CMPT 885-3: SPECIAL TOPICS: HIGH-PERFORMANCE NETWORKS
Spring 2002

Implementation of Start-Time Fair Queuing Algorithm in Opnet™

FINAL PROJECT REPORT

http://www.cs.sfu.ca/~daryn/personal/school/885/

Daryn Mitchell & Jack Man Shun Yeung
{daryn,yeung}@cs.sfu.ca

Table of Contents

Abstract 6
1 Introduction......6
2 Background......7
  2.1 Congestion Control 7
  2.2 Queuing Schemes......8
  2.3 Start-Time Fair Queuing Algorithm......9
  2.4 SFQ Details 10
3 SFQ Implementation 11
  3.1 SFQ ICI 11
  3.2 SFQ Process Model 11
    3.2.1 Algorithm Implementation Design 12
    3.2.2 The *init* State......13
    3.2.3 The *arrive* State......13
    3.2.4 *svc_start* State 13
    3.2.5 The *svc_complete* State 14
  3.3 VirtualClock Process Model 14
  3.4 Source Node Process Model 15
  3.5 Receiving Node Process Model......15
4 Performance Evaluation 16
  4.1 Network Configuration 16
  4.2 Unfairness of FIFO 16
  4.3 Flow Protection in SFQ......17
    4.3.1 Delaying Only Packets From Misbehaving Sources 17
    4.3.2 Preventing Credit-Store & Burst Misbehaviour......19
  4.4 Idle Bandwidth Use: Comparison SFQ vs. VC 20
    4.4.1 Does VC "Punish the Use of Idle Time"?......20
      Scenario 1......21
      Scenario 2......21
    4.4.2 VC Punishes Misbehaviour......22
      Misbehaviour: Auxiliary Virtual Clock in Traffic bursts (Scenario 1)......22
      Senario 3 …When Congestion Occurs 22
  4.5 Punishing & Bandwidth for VBR flows......25
      Scenario 4......26
      Scenario 5 – VBR punishing in VC 27
      Scenario 6 – VBR non- punishing in SFQ......28
      VBR punishing: Conclusion 28
5 Discussion and Conclusion 29
6 References......30

Appendix A: VirtualClock Implementation Details......32
    3.3.1 The *init* State 32
    3.3.2 The *arrive* State 32
    3.3.3 The *svc_start* State......32
    3.3.4 The *svc_complete* State......33
Appendix B: Source Code......34
  FIFO Queue Module (Opnet's acb_fifo)......34
  SFQ Queue Module......36
  VC Queue Module 44
  Source Node Module......50
  Receving Node Module 56

Abstract

Scheduling or queuing algorithms are used in packet forwarding elements such as routers or switches [Cis95]. The goal is to minimize transmission delays as well as to help in manage network congestions. In this project, we study different scheduling algorithms. We examine in particular Start-Time Fair Queuing [Goy96] which claims to achieve good fairness even with VBR traffic source present in the network. We implemented the algorithm as an Opnet™ 8.0 model on which simulations are run to verify its different characteristics. Comparisons between Start-Time Fair Queuing and other scheduling schemes were also done by ways of simulations.

1 Introduction

Routers and switches in packet-switched networks use buffers to manage data, which come from various sources but are destined to a single shared link. However, the difference between the packet arrival rate and packet departure rate as well as congestion in the network usually causes packet to wait in the buffers, and this imposes significant delays in the transmissions. Efficient buffer managements help to minimize these delays and can even help to resolve congestion in the network.

The majority of routers deployed in real networks serve packets from various sources in the order they arrive (First-in, First-out, or 'FIFO'). This is a simple and thus low cost approach which generally works. However, with FIFO scheduling, if one source floods the network with packets, it will appropriate a high proportion of the packets served and thus the bandwidth available to other well-behaved sources will dramatically decrease, resulting in an unfair allocation of bandwidth in the view of the well-behaved sources.

A number of different scheduling algorithms (a.k.a. queuing algorithms) with varying levels of complexity have been introduced over the past 15 years to address this issue. They generally achieve fairness by allocating a relatively equal share of the bandwidth to each source; under congestion, each source or incoming packet flow is guaranteed its portion. Flows may transmit faster than their allocated bandwidth only if there is idle time not being used by other flows.

Many of these fair queuing (FQ) algorithms developed in the old days, however, do not perform well in the presence of variable bit rate (VBR) traffic source. As VBR traffic becomes very common in modern networks (e.g. video traffic on the Internet), modern queuing algorithms must be able to fairly handle a combination of VBR and CBR traffic.

For this CMPT 885 project we decided to study different scheduling algorithms used in routers or switches, and specially focus on examining and implementing the "Start Time Fair Queuing" (SFQ), which is a fair queuing algorithm that performs well even in the presence of VBR traffic. Performance of SFQ will be evaluated with other algorithms. Our evaluation approach follows an

earlier project by Nazy Alborz in which she implemented and evaluated a different FQ algorithm, *Virtual Clock* (VC).

In this report, we give a general description of different scheduling algorithms and present the SFQ algorithm as the result of our study. We then describe our implementation of SFQ in the Opnet™ network modeling and simulation environment. Finally, we report the results of a network simulation that we ran for evaluating the performance of the SFQ algorithm in comparison to other scheduling schemes.

2 Background

In the section, we first present the role of scheduling or queuing algorithms in the packet-switched network. Descriptions of different widely adopted queuing algorithms, including Start-Time Fair Queuing, follow. Then, we describe our implementation of SFQ in Opnet™ 8.0. Finally, we present the results of the performance evaluation of SFQ with our implementation.

2.1 Congestion Control

In circuit-switched networks where data travels along a fixed path with dedicated bandwidth, there is no queuing delay at the switches; in packet-switched networks, however, the data rate varies depending on the packet size and the rate at which packets arrive.

When packets from multiple sources are multiplexed to the same outgoing link at the switch or router but data arrives faster than it can be routed and transmitted across, congestion may occur. In this case, the switch must buffer the packets until the outgoing link is free. The time the packet waits in the buffer adds random delay to the transmission. This added delay can be very large, and if the capacity of buffer is exceeded packets will be lost. [Wal00]

The simplest way to avoid congestion and delay is to use less than the maximum bandwidth of the network [Chu02]. However, that would by definition mean an inefficient use of the network. Instead, the goal is to keep the delay within acceptable limits by using congestion control algorithms.

Congestion control can be done at the source by limiting the amount of traffic sent, which is also referred as flow control. A typical example is the window adjustment approach of TCP [Wal00]. On the other hand, congestion can also be controlled at the switch or router by managing the buffers effectively. Different methods for the latter approach, known as scheduling or queuing algorithms, are developed and widely adopted by switch and router manufacturers.

One drawback of the controlling-at-the-source approach is that it usually depends on the responses from other network elements, either the switches or the receiver host, in order to dynamically adjust how much data should be sent out or how much data should be buffered in the future. Queuing algorithms instead usually works by only examining the incoming traffic rate, and thus needs no communications with other network elements.

2.2 Queuing Schemes

A very simple and basic queuing algorithm is to serve packets in the order they arrive from all sources, FIFO[1]. However, the router is susceptible to a serious problem. The flaw, identified nearly twenty years ago by John Nagle [Nag84], is that a source that ignores flow control and floods the network with traffic will dominate the FIFO queue - at the expense of the other well-behaved sources. This ill-behaved source effectively steals bandwidth from the other sources. Such behaviour could easily take place when people intentionally altering their algorithms in order to take advantage of the phenomenon [Der90]. But it could also happen unintentionally when a source has relatively larger packet size, or when network equipment malfunctions. In fact, this problem can even occur between well-functioning sources that use different congestion control algorithms, such as the Reno and Vegas versions of TCP [Mo99].

Priority Queuing tries to improve FIFO queuing, by examining priority values of incoming packets. Incoming packets with higher priorities will always be served before lower priority packets. So important packets with higher priorities should have smaller delays. However, if an ill-behaved source keep flooding the network with higher priority packets, the domination situation occurs as in FIFO.

Another queuing algorithm called Class-Based Queuing tries to improve FIFO by classifying incoming packets into different pre-defined sub-queues, and then serve the different sub-queues in a round-robin manner. This ensures fair bandwidth allocation among different "kinds" of packets (i.e. among different sub-queues). But since packets in the same sub-queues are still being served in a FIFO manner, the disadvantages of simple FIFO queuing still present.

The solution, known as fair queuing (FQ), is to share the bandwidth of the outgoing link among sources by guaranteeing each source a minimum bandwidth allocation. In this way, well-behaved sources can have some protections from ill-behaved sources. More Specifically, the algorithm "insures that well-behaved hosts receive better service than badly-behaved hosts" [Nag84, RFC]. Each source is served one at a time in a round robin manner. Sources with no incoming packets are skipped in that round. Nagle's concept is widely accepted and affects many later queuing algorithms, such as Virtual Clock Fair Queuing, Weighted Fair Queuing, and Start Time Fair Queuing.

Weighted Fair Queuing (WFQ) is proposed based on Nagle's simple FQ algorithm in order to better suit the needs of the real world and at the same time retains the advantages of FQ. WFQ estimates the time of finishing serving the packet, and use this finish time to determine which packets to serve first. Moreover, the "weight" concept gives room to implement quality of services in the network. The algorithm provides good fairness in general for constant bit rate (CBR) traffic.

Virtual Clock (VC) orders incoming packets by calculating a time stamp for each incoming packet. The time stamp is created by considering both a finish-serving time, like WFQ, and a time

[1] FIFO queuing (i.e. first-in, first-out) is also commonly called "first-come first-serve," or FCFS

from a virtual clock system. The packet with the earliest stamped time is served first. VC was designed to reduce the computational complexity associated with the original FQ method. However, it increases the maximum delay incurred by packets significantly. Although it is technically fair, VC does not handle VBR traffic well because it penalizes a source for the use of idle bandwidth, as stated by Parekh & Gallagher [Par93]
Zhang proposes an interesting scheme called virtual clock multiplexing [21], Virtual clock multiplexing allows a guaranteed rate and (average) delay for each session, independent of the behavior of other sessions. However, if a session produces a large burst of data, even while the system is lightly loaded, that session can be "punished" much later when the other sessions become active.
This will be examined further in Section 4.4.

2.3 Start-Time Fair Queuing Algorithm

Unlike WFQ, Start Time Fair Queuing (SFQ)[2] orders packets by calculating a start-sending time stamp for each packet. The packet with the earliest starting time will be served first. Similar to VC, it is designed to reduce computational complexity. SFQ has also been claimed to be the first queuing algorithm focusing on handling both CBR and VBR traffic, and thus has benefits when applying on modern networks where VBR traffic is common.

The SFQ algorithm can be summarized as follows:
The router serves packets in order of start time. The basic idea is as follows: a packet arriving in flow *f* is assigned a "start tag" as follows:
If there are packets from flow *f* waiting in the buffer, the new packet should be scheduled as soon as possible behind them. Thus the start tag is set to be the "finish tag" of the previous packet. The finish tag is actually pre-assigned based on the size of packet divided by the rate assigned to the flow.
However, if the start tag of the packet currently being serviced (from any flow) is higher than the finish tag of the previous packet, use the currently servicing start tag that instead. (i.e. if flow *f* is behaving, the packet can go *to* the front of the line, but it can't go *in front* of the line)

Goyal et al additionally refer to setting the first start tag after a non busy period (i.e. server not serving packets) to the maximum finish time of packets served so far, but note that this was for proving delay guarantees and that setting it to zero would be equivalent.
The algorithm provides fairness protection between flows because each packet is scheduled behind the packets from its own flow. If one flow over-transmits, then its packets won't cut in line in front of other flows. It also provides sensible fairness during less busy periods (corresponding to the times when a flow does not have any packets waiting in the buffer): the extra bandwidth is shared among all flows. When a new packet arrives from the idle flow it goes to the front of the

[2] Note that a different queuing algorithm, Stochastic Fairness Queuing, was already referred to as "SFQ" leading some to suggest the use of "STFQ" for Start-Time Fair Queuing [Vil98]. However, in most citations the acronym "SFQ" - as used by the authors - persists [Citeseer].

line and scheduling continues as before, and thus there is no penalty for other flows using the extra bandwidth.

2.4 SFQ Details

The algorithm uses two variables associated with each data flow to record the state, *maximum_finish_tag_served* to store the greatest *finish_tag* of all packets served so far, and it uses two values that it calculates and associates with each packet, *start_tag* and *finish_tag*.
The algorithm works as follows:
For each packet that arrives from flow *f* at time *t*
if (the server is busy)
  virt_time ← the *start_tag* of the packet in service at time *t*
else (server not busy)
  virt_time ← maximum_finish_tag_served
prev_finish_tag ← *finish_tag* of the previous packet in flow *f*
start_tag ← max{ virt_time , prev_finish_tag }
finish_tag ← *start_tag* + (length of packet)/weight

3 SFQ Implementation

In this section we describe our implementation of the SFQ algorithm in Opnet™ 8.0 as a Queue Model that is a process model in Opnet™ 8.0. The core of the implementation consists of a 'C' programming language file *sfq.pr.c*. With SFQ implemented as a Queue Model, a Queue Node created in the Node editor can select "sfq" as the queuing scheme.

3.1 SFQ ICI

To support the algorithm, we use the "Interface Control Information" (ICI) system in Opnet™. This allows us to associate data fields with each arriving packet to the SFQ Queue Model. The SFQ ICI is found in the file *sfq.ic.m.*, and its configuration is show below:

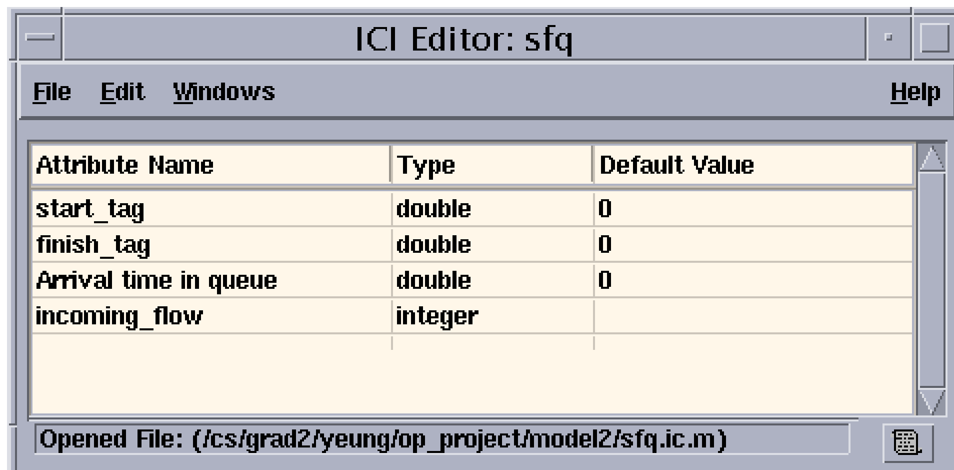


Figure 3.1-1 ICI used in SFQ scheduling algorithm

3.2 SFQ Process Model

We originally planned to implement SFQ as one of the QoS routing options in the IP model of Opnet™ following what Nazy Alborz did in her implementation of the VirtualClock algorithm [Alb01]. This was our intention at the time of the progress report in March, 2002. However, after spending time examining the code for the IP output interface and IP QoS models, we realized that the complexity of integration was too great for this project and that it would divert our focus from the implementation and testing of SFQ. Consequently, we determined to implement SFQ as an Opnet™ process model, and
were able to use the finite state machine (FSM) of Opnet™'s active FIFO queue, *acb_fifo*, as the foundation of our SFQ implementation.

The Finite State Machine of SFQ Queue Model is arranged as follows:

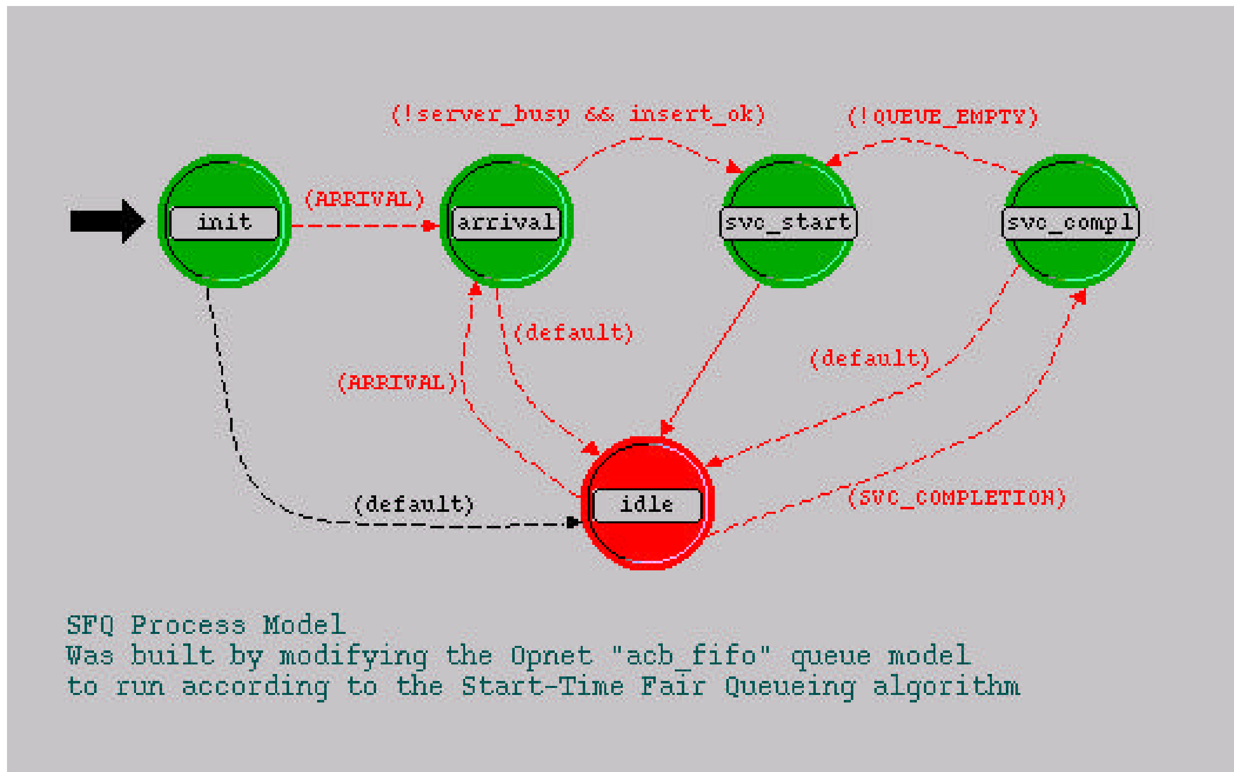


Figure 3.2-1 Process model finite state machine of FIFO, SFQ, and VirtualClock queue

3.2.1 Algorithm Implementation Design

In FIFO queuing, the arrival state simply queues the packet at the tail of a single queue. At the time of servicing, the algorithm selects the packet at the head of the queue. The queue is accessed by using Opnet™ queuing commands like

op_subq_pk_insert (queue_id, pkptr, OPC_QPOS_TAIL)
pkptr = op_subq_pk_access (queue_id, OPC_QPOS_HEAD);

In SFQ, since the packets are served in order of start tag we had the option of
building a single priority queue and inserting packets with *start_tag* as the priority.
maintaining separate FIFO subqueues for each incoming flow. Then, when it comes to the time to serve a packet, we examine the head of each queue and serve the packet with the lowest start tag.

Both techniques are suitable to implement SFQ because they both schedule packets in order of start tags.

In a real router we would likely choose option 1 because it would be adaptable to a large number of flows and it keeps most of the packet processing actions within the packet insertion stage – which would often occur while another packet was being served, thus minimizing delay between

serving packet. (Note: This priority queue concept is the one presented by Lixia Zhang for the Virtual Clock scheme [Zha90].)

For this simulation, however, we decided to use subqueues for each flow since conceptually it matched more closely with our mental model. Furthermore, it would be convenient to measure the size and delay in each subqueue using built-in statistic functions in Opnet™.

3.2.2 The *init* State

Initialize the variables and statistics.

3.2.3 The *arrive* State

Step 1: Get the incoming packet
Step 2: Get information about the packet: flow *f* that it is from, packet length, and arrival time in queue. Store the arrival time. If the flow has been assigned a weight of 0 then discard the packet.
Step 3: Calculate the start_tag and finish_tag for the packet, and update the "prev_finish_tag[flow]" variable to use in the next packet arrival from flow *f*
Step 4: Create an ICI and assign values to the data fields: start_tag, finish_tag (for SFQ) and "Arrival Time in Queue" and "incoming_flow" (for statistics); attach the ICI to the packet.
Step 5: Attempt to queue the packet in the subqueue corresponding to flow *f*. If the server is not busy, go straight to the *svc_start* state; otherwise return to the idle state.

3.2.4 *svc_start* State

Step 1: Determine which subqueue to serve. This is performed by the following sequence of actions:
Cycle through the subqueues, stop when you find the first non-empty queue
Access the packet at the head, get its *start_tag* from the ICI
Set this *start_tag* as "min" and the queue as "min_q"
Loop through any remaining subqueues and for each non-empty subqueue repeat (b)
If this start_tag is lower than *min*, update *min* and *min_q*
min_q is the subqueue to serve

Step 2: Get a pointer to the packet and retrieve its *start_tag* from the ICI. Update the start_tag_in_service and queue_in_service state variables.
Step 3: Calculate the time it will take to serve the packet (based on packet length) and schedule an interrupt for the time when service is completed.
Step 4: Update statistics
Step 5: Set the *server_busy* state variable to true and return to the idle state.

3.2.5 The *svc_complete* State

This state is entered by the self-scheduled interrupt from *svc_start* state, which indicates that the time required for packet service is completed.

Step 1: Remove the packet from its subqueue
Step 2: Retrieve its *finish_tag* from the ICI. Update the *max_finish_tag* state variable if the new *finish_tag* is greater.
Step 3: Update statistics
Step 4: Send the packet. If there are packets waiting to be sent, go straight to the *svc_start* state; otherwise return to the idle state.

3.3 VirtualClock Process Model

We originally intended to use the Virtual Clock QoS routing option in Opnet™'s IP model that Nazy Alborz implemented [Alb01]. However, we discovered that it was written for an earlier version of Opnet™ and did not work in Opnet™ Modeller 8.0. Nevertheless, we desired to compare SFQ with another fair queuing technique: it was important if we wanted to get more interesting simulation results. Thus, we decided to implement VC algorithm as a Queue Model, similar to SFQ.

Our VC process model was primarily built by transforming Nazy Alborz's VC implementation in the IP framework into the Opnet™ Process model framework. The functioning of SFQ and VC, like most Fair Queuing (FQ) algorithms, are very similar except for the scheduling method. We were able to implement VC with exactly the same structure that we did in implementing SFQ. *start_tag* and *finish_tag* in SFQ are replaced by *auxiliary_virtual_clock* in VC. Also, a VC ICI is created to associate time stamps ('virtual_clock_stamp' and 'aux_virtual_clock_stamp') to each arriving packet. The FSM for VirtualClock is the same as that of SFQ and *acb_fifo*. Since the details of the VC Queue Model are very similar to the SFQ Queue Model, the implementation details for VC are left to Appendix A.

3.4 Source Node Process Model

We modified the *simple_source* process model in Opnet to allow a more flexible way to generate different types of traffic flows. Figure 3.4-1 (right) shows the additional options available to the users in the modified version of simple source object. They are basically used to control two new functions:

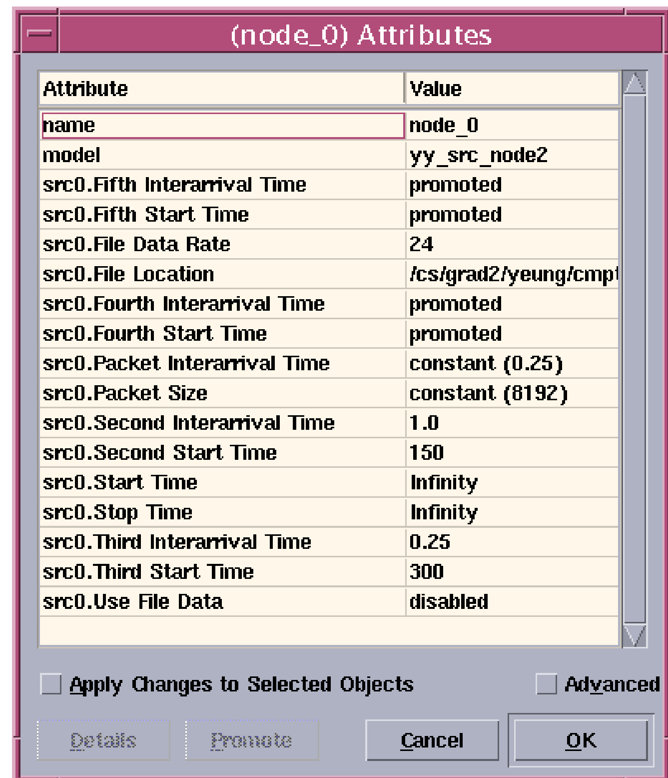


Figure 3.4-1 Source Node Attributes

1) *File reading capability*
The "Use File Data", "File Location" and the "File Data Rate" options together allow the user to tell the source node to read in data from a file and generate traffic flows based on the data (e.g. the star wars trace).
2) *Multiple packet interarrival time*
The second, third, fourth and fifth interarrival time allow the user to specify up to five different interarrival times in a traffic flows over different periods of time. With the corresponding "Start Time" options, these different periods of time are controlled.

3.5 Receiving Node Process Model

The receiving node acts as the end point in the network since all the packets from the source nodes are destined to this receiving node. The receiving node is a completely new implementation, and it has two functionalities:
1) destroy the incoming packets, and
2) record statistics about the incoming packets.

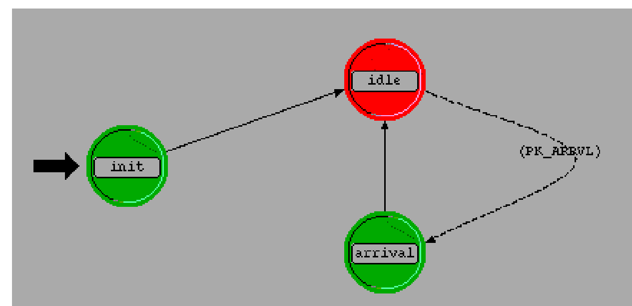


Figure 3.5-1 Receiving Node Process Model – Finite State Machine

Figure 3.5-1 (right) shows the Finite State Machine of its process model. It starts with the "init" state to initial all the state and temporary variables, and then it switches to the "idle" state. When a packet arrives, it switches from the "idle" state to the "arrival" state to record statistics, like end-to-end delay of the packets. Finally, it destroys the packets and returns to the "idle" state to wait for the next packet to arrive.

4 Performance Evaluation

4.1 Network Configuration

We designed a simple network in order to test the SFQ queue model and to verify the characteristics and performances of SFQ. There are four source nodes (on the left) for generating traffic. All source nodes are connected to the router (in the middle). In order to communicate with the receiving node (on the right), the source nodes share the same link between the router and the receiving node. The traffic generation of the source s and the bandwidth bottleneck of the router/shared link are customizable. This arrangement allowed us to create variety of source traffic situations with a bottleneck at the router.

We ran a number of simulation scenarios to verify the performance and characteristics of SFQ. The results are presented as 4 categories in the following subsections.

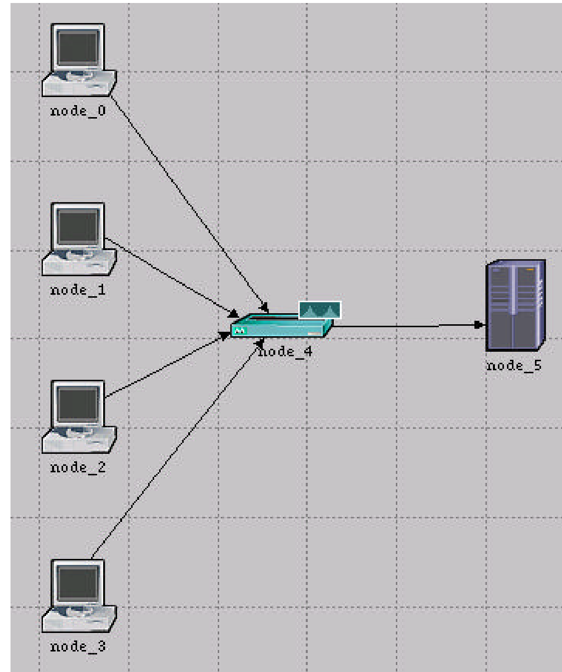


Figure 4.1-1 Network Configuration

4.2 Unfairness of FIFO

As stated in Section 2.2, FIFO is susceptible to a problem of unfairness in which an ill-behaving source can easily occupy most of the bandwidth of the outgoing link by using a relatively higher packet arrival rate at the router, leaving only a small room for the other sources to transmit their packets. Fair Queuing algorithms protect each flow from other flows' misbehaviour; in Weighted Fair Queuing schemes such as SFQ, the algorithm guarantee a certain amount of bandwidth to each flow according to its allocated 'weight.'

We created a simple scenario to observe FIFO behaviour and verify that SFQ maintains fair bandwidth allocation to each flow. The scenario uses two source nodes of our network model to generate traffic:

Source [0]	1 pkt / 8 seconds
Source [1]	10 pkts / second

The bandwidth of the link is 6 pkts/sec, so Source [1] is deemed to be misbehaving and flooding the network. We ran the scenario twice, first with FIFO queuing and then with SFQ as the scheduling algorithm in the router. The throughput of the individual traffic flows were as follows:

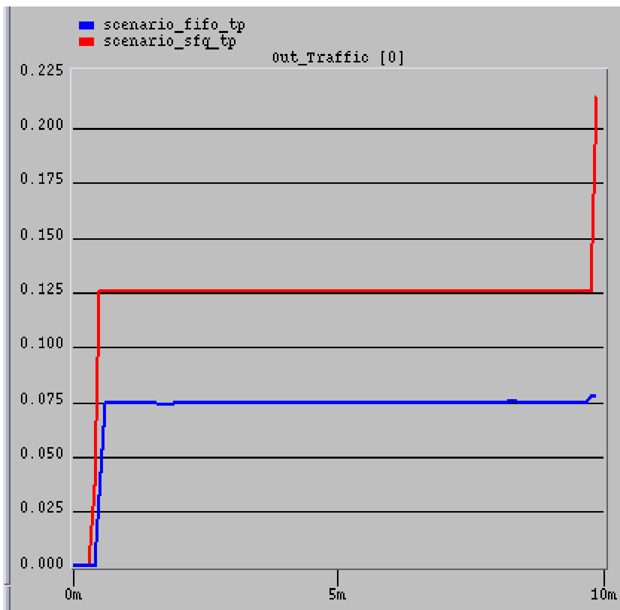


Figure 4.2-1 Throughput of Flow[0] (behaving source)

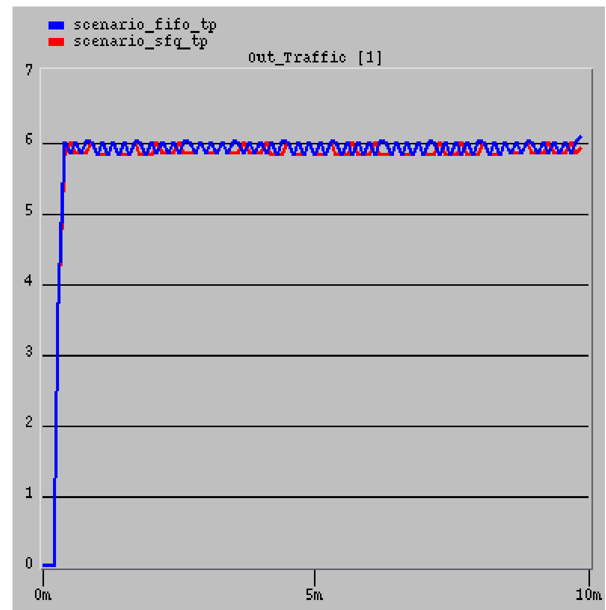


Figure 4.2-2 Throughput of Flow[1] (misbehaving source)

Figure 4.1-1 and 4.1-2 above show the throughputs of individual traffic flows at the router, and we can see that SFQ allowed packet from source 1 to go through at the expected rate of 0.125 packets per second. However, with FIFO, packets from source 1 only have an outgoing rate of 0.075 packets/sec at the router, while packets from source 2 were using most of the bandwidth.

This illustrates the fact that FIFO does not protect bandwidth for flows when there is congestion. In SFQ, on the other hand, a conforming source has a guarantee throughput even with the presence of congestion caused by a misbehaving source.

4.3 Flow Protection in SFQ

We further investigated the fairness of SFQ by verifying its ability to protect conforming sources from misbehaving sources in terms of 1) delay and 2) "credit storing."

4.3.1 Delaying Only Packets From Misbehaving Sources

A consequence of the fact that Fair Queuing algorithms guarantee bandwidth to behaving flows is that when a traffic flow misbehaves and causes congestion in the network, packets from that traffic flow will greater experience delays. To visualize this characteristic, a scenario with 2 traffic generation nodes was set up as follows:

Time:	0-3 minutes	3-5 minutes	5-10 minute
Source [1]	4 packets/sec		
Source [2]	4 packets/sec	6 packets/sec	4 packets/sec

The outgoing link bandwidth is 8 pkts/sec. The bandwidth guarantee for the flows are:
Flow 1: 4 pkts/sec
Flow 2: 4 pkts/sec
Thus, during the period of 3-5 minutes, source 2 misbehaves and causes congestion. The traffic generated by the sources and the throughputs for individual traffic flows are shown below (Figures 4.3-1 and 4.3-2):

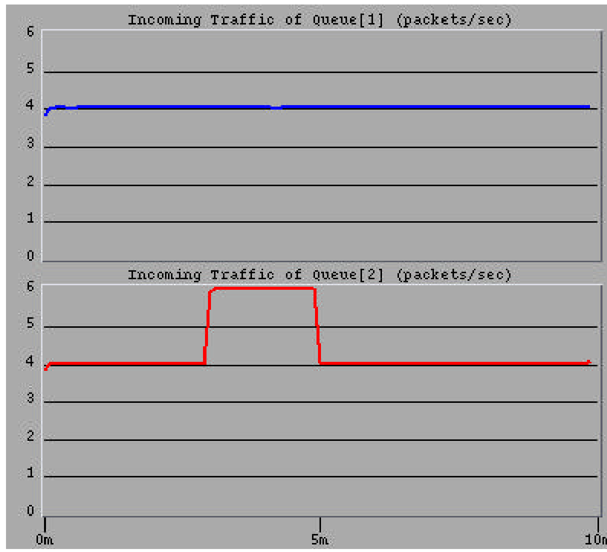


Figure 4.3-1 Source Traffic

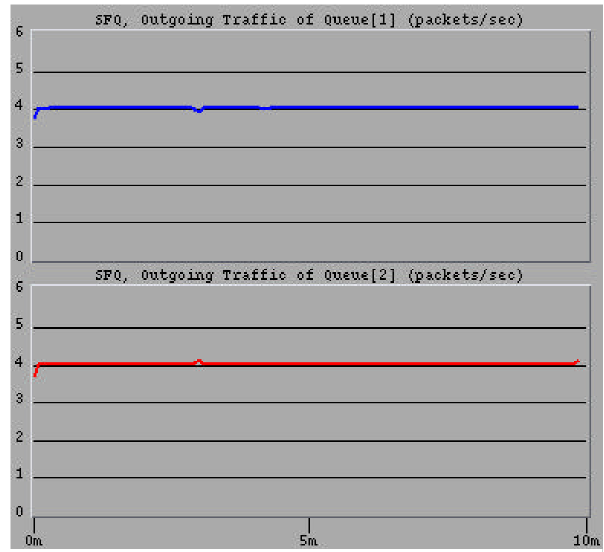


Figure 4.3-2 Throughput with SFQ

Furthermore, the end-to-end (ETE) delays of packets from different flows (Figure 4.3.-3, shown below) confirms that SFQ protects packets from the behaving source [1] from experiencing larger delays in the presence of congestion caused by the misbehaving source.

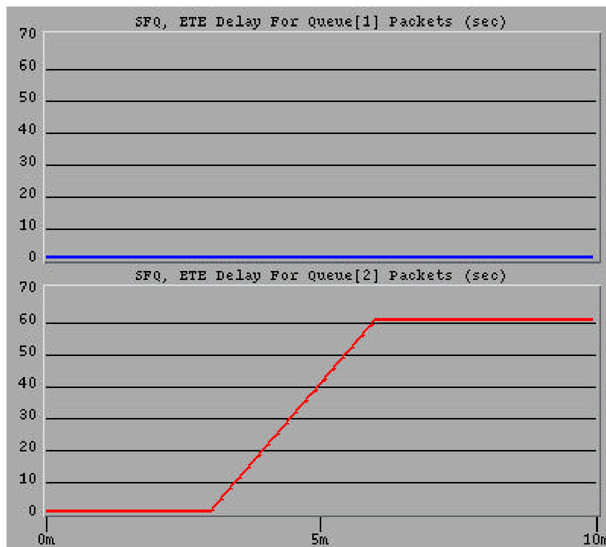


Figure 4.3-3 ETE Delay of behaving and misbehaving flows

4.3.2 Preventing Credit-Store & Burst Misbehaviour

A flow (e.g. VBR) should not be allowed to store up credit in a low period and burst later, because the idle bandwidth that existed during the low period is "gone" [Zha90]. VirtualClock prevents a source from bursting after credit-storing, as verified by Alborz and Trajkovic [Alb01]. We use the same simulation scenario to verify that SFQ has the same ability.
The scenario is created with the following settings:

	5-20sec	20-141sec	142-391sec	392-455sec	456-555sec
Source 1, CBR	–	4 packets/sec			
Source 2, CBR	–	2 packets/sec	0.5 packets/sec	8 packets/sec	–
Source 3, VBR	Average 4 packets/sec				

The outgoing link serves up to 8 packets/sec
The weight agreement is as follows:
Source 1: "4"
Source 2: "2"
Source 3: "4"
Thus even in congestion SFQ should guarantee bandwidth of:
Source 1: 3.2 pkts/sec
Source 2: 1.6 pkts/sec
Source 3: 3.2 pkts/sec
Source 2 attempts to misbehave by under-transmitting (during t= 20-141) and then bursting (during t= 392 – 455). Traffic generation for each source is shown in Fig. 4.3-1 (right).

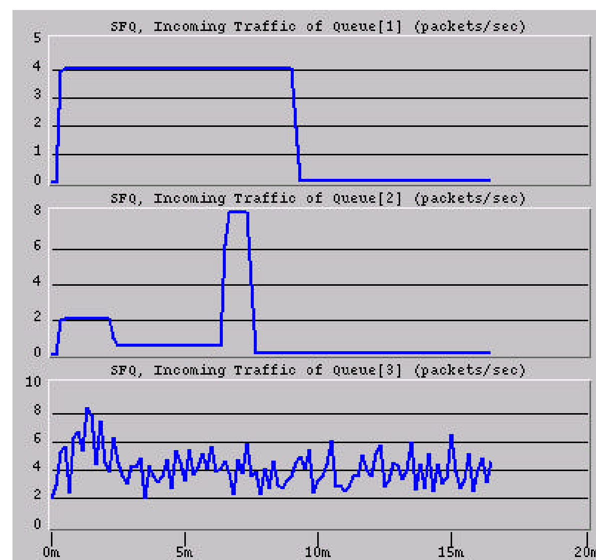


Figure 4.3-1 Source Traffic for Flows 1, 2, 3

The router throughput for each flow is found in Fig. 4.3-2 (right). This result shows that as in VirtualClock, this credit storing strategy does not work with SFQ. The throughput of flow 2 never exceeds 2 pkts /sec even when it transmits 8 pkts/sec, and the bandwidth guarantees for the other flows are maintained.
Thus we see that SFQ does not allow a misbehaving source to store credits in order to send a traffic burst at a later time.

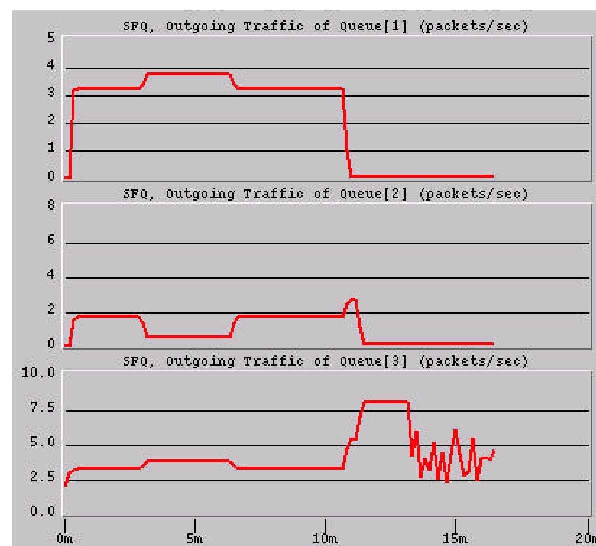


Figure 4.3-2 Throughput for Flows 1, 2, 3 (SFQ)

4.4 Idle Bandwidth Use: Comparison SFQ vs. VC

In Section 2.2 we mentioned that Parekh & Gallagher claim that VirtualClock punishes flows for large bursts of data, even long after the burst [Par93]. Goyal et al cite this, stating that VC "penalizes a traffic flow for using idle bandwidth" [Goy96]. They designed SFQ to eliminate this effect, allow a flow to use the idle bandwidth (i.e. bandwidth originally reserved for some other flows but are not being used at the time) without penalty, in order to achieve an efficient use or an efficient use of bandwidth.

In this section, we present a number of tests we used to verify this claim.

Our result shows that VC does have a "punish" effect but the claim of "punishing of using idle time" is not completely true.

4.4.1 Does VC "Punish the Use of Idle Time"?

The claim that VC punishes the use of idle time was surprising to us. We concluded that if this were true, the consequence would be as follows:

Transmission rate of source 1 and source 2

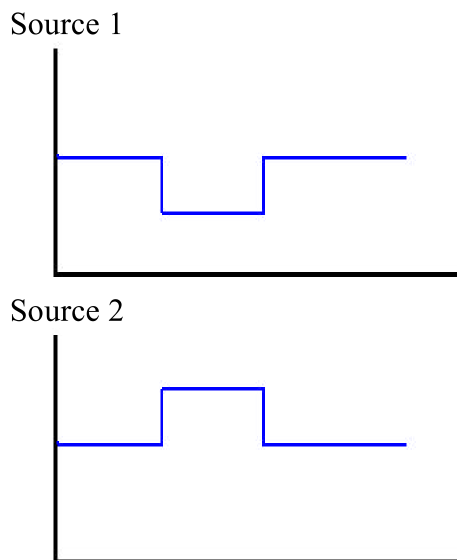


Figure 4.4-1 Source Traffic (Using "Idle Time")

Packet serving under punishing scheduling scheme
(red indicates *actual* rate)

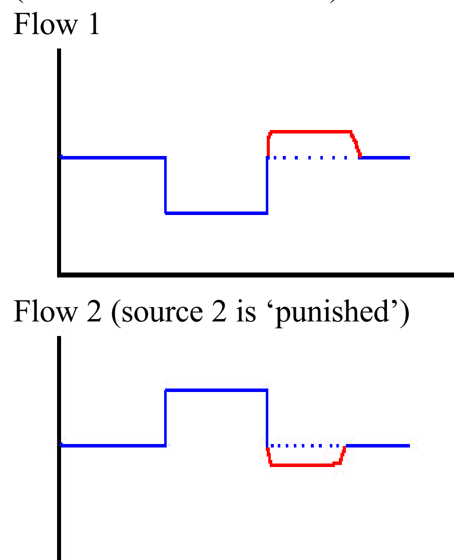


Figure 4.4-2 Expected Throughput In VC

Scenario 1

Fig. 4.4-1 defined our first test. We created a simple simulation by using two sources with traffic to mimic the traffic sources above, as follows:

Time:	0-2 minutes	2-5 minutes	5-15 minutes
Source [1]	4 pkts/sec	1 pkt / sec	4 pkts/sec
Source [2]	4 pkts/sec	6 pkts/sec	4 pkts/sec

The bandwidth of the outgoing link was 8 pkts /sec, so each flow was given 4 pkts/sec. The input traffic and outgoing throughput are shown below:

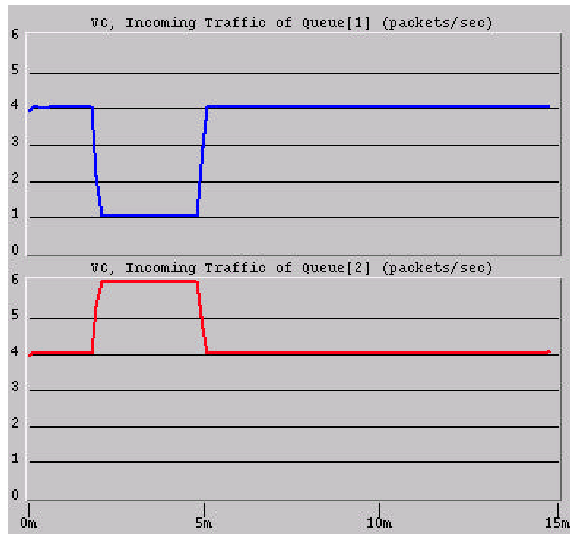


Figure 4.4-3 Source Traffic

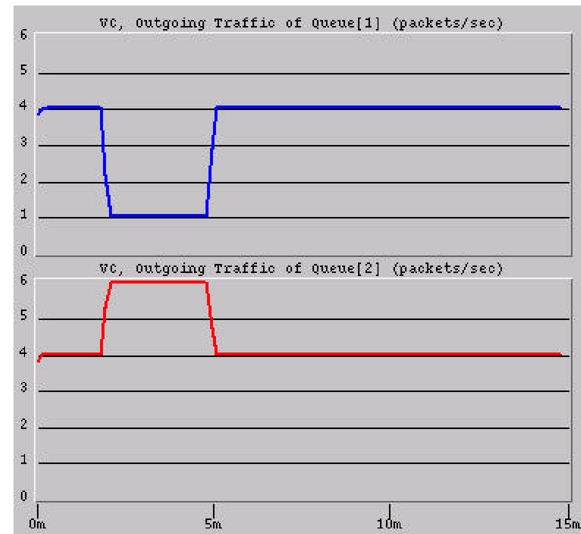
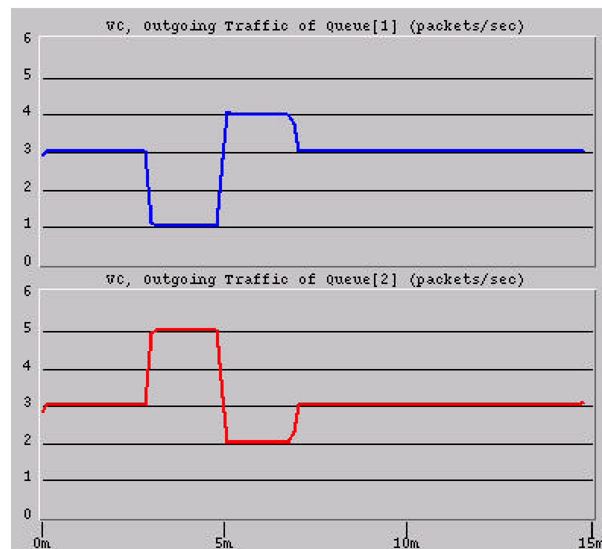


Figure 4.4-4 Throughput with VC - Scenario 1

Clearly, VC did not punish flow 2 for using idle time!

Scenario 2

It was only when we used the same source traffic and introduced congestion into the system by reducing the bandwidth of the outgoing link to 6 pkts /sec, so each flow was given 3 pkts/sec (and thus *both* were misbehaving), that we saw the result we expected for the first scenario:



The above two scenarios showed that our original understanding was not completely accurate. VirtualClock does not strictly punish the "use of idle time."

4.4.2 VC Punishes Misbehaviour

What VC actually does punish is *misbehaviour*, and this punishment is only effected when congestion occurs. These two principles are shown in the following sections:

Misbehaviour: Auxiliary Virtual Clock in Traffic bursts (Scenario 1)

In 4.4.1 Scenario 1 (no congestion), both flows maintained their throughput (Fig. 4.4-4) and had no delay (Fig 4.4-6, below left). However, let us examine the Auxiliary Virtual Clock during that scenario (Fig. 4.4-7, below right):

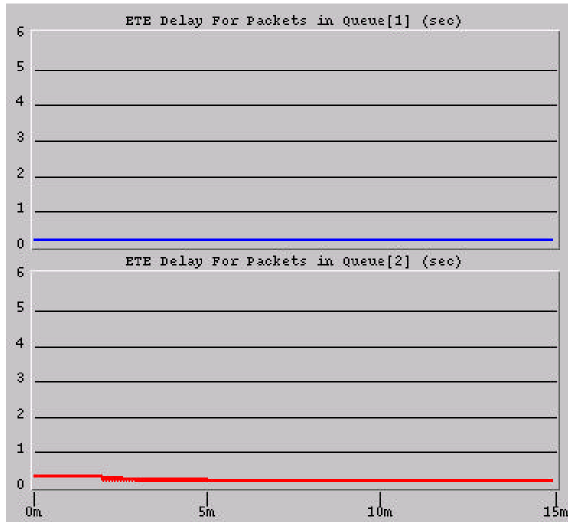


Figure 4.4-6 ETE delay in Scenario 2 (no congestion) with VC scheduling

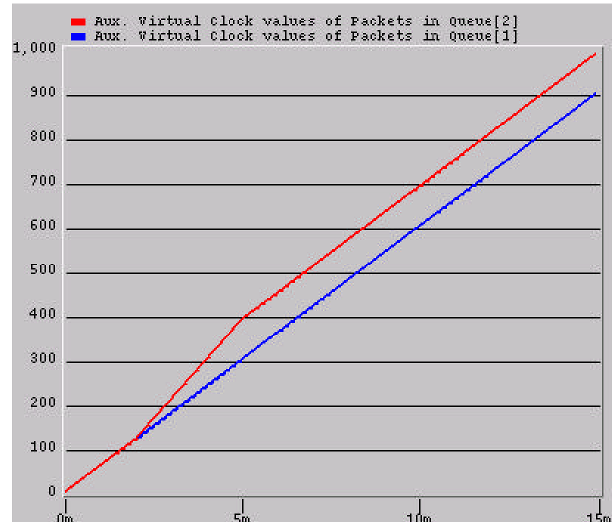


Figure 4.4-7 Throughput of flows in Scenario 2 (no congestion) with VC scheduling

Note that although Flow 2 appeared to be unaffected by misbehaving because it only used idle bandwidth, its Auxiliary Virtual Clock is now running fast because it incremented faster than agreed due to the increased arrival rate from time = 2 – 5 minutes. When the misbehaviour stopped, both internal clocks will increased at the same rate again; yet source 2's misbehaviour had been "remembered" by the values in these clocks.
This occurs whenever a flow misbehaves in VC.

Scenario 3 …When Congestion Occurs

To show the punishing effect that this results in, we will create a new Scenario 3 by *extending* Scenario 1, adding another 15 minutes to the original simulation. Recall that because the bandwidth of the link is 8 pkts/sec there has been no packet delay so far.

For the first 5 minutes both sources continue to behave by transmitting 4 pkts/sec. Now, at time = 20 minutes, source 1 takes a turn at misbehaving, transmitting 8 pkts/sec for 1 minute, causing congestion in the router. The full source traffic can be seen below in Fig. 4.4-8. We ran the simulation twice, once with VC and a second time with SFQ.

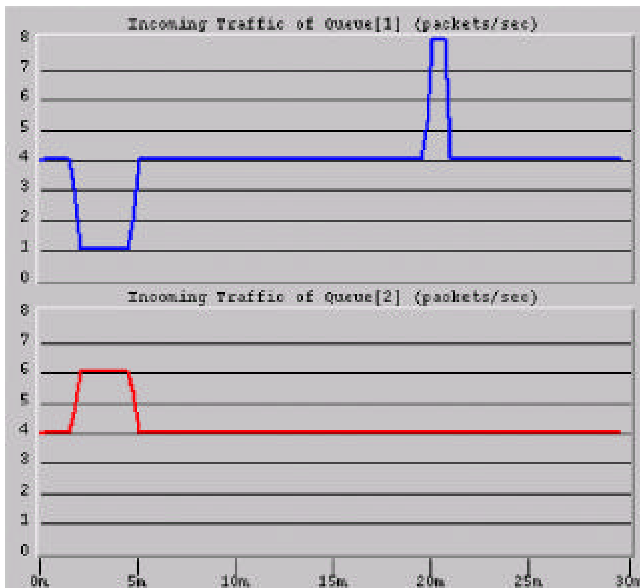


Figure 4.4-8 Source Traffic (Scenario 3)

The throughput and ETE delay for SFQ (Figs. 4.4-9 and 4.4-10, below) show that flow 1 is punished for misbehaving at time=20, creating a very large packet queue that delays packets that follow the burst:

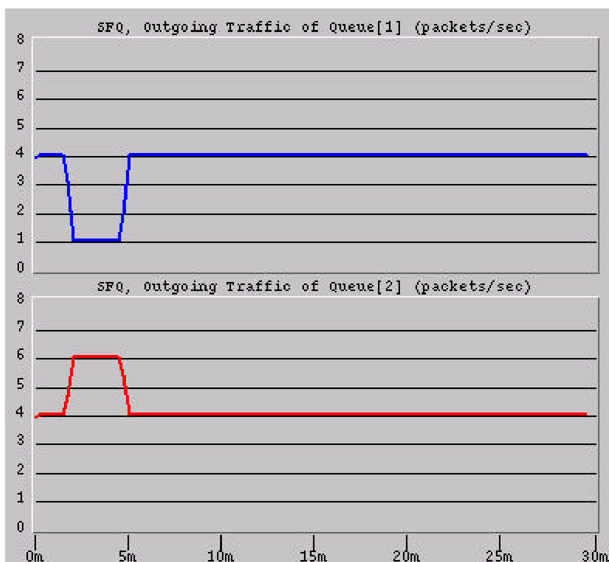


Figure 4.4-9 Throughput with SFQ (Scenario 3)

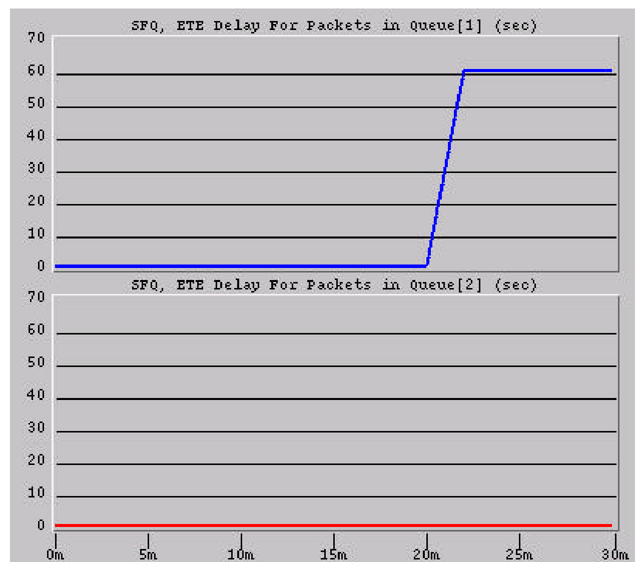
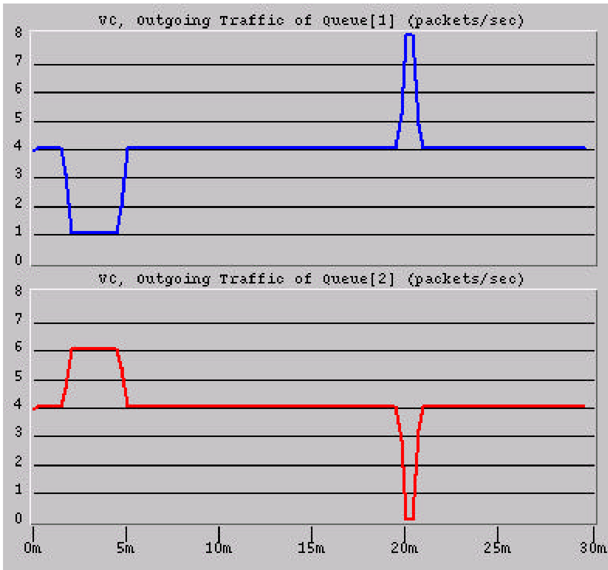


Figure 4.4-10 ETE delay with SFQ (Scenario 3)

This is fair behaviour according Goyal et al [Goy96], who believe that than bandwidth of each flow should be protected under all circumstances



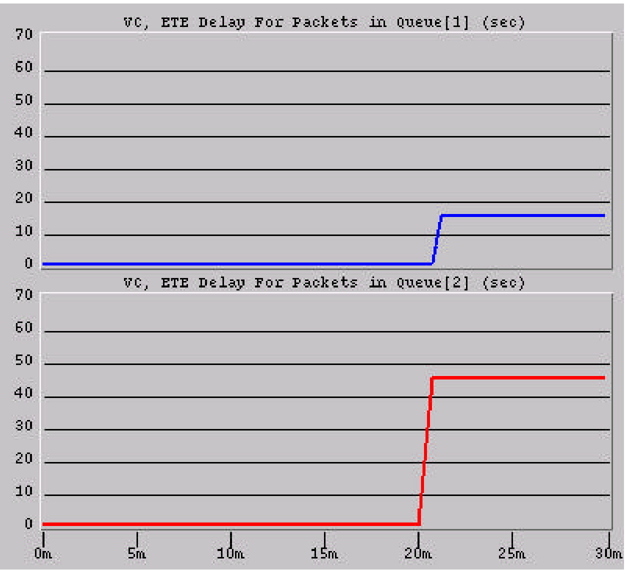


Figure 4.4-10 Throughput with VC (Scenario 3)

Figure 4.4-11 ETE delay with VC (Scenario 3)

Under VirtualClock scheduling the behaviour is very different, as shown in the throughput and ETE delay graphs above (Figs. 4.4-11 and 4.4-12). At the time when source 1 misbehaves, it is actually source 2 that is punished!
Source 2 is actually being punished for its misbehaviour 18 minutes earlier, which can be clearly seen in a graph of the values of auxiliary clocks of the 2 traffic flows:

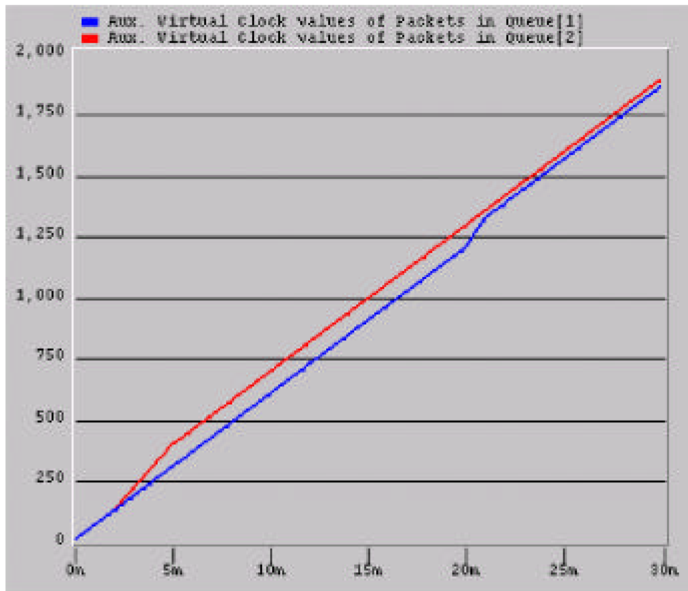


Figure 4.4-12 Auxiliary Virtual Clock of flows (Scenario 3)

As explained previously, source 2 has a higher auxiliary clock value due to an earlier misbehaviour of sending packets faster than the expected rate. During the time when both sources are transmitting packets conformingly, the gap between the auxiliary clocks stay the same; in a sense,VC remembers the misbehaviour of source 2. At the time of congestion, it was source 1 that

sent packets in a higher rate than usual. However, since its auxiliary clock value is lower than that of source 2, its packets will generally be scheduled before packets from source 2.
The punishing effect is due to the auxiliary clock value ‘running fast’ during misbehaviour. It does not take effect until there is congestion.

Thus we verified that VirtualClock does not punish the use of idle time, but it does “remember” misbehaviours that take place when there was no congestion, and then punishes them when congestion occurs.
The authors of SFQ believe that this is unfair. Some would not agree, arguing that misbehaviour should always be punished. The precise definition of fairness must be determined by the reader according to context of their particular scheduling application.

4.5 Punishing & Bandwidth for VBR flows

In Section 4.2.1 we saw that a consequence of fairness was that misbehaving flows should be punished, and in Section 4.4.2 we saw that VC punishes all misbehaviour, whereas SFQ does not punish misbehaviour that takes only idle bandwidth.

In their paper introducing SFQ, Goyal et al recognized the particular significance of VBR flows in today’s networks and identified the need to extend the notion of fairness in light of VBR behaviour:
Due to the difficulty in predicting the bit rate requirement of VBR video sources, video channels may utilize more than the reserved bandwidth. As long as the additional bandwidth used is not at the expense of other channels (i.e., if the channel utilizes idle bandwidth), it should not be penalized in the future by reducing its bandwidth allocation. [Goy96]
They relate this issue specifically to the punishing characteristic of VirtualClock and some other scheduling schemes. Let us examine the VC algorithm to see why they made this association.
Recall from section 4.3 and 4.4 above:
During the low bandwidth times, VC updates the flow’s auxiliary virtual clock to real time so that it cannot store up credits
During the high bandwidth times, VC allows the flow’s auxiliary virtual clock to go faster and get ahead of real time
If for a time the VBR source’s bandwidth is higher than the allocated bandwidth its auxiliary virtual clock will get ahead of real time just like a misbehaving flow. This characteristic can be verified by running a simple simulation.

Scenario 4

Source [1]	CBR source sending 4 pkts/sec
Source [2]	VBR source (Poisson) sending an average of 4 pkts/sec

The shared link has a bandwidth of 8.1 pkts/sec to guarantee no congestion, giving 4 pkts/sec to each flow (actually 4.05 pkts/sec).

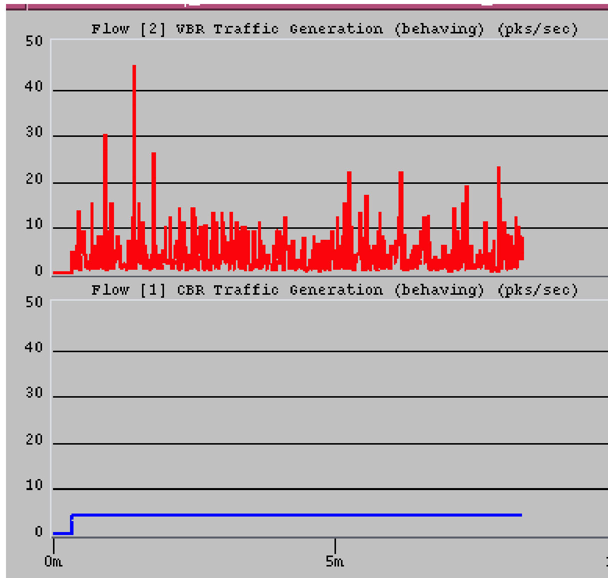


Figure 4.5-1 Source Traffic (Scenario 4)

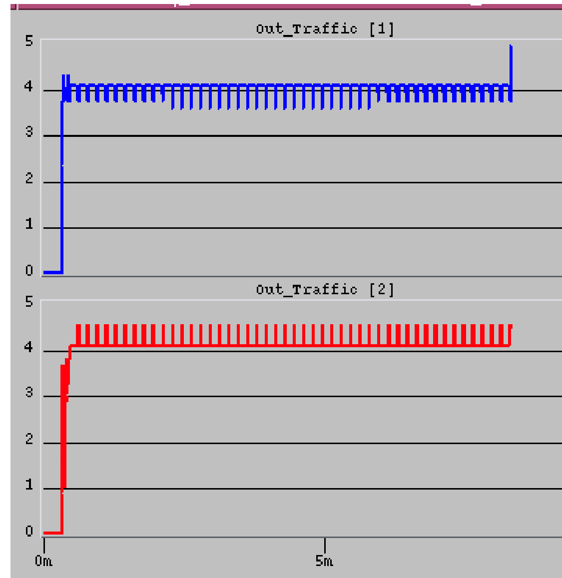


Figure 4.5-2 Throughput of flows (Scenario 4)

When there is no congestion, the throughput of both flows fair at 4 pkts/sec (Fig 4.5-2, above). However, the auxiliary virtual clock (Fig. 4.5-3, below) looks surprisingly similar to the misbehaving flow of Scenario 1 (compare to Fig. 4.4-7). The VBR flow's auxiliary virtual clock is 'running fast'.

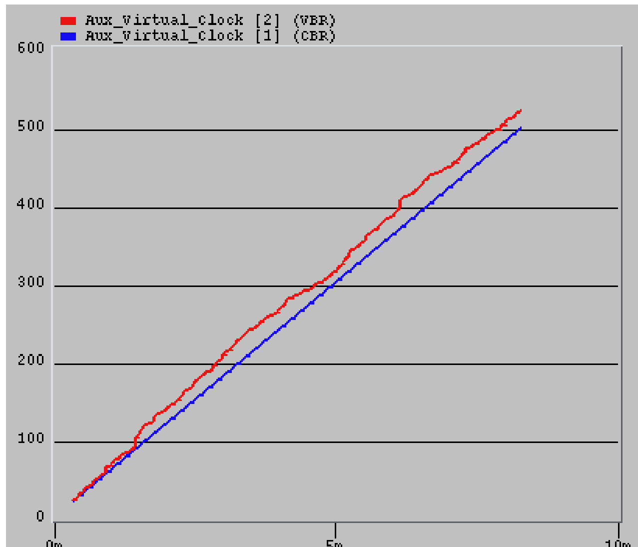


Figure 4.5-3 Auxiliary Virtual Clock of flows (Scenario 4)

Scenario 5 – VBR punishing in VC

If we introduce some congestion into Scenario 4, the VBR flow should be susceptible to punishment just like the misbehaving flow in Scenario 3.
We simply make the CBR flow - which was behaving in the above simulation - now intentionally misbehave by sending 9 pks/sec for a brief period as shown in Fig 4.4-4 (below):
The graphs of auxiliary virtual clock and throughput are included below:

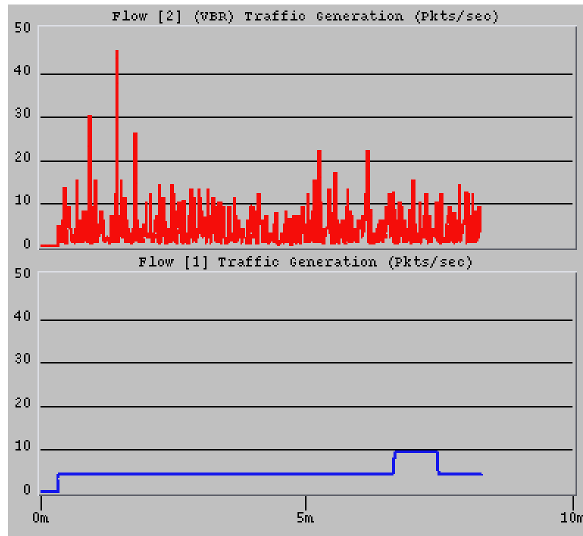


Figure 4.5-4 Source Traffic (Scenario 5)

Notice that the throughput of the behaving VBR source (red) drops as soon as there is congestion, until the auxiliary virtual clock of flow [2] 'catches up'.

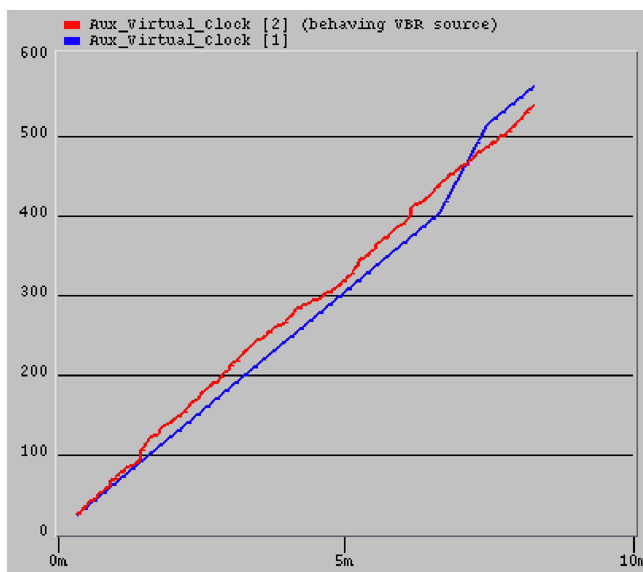


Figure 4.5-5 Auxiliary Virtual Clock of flows (Scenario 5)

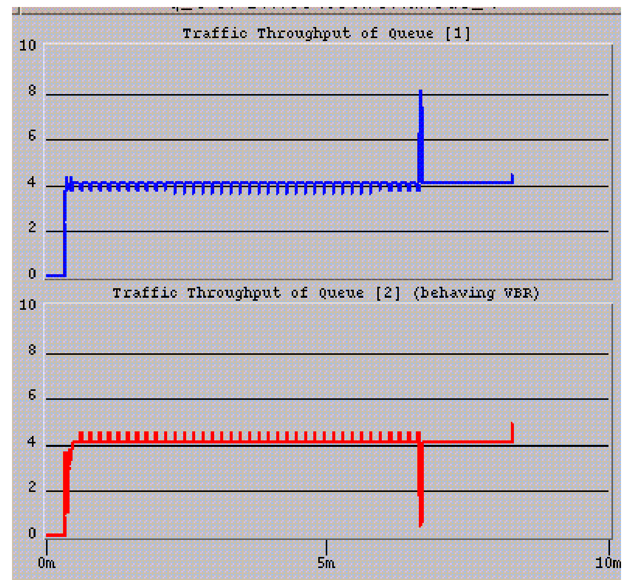


Figure 4.5-6 Throughput of flows (Scenario 5)

Scenario 6 – VBR non- punishing in SFQ

We ran the same setup as Scenario 5 with SFQ
The throughput shows how SFQ does not regard the VBR source (flow 2) as ‘misbehaving’ and thus does not punish it when congestion is introduces by flow 1.

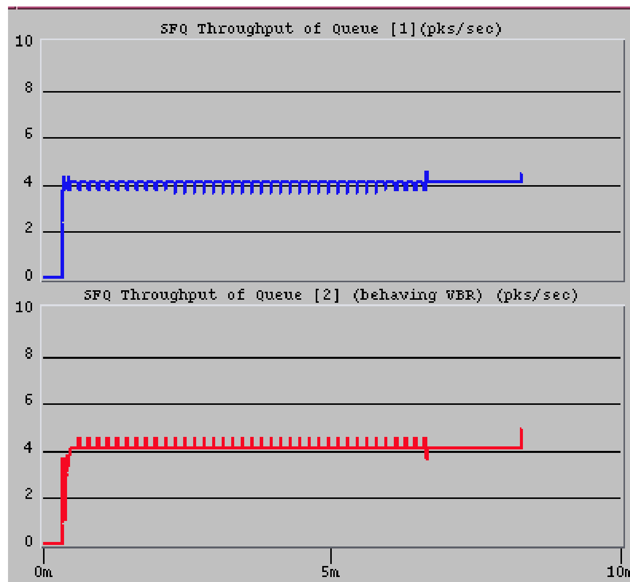


Figure 4.5-6 Auxiliary Virtual Clock of flows (Scenario 4)

VBR punishing: Conclusion

Scenario 5 and Scenario 6 illustrate the claim by Goyal et al that VBR sources whose average bandwidth is ‘behaving’ will be treated as misbehaving by the VirtualClock algorithm, but will not be punished by SFQ. [Goy96]

Again, the ‘fairness’ evaluation of these scheduling algorithms is left to the reader.

5 Discussion and Conclusion

In this course project, we studied various packet scheduling algorithms with a particular focus on the Start-Time Fair Queuing (SFQ) algorithm. We implemented SFQ as a Queue Model in Opnet™ 8.0 Modeller.

We ran simulations to verify the characteristics of SFQ and compared it to FIFO queuing and another Fair Queuing algorithm that we implemented as an Opnet Queue Model, VirtualClock (VC).
SFQ was shown

The major implementation contribution of this project was the Opnet™ model of SFQ, which can be reused in other research or simulation project in the future.

The major analysis contribution of this paper was the comparison of SFQ and VC approaches to scheduling sources which transmit more than their allotted share of service using idle bandwidth.

6 References

[Alb01] N. Alborz and L. Trajkovic, “Implementation of VirtualClock scheduling algorithm in OPNET,” OPNETWORK 2001, Washington, DC, Aug. 2001.
http://www.ensc.sfu.ca/~ljilja/papers/opnetwork01_nazy.pdf (14.Feb.02)

[Chu02] T.A. Chu, “Police Your Packets: Traffic Management Part 1; Keys to Implementation,” CommsDesign, February 1, 2002.
http://www.commsdesign.com/story/OEG20020201S0007 (14.Feb.02)

[Cis95] “Interface Queue Management,” Cisco Systems White Paper.
http://www.cisco.com/warp/public/614/16.html (14.Feb.02)

[Citeseer] NEC ResearchIndex “Citation details: Start-time fair queuing: A scheduling algorithm for integrated services packet switching.”
http://citeseer.nj.nec.com/context/23652/340968 (9.Apr.2002)

[Der90] A. Demers, S. Keshav, and S. Shenker, “Analsyis and Simulation of a Fair-queueing Algorithm,” In Proc. ACM SigComm 89, pp1-12, also to appear in Journal of Internetworking, Vol. 1, No. 1, 1990.
http://netweb.usc.edu/cs551/papers/Demers.pdf (14.Feb.02)

[Goy96] P. Goyal, H. Vin, and H. Chen. “Start-Time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks,” In Proceedings IEEE SIGCOMM'96, August 1996.
http://www.cs.columbia.edu/~danr/6762/week4/stfq.pdf (14.Feb.02)

[Mo99] J. Mo, R. J. La, V. Anantharam, and J. Warland, “Analysis and comparison of TCP Reno and Vegas,” IEEE INFOCOM, New York, May 1999.
http://www.ensc.sfu.ca/~ljilja/ENSC835/Assignments/papers/vegas.pdf (15. Feb.02)

[Nag84] J. Nagle. “Congestion Control in IP/TCP Internetworks,” Computer Communication Review, 14(4), October 1984.
http://www.acm.org/sigcomm/ccr/archive/1995/jan95/ccr-9501-nagle84.pdf (12.Mar.2002)

[Nag85] J. Nagle, “On packet switches with infinite storage,” RFC970, Dec-01-1985
http://globecom.net/ietf/rfc/rfc970.html (14.Feb.02)

[Par93] A. K. Parekh and R. G. Gallager. “A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-Node Case,” IEEE/ACM Transactions on Networking, 1(3):344--357, June 1993.
http://www.stanford.edu/class/ee384x/Papers/parekh.pdf (12.March.2002)

[Vil98] Curtis Villamizar. Message in the archives of the “Explicit Congestion Notification” Mailing List, Oct 1, 1998.
http://www-nrg.ee.lbl.gov/ecn-arch/msg00072.html (9.Apr.2002)

[Wal00] J. Walrand and P. Varaiya. High-performance Communication Networks. Second edition, San Francisco, Morgan Kaufmann, 2000, p. 78.

[Zhang90]] L. Zhang, “VirtualClock: a new traffic control algorithm for packet switching networks,” in Proc. ACM SIGCOMM, Sept. 1990.
http://portal.acm.org/citation.cfm?id=99525&coll=ACM&dl=ACM&CFID=2308333&CFTOKEN=23966879 (14.March.02)

Appendix A: VirtualClock Implementation Details

3.3.1 The *init* State

Convert the SFQ flow allocation notation, proportional weights, to the VC notation, "arrival rate" by taking
arrival_rate(flow) = weight(flow) / sum-of-weights-for-all flows
Initialize the variables and statistics.

3.3.2 The *arrive* State

Step 1: Get the incoming packet
Step 2: Get information about the packet: flow *f* that it is from, packet length, and arrival time in queue. Store the arrival time. If the flow has been assigned a weight of 0 then discard the packet.
Step 3: Check if this is the first packet to arrive on flow *f*, intialize the flow's variables ('virt_clock' and 'aux_virtual_clock', and mark 'first_packet_flag' to skip this step in to future.
Step 4: Set the variables used in the VC algorithmx, *v_clock*, and *a_v_clock*, and calculate *v_tick*
Step 5: If *a_v_clock* is less than the packet arrival time, set it equal. This prevents a flow from 'storing up credits' by transmitting under its arrival rate and then sending a burst.
Step 6: Increment and update 'virt_clock' and 'aux_virtual_clock'
Step 7: Create an ICI and assign values to the data fields: 'virtual_clock_stamp', 'aux_virtual_clock_stamp', and some statistics; attach the ICI to the packet.
Step 8: Attempt to queue the packet in the subqueue corresponding to flow *f*. If the server is not busy, go straight to the *svc_start* state; otherwise return to the idle state.

3.3.3 The *svc_start* State

Step 1: Determine which subqueue to serve. This is performed as follows:
Cycle through the subqueues, stop when you find the first non-empty queue
Access the packet at the head, get its *aux_virtual_clock_stamp* from the ICI
Set this stamp as "min" and the queue as "flow"
Loop through any remaining subqueues and for each non-empty subqueue
repeat (b)
If this start_tag is lower than *min*, update *min* and *flow*
flow is the subqueue to serve
Step 2: Get a pointer to the packet and retrieve its *aux_virtual_clock_stamp* from the ICI. Update the *queue_in_service* state variable.
Step 3: Calculate the time it will take to serve the packet (based on packet length) and schedule an interrupt for the time when service is completed.
Step 4: Update statistics
Step 5: Set the *server_busy* state variable to true and return to the idle state.

3.3.4 The *svc_complete* State

This state is entered by the self-scheduled interrupt from *svc_start* state, which indicates that the time required for packet service is completed.

Step 1: Remove the packet from its subqueue

Step 2: Update statistics

Step 3: Send the packet. If there are packets waiting to be sent, go straight to the *svc_start* state; otherwise return to the idle state.

Appendix B: Source Code

FIFO Queue Module (Opnet's acb_fifo)

```
/* Process model C form file: yy_acb_fifo.pr.c */
/* codes for the modified acb_fifo process model */


/* ***** State variable definitions ***** */
typedef struct
	{
	/* Internal state tracking for FSM */
	FSM_SYS_STATE
	/* State Variables */
	int					server_busy;
	double				service_rate;
	Objid				own_id;
	/* cmpt885 modification: State Variables for collecting statistics */
	Stathandle				In_Traffic_handle[8];
	Stathandle				Out_Traffic_handle[8];
	Stathandle				Out_Bit_Traffic_handle[4];
	Stathandle				In_Bit_Traffic_handle[4];
	} yy_acb_fifo_state;
/* ***** End of State variable definitions ***** */


/* ***** Temporary Variables definitions ***** */
		Packet*		pkptr;
		int			pk_len;
		double		pk_svc_time;
		int			insert_ok;

		/* 885 modification */
		int		flowID;
		Ici*		ici_ptr;
		int		i;
/* ***** End Temporary Variables definitions ***** */


/*-----------------------------------------------------------*/
/** state (init) enter executives **/

	/* initially the server is idle  */
	server_busy = 0;

	/* get queue module's own object id    */
	own_id = op_id_self ();

	/* get assigned value of server  */
	/* processing rate                     */
	op_ima_obj_attr_get (own_id, "service_rate", &service_rate);

	/* 885 modification */
	for (i=0; i<8; i++)
		In_Traffic_handle[i]=op_stat_reg("In_Traffic",i,OPC_STAT_LOCAL);

	for (i=0; i<8; i++)
		Out_Traffic_handle[i]=op_stat_reg("Out_Traffic",i,OPC_STAT_LOCAL);

	for (i=0; i<4; i++)
		Out_Bit_Traffic_handle[i]=op_stat_reg("Out_Bit_Traffic",i,OPC_STAT_LOCAL);

	for (i=0; i<4; i++)
		In_Bit_Traffic_handle[i]=op_stat_reg("In_Bit_Traffic",i,OPC_STAT_LOCAL);
```

```
/*--------------------------------------------------------*/

/** state (arrival) enter executives **/

      /* acquire the arriving packet                     */
      /* multiple arriving streams are supported. */

      /* 885 modification */
      /* pkptr = op_pk_get (op_intrpt_strm ()); */
      flowID = op_intrpt_strm ();
      pkptr = op_pk_get (flowID);

      ici_ptr= op_ici_create("yy_flow_stat_ici");
      op_ici_attr_set(ici_ptr, "incoming_flow", flowID);
      op_pk_ici_set(pkptr, ici_ptr);

      op_stat_write(In_Traffic_handle[flowID], 1.0);
      pk_len = op_pk_total_size_get (pkptr);
      op_stat_write(In_Bit_Traffic_handle[flowID], pk_len);


      /* attempt to enqueue the packet at tail      */
      /* of subqueue 0.                                      */
      if (op_subq_pk_insert (0, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK)
          {
          /* the inserton failed (due to to a       */
             /* full queue) deallocate the packet.  */
          op_pk_destroy (pkptr);

             /* set flag indicating insertion fail  */
             /* this flag is used to determine             */
             /* transition out of this state        */
          insert_ok = 0;
          }
      else{
             /* insertion was successful                   */
          insert_ok = 1;
          }

/*--------------------------------------------------------*/

/** state (svc_compl) enter executives **/

      /* extract packet at head of queue; this      */
      /* is the packet just finishing service       */
      pkptr = op_subq_pk_remove (0, OPC_QPOS_HEAD);

      ici_ptr=op_pk_ici_get(pkptr);
      op_ici_attr_get(ici_ptr, "incoming_flow", &flowID);

      op_stat_write(Out_Traffic_handle[flowID], 1.0);

      pk_len = op_pk_total_size_get (pkptr);
      op_stat_write(Out_Bit_Traffic_handle[flowID], pk_len);

      /* forward the packet on stream 0, causing    */
      /* an immediate interrupt at destination.     */
      op_pk_send_forced (pkptr, 0);

      /* server is idle again.                             */
      server_busy = 0;
/*--------------------------------------------------------*/
```

SFQ Queue Module

```
/* Process model C form file: sfq.pr.c */
/* codes for the SFQ Queue Model (Process Model) */





/* ***** Header Block ***** */

#define QUEUE_EMPTY      (op_q_empty ())
#define SVC_COMPLETION   op_intrpt_type () == OPC_INTRPT_SELF
#define ARRIVAL          op_intrpt_type () == OPC_INTRPT_STRM

/* SFQ */
// stdlib.h for max(a,b) function
//#include   <stdlib.h>
#define max(a,b) (((a)>(b))?(a):(b))
#define DEBUG       0
#define DEBUG2      1
#define DEBUG3      0
/* ***** End of Header Block ***** */



/* ***** State variable definitions ***** */
typedef struct
	{
	/* Internal state tracking for FSM */
	FSM_SYS_STATE
	/* State Variables */
	int				server_busy;
	double			service_rate;
	Objid			own_id;
	double			start_tag_in_service;
	double			max_finish_tag;
	double*				prev_finish_tag;
	double*				flow_weight;
	int				queue_in_service;
	int				num_subqs;
	Stathandle			bits_sent_hndl[4];
	double			bits_sent_count[4];
	Stathandle			q_delay_hndl[4];
	Stathandle			In_Traffic_stathandle[8];
	Stathandle			Out_Traffic_stathandle[8];
	Stathandle			packets_sent_hndl[4];
	double			packets_sent_count[4];
	Stathandle			In_Bit_Traffic_stathandle[4];
	Stathandle			Out_Bit_Traffic_stathandle[4];
	} sfq_state;

/* ***** End of State variable definitions ***** */



/* ***** Function Block ***** */
// double "virtual_time()" returns the value of SFQ's 'virtual time'
  // Note that according to the authors (Goyal et al, 1996) you could reset
  // the start_tag and finish_tag to zero after a non-busy period

double virtual_time(double arr_time)
	{
	if (server_busy) // "BUSY PERIOD"
		{
		return start_tag_in_service;
```

```
		}
	else // AFTER A NON-BUSY PERIOD
		{
		return max_finish_tag;
		}
	} // end virtual_time()

// double "get_start_tag()" accesses the packet's ICI and returns
// the value of the "start_tag" field

double get_start_tag(int q_index)
	{
	Packet*		pkptr;
	double	start_tag;
	Ici*	ici_ptr;

	/* Get the start_tag of head packet and set variables */
	pkptr = op_subq_pk_access (q_index, OPC_QPOS_HEAD);
	ici_ptr = op_pk_ici_get(pkptr);
	op_ici_attr_get(ici_ptr,"start_tag", &start_tag);

	return start_tag;
	} // end "get_start_tag()"


/* int "q_with_best head()" returns the subqueue id (a numerical
   index starting from 0) of the queue in which the packet with the
   lowest start_tag is found. This is found by examining the value
   of the "start_tag" for the packet at the head of each subqueue.   */
/* If two queues have the same start tag, the decision is arbitrary;
   We choose the lower queue number first.                           */

int q_with_best_head ()
	{
	int		q_index;
	double	start_tag;
	double	min_start_tag;
	int		min_q;

	// Find the first  non-empty queue and set is as min
	for (q_index = 0; q_index<num_subqs && op_subq_empty(q_index)==OPC_TRUE; q_index++)
		{
		} // end for
	min_start_tag = get_start_tag(q_index);
	min_q = q_index;

	/* Then loop through any remaining subqueues and update min if necessary*/
	for (q_index=q_index+1; q_index < num_subqs; q_index++)
		{
		if (op_subq_empty(q_index) == OPC_FALSE)
			{
			/* Examine start_tag of head packet and update min if new lowest */
			start_tag = get_start_tag(q_index);
			if (start_tag < min_start_tag)
				{
				min_start_tag = start_tag;
				min_q = q_index;
				}
			} // end if subqueue not empty
		} // end for q_index
	return min_q;
	} //end q_with_best_head ()

// This function simply passes Opnet a warning message and ends the
// simulation.

static void sfq_warning_message_print (char* message)
	{
	/** Ends the simulation and print a error message.  **/
	FIN (sfq_warning_message_print (message));
```

```
	op_sim_end ("Error in sfq process model\n", message, "", "");
	FOUT;
	}

/* ***** End of Function Block ***** */

/* ***** Temporary Variable definitions ***** */
		Packet*		pkptr;
		int			pk_len;
		double		pk_svc_time;
		int			insert_ok;

		/* Added for SFQ */
		int		i;				// For cycling through for loops!
		double	arrival_time;
		int		flow;
		Ici*	ici_ptr;		// The ICI is the information that holds start and
								// finish tags for each packet
		double	start_tag;
		double	finish_tag;
		double	pk_arrival_time;	// Used for statistics
		Objid	objid;				// Used to grab information from the node
		int		flowID;
/* ***** End of Temporary Variable definitions ***** */




/*--------------------------------------------------------*/
/** state (init) enter executives **/

	/* Initially the server is idle  */
	server_busy = 0;

	/* Get queue module's own object id    */
	own_id = op_id_self ();

	/* Get assigned value of server processing rate				*/
	op_ima_obj_attr_get (own_id, "service_rate", &service_rate);


	/* Get the number of subqueues and flow weights from node properties */
	op_ima_obj_attr_get (op_id_self (), "subqueue", &objid);
	num_subqs = op_topo_child_count (objid, OPC_OBJTYPE_SUBQ);

	/* Allocate memory for the arrays */
	flow_weight = (double*) malloc(num_subqs* sizeof(double));
	prev_finish_tag = (double*) malloc(num_subqs* sizeof(double));

	/* Initialize variables */
	if (op_ima_obj_attr_get (own_id, "weight_flow0", &flow_weight[0]) == OPC_COMPCODE_FAILURE)
	flow_weight[0] = 0.0;
	if (op_ima_obj_attr_get (own_id, "weight_flow1", &flow_weight[1]) == OPC_COMPCODE_FAILURE)
	flow_weight[1] = 0.0;
	if (op_ima_obj_attr_get (own_id, "weight_flow2", &flow_weight[2]) == OPC_COMPCODE_FAILURE)
	flow_weight[2] = 0.0;
	if (op_ima_obj_attr_get (own_id, "weight_flow3", &flow_weight[3]) == OPC_COMPCODE_FAILURE)
	flow_weight[3] = 0.0;

	for(i=0; i<num_subqs; i++)prev_finish_tag[i] = 0;
	for(i=0; i<num_subqs; i++)packets_sent_count[i] = 0;
	for(i=0; i<num_subqs; i++)bits_sent_count[i] = 0;
	max_finish_tag = 0;
	start_tag_in_service = 0;

	// Register statistics
	for (i=0; i<4; i++)
		{
```

```
		q_delay_hndl[i] = op_stat_reg ("Queuing Delay for Packets Leaving Flow",i, 
	OPC_STAT_LOCAL);
		bits_sent_hndl[i] = op_stat_reg ("Throughput of Flow (bits)",i, OPC_STAT_LOCAL);
		packets_sent_hndl[i] = op_stat_reg ("Throughput of Flow (packets)",i, 
	OPC_STAT_LOCAL);
		In_Traffic_stathandle[i]=op_stat_reg("In_Traffic",i,OPC_STAT_LOCAL);
		Out_Traffic_stathandle[i]=op_stat_reg("Out_Traffic",i,OPC_STAT_LOCAL);
		In_Bit_Traffic_stathandle[i]=op_stat_reg("In_Bit_Traffic",i,OPC_STAT_LOCAL);
		Out_Bit_Traffic_stathandle[i]=op_stat_reg("Out_Bit_Traffic",i,OPC_STAT_LOCAL);
	}


/*--------------------------------------------------------*/

/** state (arrival) enter executives **/

	/* Acquire the arriving packet                      */
	if ((pkptr = op_pk_get (op_intrpt_strm ()) ) == OPC_NIL)
		{
		/* Stop the simulation if the packet can not be accessed.  */
		sfq_warning_message_print ("Unable to get the packet from interrupt stream");
	    }

	/* Determine the flow and the packet length (in bits) */
	flow = op_intrpt_strm();
	if (flow_weight[flow] == 0.0)
	    {
		// Packet arrived from a flow not set to be serviced
		/* set flag indicating no packet was queued (for transition out of state) */
	    op_pk_destroy (pkptr);
	    insert_ok = 0;
	    }
	else{
		if (DEBUG3) printf("Arrived on flow %i\n", flow);


		/* Get information about the packet for SFQ algorithm*/
		pk_len = op_pk_total_size_get (pkptr);
		arrival_time = op_sim_time ();

		/* Determine Start time and Finish Time for SFQ algorithm,
		   and update finish tag for next packet that arrives      */
		start_tag = max ( virtual_time(arrival_time), prev_finish_tag[flow] );
		finish_tag = start_tag + pk_len/flow_weight[flow];
		prev_finish_tag[flow] = finish_tag;

		/* Create ICI to store tags and Assign SFQ information to it*/
		ici_ptr = op_ici_create("sfq");
		op_ici_attr_set(ici_ptr,"start_tag", start_tag );
		op_ici_attr_set(ici_ptr,"finish_tag", finish_tag );
		// Add information for statistics too
		op_ici_attr_set(ici_ptr,"Arrival time in queue", arrival_time);
		op_ici_attr_set(ici_ptr,"incoming_flow", flow);
		// Associate ICI with packet
		op_pk_ici_set(pkptr, ici_ptr);

		/* Attempt to enqueue  packet at tail of subqueue corresponding to the flow */
		if (op_subq_pk_insert (flow, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK)
			{
			/* the inserton failed (due to to a    */
			/* full queue) deallocate the packet.  */
			op_pk_destroy (pkptr);

			/* set flag indicating insertion fail  */
			/* this flag is used to determine             */
			/* transition out of this state        */
			insert_ok = 0;
			}
		else
			{
```

```
                /* insertion was successful                    */
                insert_ok = 1;
                } // end if() for packet insertion
        } // end if() for empty flow


        /* Write SFQ statistics */
        op_stat_write (In_Traffic_stathandle[flow], 1.0);
        op_stat_write (In_Bit_Traffic_stathandle[flow], pk_len);
/*------------------------------------------------------------*/

/** state (svc_start) enter executives **/

                        /* Determine which flow subqueue to service */
                        flow = q_with_best_head();

                        /* Get a handle on packet at head of the chosen subqueue */
                        /* (this does not remove the packet)                      */
                        /* Then get information about packet (Start Tag and  packet length) */
                        pkptr = op_subq_pk_access (flow, OPC_QPOS_HEAD);
                        pk_len = op_pk_total_size_get (pkptr);
                        ici_ptr = op_pk_ici_get(pkptr);
                        op_ici_attr_get(ici_ptr,"start_tag", &start_tag);

                        /* Update state variables */
                        start_tag_in_service = start_tag;
                        queue_in_service = flow;


                        /* Determine the time required to complete service of the packet */
                        /* (Depends on the length of the packet and the service rate     */
                        /* available to the router                                       */
                        pk_svc_time = pk_len / service_rate;

                        /* Schedule an interrupt for this process at the time where service
        ends.*/
                        op_intrpt_schedule_self (op_sim_time () + pk_svc_time, 0);

                        // Pass statistics for queue throughput
                        op_stat_write(bits_sent_hndl[flow], pk_len);
                        packets_sent_count[flow] += 1;
                        op_stat_write(packets_sent_hndl[flow], 1.0);


                        /* Set the server as "busy"            */
                        server_busy = 1;

/*------------------------------------------------------------*/
/** state (svc_compl) enter executives **/

/* Extract packet at head of queue; this is the packet just finishing service */
        /* (this removes the packet) */
        pkptr = op_subq_pk_remove (queue_in_service, OPC_QPOS_HEAD);



        /* Get SFQ info from packet and update state variable value if necessary */
        ici_ptr = op_pk_ici_get(pkptr);
        op_ici_attr_get(ici_ptr,"finish_tag", &finish_tag);
        if (finish_tag > max_finish_tag)
                {
                max_finish_tag = finish_tag;     // maximum of all packets served so far
                }

        // Update Statistics
        op_ici_attr_get(ici_ptr,"incoming_flow", &flowID); // flow ID for outgoing packet
        op_stat_write(Out_Traffic_stathandle[flowID], 1.0);
        pk_len = op_pk_total_size_get (pkptr);
        op_stat_write(Out_Bit_Traffic_stathandle[flowID], pk_len);
        op_ici_attr_get(ici_ptr,"Arrival time in queue", &pk_arrival_time);
```

```
	op_stat_write (q_delay_hndl[queue_in_service], op_sim_time () - pk_arrival_time);

	/* Rorward the packet on stream 0, causing an immediate interrupt at destination. */
	op_pk_send_forced (pkptr, 0);

	/* server is idle again.                              */
	server_busy = 0;
/*----------------------------------------------------------*/


/* Process model C form file: virtual_clock.pr.c */
/* codes for VC Queue Model (Process Model) */




/* ***** Header Block ***** */

#define QUEUE_EMPTY     (op_q_empty ())
#define SVC_COMPLETION  op_intrpt_type () == OPC_INTRPT_SELF
#define ARRIVAL         op_intrpt_type () == OPC_INTRPT_STRM

/* SFQ */
// stdlib.h for max(a,b) function
//#include   <stdlib.h>
#define max(a,b) (((a)>(b))?(a):(b))
#define DEBUG      0
#define DEBUG2     0
#define DEBUG3     0
#define DEBUG4     0

/* ***** End of Header Block ***** */







/* ***** State variable definitions ***** */
typedef struct
	{
	/* Internal state tracking for FSM */
	FSM_SYS_STATE
	/* State Variables */
	int                             	server_busy;
	double                          service_rate;
	Objid                           own_id;
	double                          arrival_rate[4];
	int                             	queue_in_service;
	int                             	num_subqs;
	Stathandle                      	bit_count_stat;
	double                          bit_count;
	double                          virt_clock[4];
	double                          aux_virtual_clock[4];
	int                             	first_packet_flag[4];
	double                          flow_weight[4];
	Stathandle                      	In_Traffic_stathandle[4];
	Stathandle                      	Out_Traffic_stathandle[4];
	Stathandle                      	vc_stathandle[4];
	Stathandle                      	aux_vc_stathandle[4];
	Stathandle                      	Out_Traffic_bandwidth_handle[4];
	Stathandle                      	In_Bit_Traffic_handle[4];
	Stathandle                      	Out_Bit_Traffic_handle[4];
	} virtual_clock_state;

/* ***** End of State variable definitions ***** */
```

```
/* ***** Function Block **** */


double get_aux_vc_stamp(int q_index)
	{
	double aux_vc_stamp;
	Ici*	ici_ptr;
	Packet*		pkptr;

	/* Get the start_tag of head packet and set variables */
	pkptr = op_subq_pk_access (q_index, OPC_QPOS_HEAD);
	ici_ptr = op_pk_ici_get(pkptr);
	op_ici_attr_get(ici_ptr,"aux_virtual_clock_stamp", &aux_vc_stamp);

	if (DEBUG3) printf("Queue [%i] aux_VC_stamp = %f\n",q_index, aux_vc_stamp);

	return aux_vc_stamp;
	} // end "get_aux_time_stamp()"

void pk_send_statistics(Ici* ici_ptr)
	{
	/*
		// Statistics are not written when the statistic dimension is exceeded.
	if ( 1 ) // @@@@@ should be if statistic_index < stat_index_max) @@@@@
		{
		// Write the statistic for queuing delay.
		op_ici_attr_get (ici_ptr, "arrival_time", &pk_arrival_time);
		stat_info_ptr = op_prg_list_access (stat_info_list_ptr, q_index);
		op_stat_write (stat_info_ptr->queuing_delay_stathandle, op_sim_time () -
	pk_arrival_time);

		// Write out a new data point for the "Traffic Sent" statistic
		// under the 'IP Interface' group.
		packet_size = op_pk_total_size_get (sending_packet_ptr);
		op_stat_write (stat_info_ptr->traffic_sent_in_pps_stathandle, 1.0);
		op_stat_write (stat_info_ptr->traffic_sent_in_bps_stathandle, packet_size);

		// Write out a zero value to signal the end of the duration to hold
		// the statistic at the previously written out value.
		op_stat_write (stat_info_ptr->traffic_sent_in_pps_stathandle, 0.0);
		op_stat_write (stat_info_ptr->traffic_sent_in_bps_stathandle, 0.0);
	  }

	*/
	} // end pk_send_statistics

void register_stats()
	{
	//virtual_clock_stathandle = op_stat_reg ( "IP Interface.Virtual Clock (Sec)", stat_index,
	OPC_STAT_LOCAL);
	} //end register_stats()

static void sfq_warning_message_print (char* message)
	{
	op_sim_end ("Error in sfq process model\n", message, "", "");
	} // end warning message


/* ***** End of Function Block ***** */
```

VC Queue Module

```
/* ***** Temporary Variables definitions ***** */

	Packet*		 pkptr;
	int		pk_len;
	double		pk_svc_time;
	int		insert_ok;

	/* Added for SFQ */
	int		i;
	double	arrival_time;
	double	total_weight;
	int		flow;
	Ici*	ici_ptr;		// The ICI is the information that holds start and
						// finish tags for each packet
	double	start_tag;
	double	finish_tag;
	Objid	objid;		// Used to grab information from the node (# and weights of
subqueues(
	char*	message;



	/* Variables added to use Nazy's  Virtual Clock algorithm		*/
	int		q_index;
	double	aux_vc_stamp;
	double	min;
	int		min_vc_q_index;
	double	pk_arrival_time;

	/* Variables borrowed from Nazy's Virtual Clock algorithm		*/
	double							v_clock;
	double						a_v_clock = 0;
	double						c_v_clock = 0;
	double						arr_rate;
	double						v_tick = 0;
	int								no_eqvc_queues;
	int								j;
	int								active_q_index[10];
	Packet *						packetptr;
	double						virtual_clock_stamp;
	double						a_virtual_clock_stamp;

	/* 885 modification */
	int flowID;

/* ***** Temporary Variables definitions ***** */

/*-----------------------------------------------------------*/
/** state (init) enter executives **/

	/* initially the server is idle  */
	server_busy = 0;

	/* get queue module's own object id	*/
	own_id = op_id_self ();

	/* get assigned value of server  */
	/* processing rate			*/
	op_ima_obj_attr_get (own_id, "service_rate", &service_rate);


	/* Determine the number of subqueues and flow weights from node properties */
	op_ima_obj_attr_get (own_id, "subqueue", &objid);
	num_subqs = op_topo_child_count (objid, OPC_OBJTYPE_SUBQ);
```

```
	if (op_ima_obj_attr_get (own_id, "weight_flow0", &flow_weight[0]) == OPC_COMPCODE_FAILURE)
	flow_weight[0] = 0.0;
	if (op_ima_obj_attr_get (own_id, "weight_flow1", &flow_weight[1]) == OPC_COMPCODE_FAILURE)
	flow_weight[1] = 0.0;
	if (op_ima_obj_attr_get (own_id, "weight_flow2", &flow_weight[2]) == OPC_COMPCODE_FAILURE)
	flow_weight[2] = 0.0;
	if (op_ima_obj_attr_get (own_id, "weight_flow3", &flow_weight[3]) == OPC_COMPCODE_FAILURE)
	flow_weight[3] = 0.0;


	/* Convert SFQ flow weights (proportional) to VC arrival rate (absolute) */
	total_weight = flow_weight[0]+flow_weight[1]+flow_weight[2]+flow_weight[3];
	for(i=0; i<4; i++)
		{
		arrival_rate[i] = ( service_rate * flow_weight[i]/total_weight );
		} // end for

	/* Notify "weight" (SFQ notation) --> "arrival-rate" (VC notation) conversion */
	printf("For flow weights:\n");
	for(i=0; i<4; i++) printf("\tflow_weight[%i] = %f\n", i, flow_weight[i]);
	printf("And total router service rate = %f\n",service_rate);
	printf("Virtual Clock has been assigned arrival rates of:\n");
	for(i=0; i<4; i++) printf("\tarrival_rate[%i] = %f\n", i, arrival_rate[i]);


	/* Initialize variables */
	for(i=0; i<4; i++)
		{
		first_packet_flag[i] = 0;		// Will be set to 1 when the first packet arrives on 
	that flow
		}

	/* Register statistics */
	for (i=0; i<4; i++)
		{
		vc_stathandle[i]=op_stat_reg("Virtual_Clock",i,OPC_STAT_LOCAL);
		aux_vc_stathandle[i]=op_stat_reg("Aux_Virtual_Clock",i,OPC_STAT_LOCAL);
		In_Traffic_stathandle[i]=op_stat_reg("In_Traffic",i,OPC_STAT_LOCAL);
		Out_Traffic_stathandle[i]=op_stat_reg("Out_Traffic",i,OPC_STAT_LOCAL);

		Out_Traffic_bandwidth_handle[i]=op_stat_reg("Out_Traffic_bandwidth",i,OPC_STAT_LOCAL
	);
		In_Bit_Traffic_handle[i]=op_stat_reg("In_Bit_Traffic",i,OPC_STAT_LOCAL);
		Out_Bit_Traffic_handle[i]=op_stat_reg("Out_Bit_Traffic",i,OPC_STAT_LOCAL);
		} // End for() -- statistics registering




/*-----------------------------------------------------------*/
/** state (arrival) enter executives **/

	if (DEBUG==1) printf("\t\t\t<--arrival-->\n");

	/* Acquire the arriving packet 				*/
	if ((pkptr = op_pk_get (op_intrpt_strm ()) ) == OPC_NIL)
		{
		/* Stop the simulation if the packet can not be accessed.  */
		sfq_warning_message_print ("Unable to get the packet from interrupt stream");
		}

	/* Determine the flow and the packet length (in bits) */
	flow = op_intrpt_strm();
	if (flow_weight[flow] == 0.0)
		{
		// Packet arrived from a flow not set to be serviced
		/* set flag indicating no packet was queued (for transition out of state) */
		op_pk_destroy (pkptr);
		insert_ok = 0;
```

```
	}
else{
	if (DEBUG3) printf("Arrived on flow %i\n", flow);


	/* attempt to insert packet*/
	pk_len = op_pk_total_size_get (pkptr);
	arrival_time = op_sim_time ();

	/* If this is the first packt comming into the queue initialize the Virtual Clock */

	if ( first_packet_flag[flow] == 0 )
		{
		virt_clock[flow] = arrival_time;
		aux_virtual_clock[flow] = arrival_time;
		first_packet_flag[flow] = 1;
		}

	/* initialization of virtual clock and auxilary virtual clock
	*/
	//v_tick = 1 / arrival_rate[flow];
	v_tick = pk_len / arrival_rate[flow];
	v_clock = virt_clock[flow];
	a_v_clock = aux_virtual_clock[flow];

	/* Calculation of auxilary virtual clock      */
	if ( arrival_time > a_v_clock )
		{
		a_v_clock = arrival_time;
		}

	/* Update and store the flow's virt_clock and aux_virtual_clock
	   for the next packet arrival                                     */
	v_clock = v_clock + v_tick;
	a_v_clock = a_v_clock + v_tick;
	virt_clock[flow] = v_clock;
	aux_virtual_clock[flow] = a_v_clock;

		/* NOTE: The virt_clock value is used by the Virtual Clock algorithm
	   to warn a source if it is transmitting more than its assigned rate.
	   In this implementation we use one-way traffic so the virt_clock is
	   not needed, but is available as a future resource
	*/

	/* The virtual Clock and Auxilary Virtual Clock stamps are inserted in */
	/* the ICI. They are used in the extract state for choosing a queue    */
	/* containing a packet with the lowest Auxilary Virtual Clock Value    */

	/* Create ICI to store tags and Assign VC information to it*/
	ici_ptr = op_ici_create("virtual_clock");
	op_ici_attr_set (ici_ptr, "virtual_clock_stamp",virt_clock[flow] );
	op_ici_attr_set (ici_ptr, "aux_virtual_clock_stamp",aux_virtual_clock[flow] );
	op_ici_attr_set (ici_ptr, "incoming_flow", flow);
	op_pk_ici_set (pkptr, ici_ptr);

	op_stat_write(In_Traffic_stathandle[flow], 1.0);
	op_stat_write(In_Bit_Traffic_handle[flow], pk_len);
	op_stat_write(vc_stathandle[flow], virt_clock[flow]);
	op_stat_write(aux_vc_stathandle[flow], aux_virtual_clock[flow]);



	/* Attempt to enqueue  packet at tail of subqueue corresponding to the flow */
	if (op_subq_pk_insert (flow, pkptr, OPC_QPOS_TAIL) != OPC_QINS_OK)
		{
		/* the inserton failed (due to to a    */
		/* full queue) deallocate the packet.  */
		op_pk_destroy (pkptr);

		/* set flag indicating insertion fail  */
```

```
                /* this flag is used to determine            */
                /* transition out of this state        */
                insert_ok = 0;
                }
        else
                {
                /* insertion was successful                  */
                insert_ok = 1;
                } // end if() for packet insertion
        } // end if() for empty flow




/*-----------------------------------------------------------*/
/** state (svc_start) enter executives **/

        if (DEBUG==1) printf("\t\t\t<--svc_start-->\n");



        /* Loop through all the queues to find a queue */
        /* with lowest Auxilary Virtual Clock value    */

        // Find the first  non-empty queue and set is as min
        for (q_index = 0; q_index<num_subqs && op_subq_empty(q_index)==OPC_TRUE; q_index++)
                { // Go until find non-empty queue
                } // end for
        min = get_aux_vc_stamp(q_index);
        min_vc_q_index = q_index;

        /* Then loop through any remaining subqueues and update min if necessary*/
        for (q_index=q_index+1; q_index < num_subqs; q_index++)
                {
                if (op_subq_empty(q_index) == OPC_FALSE)
                        {
                        /* Examine start_tag of head packet and update min if new lowest */
                        aux_vc_stamp = get_aux_vc_stamp(q_index);
                        if (aux_vc_stamp < min)
                                {
                                min = aux_vc_stamp;
                                min_vc_q_index = q_index;
                                }
                        } // end if subqueue not empty
                } // end for q_index

        /* Loop through all the queues to find out if there are more than one queue with */
        /* the same min Auxilary Virtual Clock. Then fill out the array that contains    */
        /* the index of the queues with minimum equal Auxilary Virtual Clock             */
        /* SFQ NOTE: THIS LOOP WAS WRITTEN BY NAZY AND IT'S NOT CLEAR TO US WHAT IT DOES */

        /* ---SO I'M GETTING RID OF IT AND DOING THE EQUIVALENT, AS FAR AS I CAN SEE ----*/
        /*
        for (q_index=0; q_index < num_subqs; q_index++)
                {
                if (op_subq_empty(q_index) == OPC_FALSE)
                {
                        aux_vc_stamp = get_aux_vc_stamp(q_index);
                        if ( min == aux_vc_stamp )
                                {
                                j = 0;
                                no_eqvc_queues =  0;
                                active_q_index[j] =  q_index;
                                j++;
                                no_eqvc_queues++;
                                }
                        }
                }
```

```
	// Choose the queue to service based on what we've found
	if ( no_eqvc_queues>1 )
		{
		flow = active_q_index[0];
		}
	else
		{
		flow = min_vc_q_index;
		}
	*/
	flow = min_vc_q_index;  // Replaced above loop with this line


	/* get a handle on packet at head of subqueue 0 */
	/* (this does not remove the packet)		*/
	pkptr = op_subq_pk_access (flow, OPC_QPOS_HEAD);

	/* Update state variables */
	queue_in_service = flow;

	/* determine the packets length (in bits)		*/
	pk_len = op_pk_total_size_get (pkptr);

	/* determine the time required to complete		*/
	/* service of the packet					*/
	pk_svc_time = pk_len / service_rate;

	/* schedule an interrupt for this process		*/
	/* at the time where service ends.			*/
	op_intrpt_schedule_self (op_sim_time () + pk_svc_time, 0);

	// Pass statistics for queue throughput
	/*
	bit_count = bit_count + pk_len;
	op_stat_write(bit_count_stat, bit_count);
	*/

	/* the server is now busy.					*/
	server_busy = 1;




/*-----------------------------------------------------------*/
/** state (svc_compl) enter executives **/

	if (DEBUG==1) printf("\t\t\t<--svc_compl-->\n");


	/* extract packet at head of queue; this	*/
	/* is the packet just finishing service	*/
	pkptr = op_subq_pk_remove (queue_in_service, OPC_QPOS_HEAD);

	/* 885 modification */
	ici_ptr = op_pk_ici_get(pkptr);
	op_ici_attr_get(ici_ptr,"incoming_flow", &flowID);
	op_stat_write(Out_Traffic_stathandle[flowID], 1.0);
	op_stat_write(Out_Traffic_bandwidth_handle[flowID], 1.0);
	pk_len = op_pk_total_size_get (pkptr);
	op_stat_write(Out_Bit_Traffic_handle[flowID], pk_len);

	/* forward the packet on stream 0, causing	*/
	/* an immediate interrupt at destination.	*/
	op_pk_send_forced (pkptr, 0);

	/* server is idle again.				*/
	server_busy = 0;
```

```
/*-------------------------------------------------------*/
```

Source Node Module

```
/* Process model C form file: yy_simple_source2.pr.c */
/* codes for the source node process model */





/* ***** State variable definitions ***** */
typedef struct
	{
	/* Internal state tracking for FSM */
	FSM_SYS_STATE
	/* State Variables */
	Objid				own_id;
	char				format_str [64];
	double				start_time;
	double				stop_time;
	OmsT_Dist_Handle			interarrival_dist_ptr;
	OmsT_Dist_Handle			pksize_dist_ptr;
	Boolean				generate_unformatted;
	Evhandle				next_pk_evh;
	double				next_intarr_time;
	Stathandle				bits_sent_hndl;
	Stathandle				packets_sent_hndl;
	Stathandle				packet_size_hndl;
	Stathandle				interarrivals_hndl;
	FILE*				file_ptr;
	int					use_file_data;
	char				file_name[128];
	double				file_data_rate;
	double				base_time;
	double				second_int_time;
	double				third_int_time;
	int					second_start_time;
	int					third_start_time;
	double				fourth_int_time;
	int					fourth_start_time;
	double				fifth_int_time;
	int					fifth_start_time;
	} yy_simple_source2_state;

/* ***** State variable definitions ***** */





/* ***** Function Block ***** */

static void
ss_packet_generate (void)
	{
	Packet*				pkptr;
	SimT_Pk_Size		pksize;

	/* 885 modification : variable declaration */
	char tempstr[100];

	/** This function creates a packet based on the packet generation       **/
	/** specifications of the source model and sends it to the lower layer. **/
	FIN (ss_packet_generate ());
```

```
	if (use_file_data != 1)
		{
		/* Generate a packet size outcome.					*/
		pksize = (SimT_Pk_Size) ceil (oms_dist_outcome (pksize_dist_ptr));
		}
	else
		{
		/* 885 modification : reading in the next packet size */
		if (file_ptr != NULL)
			{
			// printf("file pointer not null\n");
			if (feof(file_ptr)==0)
				{
				fscanf(file_ptr, "%d\n", &pksize);
				sprintf(tempstr, "constant (%d)", pksize);
				pksize_dist_ptr = oms_dist_load_from_string (tempstr);
				pksize = (SimT_Pk_Size) ceil (oms_dist_outcome
(pksize_dist_ptr));
				}
			else
				{
				fclose(file_ptr);
				sprintf(tempstr, "constant (0)");
				pksize_dist_ptr = oms_dist_load_from_string (tempstr);
				pksize = (SimT_Pk_Size) ceil (oms_dist_outcome
(pksize_dist_ptr));
				}
			}
		}


	/* Create a packet of specified format and size.	*/
	if (generate_unformatted == OPC_TRUE)
		{
		/* We produce unformatted packets. Create one.	*/
		pkptr = op_pk_create (pksize);
		}
	else
		{
		/* Create a packet with the specified format. */
		pkptr = op_pk_create_fmt (format_str);
		op_pk_total_size_set (pkptr, pksize);
		}

	/* Update the packet generation statistics.			*/
	op_stat_write (packets_sent_hndl, 1.0);
	op_stat_write (packets_sent_hndl, 0.0);
	op_stat_write (bits_sent_hndl, (double) pksize);
	op_stat_write (bits_sent_hndl, 0.0);
	op_stat_write (packet_size_hndl, (double) pksize);
	op_stat_write (interarrivals_hndl, next_intarr_time);

	/* Send the packet via the stream to the lower layer.	*/
	op_pk_send (pkptr, SSC_STRM_TO_LOW);

	FOUT;
	}

/* ***** End of Function Block ***** */




/* ***** Temporary Variables definitions ***** */

		/* Variables used in the "init" state.		*/
		char		interarrival_str [128];
```

```
            char         size_str [128];
            Prg_List*    pk_format_names_lptr;
            char*        found_format_str;
            int                 low, high;
            Boolean             format_found;
            int                 i;

            /* 885 modification : variable declaration */
            char         temp_interarrival_str [128];

            /* Variables used in state transitions.      */
            int                 intrpt_code;

/* ***** End of Temporary Variables definitions ***** */

/*-----------------------------------------------------------*/
/** state (init) enter executives **/

      /* 885 modification : record the simulation start time */
      base_time = op_sim_time();

      /* At this initial state, we read the values of source attributes*/
      /* and schedule a selt interrupt that will indicate our start time      */
      /* for packet generation.
            */

      /* Obtain the object id of the surrounding module.
      */
      own_id = op_id_self ();

      /* Read the values of the packet generation parameters, i.e. the */
      /* attribute values of the surrounding module.
      */
      op_ima_obj_attr_get (own_id, "Packet Interarrival Time", interarrival_str);
      op_ima_obj_attr_get (own_id, "Packet Size",              size_str);
      op_ima_obj_attr_get (own_id, "Packet Format",            format_str);
      op_ima_obj_attr_get (own_id, "Start Time",               &start_time);
      op_ima_obj_attr_get (own_id, "Stop Time",                &stop_time);

      /* Load the PDFs that will be used in computing the packet             */
      /* interarrival times and packet sizes.
      */
      interarrival_dist_ptr = oms_dist_load_from_string (interarrival_str);
      pksize_dist_ptr       = oms_dist_load_from_string (size_str);

      /* Verify the existence of the packet format to be used for
      */
      /* generated packets.
                  */
      if (strcmp (format_str, "NONE") == 0)
            {
            /* We will generate unformatted packets. Set the flag.
      */
            generate_unformatted = OPC_TRUE;
            }
      else
            {
            /* We will generate formatted packets. Turn off the flag.        */
            generate_unformatted = OPC_FALSE;

            /* Get the list of all available packet formats.
      */
            pk_format_names_lptr = prg_tfile_name_list_get
(PrgC_Tfile_Type_Packet_Format);

            /* Search the list for the requested packet format.
      */
            format_found = OPC_FALSE;
            for (i = prg_list_size (pk_format_names_lptr); ((format_found ==
OPC_FALSE) && (i > 0)); i--)
```

```
				{
				/* Access the next format name and compare with requested  */
				/* format name.
					*/
				found_format_str = (char *) prg_list_access
(pk_format_names_lptr, i - 1);
				if (strcmp (found_format_str, format_str) == 0)
					format_found = OPC_TRUE;
				}

			if (format_found == OPC_FALSE)
				{
				/* The requested format does not exist. Generate
		*/
				/* unformatted packets.
				*/
				generate_unformatted = OPC_TRUE;

				/* Display an appropriate warning.
			*/
				op_prg_odb_print_major ("Warning from simple packet generator
model (simple_source):",
								"The specified packet
format", format_str,
								"is not found. Generating
unformatted packets instead.", OPC_NIL);
				}

			/* Destroy the lits and its elements since we don't need it
		*/
			/* anymore.
					*/
			prg_list_free (pk_format_names_lptr);
			prg_mem_free  (pk_format_names_lptr);
			}


		/* Make sure we have valid start and stop times, i.e. stop time is      */
		/* not earlier than start time.
			*/
		if ((stop_time <= start_time) && (stop_time != SSC_INFINITE_TIME))
			{
			/* Stop time is earlier than start time. Disable the source.     */
			start_time = SSC_INFINITE_TIME;

			/* Display an appropriate warning.
			*/
			op_prg_odb_print_major ("Warning from simple packet generator model
(simple_source):",
							"Although the generator is not
disabled (start time is set to a finite value),",
							"a stop time that is not later
than the start time is specified.",
							"Disabling the generator.",
OPC_NIL);
			}

		/* Schedule a self interrupt that will indicate our start time for      */
		/* packet generation activities. If the source is disabled,
		*/
		/* schedule it at current time with the appropriate code value.        */
		if (start_time == SSC_INFINITE_TIME)
			{
			op_intrpt_schedule_self (op_sim_time (), SSC_STOP);
			}
		else
			{
			op_intrpt_schedule_self (start_time, SSC_START);

			/* In this case, also schedule the interrupt when we will stop   */
```

```
		/* generating packets, unless we are configured to run until	*/
		/* the end of the simulation.
		*/
		if (stop_time != SSC_INFINITE_TIME)
			{
			op_intrpt_schedule_self (stop_time, SSC_STOP);
			}
		}

	/* Register the statistics that will be maintained by this model.*/
	bits_sent_hndl		= op_stat_reg ("Generator.Traffic Sent (bits/sec)",
	OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
	packets_sent_hndl	= op_stat_reg ("Generator.Traffic Sent (packets/sec)",
	OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
	packet_size_hndl	= op_stat_reg ("Generator.Packet Size (bits)",
OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
	interarrivals_hndl	= op_stat_reg ("Generator.Packet Interarrival Time
(secs)", OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);

	/* 885 modification : get values from attributes */
	op_ima_obj_attr_get (own_id, "File Location", file_name);
	//printf("%s\n", &file_name);
	op_ima_obj_attr_get (own_id, "Use File Data", &use_file_data);
	//printf("%d\n", use_file_data);
	op_ima_obj_attr_get (own_id, "File Data Rate", &file_data_rate);
	op_ima_obj_attr_get (own_id, "Second Interarrival Time", &second_int_time);
	op_ima_obj_attr_get (own_id, "Third Interarrival Time", &third_int_time);
	op_ima_obj_attr_get (own_id, "Fourth Interarrival Time", &fourth_int_time);
	op_ima_obj_attr_get (own_id, "Fifth Interarrival Time", &fifth_int_time);
	op_ima_obj_attr_get (own_id, "Second Start Time", &second_start_time);
	op_ima_obj_attr_get (own_id, "Third Start Time", &third_start_time);
	op_ima_obj_attr_get (own_id, "Fourth Start Time", &fourth_start_time);
	op_ima_obj_attr_get (own_id, "Fifth Start Time", &fifth_start_time);

	/* 885 modification : open file operation */
	if (use_file_data == 1) {
		//printf("file name:%s\n", &file_name);
		file_ptr = fopen(file_name, "r");
		//if (file_ptr != NULL)
		//	fclose(file_ptr);
	}




/*---------------------------------------------------------*/
/** state (generate) enter executives **/

	/* At the enter execs of the "generate" state we schedule the		*/
	/* arrival of the next packet.
		*/

	/* 885 modification */
	/* next_intarr_time = oms_dist_outcome (interarrival_dist_ptr); */
	if (use_file_data != 1)
		{
		next_intarr_time = oms_dist_outcome (interarrival_dist_ptr);

		if (second_start_time > 0)
			{
			if (op_sim_time()-base_time > second_start_time)
				{
				sprintf(temp_interarrival_str, "constant (%f)",
second_int_time);
				interarrival_dist_ptr = oms_dist_load_from_string
(temp_interarrival_str);
				next_intarr_time = oms_dist_outcome
(interarrival_dist_ptr);
				}
```

```
                    }

             if (third_start_time > 0)
                    {
                    if (op_sim_time()-base_time > third_start_time)
                          {
                          sprintf(temp_interarrival_str, "constant (%f)",
third_int_time);
                          interarrival_dist_ptr = oms_dist_load_from_string
(temp_interarrival_str);
                          next_intarr_time = oms_dist_outcome
(interarrival_dist_ptr);
                          }
                    }

             if (fourth_start_time > 0)
                    {
                    if (op_sim_time()-base_time > fourth_start_time)
                          {
                          sprintf(temp_interarrival_str, "constant (%f)",
fourth_int_time);
                          interarrival_dist_ptr = oms_dist_load_from_string
(temp_interarrival_str);
                          next_intarr_time = oms_dist_outcome
(interarrival_dist_ptr);
                          }
                    }

             if (fifth_start_time > 0)
                    {
                    if (op_sim_time()-base_time > fifth_start_time)
                          {
                          sprintf(temp_interarrival_str, "constant (%f)",
fifth_int_time);
                          interarrival_dist_ptr = oms_dist_load_from_string
(temp_interarrival_str);
                          next_intarr_time = oms_dist_outcome
(interarrival_dist_ptr);
                          }
                    }

             }
      else
             {
             sprintf(temp_interarrival_str, "constant (%f)",
(double)(1.0/file_data_rate));
             //sprintf(temp_interarrival_str, "constant (1.0)");
             interarrival_dist_ptr = oms_dist_load_from_string
(temp_interarrival_str);
             next_intarr_time = oms_dist_outcome (interarrival_dist_ptr);
             }


      /* Make sure that interarrival time is not negative. In that case it */
      /* will be set to 0.
                      */
      if (next_intarr_time <0)
             {
             next_intarr_time = 0;
             }

      next_pk_evh      = op_intrpt_schedule_self (op_sim_time () + next_intarr_time,
SSC_GENERATE);




/*---------------------------------------------------------*/
/** state (stop) enter executives **/
```

```
            /* When we enter into the "stop" state, it is the time for us to */
            /* stop generating traffic. We simply cancel the generation of the     */
            /* next packet and go into a silent mode by not scheduling anything    */
            /* else.
                                */
            if (op_ev_valid (next_pk_evh) == OPC_TRUE)
                  {
                  op_ev_cancel (next_pk_evh);
                  }

            /* 885 modification : closing source data file */

            if (use_file_data == 1)
            if (file_ptr != NULL)
                  fclose(file_ptr);

/*-----------------------------------------------------------*/
```

Receving Node Module

```
/* Process model C form file: yy_rcv_node_process2.pr.c */
/* codes for the receiving node process model */




/* ***** Header Block ***** */

/* transition macros */
#define PK_ARRVL ( op_intrpt_type()== OPC_INTRPT_STRM )

/* ***** End of Header Block ***** */




/* ***** State variable definitions ***** */
typedef struct
      {
      /* Internal state tracking for FSM */
      FSM_SYS_STATE
      /* State Variables */
      Stathandle                             ete_gsh;
      int                                    packet_count;
      Stathandle                             packet_cnt_stathandle;
      Stathandle                             ete_dff_stathandle[4];
      } yy_rcv_node_process2_state;

/* ***** End of State variable definitions ***** */





/* ***** Temporary variable definitions ***** */


            Packet *pkptr;            // pointer to packet
            double ete_delay;   // end to end delay
            int i;                    // temporary counter variable
            Ici* ici_ptr;             // ici information pointer
            int flowID;               // indicates which flow the packets is from

/* ***** End of Temporary variable definitions ***** */
```

```
/*-----------------------------------------------------------*/
/** state (init) enter executives **/

      /* initialize variables and register statistics */

      ete_gsh=op_stat_reg("ETE delay", OPC_STAT_INDEX_NONE, OPC_STAT_GLOBAL);
      packet_count=0;
      packet_cnt_stathandle=op_stat_reg("packet count", OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
      for (i=0;i<4;i++)
            ete_dff_stathandle[i]=op_stat_reg("ETE Delay For Flows", i, OPC_STAT_GLOBAL);




/*-----------------------------------------------------------*/
/** state (arrival) enter executives **/

      /* get a pointer to the incoming packet */
      pkptr= op_pk_get (op_intrpt_strm() );

      /* calculate end to end delay */
      ete_delay=op_sim_time()-op_pk_creation_time_get(pkptr);
      op_stat_write(ete_gsh, ete_delay);

      /* record end to end delay for a specific flow */
      ici_ptr=op_pk_ici_get(pkptr);
      op_ici_attr_get(ici_ptr, "incoming_flow", &flowID);
      op_stat_write(ete_dff_stathandle[flowID], ete_delay);

      /* count how many packet received so far */
      packet_count=packet_count + 1;
      op_stat_write(packet_cnt_stathandle, packet_count);

      /* destroy the packet */
      op_pk_destroy(pkptr);


/*-----------------------------------------------------------*/
```