != 0) && (swcur->sw_num != sw_num))
      swcur = swcur->next_sw;
```

```
//scan the list
rmcur = swcur->FRMhead;
rmprv = swcur->FRMhead;
while ((rmcur != 0) &&
        (rmcur->vc_num != newRM->vc_num))
{
   rmprv = rmcur;
   rmcur = rmcur->next_cell;
}

if ((rmcur != 0) && (rmcur->vc_num == newRM->vc_num))
{ //found an RM cell for the same VC. Update it
   if (newRM->ER_rate <= rmcur->ER_rate)
      rmcur->ER_rate = newRM->ER_rate;
   else
   {
      rmcur->timestamp = newRM->timestamp;
      rmcur->prop_time = newRM->prop_time;
      rmcur->cur_rate = newRM->cur_rate;
      rmcur->ER_rate = newRM->ER_rate;
      rmcur->DIR = newRM->DIR;
      rmcur->CI = newRM->CI;
   }
}
else
{ //create a new cell
   rmcur = new(RMcells);
   if (rmcur == 0)
   {
      printf("Out Of Memory. - Drop_RMcells\n");
      exit(1);
   }
   else
   { //initialize it
      rmcur->timestamp = newRM->timestamp;
      rmcur->vc_num = newRM->vc_num;
      rmcur->num_sws = newRM->num_sws;
      for (i=0;i<newRM->num_sws+2;i++)
         rmcur->swt_nums[i] = newRM->swt_nums[i];
      rmcur->cur_rate = newRM->cur_rate;
      rmcur->ER_rate = newRM->ER_rate;
```

```
            rmcur->Ld_Adj_Fctr = newRM->Ld_Adj_Fctr;
            rmcur->prop_time = newRM->prop_time;
            rmcur->DIR = newRM->DIR;
            rmcur->CI = newRM->CI;
            rmcur->next_cell = 0;

            //insert it into the list
            if (swcur->FRMhead == 0)
               swcur->FRMhead = rmcur;
            else
            {
               rmprv = swcur->FRMhead;
               while (rmprv->next_cell != 0)
                  rmprv = rmprv->next_cell;
               rmprv->next_cell = rmcur;
            }
         }
      }
      return(0); //added successfully
}

/****
 Description: This routine adds an RM cell to the end of RM list
 Note: The backward Rm list is used by ERICA and PRCA for RM list
****/
int swtch::Add_RM(struct RMcells *newRM, int sw_num)
{
   struct sw_node *swcur;
   struct RMcells *rmcur,*rmprv;
   int i;

   //find the switch
   swcur = sw_head;
   while ((swcur != 0) && (swcur->sw_num != sw_num))
      swcur = swcur->next_sw;

   //create a new cell
   rmcur = new(RMcells);
   if (rmcur == 0)
   {
      printf("Out Of Memory. - Drop_RMcells\n");
```

```
        exit(1);
      }
      else
      {  //initialize it
         rmcur->timestamp = newRM->timestamp;
         rmcur->vc_num = newRM->vc_num;
         rmcur->num_sws = newRM->num_sws;
         for (i=0;i<newRM->num_sws+2;i++)
            rmcur->swt_nums[i] = newRM->swt_nums[i];
         rmcur->cur_rate = newRM->cur_rate;
         rmcur->ER_rate = newRM->ER_rate;
         rmcur->Ld_Adj_Fctr = newRM->Ld_Adj_Fctr;
         rmcur->prop_time = newRM->prop_time;
         rmcur->DIR = newRM->DIR;
         rmcur->CI = newRM->CI;
         rmcur->next_cell = 0;

         //insert it into the list
         if (swcur->BRMhead == 0)
            swcur->BRMhead = rmcur;
         else
         {
            rmprv = swcur->BRMhead;
            while (rmprv->next_cell != 0)
               rmprv = rmprv->next_cell;
            rmprv->next_cell = rmcur;
         }
      }
 return(0); //added successfully
}

/****
 Description: This routine is used to set up ERICA parameters
              appropriately.
****/
int swtch::ERICA(struct sw_node *cur_sw)
{
   int l,i,sw_num,ret;
   struct vc_info *svlist;
   struct sw_node *nxtsw;
   int cell_count = 0;
```

```
    long    SureUB = 0; //bandwidth currently used by CBR and VBR

    //find the bandwidth used by ABR and the rest
    svlist = cur_sw->sw_vclist;
    while (svlist != 0)  //scan through the list
    {
       if (svlist->type == 4)
       {
          cell_count = cell_count + svlist->cur_num_cells;
          svlist->cur_num_cells = 0;
       }
       else
          SureUB = SureUB + svlist->used_cur_rate;
       svlist = svlist->next_vclist;
    }

    //Find the target cell rate
    cur_sw->TCR = ((cur_sw->TB-SureUB)*900)/424;
    cur_sw->TCC = (cur_sw->TCR*cur_sw->interval)/1000000;

    //calculate the fair share
    cur_sw->FS = ((cur_sw->TCR/TOT_ABR)*424)/1000;
    //set the load level accordingly
    if (cell_count == 0)
       cur_sw->LL = 1.0/(double)cur_sw->TCC;
    else
       cur_sw->LL = (double)cell_count/(double)cur_sw->TCC;
    return(0);
}
/************************** end of callsetup.cpp **************************/
```

```
/*****************************************************************************
 File: list.cpp

 Description: This file contains code that reads data from the input file and
              and stores it in memory in the form of lists.

*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include "list.h"

/***
 Description: This routine creates the list and initializes the list with
              data from the given input file.
***/
int vc::init_vclist()
{

  struct vc_node   *tmp,*tmp1,*tmp2;
  int   i,count,l;
  FILE   *input;
  int   vc_num = 0;

  input = fopen("INPUT","r");
  if (input == 0)
  {
    printf("Error opening input file.\n");
    exit(1);
  }

  while (!feof(input)) // read the complete file
  {
    tmp1 = new(vc_node);  //create a new structure for each VC
    if (tmp1 == 0)
    {
      printf("Out of memory.\n");
      exit(1);
    }
    tmp1->next_vc = 0;   //initialize to 0.
    for (i=0;i<10;i++)
      tmp1->swt_nums[i]=0;
```

```
//read from file and fill in the VC information
 fscanf(input,"%d %ld %ld %ld %d %d %ld %d ", &tmp1->traffic_type,
         &tmp1->peak_rate,&tmp1->mean_rate,
         &tmp1->init_rate,&tmp1->from,
         &tmp1->to,&tmp1->Msg_Len, &tmp1->num_sws);

 for (i=0;i<tmp1->num_sws;i++)
 {
   if (i == (tmp1->num_sws - 1))
     fscanf(input,"%d\n",&tmp1->swt_nums[i]);
   else
     fscanf(input,"%d ",&tmp1->swt_nums[i]);
 }
 // Initialize the rest of the fields to 0.
 tmp1->ERdat = 0;
 tmp1->cur_rate = tmp1->init_rate;
 tmp1->vc_num = vc_num;
 tmp1->active = 0;
 tmp1->StartTime = 0;
 tmp1->NxtCellTime = 0;
 tmp1->EndTime = 0;
 tmp1->CellsSent = 0;
 tmp1->Interval_timer = 0;
 tmp1->LstRMtime = 0;
 tmp1->RMwaittime = 0;
 tmp1->tglRM = 0;
 tmp1->transmit = 0;
 tmp1->Busy = 0;
 tmp1->CI = 0;
 tmp1->CellsInNxtFrm = 0;
 tmp1->rem_cells = 0.0;

 // set the number of cells to transmit according to the traffic type
 switch (tmp1->traffic_type)
 {
   case 1: // CBR
     tmp1->CellsInCurFrm = 0;
     tmp1->CellsInLstFrm = 0;
     tmp1->Cells2Trns = tmp1->Msg_Len*1000;
     break;
```

```
        case 2: //RTVBR
          tmp1->CellsInCurFrm = 24;
          tmp1->CellsInLstFrm = 24;
          tmp1->Cells2Trns = tmp1->Msg_Len*1000;
          break;
        case 3: //NRTVBR
          tmp1->CellsInCurFrm = 198;
          tmp1->CellsInLstFrm = 198;
          tmp1->Cells2Trns = (tmp1->Msg_Len/8)/48;
          //Above, div by 8 to convert to bytes
          // and then by 48 as 5 bytes are for header
          break;
        case 4: // ABR
          tmp1->CellsInCurFrm = 0;
          tmp1->CellsInLstFrm = 0;
          tmp1->Cells2Trns = (tmp1->Msg_Len/8)/48;
          //Above, div by 8 to convert to bytes
          // and then by 48 as 5 bytes are for header
          break;
      }

      tmp1->next_vc = 0;
      // append to the end of the list
      if (vc_head == 0)
        vc_head=tmp1;
      else
      {
        tmp2 = vc_head;
        while (tmp2->next_vc != 0)
          tmp2 = tmp2->next_vc;
        tmp1->next_vc = 0;
        tmp2->next_vc = tmp1;
      }
      vc_num++;
    }
    fclose(input);  //close the input file

    return(0);
}
```

```
/***
Description: Prints the data stored in the list and logs the total time taken
             by each VC to complete the entire message transmission.
***/
int vc::print_vclist()
{
  struct vc_node *tmp1;
  FILE *total_time;

  //open the logfile
  total_time = fopen("TRANSTM.OUT","w");
  if (total_time == 0)
  {
    printf("Error opening TRANSTM.\n");
    exit(1);
  }
  tmp1 = vc_head;
  while (tmp1 != 0)
  {
    switch(tmp1->traffic_type)
    {
      case 1:
        printf("\nTYPE: CBR\n");
        printf("VC Number: %d\n",tmp1->vc_num);
        printf("Peak Rate: %ld\n",tmp1->peak_rate);
        printf("Current Rate: %ld\n",tmp1->cur_rate);
        printf("Active: %d\n",tmp1->active);
        break;
      case 2:
        printf("\nTYPE: RTVBR\n");
        printf("VC Number: %d\n",tmp1->vc_num);
        printf("Peak Rate: %ld\n",tmp1->peak_rate);
        printf("Mean Rate: %ld\n",tmp1->mean_rate);
        printf("Current Rate: %ld\n",tmp1->cur_rate);
        printf("Active: %d\n",tmp1->active);
        break;
      case 3:
        printf("\nTYPE: NRTVBR\n");
        printf("VC Number: %d\n",tmp1->vc_num);
        printf("Peak Rate: %ld\n",tmp1->peak_rate);
        printf("Mean Rate: %ld\n",tmp1->mean_rate);
```

```
            printf("Current Rate: %ld\n",tmp1->cur_rate);
            printf("Active: %d\n",tmp1->active);
            break;
          case 4:
            printf("\nTYPE: ABR\n");
            printf("VC Number: %d\n",tmp1->vc_num);
            printf("Peak Rate: %ld\n",tmp1->peak_rate);
            printf("Current Rate: %ld\n",tmp1->cur_rate);
            printf("Active: %d\n",tmp1->active);
            break;
          default:
            printf("\n\nError in type.\n");
            printf("VC Number: %d\n\n",tmp1->vc_num);
            break;
        }
        printf("VC_NUM: %d REM_CELLS: %lf\n",tmp1->vc_num,
                tmp1->rem_cells);
        fprintf(total_time,"%d %lf %lf\n", tmp1->vc_num, tmp1->StartTime,
                tmp1->EndTime); // Log to file
        tmp1 = tmp1->next_vc;
    }
    fclose(total_time); //close the log file
    return(0);
}

/***
Description: This routine frees the memory used by the VC list
***/
int vc::clean_vclist()
{
    struct vc_node *tmp1;
    while (vc_head->next_vc != 0)
    {
        tmp1 = vc_head->next_vc;
        vc_head->next_vc = vc_head->next_vc->next_vc;
        delete(tmp1);
    }
    delete(vc_head);
    return(0);
}
/************************* end of list.cpp *********************************/
```

```
/*****************************************************************************
 File: main.cpp

 Description: The file contains the main loop of simulation. It opens and
              closes all the logfiles. It adds a VC for simulation at
              piosson arrival rate. It also takes care of freeing all the
              allocated memory for simulation.
 *****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include "list.h"
#include "switch.h"
#include "source.h"

#define Total_VCs 40

int main(int argc,char **argv)
{
  vc   vclist;
  swtch   swlist;
  source   src;
  struct sw_node   *tmp;
  int   acr = 0;
  int   ret = 1;
  int   ret1 = 0;
  double   time = 0;
  int   nxtcall = 0;
  double   Sum;
  FILE   *qcount1,*qcount2,*qcount3,*qcount4,*qcount5;
  FILE   *bwidth1,*bwidth2,*bwidth3,*bwidth4,*bwidth5, *abrvc;

  if (argc == 1)
    acr = 0;
  else
    acr = atoi(argv[1]); //0 = EARA; 1 = PRCA; 2 = ERICA;

  // intialize the vc and the switch list by reading the input file.
  vclist.init_vclist();
  swlist.init_swlist();
```

```
//open all log files
qcount1 = fopen("QCOUNT1.OUT","w");
qcount2 = fopen("QCOUNT2.OUT","w");
qcount3 = fopen("QCOUNT3.OUT","w");
qcount4 = fopen("QCOUNT4.OUT","w");
qcount5 = fopen("QCOUNT5.OUT","w");
bwidth1 = fopen("BWIDTH1.OUT","w");
bwidth2 = fopen("BWIDTH2.OUT","w");
bwidth3 = fopen("BWIDTH3.OUT","w");
bwidth4 = fopen("BWIDTH4.OUT","w");
bwidth5 = fopen("BWIDTH5.OUT","w");

if ((qcount1 == 0) || (qcount2 == 0) ||
   (bwidth1 == 0) || (bwidth2 == 0) ||
   (abrvc == 0))
{
  printf("Error opening OUTPUT FILES \n");
  exit(1);
}

// Start the simulation
while ((ret == 1))
{
  if (nxtcall == 0)
  {
    swlist.call_setup(vclist.vc_head,acr,time); //add a VC
    //Piosson Arrival Rate
    Sum = 0;
    while (Sum < 1.0)
    {
      nxtcall++;
      Sum = Sum+src.expo(1.0/double(Total_VCs));
    }
    nxtcall = nxtcall * 1000;
 }
 else
 {
   if (nxtcall > 0)

     nxtcall--;
 }
```

```
        //send cells to the switch according to the rate.
        ret1 = src.send_cells(vclist.vc_head,swlist.sw_head,time,
                              acr,cell_loss,abrvc);

        //send cell to the next node
        ret = swlist.swt_all(qcount1,qcount2,qcount3,qcount4,qcount5,
                             bwidth1,bwidth2,bwidth3,bwidth4,bwidth5,
                             vclist.vc_head,time,acr);
        if (ret != 0)
        {
          printf("Error in switching.\n");
          break;
        }
        //Check if all VCs completed messages transmission
        ret = src.All_done(vclist.vc_head,swlist.sw_head);
        // increment timer in milliseconds
        time = time + 1.0;
      } //end of simulation

      // Close all opened files
      fclose(qcount1);
      fclose(qcount2);
      fclose(qcount3);
      fclose(qcount4);
      fclose(qcount5);
      fclose(bwidth1);
      fclose(bwidth2);
      fclose(bwidth3);
      fclose(bwidth4);
      fclose(bwidth5);

      // print the VC and switch list
      vclist.print_vclist();
      swlist.print_swlist();
      // Free the allocated memory
      vclist.clean_vclist();
      swlist.clean_swlist();
      return(0);
}
/***************************** end of main.cpp ******************************/
```

```
/*****************************************************************************
 File: source.cpp

 Description:  This file contains code that is excuted by the source.
               It scans the source list and for all active sources, it
               generates cells according to its traffic type and the rate.
*****************************************************************************\
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "list.h"
#include "switch.h"
#include "source.h"

/****
 Description: This routine is called by the main. It checks the source list
              and for all active VCs, it creates and sends cells to the
              appropriate switch.
****/
int source::send_cells(struct vc_node *vhead, struct sw_node *shead,
                       double time,int ACR,FILE *cell_loss, FILE *abrvc)
{
  struct vc_node   *vcnxt;
  struct sw_node   *swnxt;
  struct cells   *cellnxt,*cellprv;
  struct vc_info   *swvcnxt;
  struct Rmcells   *rmnxt,*rmprv;
  struct ER_Data   *tmpERdat,*delERdat;
  int   i=0;
  double   k=0;
  int   a,done,l;
  int   nocell;
  long   num_cells;

  vcnxt = vhead;
  while (vcnxt != 0) //scan the complete list
  {                  //active source
    a = 0;
    nocell = 0;
    num_cells = 0;
    done = 0;
```

```
if (vcnxt->active == 1)
{
  if (vcnxt->Cells2Trns == 0) //completed message transfer?
    done = 1; //yes
  swnxt = shead;
  while (swnxt != 0) //find the first switch in the path
  {
    if (swnxt->sw_num == vcnxt->swt_nums[0])
      break;
    else
      swnxt = swnxt->next_sw;
  }
  if (swnxt == 0) //could not find the switch - internal error
  {
    printf("Error in Switch List - Send_cells \n");
    exit(1);
  }
  if ((vcnxt->NxtCellTime == 0) || (vcnxt->NxtCellTime <= time) ||
    (vcnxt->transmit > 0))  // time to transmit?
  {                          // yes
    cellnxt = new(cells);    // create a new cell
    if (cellnxt == 0)        // no memory - internal error
    {
      printf("Out of memory - source \n");
      fprintf(cell_loss,"%d %d %lf ", vcnxt->traffic_type,
              vcnxt->vc_num,time);
      nocell = 1;
    }
    // initialize the cell parameters
    cellnxt->timestamp = time;
    cellnxt->num_sws = vcnxt->num_sws;
    for (i = 0; i < 10; i++)
      cellnxt->swt_nums[i] = 0;
    cellnxt->swt_nums[0] = vcnxt->from;
    for (i = 0; i < vcnxt->num_sws; i++)
      cellnxt->swt_nums[i+1] = vcnxt->swt_nums[i];
    cellnxt->swt_nums[i+1] = vcnxt->to;
    cellnxt->next_cell = 0;
    cellnxt->vc_num = vcnxt->vc_num;
    cellnxt->prop_time = 0;
    cellnxt->num_cells = 0;
```

```
cellnxt->CI = 0;
if (done == 1)  // is this the last cell to be transmitted?
  cellnxt->last_cell = 1;
else
  cellnxt->last_cell = 0;
k = 0.0;
switch(vcnxt->traffic_type)
{
  case 1: //CBR
    if (done ==  0)
    {  //find the rate
      k = (double)vcnxt->mean_rate/424000.0;  //384=8*48/ms
      vcnxt->rem_cells = vcnxt->rem_cells + k;
      num_cells = (long)vcnxt->rem_cells;
      if (num_cells > 0) //can a cell be transmitted?
        vcnxt->rem_cells = vcnxt->rem_cells - (double)num_cells;
    }
    if ((num_cells > 0) || (done == 1))
    {  // cell is transmitted - insert into the cells list of swt
      cellnxt->num_cells = num_cells;
      if (swnxt->Chead == 0)
        swnxt->Chead = cellnxt;
      else
      {
        cellprv = swnxt->Chead;
        while (cellprv->next_cell != 0)
          cellprv = cellprv->next_cell;
        cellprv->next_cell = cellnxt;
      }
      swvcnxt = swnxt->sw_vclist; //find this VC on the switch
      while ((swvcnxt != 0) &&
             (swvcnxt->vc_num != vcnxt->vc_num))
        swvcnxt = swvcnxt->next_vclist;
      if (swvcnxt == 0) // internal error
      {
        printf("Error in vc_num - source %d\n",vcnxt->vc_num);
        exit(1);
      }
      // update vc info. on switch
      swvcnxt->cur_num_cells = swvcnxt->cur_num_cells +
                                              cellnxt->num_cells;
```

```
    }
    else // not yet time to send.
      delete(cellnxt);
    break;
  case 2: //RTVBR
    if (done == 0)
    { //active
      if ((vcnxt->NxtCellTime == 0)||(vcnxt->NxtCellTime <= time))
      { // generate real time VBR traffic
        k = Gen_VBR(vcnxt);  //384 = 8 * 48
        vcnxt->transmit = vcnxt->transmit + k;
        // send cells every 33 ms
        vcnxt->NxtCellTime = time + 33000.0;
      }
      if (vcnxt->transmit >= 1)
      { //data to transmit?
        k = (double)vcnxt->peak_rate/424000.0;
                                    //384 = 8 * 48 per millisecond
        vcnxt->rem_cells = vcnxt->rem_cells + k;
        num_cells = (long)vcnxt->rem_cells;
        if (num_cells > 0) // time to transmit?
        {
          vcnxt->rem_cells = vcnxt->rem_cells - (double)num_cells;
          vcnxt->transmit = vcnxt->transmit - (double)num_cells;
          if (vcnxt->transmit == 0)
            vcnxt->rem_cells = 0;
        }
      }
    }
    if ((num_cells > 0) || (done == 1))
    { // cell is transmitted - insert into the cells list of swt
      cellnxt->num_cells = num_cells;
      if (swnxt->Rhead == 0)
        swnxt->Rhead = cellnxt;
      else
      {
        cellprv = swnxt->Rhead;
        while (cellprv->next_cell != 0)
          cellprv = cellprv->next_cell;
        cellprv->next_cell = cellnxt;
      }
```

```
      swvcnxt = swnxt->sw_vclist; // find the vc on switch
      while ((swvcnxt != 0) &&
             (swvcnxt->vc_num != vcnxt->vc_num))
        swvcnxt = swvcnxt->next_vclist;
      if (swvcnxt == 0) //internal error
      {
        printf("Error in vc_num - source %d\n",vcnxt->vc_num);
        exit(1);
      }
      // update vc info. on switch
      swvcnxt->cur_num_cells = swvcnxt->cur_num_cells +
                                                cellnxt->num_cells;
    }
    else
      delete(cellnxt);
    break;
  case 3: //NRTVBR
    if (done == 0)
    { //active
      if ((vcnxt->NxtCellTime == 0)||(vcnxt->NxtCellTime <= time))
      { //time to transmit next cell
        if ((vcnxt->Busy == 0) || (vcnxt->Busy == 1))
        { //generate VBR traffic if busy period
          k = Gen_VBR(vcnxt);  //384 = 8 * 48
          vcnxt->transmit = vcnxt->transmit + k;
          if (vcnxt->Busy == 0)
          {
            int l = rand()%2;
            //randomly set next cell generation time
            if (l == 0)
            {
              vcnxt->NxtCellTime = time + 16000.0;
              vcnxt->Busy = 2;
            }
            else
            {
              vcnxt->NxtCellTime = time + 8000.0;
              vcnxt->Busy = 1;
            }
          }
          else
```

```
        { //idle time
          vcnxt->Busy = 2;
          vcnxt->NxtCellTime = time + 8000.0;
        }
      }
      else
      { //wait for exponential time
        a = 2;
        double temp = 0.0;
        while (((long)temp < 1) || ((long)temp > 500000))
          temp = expo(1.0/160000.0);
        vcnxt->NxtCellTime = time + (long)temp;
        vcnxt->Busy = 0; // next time generate cells
      }
    }
    if (vcnxt->transmit >= 1)
    { // transmit cells
      k = (double)vcnxt->peak_rate/424000.0; //384=8*48/ms
      vcnxt->rem_cells = vcnxt->rem_cells + k;
      num_cells = (long)vcnxt->rem_cells;
      if (num_cells > 0)
      { //time to transmit
        vcnxt->rem_cells = vcnxt->rem_cells - (double)num_cells;
        vcnxt->transmit = vcnxt->transmit - (double)num_cells;
        if (vcnxt->transmit == 0)
          vcnxt->rem_cells = 0;
      }
    }
  }
  if (((num_cells > 0) && (a == 0)) || (done == 1))
  { // cell is transmitted - insert into the cells list of swt
    cellnxt->num_cells = num_cells;
    if (swnxt->Nhead == 0)
      swnxt->Nhead = cellnxt;
    else
    {
      cellprv = swnxt->Nhead;
      while (cellprv->next_cell != 0)
        cellprv = cellprv->next_cell;
      cellprv->next_cell = cellnxt;
    }
```

```
      swvcnxt = swnxt->sw_vclist; //find the vc on the switch
      while ((swvcnxt != 0) &&
             (swvcnxt->vc_num != vcnxt->vc_num))
        swvcnxt = swvcnxt->next_vclist;
      if (swvcnxt == 0) //internal error
      {
        printf("Error in vc_num - source %d\n",vcnxt->vc_num);
        exit(1);
      }
      //update vc info. on switch
      swvcnxt->cur_num_cells = swvcnxt->cur_num_cells +
                                              cellnxt->num_cells;
    }
    else
      delete(cellnxt);
    break;
  case 4: //ABR
    if (vcnxt->ERdat != 0) // if any rm cells have arrived
    {
      SortArray(vcnxt); //get the latest one
      tmpERdat = vcnxt->ERdat;
      while (tmpERdat != 0) //if any RM cell
      {
        if (tmpERdat->wait_time > 1) //wait for the transmission
          tmpERdat->wait_time = tmpERdat->wait_time - 1;//time
        else
        { // update the tranmission rate
          if (tmpERdat->ER_rate > vcnxt->peak_rate)
            tmpERdat->ER_rate = vcnxt->peak_rate;
          vcnxt->cur_rate = tmpERdat->ER_rate;

          //delete this RM cell
          delERdat = vcnxt->ERdat;
          if (vcnxt->ERdat->next == 0)
            vcnxt->ERdat = 0;
          else
            vcnxt->ERdat = vcnxt->ERdat->next;
          delete(delERdat);
        }
        tmpERdat = tmpERdat->next; //continue to scan
      }
```

```
}
if (vcnxt->cur_rate > 0) //allowed rate is not zero (minimum)
{
  k = (double)vcnxt->cur_rate/424000.0;
  vcnxt->rem_cells = vcnxt->rem_cells + k;
  if (vcnxt->rem_cells > 1.0)
  { //time to transmit a cell
    if ((((ACR == 1) ||  //PRCA
         (ACR == 0)) &&  //EARA
         (vcnxt->CellsSent >= 32) &&
         (done == 0)) ||
        ((ACR == 2) &&  //ERICA
        (vcnxt->Interval_timer >= Avg_Int) &&
        (done == 0)))
    { //create an RM cell
      rmnxt = new(RMcells);
      if (rmnxt == 0) //internal error
        printf("Out of memory. RM\n");
      else
      { //initialize the rm cell parameters
        rmnxt->next_cell = 0;
        rmnxt->timestamp = time;
        rmnxt->vc_num = vcnxt->vc_num;
        rmnxt->num_sws = vcnxt->num_sws;
        rmnxt->prop_time = 0;
        rmnxt->swt_nums[0] = vcnxt->from;
        for (i=0;i<rmnxt->num_sws;i++)
          rmnxt->swt_nums[i+1] = vcnxt->swt_nums[i];
        rmnxt->swt_nums[i+1] = vcnxt->to;
        rmnxt->cur_rate = vcnxt->cur_rate;
        if (ACR == 1) //PRCA
          rmnxt->ER_rate = vcnxt->cur_rate +
                           (long)(vcnxt->cur_rate/256);//increment
        else //ERICA
          rmnxt->ER_rate = vcnxt->cur_rate;
          rmnxt->Ld_Adj_Fctr = 0;
          rmnxt->CI = 0;
          rmnxt->DIR = 0;
        if (rmnxt->ER_rate > vcnxt->peak_rate)
          rmnxt->ER_rate = vcnxt->peak_rate;
```

```
            //insert it into the list
            if (ACR != 0)
            {
              if (swnxt->BRMhead == 0)
                swnxt->BRMhead = rmnxt;
              else
              {
                rmprv = swnxt->BRMhead;
                while (rmprv->next_cell != 0)
                  rmprv = rmprv->next_cell;
                rmprv->next_cell = rmnxt;
              }
            }
            else
            { //EARA
              if (swnxt->FRMhead == 0)
                swnxt->FRMhead = rmnxt;
              else
              {
                rmprv = swnxt->FRMhead;
                while (rmprv->next_cell != 0)
                  rmprv = rmprv->next_cell;
                rmprv->next_cell = rmnxt;
              }
            }
          }
          //reset the counters
          vcnxt->CellsSent = 0;
          vcnxt->Interval_timer = 0;
          vcnxt->rem_cells = vcnxt->rem_cells - 1.0;
        }
        num_cells = (long)vcnxt->rem_cells;
        if (num_cells > 0)
        { //time to transmit a cell?
          vcnxt->rem_cells = vcnxt->rem_cells - (double)num_cells;
          vcnxt->CellsSent = vcnxt->CellsSent + num_cells;
        }
        else
          a = 2;
      }
    }
```

```
        if (((num_cells > 0) && (a == 0)) || (done == 1))
        { // cell is transmitted - insert into the cells list of swt
          cellnxt->num_cells = num_cells;
          if (swnxt->Ahead == 0)
            swnxt->Ahead = cellnxt;
          else
          {
            cellprv = swnxt->Ahead;
            while (cellprv->next_cell != 0)
              cellprv = cellprv->next_cell;
            cellprv->next_cell = cellnxt;
          }
          swvcnxt = swnxt->sw_vclist; //find the vc on the switch
          while ((swvcnxt != 0) &&
                 (swvcnxt->vc_num != vcnxt->vc_num))
            swvcnxt = swvcnxt->next_vclist;
          if (swvcnxt == 0) //internal error
          {
            printf("Error in vc_num - source %d\n",vcnxt->vc_num);
            exit(1);
          }
          //update vc info. on switch
          swvcnxt->cur_num_cells = swvcnxt->cur_num_cells +
                                                  cellnxt->num_cells;
        }
        else
          delete(cellnxt);
        if (ACR == 2) //ERICA
          vcnxt->Interval_timer++;
        break;
      }
    }
  if (((vcnxt->traffic_type == 1) || (vcnxt->traffic_type == 2)) &&
    (vcnxt->Cells2Trns > 0))
    vcnxt->Cells2Trns--; //CBR & RTVBR
  if ((vcnxt->traffic_type == 3) || (vcnxt->traffic_type == 4))
  { //NRTVBR and ABR
    vcnxt->Cells2Trns = vcnxt->Cells2Trns - cellnxt->num_cells;
    if (vcnxt->Cells2Trns < 0)
      vcnxt->Cells2Trns = 0;
  }
```

```
        if (vcnxt->Cells2Trns == 0)
        {
          vcnxt->NxtCellTime = 0;
          vcnxt->Busy = 0;
        }
        if ((vcnxt->Cells2Trns == 0) && (done == 1))
          vcnxt->active = 2; //Message transfer complete - release bw
        }
        vcnxt = vcnxt->next_vc; //process the next source
      }
   return(0);  //successfully processed all sources
}

double source::Gen_VBR(struct vc_node *curvc)
{
   curvc->CellsInNxtFrm = a0+(a1*curvc->CellsInCurFrm)-
                                        (a2*curvc->CellsInLstFrm)+getEn();
   curvc->CellsInLstFrm = curvc->CellsInCurFrm;
   curvc->CellsInCurFrm = curvc->CellsInNxtFrm;
   return(curvc->CellsInNxtFrm);
}

double source::getEn()
{
   int i;
   double x,y,t1,t2;
   int m = 0;

   t1 = 6.28*(rand()%100);
   m = (rand()%10);
   while (m == 0)
     m = (rand()%6);
   t2 = sqrt(-2.0*-log(m));
   x = cos(t1)*t2;
   x = (StdDev*x) + Mean;
   y = sin(t1)*t2;
   y = (StdDev*y) + Mean;

   i = rand()%10;
   if (i <5)
     return(x);
```

```
    else
      return(y);
}


double source::expo(double rate)
{
  double U,result;

  U = frand();
  if (U < 0.001)
    U = 0.001;
  result = -log10(U)/rate;

  return(result);
}

double source::frand()
{
  double result = 0;
  int i;
  const double D=13849;
  const double M=32768;
  const double C=25173;

  i = rand()%10000;
  result = (C*(double)i)+D;
  result = fmod(result,M);
  result = result/M;
  return(result);
}

int source:: All_done(struct vc_node *vhead,struct sw_node *shead)
{
  struct vc_node *tmpvc;
  struct sw_node *swnxt;
  struct vc_info *swvcnxt;

  tmpvc = vhead;
  while (tmpvc != 0)
  {
```

```
      if (tmpvc->active != 2)
        return(1);
      else
        tmpvc = tmpvc->next_vc;
    }
    swnxt = shead;
    while (swnxt != 0)
    {
      swvcnxt = swnxt->sw_vclist;
      if (swvcnxt != 0)
        return(1);
      swnxt = swnxt->next_sw;
    }
    return(0);
}

void source::SortArray(struct vc_node *vcnxt)
{
  struct ER_Data *tmpERdat,*prvERdat,*nxtERdat,*insERdat,*delERdat;

  // Sort array according to wait_time
  tmpERdat = vcnxt->ERdat;
  while (tmpERdat->next != 0)
  {
    prvERdat = tmpERdat->next;
    nxtERdat = tmpERdat->next;
    while (nxtERdat != 0)
    {
      if (tmpERdat->wait_time > nxtERdat->wait_time)
      {
        if (nxtERdat == prvERdat)
        {
          nxtERdat = nxtERdat->next;
          tmpERdat->next = nxtERdat;
          prvERdat->next = tmpERdat;
          tmpERdat = prvERdat;
          prvERdat = tmpERdat->next;
        }
        else
        {
          if (nxtERdat->next == 0)
```

```
            prvERdat->next = 0;
          else
            prvERdat->next = nxtERdat->next;
          nxtERdat->next = tmpERdat;
          vcnxt->ERdat = nxtERdat;
          nxtERdat = prvERdat->next;
        }
      }
      else
      {
            prvERdat = nxtERdat;
            nxtERdat = nxtERdat->next;
      }
    }
    tmpERdat = tmpERdat->next;
}

//delete the entry whose timestamp less than the first entry
tmpERdat = vcnxt->ERdat;
while (tmpERdat != 0)
{
  if (tmpERdat->next == 0)
    break;
  nxtERdat = tmpERdat->next;
  prvERdat = tmpERdat->next;
  while (nxtERdat != 0)
  {
    if (nxtERdat->ER_Chng <= tmpERdat->ER_Chng)
    {
      delERdat = nxtERdat;
      if (tmpERdat->next == nxtERdat)
      {
        if (nxtERdat->next == 0)
        {
          prvERdat = 0;
          nxtERdat = 0;
          tmpERdat->next = 0;
        }
        else
        {
          prvERdat = prvERdat->next;
```

```
                nxtERdat = nxtERdat->next;
                tmpERdat->next = nxtERdat;
              }
            }
            else
            {
              if (nxtERdat->next == 0)
              {
                prvERdat->next = 0;
                nxtERdat = 0;
              }
              else
              {
                prvERdat->next = nxtERdat->next;
                nxtERdat = prvERdat->next;
              }
            }
            delERdat->next = 0;
            delete(delERdat);
          }
          else
          {
            prvERdat = nxtERdat;
            nxtERdat = nxtERdat->next;
          }
        }
        tmpERdat = tmpERdat->next;
      }
      return;
}
/************************ end of source.cpp *********************************/
```

```
/*****************************************************************************
 File: switch.cpp

 Description: This file contains the switching algorithms. It also initalizes
              the switch and receives/sends the cells from node to node.
*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "switch.h"
#include "list.h"

/****
Description: This routine intializes all the switches used for simulation.
             It sets the correct default values.
****/
int swtch::init_swlist()
{
  struct sw_node   *tmp,*tmp1;
  struct cells   *init_cell,*Icell;
  struct RMcells   *RM_init_cell, *RM_Icell;
  int   i,j,k;
  int   sw_num=100;

  for (k=0;k<TOT_SWT;k++)
  {
    tmp = new(sw_node);   //create a new switch
    if (tmp == 0)      //internal error
    {
      printf("Out of memeory in switch module.\n");
      exit(1);
    }

    tmp->AB = 155000; //Available Bandwidth
    tmp->UB = 0;   //Used Bandwidth
    tmp->TB = 155000; //Total Bandwidth
    tmp->TCR = ((double)tmp->TB*1000*.9)/424; //used for ERICA
    tmp->TCC = ((long)tmp->TCR*300)/1000000; //used for ERICA
    tmp->FS = (tmp->TCR*424)/(TOT_ABR*1000); //used for ERICA
    tmp->LL = 1; //used for ERICA
    tmp->Cellspermicsec = (double)tmp->TB/424000.0;
```

```
        tmp->sw_num = sw_num; //switch number
        tmp->Ready = 0;
        tmp->qued_cells= 0;
        tmp->rem_cells=0;
        tmp->Ready_time = 0;
        tmp->interval = 300; //300 micro sec - ERICA
        tmp->timer = 0;
        tmp->sw_vclist = 0;
        tmp->Chead = 0; //CBR Data Cells Queue
        tmp->Rhead = 0; //RTVBR Data Cells Queue
        tmp->Nhead = 0; //NRTVBR Data Cells Queue
        tmp->Ahead = 0; //ABR Data Cells Queue
        tmp->BRMhead = 0; //Backward RM Cell Queue
        tmp->FRMhead = 0; //Forward RM Cell Queue
        tmp->next_sw = 0; //Pointer to the next switch

        if (sw_head == 0) // insert into the list
          sw_head = tmp;
        else
        {
          tmp1 = sw_head;
          while (tmp1->next_sw != 0)
            tmp1 = tmp1->next_sw;
          tmp1->next_sw = tmp;
        }
        sw_num++;
    }
    return(0); //successfully initialized all switches
}

/****
Description: This routine prints the number of cells in each queue as well
             as the total, including RM cells.
****/
int swtch::print_swlist()
{
  struct sw_node *tmp1;
  struct cells *ctmp;
  struct RMcells *rtmp;
  int c,r,n,a,brm,frm;
```

```
tmp1 = sw_head;

while (tmp1 != 0)
{
  c=r=n=a=brm=frm=0;
  rtmp = tmp1->BRMhead;
  while (rtmp != 0) //find out the number of cells in backward RM queue
  {
    brm++;
    rtmp = rtmp->next_cell;
  }
  rtmp = tmp1->FRMhead;
  while (rtmp != 0) //find out the number of cells in forward RM queue
  {
    frm++;
    rtmp = rtmp->next_cell;
  }
  ctmp = tmp1->Chead;
  while (ctmp != 0) //find out the number of cells in CBR queue
  {
    c++;
    ctmp = ctmp->next_cell;
  }
  ctmp = tmp1->Rhead;
  while (ctmp != 0) //find out the number of cells in RTVBR queue
  {
    r++;
    ctmp = ctmp->next_cell;
  }
  ctmp = tmp1->Nhead;
  while (ctmp != 0) //find out the number of cells in NRTVBR queue
  {
    n++;
    ctmp = ctmp->next_cell;
  }
  ctmp = tmp1->Ahead;
  while (ctmp != 0) //find out the number of cells in ABR queue
  {
    a++;
    ctmp = ctmp->next_cell;
  }
```

```
      //Print to screen
      printf("SW_NUM: %d TB: %ld UB: %ld AB: %ld\n",
              tmp1->sw_num,tmp1->TB,tmp1->UB,tmp1->AB);
      printf("QUEUE - BRM: %d CBR: %d RTVBR: %d NRTVBR: %d
              ABR: %d FRM: %d\n", brm,c,r,n,a,frm);
      tmp1 = tmp1->next_sw;
    }
    return(0); //completed scanning through all queues
}

/****
Description: This routine frees all the memory allocated for the switch
***/
int swtch::clean_swlist()
{
  struct sw_node *tmp1,*tmp;
  struct cells *ctmp,*ctmp1;
  struct RMcells *rtmp,*rtmp1;

  tmp1 = sw_head;
  while (tmp1 != 0)
  {
    rtmp = tmp1->BRMhead;
    while (rtmp != 0) //Clean up backward RM queue
    {
      rtmp1 = rtmp;
      rtmp = rtmp->next_cell;
      delete(rtmp1);
    }
    rtmp = tmp1->FRMhead;
    while (rtmp != 0) //Clean up forward RM queue
    {
      rtmp1 = rtmp;
      rtmp = rtmp->next_cell;
      delete(rtmp1);
    }
    ctmp = tmp1->Chead;
    while (ctmp != 0) //Clean up CBR queue
    {
      ctmp1 = ctmp;
      ctmp = ctmp->next_cell;
```

```
        delete(ctmp1);
      }
      ctmp = tmp1->Rhead;
      while (ctmp != 0) //Clean up RTVBR queue
      {
        ctmp1 = ctmp;
        ctmp = ctmp->next_cell;
        delete(ctmp1);
      }
      ctmp = tmp1->Nhead;
      while (ctmp != 0) //Clean up NRTVBR queue
      {
        ctmp1 = ctmp;
        ctmp = ctmp->next_cell;
        delete(ctmp1);
      }
      ctmp = tmp1->Ahead;
      while (ctmp != 0) //Clean up ABR queue
      {
        ctmp1 = ctmp;
        ctmp = ctmp->next_cell;
        delete(ctmp1);
      }
      tmp = tmp1;
      tmp1 = tmp1->next_sw; //get next witch
      tmp->next_sw = 0;
      delete(tmp);
   }
   return(0); //freed all allocated memory
}

/****
Description: This routine switches all the cells in the queue according to
             the priority or traffic type.
****/
int swtch::switch_cells(struct sw_node *swcur,double time,
                        struct vc_node *vhead,int ACR)
{
  struct cells *head;
  struct RMCells *rmh;
  int type=0;
```

```
      type=1;
      while ((swcur->Ready >= 1) && (type < 5)) //time to transmit?
      {
        switch (type)
        {
          case 1:
            if (swcur->Chead != 0) //any CBR cells?
              swap_cells(swcur,swcur->Chead,type,time,vhead,ACR);
            break;
          case 2:
            if (swcur->Rhead != 0) //any RTVBR cells?
              swap_cells(swcur,swcur->Rhead,type,time,vhead,ACR);
            break;
          case 3:
            if (swcur->Nhead != 0) //any NRTVBR cells?
              swap_cells(swcur,swcur->Nhead,type,time,vhead,ACR);
            break;
          case 4:
            if (swcur->Ahead != 0) //any ABR cells?
              swap_cells(swcur,swcur->Ahead,type,time,vhead,ACR);
            break;
        }
        type++;
      }
    if ((swcur->BRMhead == 0) && (swcur->Chead == 0) &&
       (swcur->Rhead == 0) && (swcur->Nhead == 0) &&
       (swcur->Ahead == 0) && (swcur->FRMhead == 0))
      swcur->Ready = 0; // no cells in queue
    return(0); //completed transmission successfully
}

/****
Description: This routine finds the next node to whcih the cell must be
             passed to and sets up all the parameters correclty
             and transmits them.
****/
int swtch::swap_cells(struct sw_node *swcur,struct cells *head,int type,
                      double time,struct vc_node *vhead,int ACR)
{
  int  j=0;
```

```
struct sw_node   *swnxt;
struct cells   *nxtcell,*newcell,*cur;
struct vc_node   *vcnxt;
struct vc_info   *swvcprv,*swvcnxt;


  while ((head != 0) && (swcur->Ready >= 1)) //switch all cells in the
  {                          // given queue if we have the bandwidth
    for (j = 0; j < head->num_sws+2; j++)
    {                        //find the switch number
      if (head->swt_nums[j] == swcur->sw_num)
        break;
    }
    if (head->swt_nums[j-1] < 100) //from source
      head->prop_time = head->prop_time+SRC_SWT; //update the
    else                                  // propagation delay accordingly
      head->prop_time = head->prop_time+SWT_SWT;
    if (head->swt_nums[j+1] < 100)
    { //reached destination.
      printf("DST: %d Time: %lf\n",
           head->swt_nums[j+1],(time+head->prop_time+SRC_SWT));
      if (ACR == 1) // if PRCA
      {
        if (head->CI == 1) // Congestion?
          vcnxt->CI = 1;  // Set it and store
        else
          vcnxt->CI = 0;
      }
      if ((head->last_cell == 1) &&
        ((long)swcur->Ready >= head->num_cells))
      {                  // Last cell to be transmitted on this VC
        vcnxt = vhead;
        while ((vcnxt != 0) && (vcnxt->vc_num != head->vc_num))
          vcnxt = vcnxt->next_vc;
        //update end time
        vcnxt->EndTime = time+head->prop_time+SRC_SWT;
      }
    }
    else
    {
      swnxt = sw_head;
```

```
while ((swnxt->sw_num != head->swt_nums[j+1]) &&
        (swnxt != 0)) //find the next switch
  swnxt = swnxt->next_sw;
if (swnxt == 0) //internal error
{
  printf("Error: No such switch number: %d
          Module switch_cells.\n", nxtcell->swt_nums[j+1]);
  exit(1);
}
newcell = new(cells); //create a new data cell
if (newcell == 0) //internal error
{
  printf("Out of memory.\n");
  exit(1);
}
for (j=0;j<10;j++) //initialize all info. on the cell
  newcell->swt_nums[j] = 0;
newcell->num_sws = 0;
newcell->next_cell = 0;
switch (type) //according to the type insert into the queue of the
{          //next switch
  case 1:
    if (swnxt->Chead == 0)
      swnxt->Chead = newcell;
    else
    {
      nxtcell = swnxt->Chead;  // In the next switch
      while (nxtcell->next_cell != 0)
        nxtcell = nxtcell->next_cell;
      nxtcell->next_cell = newcell;
    }
    break;
 case 2:
    if (swnxt->Rhead == 0)
      swnxt->Rhead = newcell;
    else
    {
      nxtcell = swnxt->Rhead;  // In the next switch
      while (nxtcell->next_cell != 0)
        nxtcell = nxtcell->next_cell;
      nxtcell->next_cell = newcell;
```

```
        }
        break;
      case 3:
        if (swnxt->Nhead == 0)
          swnxt->Nhead = newcell;
        else
        {
          nxtcell = swnxt->Nhead;  // In the next switch
          while (nxtcell->next_cell != 0)
            nxtcell = nxtcell->next_cell;
          nxtcell->next_cell = newcell;
        }
        break;
      case 4:
        if (swnxt->Ahead == 0)
          swnxt->Ahead = newcell;
        else
        {
          nxtcell = swnxt->Ahead;  // In the next switch
          while (nxtcell->next_cell != 0)
            nxtcell = nxtcell->next_cell;
          nxtcell->next_cell = newcell;
        }
        break;
    }
    nxtcell = newcell;
    //set up all the values correctly.
    nxtcell->timestamp = time;
    nxtcell->vc_num = head->vc_num;
    nxtcell->prop_time = head->prop_time;
    nxtcell->num_sws = head->num_sws;
    for (j=0; j < head->num_sws+2;j++)
      nxtcell->swt_nums[j] = head->swt_nums[j];
    if ((long)swcur->Ready >= head->num_cells)
    {
      nxtcell->num_cells = head->num_cells;
      nxtcell->last_cell = head->last_cell;
    }
    else
    {
      nxtcell->num_cells = (long)swcur->Ready;
```

```
      nxtcell->last_cell = 0;
    }
    if ((nxtcell->CI == 0) && (swcur->qued_cells > 9)) //congested?
      nxtcell->CI = 1; //used by PRCA
    else
      nxtcell->CI = 0;
    swvcnxt = swnxt->sw_vclist;
    while ((swvcnxt->vc_num != head->vc_num) && (swvcnxt != 0))
      swvcnxt = swvcnxt->next_vclist; //look for that VC info
    if (swvcnxt == 0) //internal error
    {
      printf("Error in vc number in sw %d\n",swnxt->sw_num);
      exit(1);
    }
    //update the number of cells received
    swvcnxt->cur_num_cells = swvcnxt->cur_num_cells+
                                                nxtcell->num_cells;
}
if ((long)swcur->Ready < head->num_cells)
{
  head->num_cells = head->num_cells - (long)swcur->Ready;
  swcur->Ready = swcur->Ready - (long)swcur->Ready;
}
else
{
  if (head->last_cell == 1) //last cell to be transmitted ?
  {
    swvcnxt = swcur->sw_vclist;
    swvcprv = swcur->sw_vclist;
    while ((swvcnxt != 0) && (swvcnxt->vc_num != head->vc_num))
    {
      swvcprv = swvcnxt;
      swvcnxt = swvcnxt->next_vclist;
    }
    if (swvcnxt == 0) //find the VC
    {
      printf("Error in vc numbers.\n");
      exit(1);
    }
    //release bandwidth
    swcur->UB = swcur->UB - swvcnxt->allowed_cur_rate;
```

```
    if (swcur->UB < 0)
      swcur->UB = 0;
    //connect the list and remove this VC
    if (swvcnxt == swcur->sw_vclist)
    {
      if (swvcnxt->next_vclist != 0)
        swcur->sw_vclist = swcur->sw_vclist->next_vclist;
    }
    else
    {
      if (swvcnxt->next_vclist != 0)
        swvcprv->next_vclist = swvcnxt->next_vclist;
      else
        swvcprv->next_vclist = 0;
    }
    if ((swvcnxt == swcur->sw_vclist) &&
      (swvcnxt == swvcprv) &&
      (swvcnxt->next_vclist == 0))
      swcur->sw_vclist = 0;
    else
    {
      swvcnxt->next_vclist = 0;
      delete(swvcnxt); // free the memory
    }
  }
  //update the available bandwidth
  swcur->Ready = swcur->Ready - (double)head->num_cells;

  head->timestamp = 0; //clear all values
  for (j=0; j < head->num_sws+2;j++)
    head->swt_nums[j] = 0;
  head->num_sws = 0;
  head->vc_num = 0;
  head->last_cell = 0;
  head->CI = 0;
  head->num_cells = 0;
  head->prop_time = 0;
  if (head->next_cell == 0) //remove this cell from the current
  {                         //switch
    delete(head);
    head = 0;
```

```
    switch (type)
    {
      case 1:
        swcur->Chead = 0;
        break;
      case 2:
        swcur->Rhead = 0;
        break;
      case 3:
        swcur->Nhead = 0;
        break;
      case 4:
        swcur->Ahead = 0;
        break;
    }
  }
  else
  {
    cur = head;
    switch (type) // update the list
    {
      case 1:
        swcur->Chead = head->next_cell;
        head = swcur->Chead;
        break;
      case 2:
        swcur->Rhead = head->next_cell;
        head = swcur->Rhead;
        break;
      case 3:
        swcur->Nhead = head->next_cell;
        head = swcur->Nhead;
        break;
      case 4:
        swcur->Ahead = head->next_cell;
        head = swcur->Ahead;
        break;
    }
    cur->next_cell = 0;
    delete(cur);
  }
```

```
      }
    }
  return(0); //successfully transmitted all cells
}

/****
Description: This routine logs the number of cells in each queue at
             runtime.
****/
void swtch::QCount(struct sw_node *tmp1,FILE *qcount1,
                   FILE *qcount2,double time)
{
  struct cells   *ctmp;
  struct RMcells   *rmcell;
  long   cbr,rtvbr,nrtvbr,abr,brm,frm;
  long   count;
  long   total;

    cbr=rtvbr=nrtvbr=abr=brm=frm=0;
    total = 0;
    rmcell = tmp1->BRMhead;
    while (rmcell != 0) // scan throught the queue and find the total
    {              // number of backward RM cells
      brm++;
      rmcell = rmcell->next_cell;
    }
    printf("BRM Q: %ld\n",brm);
    rmcell = tmp1->FRMhead;
    while (rmcell != 0) // scan throught the queue and find the total
    {              // number of forward RM cells
      frm++;
      rmcell = rmcell->next_cell;
    }
    printf("FRM Q: %ld\n",frm);

    ctmp = tmp1->Chead;
    while (ctmp != 0) // scan throught the queue and find the total
    {            //number of CBR cells
      cbr = cbr + ctmp->num_cells;
      ctmp = ctmp->next_cell;
    }
```

```
 printf("CBR Q: %ld\n",cbr);

 ctmp = tmp1->Rhead;
 while (ctmp != 0) // scan throught the queue and find the total
 {              // number of RTVBR cells
   rtvbr = rtvbr + ctmp->num_cells;
   ctmp = ctmp->next_cell;
 }
 printf("RTVBR Q: %ld\n",rtvbr);

 ctmp = tmp1->Nhead;
 while (ctmp != 0) // scan throught the queue and find the total
 {              // number of NRTVBR cells
   nrtvbr = nrtvbr + ctmp->num_cells;
   ctmp = ctmp->next_cell;
 }
 printf("NRTVBR Q: %ld\n",nrtvbr);

 ctmp = tmp1->Ahead;
 while (ctmp != 0) //  scan throught the queue and find the total
 {                 //  number of ABR cells
   if (ctmp->timestamp != 0)
     abr = abr + ctmp->num_cells;
   ctmp = ctmp->next_cell;
 }
 printf("ABR Q: %ld\n",abr);

 total = cbr+rtvbr+nrtvbr+abr; // find the total number of data cells
 count = total+brm+frm; // total count of all cells , data & RM

 printf("SW_NUM: %d TOTAL CELLS: %ld\n",tmp1->sw_num,total);
 if (total > 5)
 { //log to a file
   if (tmp1->sw_num == 100)
     fprintf(qcount1,"%lf %ld %ld %ld %ld %ld %ld %ld\n",
             time,brm,cbr,rtvbr,nrtvbr,abr,frm,total,count);
   else
     fprintf(qcount2,"%lf %ld %ld %ld %ld %ld %ld %ld\n",
             time,brm,cbr,rtvbr,nrtvbr,abr,frm,total,count);
 }
 tmp1->qued_cells = total;
```

```
  return; // scanned and logged the number of cells in queue
}

/****
Description: This routine check if the input and the output loads are equal
             and according to the algorithm used it may calculate the fair
             share and notify ABR sources.
****/
int swtch::chk_sw_load(struct sw_node *swnxt,struct vc_node *vhead,
                       double time,int ACR)
{
  struct vc_info *swvcnxt;
  int ret=0;

    swnxt->UB = 0;
    swvcnxt = swnxt->sw_vclist;
    while (swvcnxt != 0) //Check each VC on the switch
    {                    //and accordingly find the used rate
      if (swvcnxt->type == 4)
        swnxt->UB = swnxt->UB + swvcnxt->allowed_cur_rate;
      else
        swnxt->UB = swnxt->UB + swvcnxt->used_cur_rate;
      swvcnxt = swvcnxt->next_vclist;
    }

    if (swnxt->UB != swnxt->TB) //if used rate is not equal to total rate
    {
      if (ACR == 0) // execute EARA
        ret = EARA(swnxt,vhead,time);
    }

    if (ACR == 2) // execute ERICA
      ret = ERICA(swnxt);

    if (ret != 0)
      return(ret);

  // successfully monitored the load and executed the switch algo.
  return(0);
}
```

```
/****
Description: This routine monitors the rate of each VC. This is used for
             simulation verification only.
****/
int swtch::monitor_rate(struct sw_node *swnxt,int ACR)
{

  struct vc_info *swvcnxt;
  long bits;

  swvcnxt = swnxt->sw_vclist;
  while (swvcnxt != 0)
  {
   if (swvcnxt->timer == swvcnxt->interval) // Is it time to update ?
   {
    if (swvcnxt->type == 4)
    {
     if (swvcnxt->cur_num_cells != 0)
     { // find out the used rate
       swvcnxt->used_cur_rate = (swvcnxt->cur_num_cells*424000)/
                                                    swvcnxt->interval;
       if (ACR != 2)
         swvcnxt->cur_num_cells = 0;
     }
    }
    else
    { //find the used rate
     swvcnxt->used_cur_rate = (swvcnxt->cur_num_cells*424000)/
                                                    swvcnxt->interval;
     swvcnxt->cur_num_cells = 0;
    }
    swvcnxt->timer = 0; //reset timer
   }
   else
    swvcnxt->timer++; // or update it

    swvcnxt = swvcnxt->next_vclist; // go to next VC
 }
 return(0); //succefully updated the used rates
}
```

```
/****
Description: This routine switches the RM cells send in the backward
             birection queue.
****/
int swtch::switch_BRMcells(struct sw_node *swcur,struct vc_node *vhead,
                           double time,int ACR)
{
  struct RMcells *rmcur,rmnxt,*cur_cell;
  struct sw_node *swnxt;
  struct vc_node *vccur;
  struct ER_Data *tmpERdat,*newERdat;
  int i,j,prv_sw_num,nxt_sw_num,ret,k,l;
  struct vc_info *swvccur;
  long N_TCR=0;
  double tmpLd_Adj_Fctr=0;

  if (swcur->Ready_time <= time)
  {
    if ((swcur->BRMhead != 0) ||
       (swcur->Chead != 0) ||
       (swcur->Rhead != 0) ||
       (swcur->Nhead != 0) ||
       (swcur->Ahead != 0) ||
       (swcur->FRMhead !=0))
         swcur->Ready = swcur->Ready + swcur->Cellspermicsec; //update
    else
      swcur->Ready = 0;
  }
  swcur->Ready_time = time;
  ret = 0;
  rmcur = swcur->BRMhead;
  while ((rmcur != 0) && (swcur->Ready >= 1))
  {
    k = 0;
    swvccur = swcur->sw_vclist;
    // find the VC that number for which an RM cell is received
    while ((swvccur != 0) && (swvccur->vc_num != rmcur->vc_num))
      swvccur = swvccur->next_vclist;
    if (swvccur == 0) // cannot find - probably completed transmission
    {                 // clear all values and drop it
      rmcur->timestamp=0;
```

```
    for (j=0;j<rmcur->num_sws+2;j++)
      rmcur->swt_nums[j] = 0;
    rmcur->num_sws = 0;
    rmcur->cur_rate = 0;
    rmcur->prop_time = 0;
    rmcur->ER_rate = 0;
    rmcur->Ld_Adj_Fctr = 0;
    rmcur->vc_num=0;
    rmcur->DIR = 0;
    rmcur->CI = 0;
    cur_cell = rmcur;
    rmcur = rmcur->next_cell;
    swcur->BRMhead = rmcur;
    cur_cell->next_cell = 0;
    delete(cur_cell);
  }
  else // found the VC
  {
   for (i=0;i<rmcur->num_sws+2;i++) // find the switch on the path
   {
     if (rmcur->swt_nums[i] == swcur->sw_num)
       break;
   }

   if (rmcur->swt_nums[i] != swcur->sw_num) //internal error
   {
     printf("Error in switch numbers. - switch_RMCells \n");
     exit(1);
   }

   if (rmcur->DIR == 0) // if going to destination
   {
      nxt_sw_num = rmcur->swt_nums[i+1];
      prv_sw_num = rmcur->swt_nums[i-1];
   }
   else  //going to source
   {
      nxt_sw_num = rmcur->swt_nums[i-1];
      prv_sw_num = rmcur->swt_nums[i+1];
   }
```

```
if (ACR == 2)  //ERICA
{ //check the load.
  if ((swcur->LL >= LLB) && (swcur->LL <= ULB))
  {
     if (rmcur->cur_rate > swcur->FS)
       tmpLd_Adj_Fctr = swcur->LL/LLB;
     else
       tmpLd_Adj_Fctr = swcur->LL/ULB;
  }
  else
    tmpLd_Adj_Fctr = swcur->LL;

  if (tmpLd_Adj_Fctr > rmcur->Ld_Adj_Fctr)
    rmcur->Ld_Adj_Fctr = tmpLd_Adj_Fctr;

  //backward indication - input more than output
  if (tmpLd_Adj_Fctr > 1)
  {
     if (rmcur->DIR == 0)
     { //send a copy to source
        rmnxt.timestamp = rmcur->timestamp;
        rmnxt.vc_num = rmcur->vc_num;
        rmnxt.num_sws = rmcur->num_sws;
        for (j=0;j<rmcur->num_sws+2;j++)
          rmnxt.swt_nums[j] = rmcur->swt_nums[j];
        if (prv_sw_num < 100)
          rmnxt.prop_time = rmcur->prop_time+SRC_SWT;
        else
          rmnxt.prop_time = rmcur->prop_time+SWT_SWT;
        rmnxt.cur_rate = rmcur->cur_rate;
        rmnxt.ER_rate = rmcur->ER_rate;
        rmnxt.Ld_Adj_Fctr = rmcur->Ld_Adj_Fctr;
        rmnxt.DIR = 1;
        rmnxt.CI = rmcur->CI;
        if (prv_sw_num < 100)
          ret = Add_RM(&rmnxt,swcur->sw_num);
        else
          ret = Add_RM(&rmnxt,prv_sw_num);
        if (ret != 0)
          exit(1);
     }
```

```
  }
}
if ((ACR == 0) &&
   (swvccur->allowed_cur_rate >= rmcur->ER_rate) ||
   (ACR !=0))
{
  if ((nxt_sw_num < 100) && (rmcur->DIR == 1))
  {
    //find vc & update it
    vccur = vhead;
    while ((vccur != 0) && (vccur->vc_num != rmcur->vc_num))
      vccur = vccur->next_vc;
    if (vccur->LstRMtime <= rmcur->timestamp)
    {
      newERdat = new(ER_Data); //create a new cell
      if (newERdat == 0)  //internal error
      {
        printf("Out of Memory- ER data\n");
        exit(1);
      }
      newERdat->wait_time = rmcur->prop_time + SRC_SWT;
      if (ACR == 2) //ERICA
      {
        if (rmcur->Ld_Adj_Fctr == 0)
          rmcur->Ld_Adj_Fctr = 0.01;
        N_TCR =(long)((double)rmcur->cur_rate/
                                  rmcur->Ld_Adj_Fctr);
        if (rmcur->Ld_Adj_Fctr >= 1)
        {    //set ER according to load level
          if (N_TCR < rmcur->ER_rate)
            rmcur->ER_rate = N_TCR;
        }
        else
        {
          if (N_TCR > rmcur->ER_rate)
            rmcur->ER_rate = N_TCR;
        }
        if (rmcur->ER_rate > vccur->peak_rate)
          swvccur->allowed_cur_rate = vccur->peak_rate;
        else
          swvccur->allowed_cur_rate = rmcur->ER_rate;
```

```
        }
        newERdat->ER_rate = rmcur->ER_rate;
        if (ACR == 1 && rmcur->CI == 1)    //PRCA
          newERdat->ER_rate = vccur->cur_rate -
                                        (long)(vccur->cur_rate/16);
        if (ACR == 1)
        {
          if (rmcur->ER_rate > vccur->peak_rate)
            swvccur->allowed_cur_rate = vccur->peak_rate;
          else
            swvccur->allowed_cur_rate = rmcur->ER_rate;
        }
        newERdat->ER_Chng = rmcur->timestamp;
        newERdat->next = 0;

        if (vccur->ERdat == 0)  //insert into queue
          vccur->ERdat = newERdat;
        else
        {
          tmpERdat = vccur->ERdat;
          while (tmpERdat->next != 0)
            tmpERdat = tmpERdat->next;
          tmpERdat->next = newERdat;
        }
        vccur->LstRMtime = rmcur->timestamp;
      }
    }
    else
    {
      if ((nxt_sw_num < 100) && (rmcur->DIR == 0))
      {  //set progation time
        rmcur->prop_time = rmcur->prop_time+SRC_SWT;
        rmcur->DIR = 1;
        vccur = vhead;
        while ((vccur != 0) && (vccur->vc_num != rmcur->vc_num))
          vccur = vccur->next_vc;
        if (vccur->CI == 1)
          rmcur->CI = 1; //update congestion bit for PRCA
        else
          rmcur->CI = 0;
        k = 1;
```

```
        }
        if (k == 0)
        {  //get all the parameters and send to next node
          rmnxt.timestamp = rmcur->timestamp;
          rmnxt.vc_num = rmcur->vc_num;
          rmnxt.num_sws = rmcur->num_sws;
          for (j=0;j<rmcur->num_sws+2;j++)
            rmnxt.swt_nums[j] = rmcur->swt_nums[j];
          if (prv_sw_num < 100)
            rmnxt.prop_time = rmcur->prop_time+SRC_SWT;
          else
            rmnxt.prop_time = rmcur->prop_time+SWT_SWT;
          rmnxt.cur_rate = rmcur->cur_rate;
          rmnxt.ER_rate = rmcur->ER_rate;
          rmnxt.Ld_Adj_Fctr = rmcur->Ld_Adj_Fctr;
          rmnxt.DIR = rmcur->DIR;
          rmnxt.CI = rmcur->CI;
          if (ACR == 0)
            ret = Add_BRMcell(&rmnxt,nxt_sw_num);
          else
            ret = Add_RM(&rmnxt,nxt_sw_num);
          if (ret != 0)
            exit(1);
        }
      }
      swcur->Ready = swcur->Ready - 1;
    }
    if (k == 0) //clear all values and delete it
    {
      rmcur->timestamp=0;
      for (j=0;j<rmcur->num_sws+2;j++)
        rmcur->swt_nums[j] = 0;
      rmcur->num_sws = 0;
      rmcur->cur_rate = 0;
      rmcur->prop_time = 0;
      rmcur->ER_rate = 0;
      rmcur->Ld_Adj_Fctr = 0;
      rmcur->vc_num=0;
      rmcur->DIR = 0;
      rmcur->CI = 0;
      if (rmcur->next_cell == 0)
```

```
            {
              delete(rmcur);
              swcur->BRMhead = 0;
            }
            else
            { //remove the cell and connect the list
              cur_cell = rmcur;
              swcur->BRMhead = rmcur->next_cell;
              cur_cell->next_cell = 0;
              delete(cur_cell);
            }
            rmcur = swcur->BRMhead;
          }
        }
      }
   return(0); //successfully swaped backward RM cells
}

/****
Description: This routine switches the RM cells send in the forward
             direction queue.
****/
int swtch::switch_FRMcells(struct sw_node *swcur,struct vc_node *vhead,
                           double time,int ACR)
{
  struct RMcells *rmcur,rmnxt,*cur_cell;
  struct sw_node *swnxt;
  struct vc_node *vccur;
  struct ER_Data *tmpERdat,*newERdat;
  int i,j,prv_sw_num,nxt_sw_num,ret,k,l;
  struct vc_info *swvccur;

  ret = 0;
  rmcur = swcur->FRMhead;
  while ((rmcur != 0) && (swcur->Ready >= 1))
  {
    k = 0;
    swvccur = swcur->sw_vclist;
    while ((swvccur != 0) && (swvccur->vc_num != rmcur->vc_num))
      swvccur = swvccur->next_vclist;
```

```
if (swvccur == 0) // cannot find - probably completed transmission
{                 // clear all values and drop it
  rmcur->timestamp=0;
  for (j=0;j<rmcur->num_sws+2;j++)
    rmcur->swt_nums[j] = 0;
  rmcur->num_sws = 0;
  rmcur->cur_rate = 0;
  rmcur->prop_time = 0;
  rmcur->ER_rate = 0;
  rmcur->Ld_Adj_Fctr = 0;
  rmcur->vc_num=0;
  rmcur->DIR = 0;
  rmcur->CI = 0;
  cur_cell = rmcur;
  rmcur = rmcur->next_cell;
  swcur->FRMhead = rmcur;
  cur_cell->next_cell = 0;
  delete(cur_cell);
}
else // found the VC
{
  for (i=0;i<rmcur->num_sws+2;i++) // look for the switch number
  {
    if (rmcur->swt_nums[i] == swcur->sw_num)
      break;
  }
  if (rmcur->swt_nums[i] != swcur->sw_num) // internal error
  {
    printf("Error in switch numbers. - switch_RMCells \n");
    exit(1);
  }
  if (rmcur->DIR == 0) // going to destination
  {
    nxt_sw_num = rmcur->swt_nums[i+1];
    prv_sw_num = rmcur->swt_nums[i-1];
  }
  else  // going to source
  {
    nxt_sw_num = rmcur->swt_nums[i-1];
    prv_sw_num = rmcur->swt_nums[i+1];
  }
```

```
if ((ACR == 0) && (swvccur->allowed_cur_rate >= rmcur->ER_rate) ||
   (ACR !=0))
{
  if ((nxt_sw_num < 100) && (rmcur->DIR == 1))
  {
    //find vc & update it
    vccur = vhead;
    while ((vccur != 0) && (vccur->vc_num != rmcur->vc_num))
      vccur = vccur->next_vc;
    if (vccur->LstRMtime < rmcur->timestamp)
    {
      newERdat = new(ER_Data);
      if (newERdat == 0)
      {
        printf("Out of Memory- ER data\n");
        exit(1);
      }
      //update propagation time
      newERdat->wait_time = rmcur->prop_time + SRC_SWT;
      newERdat->ER_rate = rmcur->ER_rate;
      if (ACR == 1 && rmcur->CI == 1) //PRCA
        newERdat->ER_rate = vccur->cur_rate -
                                    (long)(vccur->cur_rate/16);
      newERdat->ER_Chng = time;
      newERdat->next = 0;

      if (vccur->ERdat == 0) // add to list
        vccur->ERdat = newERdat;
      else
      {
        tmpERdat = vccur->ERdat;
        while (tmpERdat->next != 0)
          tmpERdat = tmpERdat->next;
        tmpERdat->next = newERdat;
      }

      vccur->LstRMtime = rmcur->timestamp;
    }
}
else
```

```
    {
      if ((nxt_sw_num < 100) && (rmcur->DIR == 0))
      { //update propagation time
        rmcur->prop_time = rmcur->prop_time+SRC_SWT;
        rmcur->DIR = 1;
        vccur = vhead;
        while ((vccur != 0) && (vccur->vc_num != rmcur->vc_num))
          vccur = vccur->next_vc;
        if (vccur->CI == 1) // congested?
          rmcur->CI = 1;
        else
          rmcur->CI = 0;
        k = 1;
      }
      if (k == 0)
      { // set all values and send it to next node
        rmnxt.timestamp = rmcur->timestamp;
        rmnxt.vc_num = rmcur->vc_num;
        rmnxt.num_sws = rmcur->num_sws;
        for (j=0;j<rmcur->num_sws+2;j++)
          rmnxt.swt_nums[j] = rmcur->swt_nums[j];
        if (prv_sw_num < 100)
          rmnxt.prop_time = rmcur->prop_time+SRC_SWT;
        else
          rmnxt.prop_time = rmcur->prop_time+SWT_SWT;
        rmnxt.cur_rate = rmcur->cur_rate;
        rmnxt.ER_rate = rmcur->ER_rate;
        rmnxt.Ld_Adj_Fctr = rmcur->Ld_Adj_Fctr;
        rmnxt.DIR = rmcur->DIR;
        rmnxt.CI = rmcur->CI;
        ret = Add_FRMcell(&rmnxt,nxt_sw_num);
        if (ret != 0)
          exit(1);
      }
    }
    swcur->Ready = swcur->Ready - 1;
  }
  if (k == 0)
  { //clear all values and delete from memory
    rmcur->timestamp=0;
```

```
            for (j=0;j<rmcur->num_sws+2;j++)
              rmcur->swt_nums[j] = 0;
            rmcur->num_sws = 0;
            rmcur->cur_rate = 0;
            rmcur->prop_time = 0;
            rmcur->ER_rate = 0;
            rmcur->Ld_Adj_Fctr = 0;
            rmcur->vc_num=0;
            rmcur->DIR = 0;
            rmcur->CI = 0;
            if (rmcur->next_cell == 0)
            {
              delete(rmcur);
              swcur->FRMhead = 0;
            }
            else
            {
              cur_cell = rmcur;
              swcur->FRMhead = rmcur->next_cell;
              cur_cell->next_cell = 0;
              delete(cur_cell);
            }
            rmcur = swcur->FRMhead;
          }
        }
      }
      return(0); // successfully switched the cell to the next node1
}

/****
Description: This routine logs the bandwidth usage
****/
void swtch::log_BW(struct sw_node *swnxt,double time,FILE *bwidth1,
                   FILE *bwidth2,int ACR)
{
  struct vc_info *swvcnxt;

    if (ACR != 0)
    {
      swnxt->UB = 0;
      swvcnxt = swnxt->sw_vclist;
```

```
        while (swvcnxt != 0) // find out the total used rate
        {
          if (swvcnxt->type == 4)
            swnxt->UB = swnxt->UB + swvcnxt->allowed_cur_rate;
          else
          swnxt->UB = swnxt->UB + swvcnxt->used_cur_rate;
          swvcnxt = swvcnxt->next_vclist;
        }
      }
      if (swnxt->sw_num == 100) // log to file
        fprintf(bwidth1,"%lf %ld %ld %ld\n",
              time,swnxt->AB,swnxt->TB,swnxt->UB);
      else
        fprintf(bwidth2,"%lf %ld %ld %ld\n",
            time,swnxt->AB,swnxt->TB,swnxt->UB);

  return; //completed log
}

/****
Description: This routine fluses all the forward RM cells if there are cells
             in the backward RM cell for that VC as the values in forward
             RM cells is no longer valid.
****/
void swtch::FlushFRM(struct sw_node *swcur)
{
  struct RMcells *rmprv,*rmcur,*rmtmp;
  struct RMcells *bcur;

  if (swcur->BRMhead != 0)
  {
    bcur = swcur->BRMhead;
    while (bcur != 0)  //if any backward RM cells
    {
      if (swcur->FRMhead != 0) // if any forward RM cells
      {
        rmcur = swcur->FRMhead;
        rmprv = swcur->FRMhead;
        while (rmcur != 0) // find if there is a forward RM cell
        {                  // of that VC which has a bacward RM cell
          if (rmcur->vc_num == bcur->vc_num)
```

```
            {
              if (rmprv == rmcur) // remove it from forward queue
              {
                  rmtmp = swcur->FRMhead;
                if (swcur->FRMhead->next_cell == 0)
                  swcur->FRMhead = 0;
                else
                  swcur->FRMhead = swcur->FRMhead->next_cell;
                rmcur = rmprv = swcur->FRMhead;
              }
              else
              {
                rmtmp = rmcur;
                rmprv->next_cell = rmcur->next_cell;
                rmcur = rmprv->next_cell;
              }
              delete(rmtmp);  // flush it
            }
            else
            { // reconnect the queue
              rmprv = rmcur;
              rmcur = rmcur->next_cell;
            }
          }
        }
        bcur = bcur->next_cell; // go to next RM cell
      }
    }
    return; //successfully flushed all cells
}

/****
Description: This routine is the main routine of the switch. It calls the
             appropriate routines and executes the required algorithms.
****/
int swtch::swt_all(FILE *qcount1,FILE *qcount2,FILE *bwidth1,
                   FILE *bwidth2, vc_node *vhead,double time,int acr)
{
  sw_node *swcur;
  int ret1;
  double result;
```

```
swcur = sw_head;
while (swcur != 0)
{
  ret1 = 0;
  FlushFRM(swcur); // flush Rm cells if required
  // log the number of cells in queue
  QCount(swcur,qcount1,qcount2,time);
  // switch the backward RM
  ret1 = switch_BRMcells(swcur,vhead,time,acr);
  if (ret1 != 0) //cells
  { // internal error
    printf("Error in Switch_RMcells.\n");
    break;
  }
  // transmit cells to next node
  ret1 = switch_cells(swcur,time,vhead,acr);
  if (ret1 != 0) // internal error1
  {
    printf("Error in Switch_cells.\n");
    break;
  }
  // switch the forward RM
  ret1 = switch_FRMcells(swcur,vhead,time,acr);
  if (ret1 != 0)  // cell
  { //internal error
    printf("Error in Switch_RMcells.\n");
    break;
  }
  ret1 = monitor_rate(swcur,acr); // monitor the rate used by each VC
  if (ret1 != 0) //internal error
  {
    printf("Error in Update SW Rates.\n");
    break;
  }
  if ((acr == 0)|| (acr == 2)) //EARA and ERICA
  {
    if (swcur->timer >= swcur->interval) // if the timer expired
    {
      if (acr == 0)
        swcur->rem_cells = swcur->qued_cells;
```

```
        ret1 = chk_sw_load(swcur,vhead,time,acr); // check the load
        if (ret1 != 0) //internal error
        {
          printf("Error in Check SW Load.\n");
          break;
        }
        swcur->timer = 0; //reset timer
      }
      else
        swcur->timer++; // increment timer count
    }

 result = fmod(time,1000.0);
 if (result == 0) //log the bandwidth usage
    log_BW(swcur,time,bwidth1,bwidth2,acr);
    swcur = swcur->next_sw;
 }
return(0); //successfully completed the switch algorithm
}
/************************** end of switch.cpp ******************************/
```