ENSC 833-3 : NETWORK PROTOCOLS AND PERFORMANCE

ATM Traffic Control Based on Cell Loss Priority and Performance Analysis

Spring 2001/4/13

Final Report

Shim, Heung-Sub

shim_hs@hanmail.net

Contents

Index of figures and tables

1. Abstract

ATM, an ultimate solution of B-ISDN to provide integrated multimedia services including voice, video, and data, has entered into the limelight with increased demand

for such services. Hence, ATM is to be capable of supporting a variety of service classes and providing appropriate QoS according to classes. This may force us to sacrifice low priority traffic classes for high priority traffic classes to satisfy QoS requirements for the high priority traffic classes in case of congestion. There have been many possibilities suggested for traffic control in terms of QoS and 'cell loss priority (CLP) control', which was originally introduced in ATM networks for the purpose of congestion control, must be one of them. The capability of CLP control can apply to several buffer priority schemes: push-out, partial buffer sharing, buffer separation, hybrid schemes and so on. These are all priority queuing disciplines that are used to secure the cell loss ratio (CLR) of higher priority cells at the cost of loss of low priority cells by examining the CLP bit of each incoming cell set to '0' or '1' and will be discussed later in full detail. Priority queuing is especially appropriate in cases where WAN (Wide Area Network) links are congested from time to time. Therefore, if the WAN links are never congested, such mechanism is unnecessary. Because priority queuing requires extra processing and can cause performance degradations for low priority traffic, it should not be recommended unless necessary.

2. Introduction

A. Overview of relevant technologies

1) ATM cell structure

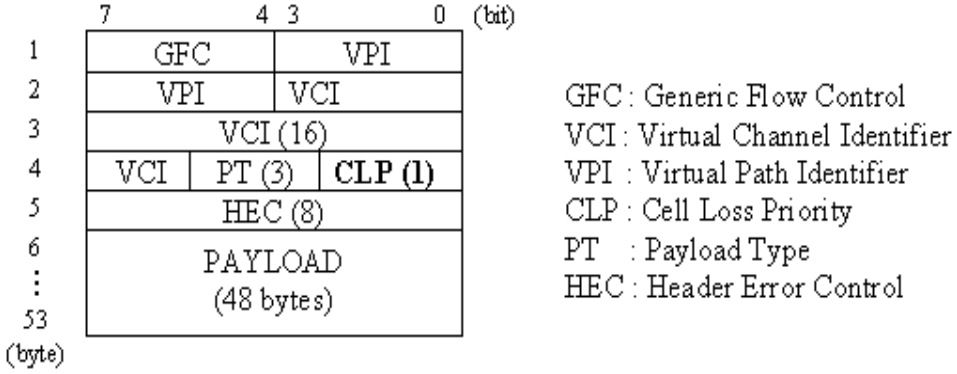


Figure 1. ATM cell header for UNI (User-Network Interface)

The above figure shows the header structure of ATM cell for UNI. On the other hand, the header of ATM cell for NNI (Node-Network Interface) does not include the GFC field, which contains the information of flow control mechanism to be used to prevent cells from colliding among terminals by supporting multiple access to the network.

2) Schemes to be implemented

An application can offer two types of traffic streams to the ATM networks. The CLP bit in the header of the ATM cell may be used to declare two levels of QoS. This CLP based cell loss priority control can be implemented by a variety of schemes. However, I will focus on three major methods: push-out, partial buffer sharing and buffer separation.

i) Push-out

In this scheme, cells of both high and low priorities share a common buffer. An incoming cell with CLP = 1 may be accepted irrespective of the queue length it sees. However, if the buffer is full and a high priority cell (CLP = 0) arrives, a low priority cell, which already resides in the buffer, will be pushed out and lost. A very important feature is that when the low priority cell gets pushed out, its place is taken by the next cell in the queue, and so on, until the last slot becomes available to the incoming high priority cell. The buffer management, however, should maintain the cell sequence integrity, requiring a complex mechanism.

ii) Partial buffer sharing

An incoming cell with CLP = 1 is dropped if the queue length it sees is greater than or equal to a specified threshold level which is less than the total buffer capacity. On the other hand, high priority cells can access the buffer unless it remains full. The threshold level can be adjusted for optimal load conditions adaptively. Otherwise, the incoming cells are served on a first-come-first-serve (FCFS) basis.

ii) Buffer separation

We may want to have two separate buffers, one of which is for high priority cells and the other is for low priority cells so as to make the cell loss ratio (CLR) of high priority cells as low as possible. The high priority queue is always emptied before the low priority queue is serviced. The cell sequence integrity can be maintained only if a single priority is assigned to each connection.

B. Scope of project

As described above, the cell loss priority queuing schemes to be implemented in the project are push-out, partial buffer sharing and buffer separation, each of which has its own pros and cons. These are what we wish to see through simulations. There are many parameters to be taken into account and complicated elements to be constructed in implementing such mechanisms. However, OPNET Modeler, a powerful network simulation tool, has made it far simpler and easier to design and simulate network models by providing a variety of element templates and allowing us to conveniently customize our model using them. Also, the simulator enables us to easily collect and visualize simulation results.

Since the most crucial thing is what is going to happen in the switch node during a simulation and the only difference among the three implementations is queuing strategy, their network topology will basically remain the same through all simulations but the switch node where the proper queuing method is to be implemented (actually, those queuing schemes are implemented in Process Model of OPNET Modeler). The topology to be used is as shown below.

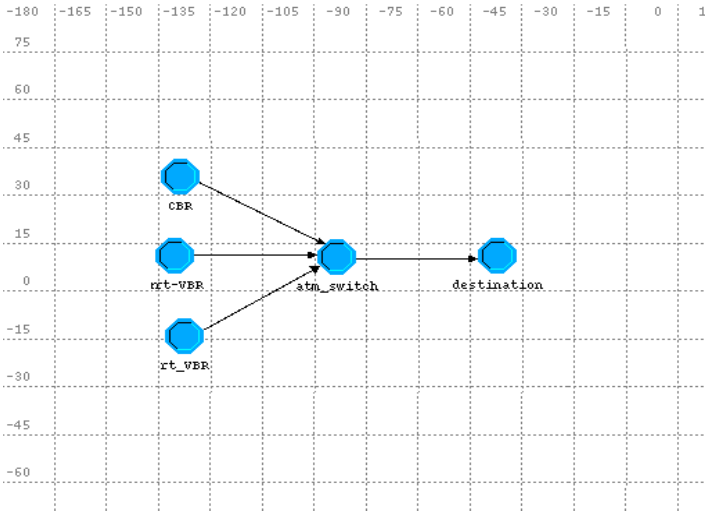


Figure 2. Project topology

There are a total of 5 nodes in the network. The first three nodes are source nodes that generate CBR (Constant Bit Rate), nrt-VBR (non-realtime Variable Bit Rate) and rt-VBR (realtime Variable Bit Rate) cells whose CLP bits will be set to '0', '1' and '0', respectively. Especially, the rt-VBR node will cause all cell losses by generating bursty traffic during the simulations. The middle one in the figure above is the ATM switch node which has the most interest of the project. Then, those cells accepted and served in the queue are sent to the destination node where they will sink. Of course, the most important parameter is the cell loss ratio (CLR) of the high priority cells (CBR and rt-VBR). Furthermore, how much we should sacrifice low priority cells in order to obtain the target CLR of high priority cells is another issue. For a fair comparison, the same physical conditions, that is, the same topology, the same cell generation rates, the same link and so on are to apply to the implementations. Also, the comparison will be made on the basis of a total buffer capacity of 100 cells (Assume we do not consider any abstractive factors such as interaction among cells during simulations. However, they may be discussed later in the paper).

3. Main Sections

As described in the previous section, the three source nodes, CBR, nrt-VBR and rt-VBR generate a constant bit traffic, a variable bit traffic and a bursty bit traffic, respectively. Specifically, the CBR source and the nrt-VBR source approximately generate cells at a fixed rate of 3537 cells/sec ($\approx$ 1.5Mbps = DS1) and at a mean rate of 3537 cells/sec (exponential distribution) while the bursty rt-VBR source generates at a mean rate of 2358 cells/sec for the off-duration and 7075 cells/sec for the on-duration until the simulation terminates since after it starts. The mean off-duration and on-duration of the bursty rt-VBR traffic are to be 0.125 seconds and 0.875 seconds, respectively. In the meanwhile, the ATM switch node processes incoming cells at a service rate of 10613 cells/sec ($\approx$ 4.5Mbps = 3 $\times$ DS1) so as to support three DS1 channels and drops any excessive cells. In order to satisfy the QoS requirements, the minimum average CLR of 0.75% and the maximum queuing delay of 10msec for the CBR traffic must be secured for the given traffic load, which will be fixed during simulations. Simulations each will

go on for 7 seconds and the traffic sources are to start generating cells immediately after the simulation begins. Incoming cells are dropped when the total cell generation rate exceeds the service rate of the switch. Assume that no cell losses occur due to the link capacity. The following subsections each include the implementations of queuing schemes and simulation results.

A. Push-out

This mechanism requires a relatively complex queue management. The simulation procedure is as shown below.

1) Diagram and flowchart

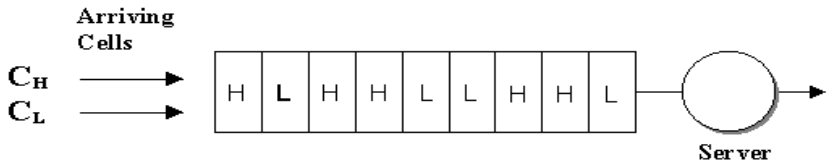


Figure 3. Diagram of push-out scheme

The letters 'H' and 'L' represent a high priority cell and a low priority cell, respectively. The last low priority cell (bold letter 'L'), which already resides in the queue, will be pushed out to be lost if the next incoming cell is a high priority cell while the status of the queue is as shown in the above figure. The following flowchart shows how the queuing scheme works if any incoming cell arrives in the queue.

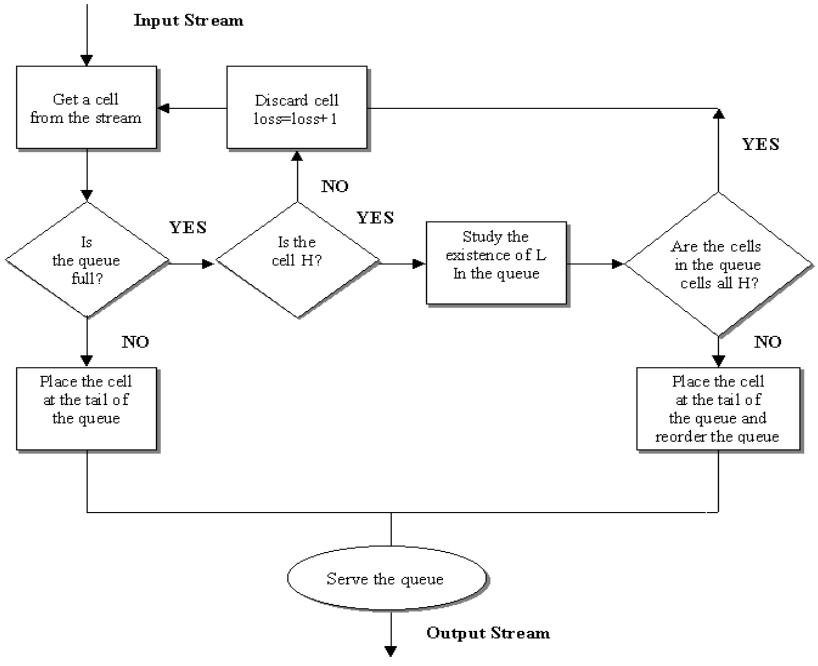
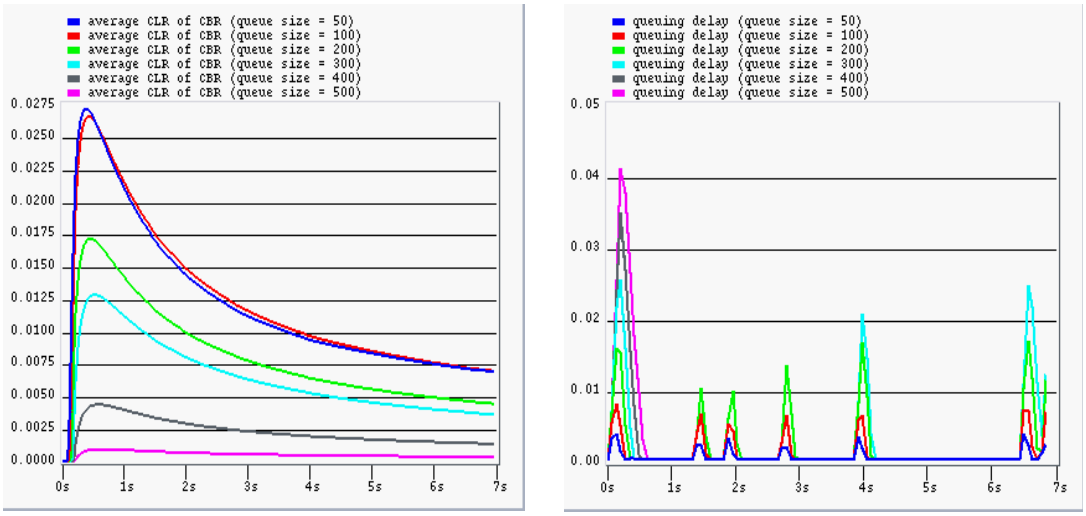


Figure 4. Flowchart of push-out algorithm

※ Refer to the Attachment for the simple test code of push-out mechanism.

2) Simulation

The following graphs show the average CLR and the queuing delay of the CBR traffic according to various buffer sizes.



(a) Average CLR (b) Queuing delay

Figure 5. Average CLR and queuing delay of CBR traffic by queue size

We can easily see that there is no significant benefit of having a 100-cell buffer instead of a 100-cell buffer with respect to the CLR of CBR traffic. However, we have to pay twice the queuing delay in case of the 50-cell buffer. Also, as the buffer size increases, the queuing delay proportionally increases, deteriorating the QoS of the CBR traffic in sense of delay. The above argument supports the fact that, in general, the ATM network does not require large buffers. The graphs (a) and (b) show that, given the amount of traffic load and QoS requirements, it is reasonable to have a 100-cell buffer in order for the buffer to accommodate more cells and, therefore, the choice of the 100-cell buffer will apply to the simulations performed hereinafter.

B. Partial buffer sharing

This has over the push-out scheme an advantage that not only its implementation is much simpler than the push-out scheme but also it is possible to implement it using a hardware device though its efficiency is inferior to the push-out mechanism.

1) Diagram and flowchart

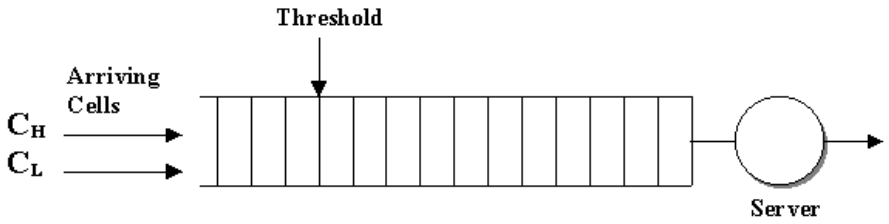


Figure 6. Diagram of partial buffer sharing scheme

The overall queuing operation is thoroughly dependent on the threshold and, therefore, it should be so properly chosen as to optimize the network performance. The partial buffer sharing algorithm governs the whole queue operation in case of congestion. Otherwise, the queue operates based on the FCFS. The following flowchart shows how the queuing scheme works if any incoming cell arrives in the queue.

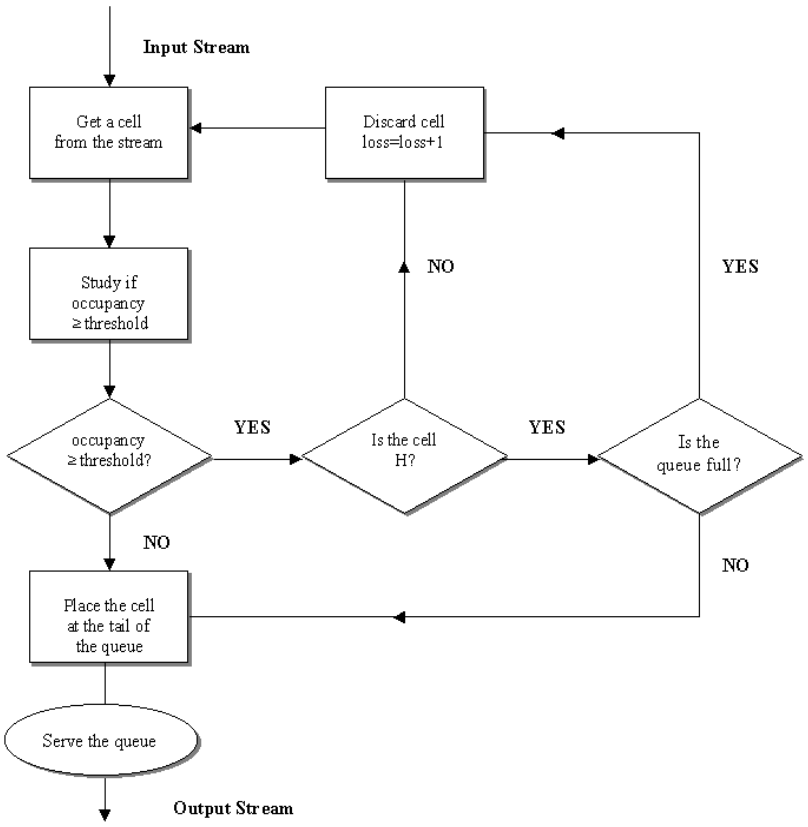
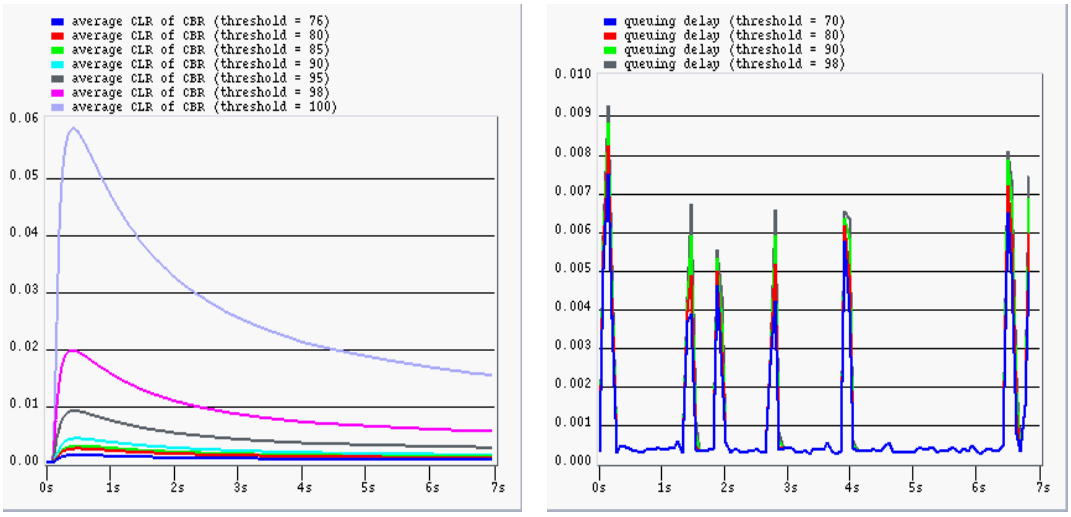


Figure 7. Flowchart of partial buffer sharing algorithm

※ Refer to the Attachment for the simple test code of partial buffer sharing mechanism.

2) Simulation

The following graphs show the average CLR and the queuing delay of the CBR traffic according to various thresholds. In this case, we need to have a closer look at how the queuing delay changes as the threshold varies. The result is comparable to that in the case of the buffer separation, which is discussed later in the report.



(a) Average CLR (b) Queuing delay

Figure 8. Average CLR and queuing delay of CBR traffic by threshold

As shown in the figure (a), CBR cells start being lost with the threshold equal to 76 and the threshold for the CBR traffic that satisfies the minimum required QoS is found to be very high (98 in this case). We can intuitively anticipate that, in order to prevent an excessive loss of the low priority traffic, the threshold should be as high as possible as long as the QoS requirements for the CBR traffic are secured. Seemingly, there would be no difference among queuing delays according to various thresholds once the queue size is determined, but it is an interesting result that the queuing delay varies with threshold though the variations are still slight. In terms of the CLR of the CBR traffic, the simulation result shows that the model performs as well as the push-out scheme in case of threshold 98 (98% of the whole buffer). More details related to their performance comparisons will be given in Discussion and Conclusions.

C. Buffer separation

This is a very simple priority queuing scheme that ensures important traffic is processed first. It was originally designed to give strict priority to a critical application, and is particularly useful for time-sensitive protocols.

1) Diagram and flowchart

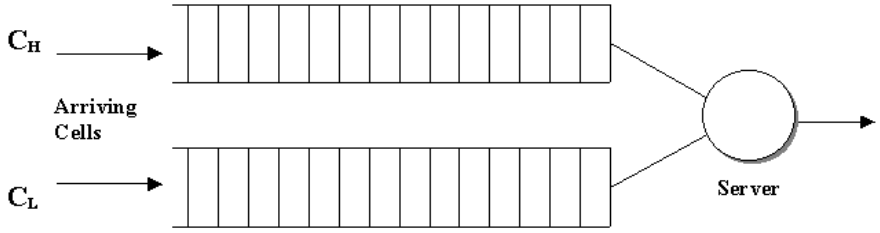


Figure 9. Diagram of buffer separation scheme

Different from the push-out or partial buffer sharing scheme where high and low priority cells share the same memory space, the buffer separation scheme allocates a

separate queue to each traffic class as shown above. The following flowchart shows how the queuing scheme works if any incoming cell arrives in the queue.

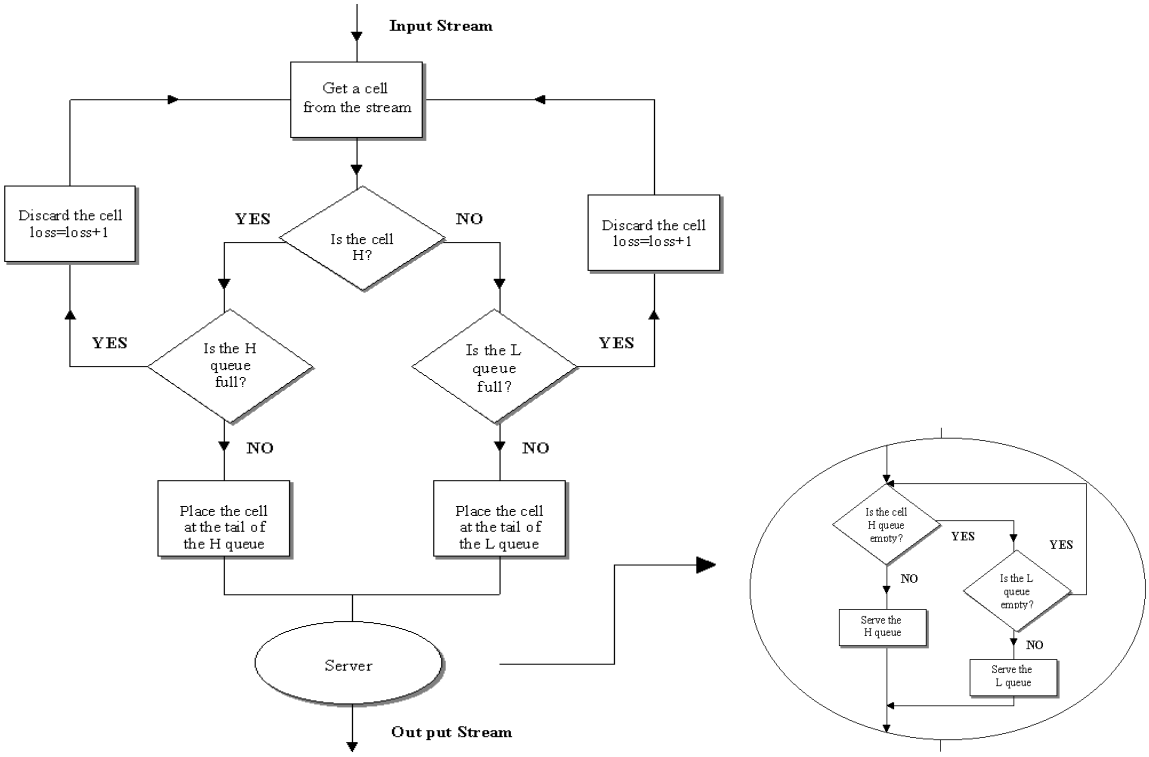
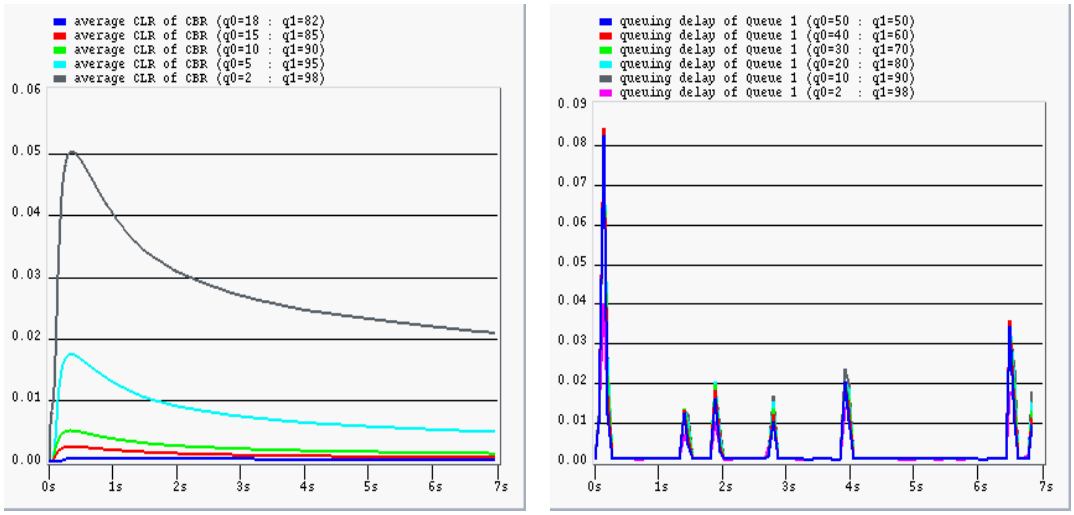


Figure 10. Flowchart of buffer separation algorithm

2) Simulation

The following graphs show the average CLR and the queuing delay of the CBR traffic according to various queue size ratios.



(a) Average CLR (b) Queuing delay

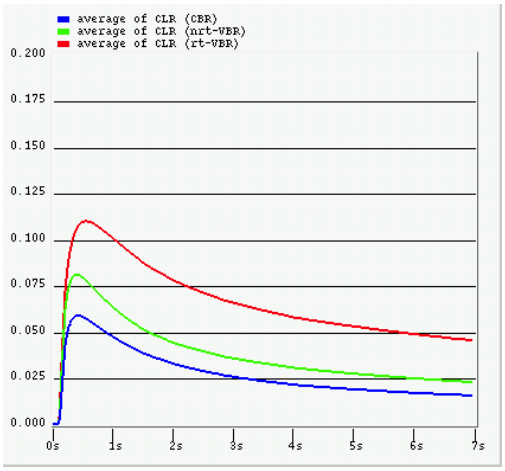
Figure 11. Average CLR and queuing delay of CBR traffic by queue size ratio

In this simulation, the queue size ratio between Queue 0 (high priority queue) and Queue 1 (low priority queue) plays a similar role to threshold in case of the partial buffer sharing scheme. The simulation result, however, contains quite different aspects. Given the amount of traffic load and the total queue size, the queuing delay of Queue 1 does not change a lot with respect to its queue size varying. Furthermore, we can say that the CLR behavior of this queuing scheme is biased against the low priority queue because the queue is never going to be served as long as there is or are always any cell or cells in the high priority queue so that a serious degradation in the QoS for the low priority traffic may occur. Nevertheless, the simulation results show that the partial buffer sharing scheme is most biased against the low priority traffic. This will be numerically discussed in the next section.

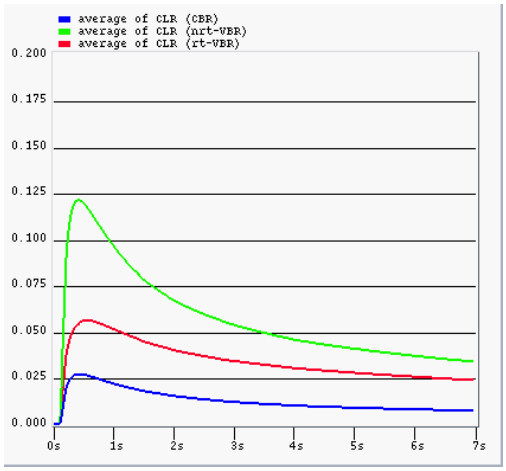
4. Discussion and Conclusions

The simulation environment was set up so that the intended load would be a total of 70151 cells during a simulation. Anyway, the actual load has turned out to be 67953 cells per simulation. This is because the bursty rt-VBR source did not generate as many cells as intended by the actual settings during a simulation. The followings discuss the comparison among the different priority queuing schemes by providing some graphs and numerical results.

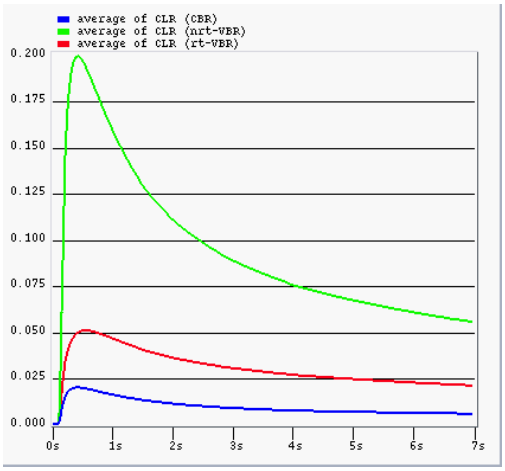
What is going to happen if there has been no priority queuing mechanism used in case of congestion? We may easily come up with the inference that the higher bit rate traffic will see more cells lost than the lower bit rate traffic while the queue overflows. This is what we can see in (a) of the figure below. The graph (a) represents the average CLR of each traffic source in case of no priority queuing scheme applied. The queue, in this case, operates on the basis of FCFS and any incoming cells arriving during a congestion are dropped no matter what their CLP bit may be set to. In the meanwhile, the simulation results agree to the fact that all cell loss priority queuing schemes are biased against the low priority traffic more or less and, at the same time, the CLR of the high priority traffic gets improved. This is exactly what the other graphs describe.



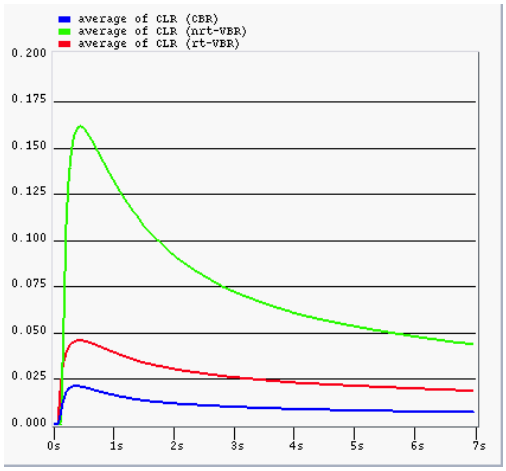
(a) No traffic control



(b) Push-out



(c) Partial buffer sharing (threshold = 98)



(d) Buffer separation (queue size ratio = 5 : 95)

Figure 12. Comparison among simulated queuing schemes

Given the amount of the traffic load and the conditions, the three priority queuing schemes have showed similar performances with respect to the CLR of the CBR source. However, as far as the overall cell loss is concerned, the push-out scheme outperforms the others, the partial buffer sharing and the buffer separation. The following table proves the result.

Traffic type / Scheme	CBR (CLR)	rt-VBR (CLR)	nrt-VBR (CLR)	Total (CLR)
No control	208 (0.0084)	502 (0.027)	279 (0.0114)	989 (0.0146)
Push-out	90 (0.0036)	242 (0.013)	435 (0.0177)	767 (0.0113)
Partial buffer sharing (threshold = 98)	86 (0.0035)	221 (0.0119)	690 (0.0281)	997 (0.0147)
Buffer separation (queue size ratio = 4 : 96)	93 (0.0038)	209 (0.0112)	482 (0.0196)	784 (0.0115)

Table 1. Cell loss comparison (queue size = 100, total load = 67953)

The above simulation results converge to the following three conclusions. Firstly, as expected, all the three implemented priority queuing schemes improve the CLR of the high priority traffic by sacrificing the low priority traffic. Secondly, some queuing schemes may bring an improvement in the CLR of the whole traffic (in the push-out mechanism and the buffer separation). Finally, queuing delay could vary with queuing schemes used though the queue size is fixed (especially in the partial buffer sharing scheme).

There have been several difficulties doing the project, such as time-consuming OPNET debugging process, determination of simulation scale for better comparison, clear understanding of relevant existing OPNET models required to create user-defined models or attributes and so on. Also, at the very beginning of my project, I planned to

establish a mathematical model for the simulated network and found it is much more difficult than I actually thought. So, I want to end up this report with introducing a useful paper for those who are interested in cell loss calculation in ATM networks, "Improved Loss Calculations at an ATM Multiplexer" written by Ness B. Shroff and Mischar Schwartz [8]. In the paper, they classify arrival processes into the Markov-modulated Poisson process (MMPP) and the Markov-modulated fluid (MMF) process, and apply those processes to a finite-buffer system. In other words, they develop a simple analytical technique to determine the probability of loss at a finite-buffer ATM multiplexer. Their loss calculation technique is shown to be valid for both heterogeneous and homogeneous sources.

5. References

[1] P.S. Neelakanta, *ATM Telecommunications*, CRC Press, 2000

[2] Dominique Gaiti and Guy Pujolle, "Performance Management Issues in ATM Networks: Traffic Congestion Control", IEEE/ACM Transactions on Networking, Vol. 4, No. 2, April 1996

[3] Sridhar Ramesh, Gatherine Rosenberg and Anurag Kumar, "Revenue Maximization in ATM Networks Using the CLP Capability and Buffer Priority Management", IEEE/ACM Transactions on Networking, Vol. 4, No. 6, December 1996

[4] Ness B. Shroff and Mischa Schwartz, "Improved Loss Calculations at an ATM Multiplexer", IEEE/ACM Transactions on Networking, Vol. 6, No. 4, August 1998

[5] Todd Lizambri, Fernando Duran and Shukri Wakid, "Priority Scheduling and Buffer Management for ATM Traffic Shaping".
http://w3.antd.nist.gov/Hsntg/publications/Papers/lizambri_1299.pdf

[6] Viet L. Do and Kenneth Y. Yun, "A Scalable Priority Queue Manager Architecture for Output-Buffered ATM Switches".
http://paradise.ucsd.edu/PAPERS/ICCCN-99-PQM.pdf

[7] Center For Telecommunication Networks, “Space Priority Algorithms”.
http://www.dur.ac.uk/~des0www3/space/space2.html

[8] CISCO, “Optimizing Your Network Design”.
http://www.cisco.com/cpress/cc/td/cpress/design/topdown/td0512.htm

Appendix A. OPNET-implemented codes (Arrival Process)

1) Push-out

```
/* push-out mechanism */
if (op_subq_pk_insert (0, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK)
       {
       /* the insertion failed (due to a full queue) */
       if (CLP != 0)
            {
             /* insertion of low priority cell failed (due to a full queue) */
             /* static to record cell loss */
             op_pk_destroy (pkptr);
             insert_ok = 0;
             L_loss_src1 = L_loss_src1 + 1;
            }
       else
            {
              /* function to find out if there are any low-priority cells in the queue
and, */
              /* if so, where the last one is in the queue when the queue is full */
              L_pos_index =  OPC_QPOS_TAIL;
              L_found = FALSE;
              while (!L_found && L_pos_index >= OPC_QPOS_HEAD)
                     {
                     op_pk_nfd_access (op_subq_pk_access (0, L_pos_index), "CLP",
&CLP);
                     if (CLP == 1)
                            L_found = TRUE;
                     else
                            L_pos_index = L_pos_index - 1;
                     }
           if (!L_found)
                     {
                     if (src == 0)
                            H_loss_src0 = H_loss_src0 + 1;
                     else if (src == 2)
                            H_loss_src2 = H_loss_src2 + 1;
                     op_pk_destroy (pkptr);
                     insert_ok = 0;
                     }
              else
                     {
                     op_pk_destroy (op_subq_pk_remove (0, L_pos_index));
                     for (i=L_pos_index; i<OPC_QPOS_TAIL ; i++)
                            op_subq_pk_swap (0, i+1, i);
                     op_pk_destroy (op_subq_pk_remove (0, OPC_QPOS_TAIL));
                     op_subq_pk_insert (0, pkptr, OPC_QPOS_TAIL);
                     insert_ok = 1;
                     L_loss_src1 = L_loss_src1 + 1;
                     }
              }
       }
else
       insert_ok = 1;
```

2) Partial buffer sharing

```
/* partial buffer sharing mechanism */
num_cells = op_subq_stat (0, OPC_QSTAT_IN_PKSIZE);
num_free_slots = op_subq_stat (0, OPC_QSTAT_FREE_PKSIZE);

/* examine if the queue overflows */
if (num_free_slots > 0)
       {
       /* examine if the queue threshold has been met */
       /* this is the case that the threshold is met */
       if (num_cells >= queue_thres)
              {
              /* if the threshold is met, the cell will be enqueued or discarded
depending on the priority of the cell */
```

```
                if (CLP != 0)
                        {
                        op_pk_destroy (pkptr);
                        insert_ok = 0;
                        L_loss_src1 = L_loss_src1 + 1;
                        }
                else
                        {
                        op_subq_pk_insert (0, pkptr, OPC_QPOS_TAIL);
                        insert_ok = 1;
                        }
                }
        /* this is the case that the threshold is not met */
        else
                {
                op_subq_pk_insert (0, pkptr, OPC_QPOS_TAIL);
                insert_ok = 1;
                }
        }
/* the cell is discarded due to the full queue */
else
        {
        if (CLP != 0)
                {
                L_loss_src1 = L_loss_src1 + 1;
                }
        else
                {
                if (src == 0)
                        H_loss_src0 = H_loss_src0 + 1;
                else if (src == 2)
                        H_loss_src2 = H_loss_src2 + 1;
                }
        op_pk_destroy (pkptr);
    insert_ok = 0;
    }
```

3) Buffer separation

```
/* buffer separation mechanism */
if ( CLP == 0)
        {
        if (op_subq_pk_insert (0, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK)
                {
                /* the insertion failed (due to a full queue) */
                /* insertion of low priority cell failed (due to a full queue) */
                /* static to record cell loss */
                if (src == 0)
                        H_loss_src0 = H_loss_src0 + 1;
                else if (src == 2)
                        H_loss_src2 = H_loss_src2 + 1;
                op_pk_destroy (pkptr);
                insert_ok = 0;
                }
        else
                insert_ok = 1;
        }
else
        {
        if (op_subq_pk_insert (1, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK)
                {
                /* the insertion failed (due to a full queue) */
                /* insertion of low priority cell failed (due to a full queue) */
                /* static to record cell loss */
                op_pk_destroy (pkptr);
                insert_ok = 0;
                L_loss_src1 = L_loss_src1 + 1;
                }
        else
                insert_ok = 1;
        }
```

Attachment. Simple test codes (C++ Language)

1) Push-out

```
// Push-out Mechanism Implementation //

// Input/output and null
#include<iostream.h>
#include<stddef.h>

typedef int Boolean;
const Boolean TRUE=1;
const Boolean FALSE=0;

// Refer to the blocks for the full descriptions of the functions below
void Discard_cell(char&, int&);
void Find_index(char[], int, Boolean&, int&);
void Place_tail(char&, char[], int&);
void Insert_cell(char&, char[], int, int&, int&);

int main()
{
       cout << "####    Assume that all valid inputs are H (high priority cell),
####" << endl
              << "####    L (low priority cell) and X (flag for termination) and
####" << endl
              << "####    that the values you will enter are incoming cells arriving
####" << endl
              << "####    within one service time unit so the cell at the head keeps
####" << endl
              << "####    waiting to be served.
####" << endl;

       int Q_SIZE;
       char cell;
       char queue[100];
       char* cell_ptr;
       Boolean Q_overflow;
       Boolean L_found;
       int L_index;
       int queue_occupancy;
       int H_loss;
       int L_loss;

       cout << "Enter the queue size : ";
       cin >> Q_SIZE;
       if (!cin)
       {
              cout << "Invalid value entered (non-integer value)!" << endl;
              return 1;
       }
       cout << "You have entered queue size = " << Q_SIZE << ". " << endl << endl <<
              "Enter cell values (H or L), followed by X to terminate. " << endl;

       // Initialization of variables
       queue_occupancy = 0;
       Q_overflow = FALSE;
       L_found = FALSE;
       H_loss = 0;
       L_loss = 0;

       for (int i=Q_SIZE-1; i>=0; i--)
              queue[i] = ' ';

       // Implementation of the push-out mechanism
       cin >> cell;
       while (cell != 'X')
       {
              if (cell == 'H' || cell == 'L')
              {
                     cell_ptr = &cell;
```

```
                        if (Q_overflow)
                        {
                                if (!(*cell_ptr=='H'))
                                        Discard_cell(*cell_ptr, L_loss);
                                else
                                {
                                        Find_index(queue, Q_SIZE, L_found, L_index);
                                        if (!L_found)
                                                Discard_cell(*cell_ptr, H_loss);
                                        else
                                                Insert_cell(*cell_ptr, queue, Q_SIZE,
L_index, L_loss);
                                }
                        }
                        else
                                Place_tail(*cell_ptr, queue, queue_occupancy);
                        if(queue_occupancy>=Q_SIZE)
                                Q_overflow = TRUE;
                }
                else
                        cout << "Invalid value! Please, enter again." << endl;
                cin >> cell;
        }
        cout << endl;

        // Print-out of cells in the queue arranged by Push-out
        cout << "Note that the right-most letter represents the cell at the head of the
queue"
                << endl << "while the left-most one represents the cell at the tail."
                << endl << endl;
        for (int row_1=Q_SIZE; row_1>0; row_1--)
                cout << "****";
        cout << endl;
        for (int row_2=Q_SIZE; row_2>0; row_2--)
                cout << "   *";
        cout << endl;
        for (int row_3=Q_SIZE; row_3>0; row_3--)
                cout << " " << queue[row_3-1] << " *";
        cout << endl;
        for (int row_4=Q_SIZE; row_4>0; row_4--)
                cout << "   *";
        cout << endl;
        for (int row_5=Q_SIZE; row_5>0; row_5--)
                cout << "****";
        cout << endl << endl;
        cout << "Remark :" << " Once the queue is full, the incoming high priority cell
enters the"
                << endl << "        queue, pushing out the last low priority cells (L) in
the queue. If"
                << endl << "        there is no low priority cell in the queue, the
incoming cell will"
                << endl << "        be dropped." << endl << endl;
        cout << "    Number of high priority cells lost : " << H_loss << endl
                << "    Number of low priority cells lost  : " << L_loss << endl;
        return 0;
}

// Function to discard cells based on Push-out when the queue is full
void Discard_cell (char& cel, int& loss)
{
        cel = NULL;
        loss = loss + 1;
}

// Function to place cells at the tail of the incompletely occupied queue
void Place_tail(char& cl,char queue[], int& q_occupancy)
{
        queue[q_occupancy] = cl;
        q_occupancy = q_occupancy + 1;
}

// Function to find out if there are any low-priority cells in the queue and,
```

```
// if so, where the last one is in the queue when the queue is full
void Find_index(char queue[], const int Q_length, Boolean& found, int& index)
{
	index = Q_length - 1;
	found = FALSE;
	while (!found && index>=0)
	{
		if (queue[index]=='L')
			found = TRUE;
		else
			index = index - 1;
	}
}

// Function to push out the last low-priority cell and place the incoming
// high-priority cell at the tail of the queue
void Insert_cell(char& cll, char queue[], int Q_len, int& del_index, int& l_loss)
{
	int i;
	for (i=del_index; i<Q_len-1 ; i++)
		queue[i] = queue[i+1];
	queue[Q_len-1] = cll;

	l_loss = l_loss + 1;
}
```

2) Partial buffer sharing

```
// Partial Buffer Sharing Mechanism Implementation //

// Input/output and null
#include<iostream.h>
#include<stddef.h>

typedef int Boolean;
const Boolean TRUE=1;
const Boolean FALSE=0;

// Refer to the blocks for the full descriptions of the functions below
void Discard_cell(char&, int&);
void Place_tail(char&, char[], int&);

int main()
{
	cout << "####    Assume that all valid inputs are H (high priority cell),
####" << endl
		<< "####    L (low priority cell) and X (flag for termination) and
####" << endl
		<< "####    that the values you will enter are incoming cells arriving
####" << endl
		<< "####    within one service time unit so the cell at the head keeps
####" << endl
		<< "####    waiting to be served.
####" << endl;


	int Q_SIZE;
	int Q_THRES;
	char queue[100];
	char cell;

	char* cell_ptr;
	Boolean Q_overflow;
	Boolean Q_thres_met;
	int queue_occupancy;
	int H_loss;
	int L_loss;

	cout << "Enter the queue size : ";
	cin >> Q_SIZE;
	while (!cin)
```

```
	{
		cout << "Invalid value entered (non-integer value)!" << endl;
		return 1;
		//cin >> Q_SIZE;
	}
	cout << "Enter the threshold : ";
	cin >> Q_THRES ;
	while (!cin || Q_THRES > Q_SIZE)
	{
		if (!cin)
		{
			cout << "Invalid value entered (non-integer value)!" << endl;
			return 1;
		}
		else
			cout << "Threshold couldn't be greater than queue size! Please,
enter again."
				<< endl << "Queue threshold = ";
		cin >> Q_THRES;
	}

	cout << "You have entered queue size = " << Q_SIZE << " and threshold = " <<
Q_THRES << ". "
		<< endl << endl << "Enter cell values (H or L), followed by X to
	terminate. " << endl;

	// Initialization of variables
	queue_occupancy = 0;
	Q_overflow = FALSE;
	Q_thres_met = FALSE;
	H_loss = 0;
	L_loss = 0;

	for (int i=Q_SIZE-1; i>=0; i--)
		queue[i] = ' ';



	// Implementation of the partial buffer sharing mechanism
	cin >> cell;
	while (cell !='X')
	{
		if (cell == 'H' || cell == 'L')
		{
			cell_ptr = &cell;

			if (Q_thres_met)
			{
				if (!(*cell_ptr=='H'))
					Discard_cell(*cell_ptr, L_loss);
				else
				{
					if (Q_overflow)
						Discard_cell(*cell_ptr, H_loss);
					else
						Place_tail(*cell_ptr, queue,
queue_occupancy);
				}
			}
			else
				Place_tail(*cell_ptr, queue, queue_occupancy);

			if(queue_occupancy>=Q_THRES)
				Q_thres_met = TRUE;
			if(queue_occupancy>=Q_SIZE)
				Q_overflow = TRUE;
		}
		else
			cout << "Invalid value! Please, enter again." << endl;
		cin >> cell;
	}
	cout << endl;
```

```
	// Print-out of cells in the queue arranged by Partial Buffer Sharing
	cout << "Note that the right-most letter represents the cell at the head of the
queue"
		<< endl << "while the left-most one represents the cell at the tail."
		<< endl << endl;
	for (int row_1=Q_SIZE; row_1>0; row_1--)
		cout << "****";
	cout << endl;
	for (int row_2=Q_SIZE; row_2>0; row_2--)
		cout << "   *";
	cout << endl;
	for (int row_3=Q_SIZE; row_3>0; row_3--)
		cout << " " << queue[row_3-1] << " *";
	cout << endl;
	for (int row_4=Q_SIZE; row_4>0; row_4--)
		cout << "   *";
	cout << endl;
	for (int row_5=Q_SIZE; row_5>0; row_5--)
		cout << "****";
	cout << endl << endl;
	cout << "Remark :" << "Once the threshold is met, the low priority cells (L)
arriving are "
		<< endl << "        dropped. Also, the high priority cells (H) are lost
once the queue"
		<< endl << "        gets full."
		<< endl << endl;
	cout << "   Number of high priority cells lost : " << H_loss << endl
		<< "   Number of low priority cells lost  : " << L_loss << endl;

	return 0;
}

// Function to discard cells depending on the queue condition (threshold met or queue
full)
void Discard_cell (char& cel, int& loss)
{
	cel = NULL;
	loss = loss + 1;
}

// Function to place cells at the tail of the queue
void Place_tail (char& cl,char queue[], int& q_occupancy)
{
	queue[q_occupancy] = cl;
	q_occupancy = q_occupancy + 1;
}
```