An Efficient Implementation of a Hierarchical Weighted Fair Queue Packet Scheduler

by

Oumar Ndiaye

B.S, Computer Science, Lehman College
(City University of New York), 1992

Submitted to the Department of
Electrical Engineering and Computer Science
In Partial Fulfillment of the Requirements
for the Degree of

Master of Science in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology
May, 1994

Signature of Author ______________________________
Department of Electrical Engineering and Computer Science
May 6, 1994

Certified by ______________________________
Senior Research Scientist David D. Clark
Thesis Supervisor

Accepted by ______________________________
Professor F. R. Morgenthaler
Chairman, Committee on Graduate Students

An Efficient Implementation of a Hierarchical Weighted Fair Queue Packet Scheduler

by

Oumar Ndiaye

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 1994
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Electrical Engineering and Computer Science

ABSTRACT

The technical developments in computer networks in recent years have spawned the possibility of merging different services into a single Integrated Service Packet Network (ISPN). The types of service quality required by each of the individual services in an ISPN often differ greatly. Thus, the packet scheduling algorithms used in such networks must be flexible enough to allocate the available link shares according to the service quality requirements of the different services.

In this thesis we propose an efficient implementation of a link sharing algorithm for an Integrated Services Packet Switch (ISPS). The sharing algorithm guarantees to each customer a percentage of the link share no less than the portion of the bandwidth he owns. A customer can split its link share among different services whose link shares may in turn be split among child-services. The link sharing is hierarchical; each service gets its link share from that of its parent service.

Because of the complex nature of the link sharing the algorithm provides, the run time of its implementation can be large. However, in order to benefit from the fast links of networks like ATM, it is desirable to have reasonably short packet processing delays. The implementation presented here maps the hierarchical sharing into an equivalent single level sharing which resulted in a speed up of the packet processing executor by a factor of 10.

Thesis Supervisor: Dr. David D. Clark
Title: Senior Research Scientist, MIT

Acknowledgments

The source code of the original implementation of the algorithm by David Clark was used as a basis of this work.

I would like to take this opportunity to thank my thesis supervisor, David Clark, who suggested I work on this problem and provided me with valuable intuitive insights on the problem. He provided guidance, support and the freedom to do this thesis. I am specially grateful to his patience and numerous suggestions during our many discussions. Working with him and in his group has been an enriching experience.

I want to thank Timothy J Shepard and Christopher Lefelhocz for their attentiveness to my sometimes distracting questions.

I also thank Anna Charny, Ye Gu, Janey Hoe, Karen Sollins, Lisa Taylor and every member of the ANA group for being so nice and friendly.

I am grateful to Mark A. Smith and James Njeru for proof reading this document.

I am grateful to my brother and sister in law; Abdoulaye and Margaret Ndiaye, and MIT for giving me the opportunity to further my education.

This research was partially made possible by the MIT *Vinton Hayes* Fellowship. This research was also supported by the Advanced Research Project Agency under contract DABT63-92-C-0002.

This document was typeset with the LaTeX Document Preparation System.

To my mother.

Contents

Chapter 1

Introduction

The objective of this thesis is the optimization of an implementation of a packet scheduling algorithm for shared services in an Integrated Service Packet Switch (ISPS). The algorithm is based on an paper by D. Clark, S. Shenker, L. Zhang ([1]). The integrated service packet switch algorithm was originally implemented by D.D. Clark. Unfortunately there was a problem in the original implementation. It implemented the algorithm correctly, but it turns out that its running time was undesirably large. Thus, the goal of this work is to modify the implementation of the algorithm so as to bring its time complexity as low as possible.

1.1 Background

The algorithm accepts service requests according to the present load of the network and the service requirements it has promised to the already accepted services. See reference [2] for more details on the admission control algorithm. The nature of the promise made to a service depends on the type of the service. There are three type of services: *Guaranteed, Predicted* and *Shared.* In this paper we will focus our attention on the shared services.

As the name suggests, the shared services share the link bandwidth according to the arbitrary portion of the bandwidth that each one owns. As in every sharing environment, fairness needs to be addressed here. The sharing is considered fair if the amount of bandwidth owned by any service is available to it regardless of the behavior of other services in the network. To insure fairness, packets are scheduled using a Weighted Fair Queuing (WFQ) scheme, where the weights are the link shares. In the next paragraphs we will justify the choice of WFQ for the queueing discipline and then present its basic mechanism.

Queueing algorithms can be thought of as mechanisms that allocate three quantities: bandwidth (which packets get transmitted), promptness (when will these packets leave) and buffer space (which packets get dropped) ([5]). The most currently used queueing discipline is first-in-first-out (FIFO). In FIFO queueing, the rate at which a source sends packets relative to the other sources in the network essentially determines the amounts of the three quantities allocated to it. Since packets are being serviced in their order of arrival, the faster a source generate packets the more likely its packets will arrive first at a gateway, thus its packets will be serviced more often. This implies that the service quality that a session gets from the network is partly dictated by the behavior of its source. Thus, FIFO is not adequate for fairness in the sense that it does not guarantee service quality to a particular service; a more effective queueing mechanism is needed.

Following a similar line of reasoning, Nagle [10,11] proposed a *fair queueing* (FQ) algorithm in which separate queues were maintained for packets from each source. The queues are serviced in round-robin manner. His algorithm was effective in providing fairness in terms of the number of packets serviced for each session. If a source attempts to get over by sending packets at a faster rate, it will simply overflow its queue and hence suffer packet drop but won't get more packets sent than other sources in the network. However, because packets can be of varying length, round-robin does not guarantee fair share of the bandwidth. In FQ, sessions with long packets will be allocated a greater portion of the bandwidth. Furthermore, even with fixed packet length, round-robin is not flexible since it attempts to treat each session or packet equally. In integrated services networks, since the different services have different requirements, one needs a mechanism that can treat the different services in different ways. Thus, in integrated networks a more attractive queueing discipline is needed.

Weighted Fair Queue (WFQ), presented by Parekh and Gallager in [8], which also works for non uniformly weighted shares, provides fairness in the allocation of the bandwidth. Here a weight can be thought as the percentage of a link bandwidth allocated to a given service.

We now present the mechanism of WFQ. Assume, for simplicity, that there are two sessions S_1 and S_2 and we want to allocate $\frac{3}{4}$ of the link bandwidth to S_1 and the remaining $\frac{1}{4}$ to S_2. Furthermore, each session has a packet to send whenever it is time to send one. A simple way of implementing the sharing is to fill each frame with the following sequence of bits below (s^i_m is the m^{th} bit of session i). Here a frame is defined as a string of bits composed of control bits (*header*) and data bits obtained from the packets. In other words a frame is just a packet formed of bits from the different packets to service.

$$\underbrace{\overbrace{H s^1_1 s^1_2 s^1_3 s^2_1 \cdots s^1_{3n-2} s^1_{3n-1} s^1_{3n} s^2_n}^{frame0} \overbrace{H s^1_{3n+1} s^1_{3n+2} s^1_{3n+3} s^2_{n+1} \cdots s^1_{6n-2} s^1_{6n-1} s^1_{6n} s^2_{2n}}^{frame1} \cdots \overbrace{H s^1_{i3n+1} s^1_{i3n+2} s^1_{i3n+3} s^2_{in+1} \cdots}^{framei}}_{seq-of-bits-sent}$$

where H is the *header* of the frame, and $4n$ is the length of every frame.

Once the right length of the frame is chosen, the implementation is fairly straight forward. However, this model is not very realistic. The processing overheads resulting from the filling of the different frames is too large. For example, the CRC of the new frames will need to be computed. An alternative approach is to attain the same goals without the processing overhead mentioned above. This can be accomplished by keeping weighted counters ($C_1(t)$ and $C_2(t)$ respectively for S_1 and S_2 at time t) of the number of bits serviced for each of the sessions. The counters are initially zero. Every time that a packet is serviced the different counters are updated as follow:

$$C_1(t_{(1,n)}) = C_1(t_{(1,(n-1))}) + \frac{4}{3}L_{(1,n)}$$

where $L_{(1,n)}$ is the length of the n^{th} packet of session 1 to be serviced and $C_1(t_{(1,n)})$ is the value of the counter at time $t_{(1,n)}$; when the n^{th} packet of session 1 finished service.

$$C_2(t_{(2,n)}) = C_2(t_{(2,(n-1))}) + 4L_{(2,n)}$$

where $L_{(2,n)}$ is the length of the n^{th} packet of session 2 to be serviced and $C_2(t_{(2,n)})$ is the value of the counter at time $t_{(2,n)}$; when the n^{th} packet of session 2 finished service.

Whenever it is time to service a packet, the first packet of the session with the smallest counter is chosen and then its counter is updated as described above. One draw back with this approach is that it does not cope with the situation where a session may remain inactive[1] for some period of time in which case its counter will lag way begin the counter of the other session. Thus, the inactive session will be credited for the time it was inactive, we don't want that. We resolve this problem by replacing the weighted counter with a new parameters Fi_x, for each sessions x. Now we define Fi which is slightly different from the weighted counter. When updating Fi, if it is smaller than the real time (now), it is first set to now before it gets incremented. Thus, if Fi lags behind because the session remained inactive for some time, the session won't be credited for the inactive period. We update Fi as follows:

$$Fi_x(t_{(x,n)}) = max(Fi_x(t_{((x,n-1))}), now) + \frac{L_{(x,n)}}{(r_x)v}$$

where $Fi_x(t_{(x,n)})$ is the value of the Fi_x at time $t_{(x,n)}$; when the n^{th} packet of session x finished service, $L_{(x,n)}$ is the length of the n^{th} packet of session x to be serviced, r_x is the percentage of the total bandwidth allocated to session x and v is the link speed.

[1] an active (inactive) service is just one that does (not) have packets to send.

WFQ algorithms have been studied in different papers ([5],[8]). Parekh and Gallager ([8]) showed that WFQ is an efficient way to guarantee service quality. More analytical discussion on Weighted Fair Queue can be found in references [1] and [5]. Using WFQ, a given customer that purchases $r\%$ of the link bandwidth can be guaranteed an effective throughput rate no less than $(rv)/(\sum r_i)$ where v is the link speed and the r_i's are the percentages of the bandwidth allocated to the currently active services.

1.2 Organization of this thesis

In chapter 2 the properties of the sharing and how the original implementation, the old scheme, achieved them are discussed. The reasons for the large processing delay of the old scheme is also discussed in this chapter. We present the new scheme which overcomes the large processing delay in chapter 3. We also examine in this chapter the two different methods used by the new scheme to distinguish between active and inactive nodes. The first method is not very realistic, its merit of being presented in this thesis is merely to give a better insight of the effectiveness of the second method. A Series of simulations are conducted in chapter 4 to verify the effectiveness of the second method. Chapter 5 concludes the thesis with a brief summary of the results of this work, some implementation issues and future work points that could enhance the new implementation in the presence of priority services.

Chapter 2

The Problem

2.1 Sharing in ISPS

In the Integrated Service Packet Network architecture we allow a customer to divide its service into several sub-services[1]. We define a service class as a service that has derivative services. In Figure 2.1.1, a service class is simply a non-leaf service node. In figure 2.1.1, every node other than nodes labeled svc7 through svc16 is a service class. Thus, a customer can divide its share of the link according to the needs of its sub-services; $s_1, s_2, \ldots, s_n$ and each of the s_i may be recursively split into lower level sub-services, $s_{i1}, s_{i2}, \ldots, s_{ij}$, for some j.

[1] a sub-service is a service that derived from another service, in Fig 2.1.1, a sub-service is a non-root node.

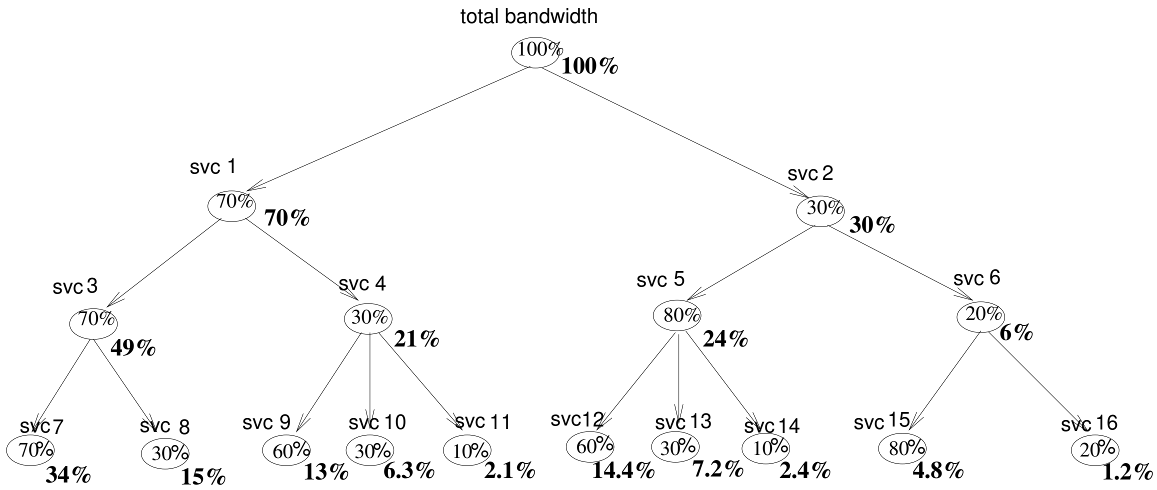


Figure 2.1.1: Sharing in ISPS

For example, in figure 2.1.1, service1 (svc1) distributed its link share (70%) to svc3 (49%) and svc4 (21%). Service 4 (svc4) in turn splits its share between svc9 (12.6%), svc10 (6.3%) and svc11 (2.1%). Thus, the architecture provides to customers a mechanism to prioritize their different services. For example, one may choose to allocate 80% of his link share to video services and just 20% to data services. Typically the leaf node services in figure 2.1.1 are the only real services(traffic), the inner node services (svc1, svc2,…,svc6) are simply classes of services. For example svc3 is a service class composed of svc7 and svc8, and svc1 is a service class composed of service class svc3 and service class svc4.

Properties of the Sharing

Through the sharing in ISPS, we are trying to achieve two major goals. We consider the sharing to be fair if the two major goals described below are met.

First, we want the sharing to be such that the link share allocated to any service is available to it at all the time, hence available to all its sub-services. For example in figure 2.1.1, we want the 21% of the total bandwidth to be available to svc4 at all time, hence available to svc9, svc10 and svc11.

Second, we want the ratio between the bandwidth shares of any two sub-services of the same parent service to be constant at all time. For example in figure 2.1.1, if every service other than svc10 has traffic to send we want the bandwidth share of svc10 shifted to svc9 and svc11. Further more the ratio between the new bandwidth shares of svc9 and svc11 after the shift should be equal to the ratio of their original shares ($\frac{12.6}{2.1} \approx 6$). The way to accomplish that, in our example, is to shift $\frac{6}{7}$ and $\frac{1}{7}$ of svc10's share respectively to svc9 and svc11. Thus, the new shares of svc9 and svc11 become respectively 18% (=$12.6 + \frac{6}{7}6.3\%$) and 3% (=$2.1 + \frac{1}{7}6.3\%$), and the new ratio is still 6 (=$\frac{18}{3}$). The motivation behind keeping the ratio constant is consistency. If for example, a customer decides to split its bandwidth share into two services; A : 80% and B : 20%, then it is clear that the customer desires service A to be four times better serviced than service B. Thus, when the same customer acquire $x\%$ extra of the total bandwidth it must be the case that $\frac{4}{5}x\%$ should go to A and $\frac{1}{5}x\%$ to B.

We summarize the two fundamental goals of the sharing mentioned above as follows.

- *i)* At any active period of the link (when there are packets to transmit), each active customer is guaranteed a share of the link no less than the portion of the link bandwidth it owns.

- *ii)* For any sub-service, the amount of link resource at its disposal that it is not using is distributed to all its sibling sub-services that need more link usage than the amount they are entitled to. This distribution is done proportionally to the rates of the sub-services.

In other words when a service (node in fig 2.1.2) s of share x is not fully using its share, then the unused portion of x is distributed to the other active services (nodes) that have the same parent as s proportionally to their rates.

The WFQ packet scheduling mechanism presented in the previous chapter is too simplistic to suffice for the achievement of the two major goals of the sharing in

ISPS. Any algorithm that will accomplish these two goals will have to both deal with the different levels of the sharing hierarchy and the states (active or inactive) of the different services. This is because every time that a service becomes newly inactive or active, the current link shares of all its siblings (of the same parent service) services will have to be updated. We will present two different schemes in the next section and in section 3.2 that achieve the two criteria of the sharing.

Both schemes use a tree structure for the packet scheduling. The tree is an abstraction of the service sharing. For example the tree in figure 2.1.2 corresponds to the sharing presented in figure 2.1.1.

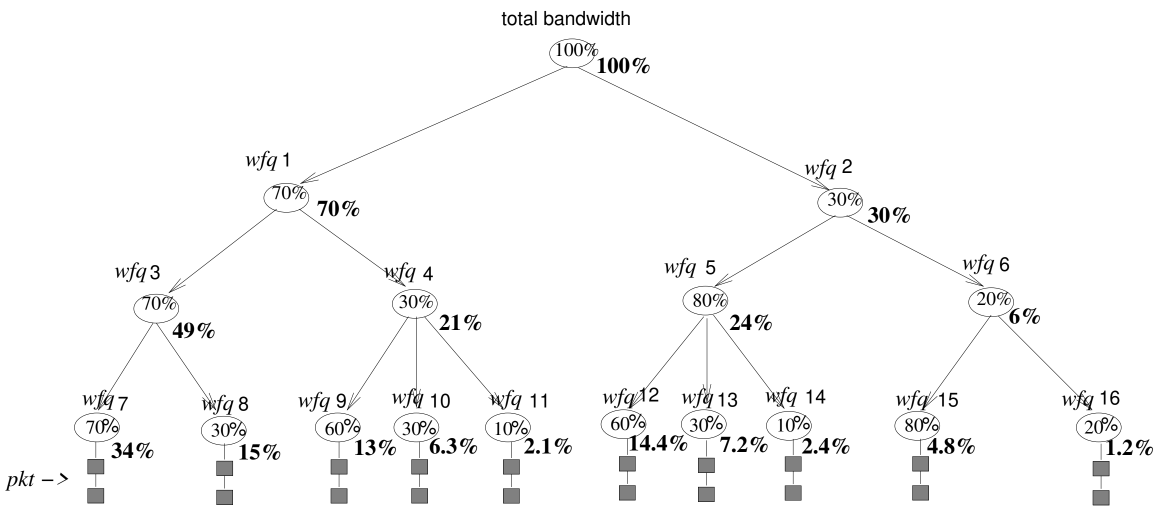


Figure 2.1.2: Tree structure

Each service is associated with a node in the tree. The tree supports three types of nodes: *weighted fair queue(wfq)*, *priority*, and *root*. *Priority* nodes are presently implemented, but for clarity we will ignore them as much as possible. The root node has the full bandwidth of the link which is distributed among the remaining nodes. Each node other than the root, gets its share of the bandwidth from that of its parent. That is the total share of a tree node is equal to the sum of its children's. *Root* and *wfq* nodes have child node type *wfq* (service) or *pkt* (packets). The leaf *wfq* nodes are the only *wfq* nodes that can have a child type *pkt*. They also correspond to leaf nodes in the corresponding sharing tree. The tree nodes of child type *wfq* correspond to service classes in the sharing hierarchy.

2.2 The Old Scheme

In the original version of the implementation, the varying link shares of the services are captured by a Hierarchical Weighted Fair Queuing (HWFQ) mechanism which consists of nested WFQ mechanisms at each level of the tree; a WFQ of WFQ's. In this implementation every tree node of child type *wfq* had a child queue that linked all its active child nodes. An inner node is active if at least one of its child node is active, and inactive if all its child nodes are inactive. A leaf node is inactive (active) if its packet queue is (not) empty. At each level of the tree, the *wfq* nodes of the same parent are sorted in ascending order of their *Fi's* (finish time). The finish time (*Fi*) of a node of child type *pkt* is the same as defined in the previous chapter:

$$Fi_x(t_{(x,n)}) = max(Fi_x(t_{((x,n-1))}), now) + \frac{L_{(x,n)}}{(r_x)v} \tag{2.1}$$

It is the latest time the node's first packet must finish service. And the Fi of each *wfq* node of child type *wfq* (inner node) is computed using the length of the first packet of the first leaf node below the inner node for $L_{(x,n)}$ in (2.1). The first leaf node below an inner node is the leaf node reached by walking the tree up-down starting from the inner node and moving recursively from a node to the node at the head of its child queue until a leaf node is reached. Note that the *wfq* nodes are sorted into their parent's child queue in ascending order of their Fi's. Hence, the Fi of the first on the queue will be minimal, and the first leaf node below an inner node will be the leaf node with the smallest Fi among the leaf nodes below the inner node.

When a switch receives a packet p from a service *svc* it appends the packet to the associated node's packet queue. If the node's packet queue is empty, the node's Fi is computed using the length of the packet in (2.1) and the node is sorted into its parent's child queue. If after the sorting the node happens to be at the first position, its parent's Fi will be updated using the length of this new packet in (2.1). The process is recursively carried over along the tree hierarchy as long as the node at the head of the ancestor's child queue changes after the sorting.

Whenever the switch is ready to send a packet, it walks the tree top-down (from root to leaf) choosing at each level the first node of the child queues. Each time that a node is chosen, it is taken off its parent's child queue. The walk continues until a leaf node is reached, in which case the first packet of the leaf node is sent. In this implementation it is always guaranteed that this leaf node will have a packet on its queue. If the node still has packets, its Fi is computed with the length of its next packet and the node is resorted into its parent's packet queue according to its newly computed Fi. To conclude the sending, the switch walks the tree in reverse order updating each ancestor's Fi with the length of the first packet of its first leaf node. After each Fi update the corresponding ancestor is sorted into its parent's child queue. Enqueuing a packet and dequeuing a packet are handled by two routines; respectively *Tree_enq* and *Tree_deq*.

Even though this implementation guarantees fairness in the sense of the definition given at the beginning of section 2.1.1, one disadvantage of this scheme is that the different child queues must be updated dynamically as their nodes transit from an inactive state to an active one. Recall that the child queues only contain active nodes.

Another disadvantage with this scheme is that every time that a packet departs from a node, the *Tree_deque* routine has to make sure that all the ancestors are resorted correctly into their parent's child queues. At each instance that a packet departs from a node, the node needs to be taken off its parent's child queue. If the node still has packets, it needs to be sorted according to its new Fi into its parent's child queue. The process has to be carried over as far as up to the root node.

Thus, if the tree is large, the switch will spend a relatively large amount of its time updating the child queues of the different nodes even though the link may be ready to accept another packet. Hence, to take advantage of the link speed, the faster the link is the less time the switch has to resort the child queues. For example, if the link can transmit say 100 packets per second at maximum, then the switch only needs to be fast enough to execute the necessary operations after a packet dequeue within $10msec$. However if the link can transmit say 10,000 per second, then the switch must be fast enough to execute the necessary operations after packet dequeue within $0.1msec$. This may not be an issue in the case of a slow network. However, the increasing widespread use of fast networks makes the need for optimization obvious.

Chapter 3

The Solution

3.1 Some Observations

Before optimizing, one needs, a priori, to locate the routines in which the code performs worst. This led us to an instruction count that helped pin point parts of the code that one would like to see modified so as to reduce the time complexity of the software package. The result of the instructions count confirmed that *Tree_enq* and *Tree_deq* contributed most to the run time.

Hence, a first step toward solving the problem was to modify the routines that manage the enqueue and dequeue of packets namely *Tree_enque* and *Tree_deque*. The main problem with these routines is that they had to update the *Fi's* and the child queues of the inner *wfq* nodes whenever the *Fi* of a leaf node changes. After several modifications of the original implementation, we came to the conclusion that the only way we can meet our optimization goal is to break the implementation into two independent components using some heuristics.

3.2 The New Scheme

One way to speed up the software is to avoid re-sorting the different child queues every time that a packet departs. That is the purpose of the new scheme. The new scheme uses the same tree structure as the old scheme but uses it in a different way. In this scheme *Tree_enque* and *Tree_deque* won't have to sort beyond the first *wfq* node. This is accomplished by mapping the hierarchical weighted fair queue mechanism (HWFQ) into an equivalent flat weighted fair queue mechanism in which the changing weights are computed periodically. The tree is used just to update the rates.

- First we need to define few new parameters.

  Instead of associating a node with a rate that represent its percentage of its parent node's share, we associate a node with two rates: *rate* and *cur_rate*. *rate* is the predetermined effective share of the total link bandwidth that is allocated to a node, For example the effective rate of *svc10* (in figure 2.1.2) is 6.3% (= .70 × .30 × .30). *cur_rate* is the current effective share of the total bandwidth, it changes with time as some other services of the parent service become newly inactive or active.

  As in the old scheme, each leaf node keeps a Fi parameter which is computed the same way as in the old scheme.

  A sorted low level queue (*llq*), that links the active leaf level *wfq* nodes is maintained. The sorting of the *llq* queue will be performed by a special purpose traffic scheduler chip by J. Chao ([9]) which will operate in parallel.

3.2.1 Receiving a Packet

Whenever a packet arrives at a node, it is appended at the end of its packet queue. If the packet queue was empty the Fi of the node is computed using the length of the newly arrived packet, and the node is sorted into the low level queue, *llq*, according to Fi.

- **Tree_enq**(*pkt,node*)

  1. **if** NOT empty(*node*(PQ[1])) **then**
     (a) append *pkt* into *node*'s packet queue.
  2. **else**
     (a) append *pkt* into *node*'s packet queue.
     (b) $node{\rightarrow}Fi = \max(node \rightarrow Fi, now) + \frac{pkt{\rightarrow}len}{(node{\rightarrow}cur\_rate)\times LINK\_SPEED}$
     (c) pass(*node*) to the scheduler chip for sorting into *llq* according to Fi
  3. **return**;

This code requires at most 55 instructions with *priority* nodes and 47 without *priority* nodes independently of the size of the tree.

3.2.2 Sending a Packet

To send a packet, the node at the head of the (*llq*) queue, if not nil, is chosen. Once the node is chosen, the packet at the head of its packets queue is sent. Once the packet is sent the new Fi is computed using the length of the next packet to go and the node resorted into the *llq* queue.

- **Tree_deq**()

  1. **if** not null(*llq*) **then**
     (a) temp= *llq*.head;
     (b) pkt_to_send= temp→first[2];
     (c) temp→first= pkt_to_send→next[3];

[1]PQ is the packet queue of the node.

[2]first is the packet at the head of the node's packet queue.

[3]next is the next structure in the list. In a packet queue it's the next packet in the queue and it's the next node in a queue of nodes.

(d) *send*(pkt_to_send);

(e) *llq*.head= temp→next;

(f) **if** (temp→first) **then**

i. temp→Fi= temp→Fi+ $\frac{temp \rightarrow first \rightarrow len}{(temp \rightarrow cur\_rate) \times LINK\_SPEED}$

ii. pass(temp) to the scheduler chip for sorting into *llq* according to Fi;

2. **return;**

The code above will execute at most 70 instructions with priority nodes and 31 without priority nodes independently of the size of the tree.

3.2.3 Updating the rates

Beside *Tree_enque* and *Tree_deque*, there is now a new routine, *Update_rate*. *Update_rate* periodically updates the current shares (*cur_rate*) of the active services by allocating the current shares of the newly inactive services to their active siblings proportionally to their rates. The *Update_rate* routine achieves this by recomputing the *cur_rates* of the active nodes. It does so by going through the tree structure top down and at each node p it assigns to each of its active children i, $\frac{r_i cur\_rate_p}{\sum_{j \in A(p)} r_j}$, where $A(p)$ is the set of the children of p that are active and r_i is the effective rate of node i. This implies the need to keeps track of who is active and who is not. In section 3.3.1 we will present a first method of detecting the state of a node. As we will see this method has some fundamental limitations. In section 3.3.2 we will present a more realistic method; the second method. The state changes of the nodes are captured in *Update_rate* by two routines; *Mark_active* and *Mark_inactive*, we present their pseudo codes below. *Mark_active* detects the low level *wfq* nodes that just change state and mark them with their new state. It makes sure that all ancesters of a node that just became active are marked active. *Mark_inactive* makes sure that any node that has all its child nodes inactive is marked inactive.

- **Update_rate**(*isps,tree*)

  1. **If** (*tree*=ROOT) **then**
     (a) call Mark_active_nodes
     (b) call Mark_inactive_nodes
     (c) **If** (no new active and no new inactive) **then**
        i. return;
  2. **for** (every active child node n of *tree*) **do**
     (a) i++
     (b) ar[i]= n
     (c) sum_rate=+ n→rate
  3. **for** (every active child n of *tree*) **do**
     (a) $n \rightarrow cur\_rate = \frac{n \rightarrow rate}{sum\_rate} \times tree \rightarrow cur\_rate$
  4. **If** (tree is of child type *WFQ*) **then**
     (a) **for** (every child *child_node* of tree) **do**
        i. *Update_rate(isps, child_node)*
  5. return;

Notice that *Update_rate* leaves the *cur_rate* of the newly inactive nodes unchanged. Thus, the next time that an inactive node will become active the value of its *cur_rate* from the last update period it was active will be used to compute its Fi.

- **Mark_active**(*isps*)

  1. **for** (every node, n, in potential_state_change_q) */* potential_state_change_q is the queue of the nodes that may change state */*
     (a) **If** (n just became active) **then**
        i. $n \rightarrow active = 1$

ii. **If**(NOT($n \rightarrow counted\_in\_sum\_active$)) **then**

A. $n \rightarrow parent \rightarrow sum\_rate = +n \rightarrow rate$

B. $n \rightarrow counted\_in\_sum\_active = 1$

iii. **for**(every inactive ancester, an, of n)

A. $an \rightarrow active = 1$

B. **If**(NOT($an \rightarrow counted\_in\_sum\_active$)) **then**

C. $an \rightarrow parent \rightarrow sum\_rate = +n \rightarrow rate$

D. $an \rightarrow counted\_in\_sum\_active = 1$

iv. **If** (n's pkt queue NOT empty)

A. take n off potential_state_change_q

(b) **else**

(c) put n in ar; the array of newly inactive nodes

(d) $n \rightarrow active = 0$

(e) **If**($n \rightarrow counted\_in\_sum\_active$) **then**

i. $n \rightarrow parent \rightarrow sum\_rate = -n \rightarrow rate$

ii. $n \rightarrow counted\_in\_sum\_active = 0$

(f) take n off potential_state_change_q

2. return

- **Mark_inactive**(ar, ar_len) */* ar is the arrray of newly inactive nodes */*

1. **if** (empty(ar)) **then**

return

2. **for** every node, n, in ar

(a) **If** (all siblings of n are inactive n's parent active) **then** */* the test of parent's activity is to avoid putting it twice in the next_level_ar */*

i. put n's parent in next_level_ar   */* next_level_ar is ar for the next level of the tree */*

ii. $n \rightarrow parent \rightarrow active = 0$

iii. **If**($n \rightarrow parent \rightarrow counted\_in\_sum\_active$) **then**

A. $n \rightarrow parent \rightarrow parent \rightarrow sum\_rate = -n \rightarrow parent \rightarrow rate$

B. $n \rightarrow parent \rightarrow counted\_in\_sum\_active = 0$

3. **If** (NOT empty(next_level_ar))

   (a) *Mark_insactive (next_level_ar, next_level_ar_len)*

4. return

Given a tree of n nodes with a total of p *wfq* leaf nodes, a an average number of children per node and update interval T equal to u packet-times, *Update_rate* will execute $68n - 65p + 11 + (11 + 18a + 7log_a(u))u + (5 + 37a)log_a(\frac{u}{2})$ instructions. This includes an average of $6 + 11u + 7ulog_a(u)$ instructions from *Mark_active* and average of $5 + 18ua + (5 + 37a)log_a(\frac{u}{2})$ instructions from *Mark_inactive. Mark_active* finds the nodes that just changed state in the queue of *wfq* nodes that may change state. The nodes in this queue are those nodes that experienced a packet arrival following an empty packet queue or whose packet queue became empty within the last update interval. Because the maximum numbers of packet departures and arrivals within one update interval are both u, this queue will be at most $2u$ nodes long. Given a full *5-nary* tree of depth 5 and a 10 packet-times update interval, the code for *Update_rate* will execute $63,543$ instructions.

In the computation of these averages we used the average numbers of time the different blocks of code will be executed whenever it was possible to reasonably assess these average numbers, otherwise the worst cases were used. Thus, one can expect to get the average number of instructions executed in *Mark_active* and *Mark_inactive* in real life to be better than the average instruction counts presented here.

Even though *Tree_enque* and *Tree_deque* no longer have to sort the child queues, they still have to sort a leaf nodes into the *llq* queue whenever their *Fi*'s change. However, because of the modularity of the new implementation, we are able to perform the sorting into *llq* with the traffic scheduler chip in parallel.

Thus, the time required to decide on the packet to send and the time required to enqueue a packet are now significantly reduced. The pseudo codes of *Tree_enque* and *Tree_deque* sketched above are presently implemented in the C programming language. Their assembler code listings resulted in a significant reduction of the number of instructions executed by each of the two routines. For a full *5-nary* tree configuration the run time of *Tree_enque* now requires the execution of at most 46 machine instructions (with priority nodes and 38 without) as opposed to 679 for the same tree configuration in the original implementation. For the same tree, *Tree_deque* requires at most 70 machine instructions (with priority and 31 without) compared to 875 in the original implementation. Hence, assuming a 1-Mips processor, not only that the switch can decide on the next packet to send or perform the required operations to enqueue a packet within 125 μsec, but it now can both send and enqueue a packet within 125 μsec.

3.3 Validity of the solution for the continuous case

The following tree configuration (see fig 3.3.1) was simulated and the results in the table (fig 3.3.2) were observed. In this simulation there were eleven services with deterministic traffic sources of packets generation rate $\mu = .00052$ $packets/\mu sec$. All packets have 1600 bits length and the bottleneck link speed between the sources and the ISPS component is $10,000 Kbits/sec$. The out going bottleneck link from the ISPS component is $300 Kbits/sec$. Notice that μ $(= 1600 bits * .52K/sec = 8320 Kbits/sec)$

is big enough that any service can saturate the link alone. All sources generate packets continuously at the rate specified above.

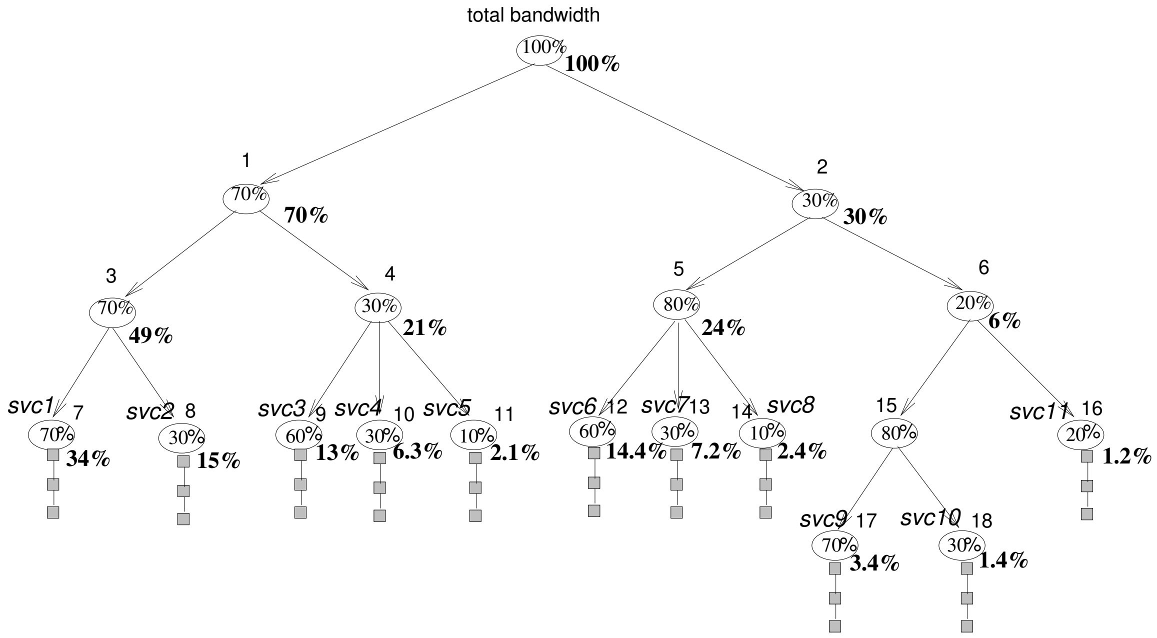


Figure 3.3.1: Configuration of the simulation

The number inside every *wfq* node represents its bandwidth rate with respect to its parent's. The numbers (in bold face) below the nodes represent their rates relative to the total bandwidth (effective rate), they correspond to the promised rates in fig 3.3.2. Each service svc_i is associated with the i^{th} leaf node counting from left to right. For example svc1 is associated with the node7 whose rate is 34%.

In this simulation below all eleven sources were continuously sending packets.

Services	*svc1*	*svc2*	*svc3*	*svc4*	*svc5*	*svc6*	*svc7*	*svc8*	*svc9*	*svc10*	*svc11*
Promised Rate	*34%*	*15%*	*13%*	*6.3%*	*2.1%*	*14.4%*	*7.2%*	*2.4%*	*3.4%*	*1.4%*	*1.2%*
Observed Rate	*34.01%*	*14.78%*	*12.46%*	*6.3%*	*2.07%*	*14.17%*	*7.09%*	*2.36%*	*3.33%*	*1.42%*	*1.21%*

Figure 3.3.2: Table of results for continuous sources

The table shows that the promised rates (the rates that the nodes are entitled to) are indeed the same as the the observed ones (the rates that the different services actually received). Even though the packet generation rates of the different sources is the same in this simulation, the table above would still be the same otherwise as long as the packet generation rate of every source is large enough to use up the link share its service. Using the same parameters, we obtained similar results for an identical simulation with poisson sources. Every simulation we present in this thesis is performed for both the cases where the sources are deterministic and poisson. For the most part we will present one case as long as the results of the two cases are similar.

The simulation results above confirm the correctness of the new implementation in the case where each service is continuously sending enough packets to use up its share of the link. We will therefore focus the rest of the discussion of the validity of the new scheme to the case where some services may not be generating enough packets to fill up their link shares. That is when some services may become alternatively active and inactive over the course of time. In the next two sections we present the two different methods of determining the state (active or inactive) of a service.

3.3.1 The First Method

The initial implementation of the new scheme followed this method. It is very simple, but has some problems as we will see. The first method distinguishes active and inactive nodes as follow: At the end of each rate update all low level *wfq* nodes are marked inactive, and within this time period every time that a given number n_p of packets arrive at a node, the node is marked active. n_p is presently equal to one. Thus, we need to add the code below at the top level of *Tree_enq*. The code marks a node active whenever it receives its first packet.

- **if** NOT(*node*$\rightarrow$*active*) **then**

1. num_new_active++; */* if the node just became active put it in */*
2. array_new_active[num_new_active]= node; */* the array of newly */*
3. node→active=1; */* active nodes. */*

If the minimum number of packet arrivals, n_p, required within the interval time T for a node to be active is not chosen properly, property (ii) of the algorithm may not be preserved by the implementation. For example, the rate corresponding to the total bandwidth shares of the services, s_i's, associated with a node i may be r_{max}, but these services may only generate packets at a rate $r_t < r_{max}$ during T. If r_t is such that the number of packet arrivals at i during T is greater or equal to n_p then at the next rate update, *Update_rate* will distribute the shares the way as it would if the s_i's were generating packets at r_{max}. In this case the unused share $(r_{max} - r_t)$ will be shared equally by all other active services. This problem can be solved by minimizing the update interval time T so that the services transmitting below their rates will be alternatively marked inactive and active (inactive most of the time). Thus, in the long run their siblings will be allocated in *Update_rate* approximately the exact unused rates that should go to them. However, a very small T may not be very attractive. The smaller T is the greater the likelihood of the existence of a node with a packet inter-arrival greater than T. Such a node will tend to be marked inactive most of the time even though it may still have packets in queue for transmission. Thus, its parent service will be serviced with favor because having packets for transmission, the node is using its share while its share is reallocated to its siblings because it was mistakenly considered inactive. Thus, its parent service will enjoy the node's associated share twice. The next simulations results based on different values of T will give a better insight of this problem.

We simulated the same tree configuration in fig 3.3.1 except that this time the source associated with the leaf node of rate 15% (svc2) was generating packets as follows: It alternately sends packets at a rate μ =.0000284 $pkts/\mu sec$ which is just enough to feel svc2's bandwidth share during a time period equal to 10,000 μsec simulator time and then stop sending for a period equal to 50,000 μsec.

With this new configuration, the update period T was set to 10,000 μsec. Since the incoming bottleneck link speed is $10,000 Kbits/sec$, a packet will arrive at a node every $160\mu sec$. Hence during each period T 62 (=10,000/160) packets will arrive at the nodes. And since $62 > 11$ and the sources have an identical packet generation rates, within T each of the eleven sources will have a packet that will arrive at the node associated with its service. Therefore the inactive sources are the only ones that may be marked inactive during an update. The rate table in figure 3.3.1.1 was observed.

Services	svc1	svc2	svc3	svc4	svc5	svc6	svc7	svc8	svc9	svc10	svc11
Promised Rate	34%	15%	13%	6.3%	2.1%	14.4%	7.2%	2.4%	3.4%	1.4%	1.2%
Observed Rate	42%	12.9%	11.13%	5.2%	1.82%	13%	6.33%	2.1%	3%	1.2%	1%

Figure 3.3.1.1: Table of results with one non continuous source, $T = 10,000\mu sec$

Beside the columns corresponding to svc1 and svc2 we almost have the same table as before. The rate of svc1 changed from 34.01% to 42% and that of svc2 changed from 14.78% to 13%. This is approximately what was expected. The unused portion of svc2 went to its sibling svc1. The total rate received by svc1 and svc2 is 55% instead of 49%. The extra 5% can be explained by the fact that svc2 may have been taken to be inactive during the first update intervals that it becomes active. This result gives a hint to the potential problems with this method.

Limitations of the first method

The results presented here showed that the new implementation achieved a good approximation for appropriate values of T. However, the limitations of this method is in choosing the right value of T.

On one hand if T is too small compared to the packet inter-arrivals (as in figure 3.3.1.2) all services tend to have an equal share of the link. In figure 3.3.1.2 we repeat the same simulation as in figure 3.3.1.1 with a very small T. T equals to $200\mu sec$, $160\mu sec < T < 320\mu sec$, hence exactly one packet can arrive and its recipient will be the single active node during the update period. Thus, this node will be allocated 100% of the bandwidth, i.e every node that has a packet will be allocated the total bandwidth.

Services	*svc1*	*svc2*	*svc3*	*svc4*	*svc5*	*svc6*	*svc7*	*svc8*	*svc9*	*svc10*	*svc11*
Promised Rate	*34%*	*15%*	*13%*	*6.3%*	*2.1%*	*14.4%*	*7.2%*	*2.4%*	*3.4%*	*1.4%*	*1.2%*
Observed Rate	*9.81%*	*7.8%*	*9.32%*	*8.82%*	*9.01%*	*9.27%*	*8.92%*	*8.91%*	*9.4%*	*9.5%*	*9.3%*

Figure 3.3.1.2: Table of results with one non continuous source, $T = 200\mu sec$

We see from the table that all services share equally the link bandwidth.

On the other hand if T is too large the very short inactive period of a service won't be detected. The most limiting factor that makes this method unrealistic is that the magnitude of T is relative to the packet inter-arrival time of each source. Regardless of the constant chosen for T, there may always be a source with ON and OFF periods such that any two adjacent ON and OFF periods are included in a single update interval. Such source will be often considered continuously active even though it is not. Hence, given a value of T the performance of this method depends very much on the lengths of the ON and OFF periods of each source. To get a better insight of this phenomena, we repeated the same simulation configuration of figure 3.3.1.1. This time, using the same packet generation rate, svc2 generates packets in the ON and OFF pattern described in figure 3.3.1.3.a. It periodically generates packets for 1,000μsec and stop sending for a time interval equal to 5,000μsec. Figure 3.3.1.3.b is the ON and OFF pattern with which the same service, svc2, was generating packets in the previous simulation in figure 3.3.1.1.

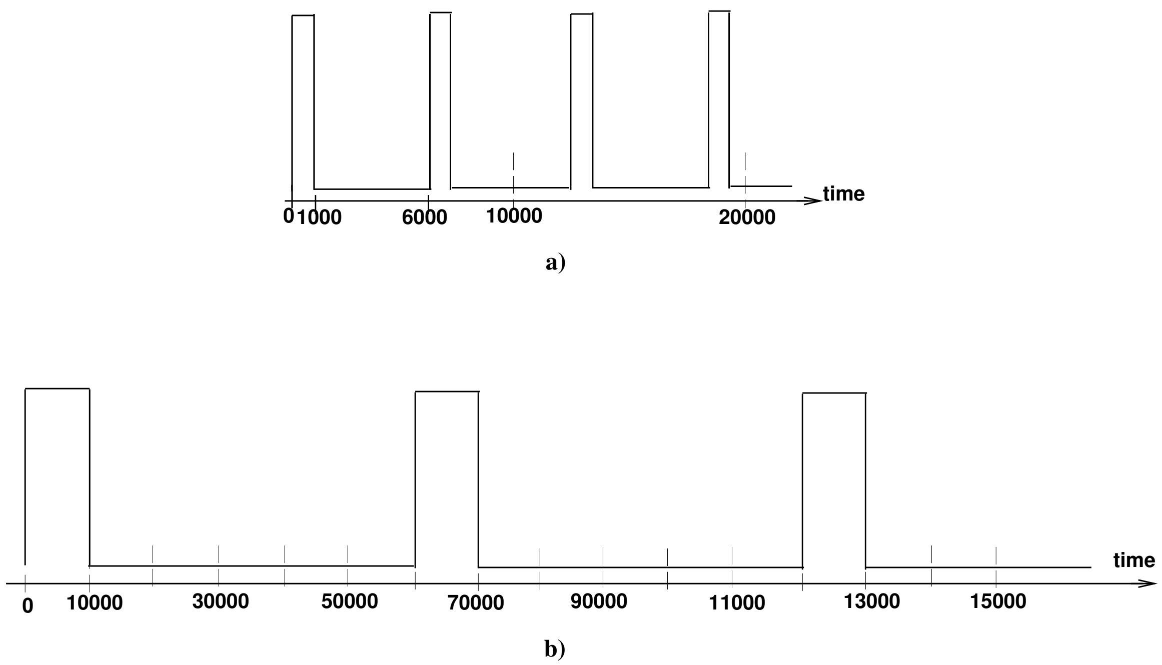


Figure 3.3.1.3: Two different ON-OFF paterns.

The ratio r between the ON and OFF periods is the same in both patterns, $r = \frac{1{,}000}{5{,}000} = \frac{10{,}000}{50{,}000}$. The update interval $T = 10,000$ is also the same in both cases. Hence, the same unused portion of svc2's share should be allocated to svc1 in the new ON and OFF patern (figure 3.3.1.3.a). However, because T is greater than the sum of one ON period and one OFF period (6,000=1,000+5,000), svc2 will be perceived as active in every update period. Thus the unused share of svc2 won't be transferred to svc1. This shows that the correctness of the implementation depends on the magnitudes of the ON and OFF intervals, which change with time, relative to the value of T which is a constant.

Since this algorithm guarantees fair sharing (in a sense; a regulator) of the link to all services, the correctness of its implementation can not depend on the behavior of the dynamically changing traffic characteristics (ON and OFF intervals) of the services. These concluding remarks about the first method motivate the need for an alternative method of coping with the ON and OFF sources. The next section presents the second method in which the concept of active and inactive states are redefined. This second method does not have the limitations presented above.

3.3.2 The Second Method

Instead of looking at the number of packets that arrived at a node during an update interval, a better way for detecting an active or inactive node is to compare its Fi to *now*; the real time. That is how the second method that we now present detects active and inactive nodes. A node is marked active if its Fi is greater or equal to *now* and inactive otherwise.

Recall that Fi measures the finish time of the packet at the head of the node, hence a leaf node that has at least one packet in its packet queue must have a Fi greater than *now*.

$$Fi_x(t_{(x,n)}) = max(Fi_x(t_{((x,n-1))}), now) + \frac{L_{(x,n)}}{(r_x)v}$$

And since Fi is updated only when a packet leaves a node and there is another packet in the node's packet queue, a node's Fi will remain non-incremented for the entire period that the node remained without a packet to send (inactive). Thus, eventually Fi will become less than *now* unless a new packet arrives at the node before its last packet finishes transmission.

In this new method, regardless of the value chosen for T, only those nodes whose Fi's are less than *now* will be marked inactive during the rate update. As in the previous method a large T will cause small inactive periods to go undetected, hence will cause less accurate results. However, we will see in section 4.3 that results for large values of T are very good approximations to the correct results.

Advantages of the Second Method

The advantage of this method over the first one is that the smaller the update interval, T, is the more accurate the results obtained are. For example, in the previous method if T is such that at most one packet arrival can occur within the time interval T the different nodes will tend to enjoy an equal share of the bandwidth (as simulated in figure 3.3.1.2). To illustrate the advantages of this method over the previous one we performed the series of simulations that we present below.

We repeated the simulations performed in figure 3.3.1.1 ($T = 10,000\mu sec$) and in figure 3.3.1.2 ($T = 200\mu sec$) with the second method.

In the case where $T = 10,000\mu sec$ we obtained similar results in both methods. However, the results were very different for $T = 200\mu sec$. In the next figure we display the rate table for $T = 200\mu sec$ (very small). As expected, the results are much better in the second method. The results collected here are also better than those collected in figure 3.3.1.1 where T=$10,000\mu sec$.

Services	*svc1*	*svc2*	*svc3*	*svc4*	*svc5*	*svc6*	*svc7*	*svc8*	*svc9*	*svc10*	*svc11*
Promised Rate	*34%*	*15%*	*13%*	*6.3%*	*2.1%*	*14.4%*	*7.2%*	*2.4%*	*3.4%*	*1.4%*	*1.2%*
Observed Rate	*38%*	*12.4*	*12.3%*	*6.1%*	*1.82%*	*13.6%*	*6.33*	*2.1%*	*3%*	*1.2%*	*1%*

Figure 3.3.2.1: Table of results with one non continuous source, $T = 200\mu sec$

However, one should bare in mind that the $200\mu sec$ value chosen for T is not realistic. The primary goal of this work is to minimize the computation necessary for the packet scheduling, hence the frequency at which the rates are updated. The $200\mu sec$ was chosen just for the purpose of comparing the two methods for very small values of T. Ideally we want T to be in the order of ten packets transmission time.

Chapter 4

Simulations over the Second Method

The simulations of the second method presented in the previous chapter were primarily done to compare it to the first method. Having shown the advantages of the second method over the first one we will, in this chapter, investigate the performance of the second method in greater details. Here we will first look at the behavior of the sharing over small time intervals. The results presented in the previous chapters presented the different services' average bandwidth shares which were computed over the entire length of the simulation. It is of sspecial interest to look at the behavior of the sharing over small time intervals. For example, it is crucial to an interactive video service to be guaranteed its link share at all points in time. Furthermore, we need to examine the interdependence between the shares of the different services over very short intervals. That is, we need to see how the reallocation of a share to some services will affect the shares of the others. Second we will present what happens with very small inactive periods. And finally we will look at the performance of the implementation for values of T that are in the magnitude of 10 packet-departures-times.

4.1 Results over Small Update Intervals

For the purpose of exposing some issues that the previous simulation configuration did not expose, we make small changes to the configuration. The resulting configuration is the same as the previous one except that the source associated with svc1 has ON and OFF periods respectively equal to $100,000\mu sec$ and $10sec$, and generates packets during its active period at the rate of $64.3469 \times 10^{-6} pkt/\mu sec$ which is just enough to fill its link share. The source associated with svc2 now generate packets continuously at a rate of $28.38792 \times 10^{-6} pkt/\mu sec$ which is just enough to use up its link share. The update interval T is $0\mu sec$. We chose $0\mu sec$ not to be distracted in our analysis by the effects of large update intervals. We will examine these effects in section 4.3. We present in the table below the shares received by the different services over the total length of the simulation and in the following graphs we present the shares of each individual service over short time intervals of 100 packet-departure-times.

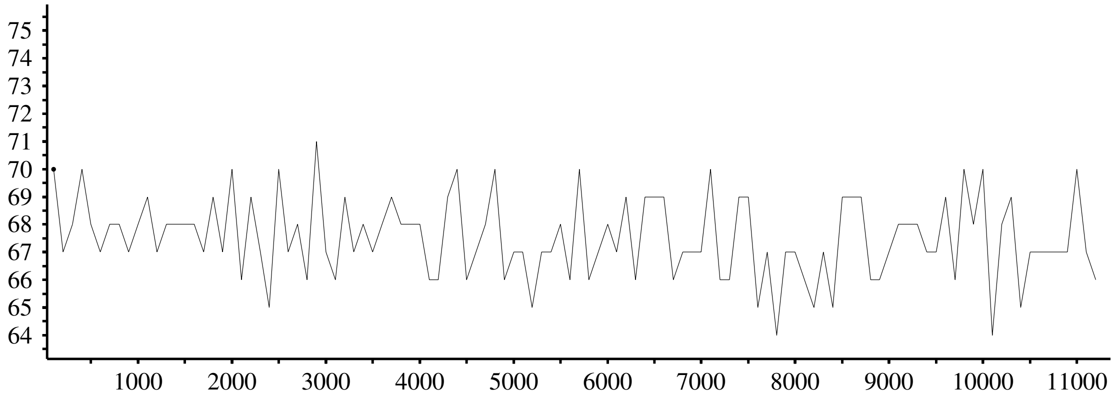
sim_snap94:04:20@18:10

Services	*svc1*	*svc2*	*svc3*	*svc4*	*svc5*	*svc6*	*svc7*	*svc8*	*svc9*	*svc10*	*svc11*
Promised Rate	*34%*	*15%*	*13%*	*6.3%*	*2.1%*	*14.4%*	*7.2%*	*2.4%*	*3.4%*	*1.4%*	*1.2%*
Observed Rate	*.44%*	*15.1%*	*30.3%*	*16%*	*5.7%*	*15.6%*	*7.8%*	*2.58%*	*3.6%*	*1.5%*	*1.29%*

Figure 4.1.1: Table of average shares of the different services, $T = 0\mu sec$

We note however that the sum of the shares of services 1,2,3,4 and 5 is 67.54% instead of 70%. The missing 2.46% is shared among the others services. We will investigate this phenomena in greater details in section 4.2.

sumsvc12345_sim_snap94:04:20@18:10.ps



o **Figure 4.1.2:** Sum of services 1,2,3,4 and 5 share over time. It fluctuates between 66% and 70%.

Even though svc1 and svc2 have inactive periods, we see from this curve that the share of *node1* which is the sum of services 1,2,3,4 and 5 stays constant around 67.54%. 67.54% is the average of the share of *node1* computed over the total length of the simulation. Thus, the implementation keeps the share allocated to *node1* no less than its share, hence achieves criteria *i)* of the sharing for svc1 (associated with *node1*). Criteria *i)* of the sharing requires that each active service be guaranteed a share of the link no less than the percentage of the link it owns. We show that criteria *i*) is also satisfied for all services by presenting the shares of each individual service in the next graphs.

svc1_sim_snap94:04:20@18:10.ps

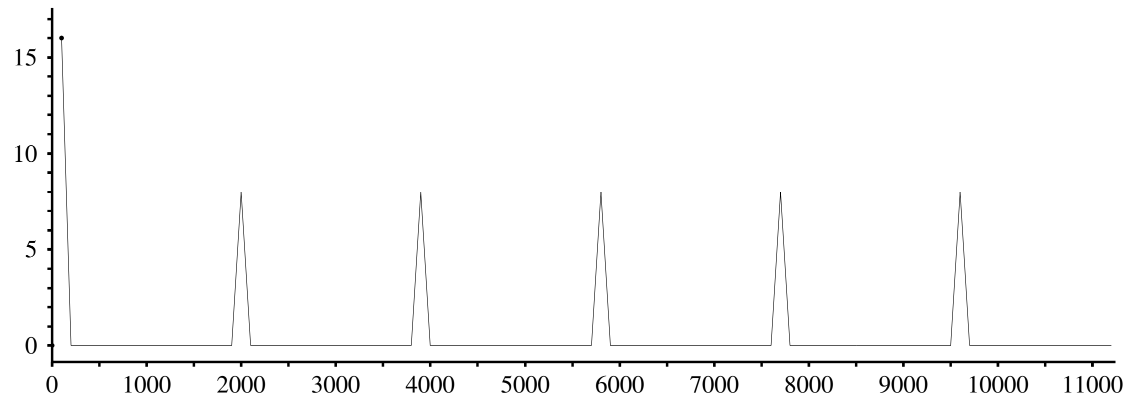


Figure 4.1.3: svc1 share over time; originally 34%.

Given an outgoing bottleneck link speed of $.300 bit/\mu sec$, 100 packet-departure-times equal $533,333 \mu sec$.

$$\frac{100 \times 1600 bits}{.300 bits/\mu sec} = 533,333 \mu sec$$

Since svc1 can generate exactly 8 packets during one active period of $100,000 \mu sec$ and its share is 34% (34 in a sample of 100), all 8 packets will be sent within the next 100 packets sent. That show that svc1's share is available to it whenever it becomes active.

svc2_sim_snap94:04:20@18:10.ps

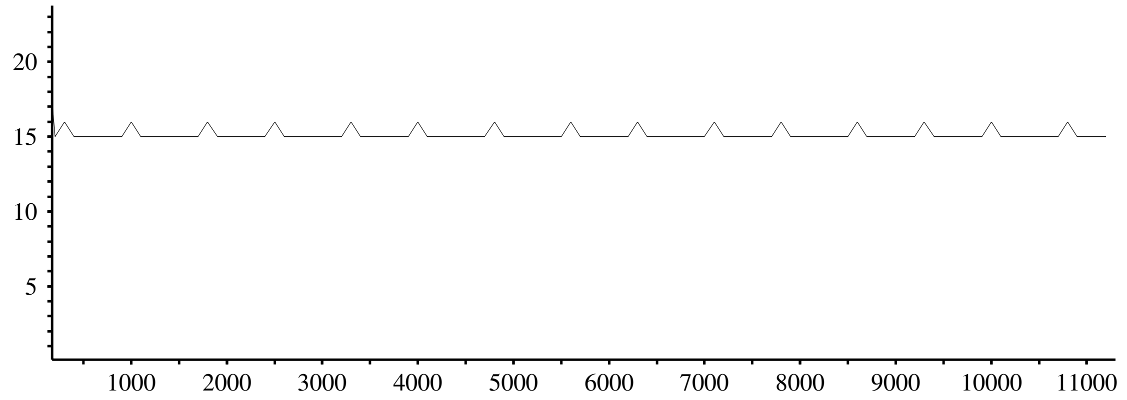


Figure 4.1.4: svc2 share over time; originally 15%.

Within 100 packet departures svc2's source generates 16 packets ($=28.38792 \times 10^{-6} pkt/\mu sec \times 533,333 \mu sec$). With a 15% share of the total bandwidth, svc2 can only send 15 packets within 100 packet departures. During svc1's inactive period svc2 can have up to 49 ($= 15 + 34$ (from svc1's share)) packets sent in 100 packet departures using svc1's share. Because svc2 has only 16 packets to send in every 100 packet departures it will get all its 16 packets sent during the 100 packet departures. Since svc2 generates packets at a rate slower than its rate of service when svc1 is inactive, there will be periods when svc2 will go inactive. The inactivity of svc2 which is caused by the inactivity of svc1 will causes in turn together with the inactivity of svc1 the fluctuation in the shares of services 3,4 and 5 as we will see.

We see in the next three graphs that the unused shares of services 1 and 2 are redistributed to services 3,4 and 5. We also see that the redistribution of the shares conserves the ratios between the different shares. Thus, criteria *ii)* of the sharing is continuously met. Criteria *ii*) requires that the ratio between the shares of any two services of the same parent to be always constant. The fluctuations in these curves are due to the extra shares that their corresponding services receive periodically when *node1* is inactive.

svc3_sim_snap94:04:20@18:10.ps

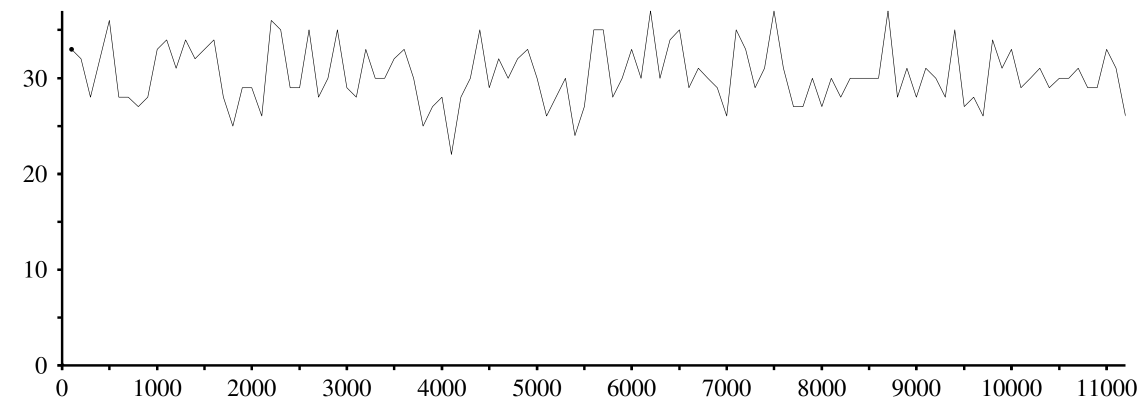


Figure 4.1.5: svc3 share over time; originally 13%.

svc3's share fluctuates around 30%. The ratio between the share of svc3 and that of svc4 is $\frac{30}{17.5} = 1.7$ which is approximately equal to the ratio between the original shares of the two services ($2 \approx \frac{13}{6.3}$). The minimum value of the curve is 23 which is greater than the original share; 13.

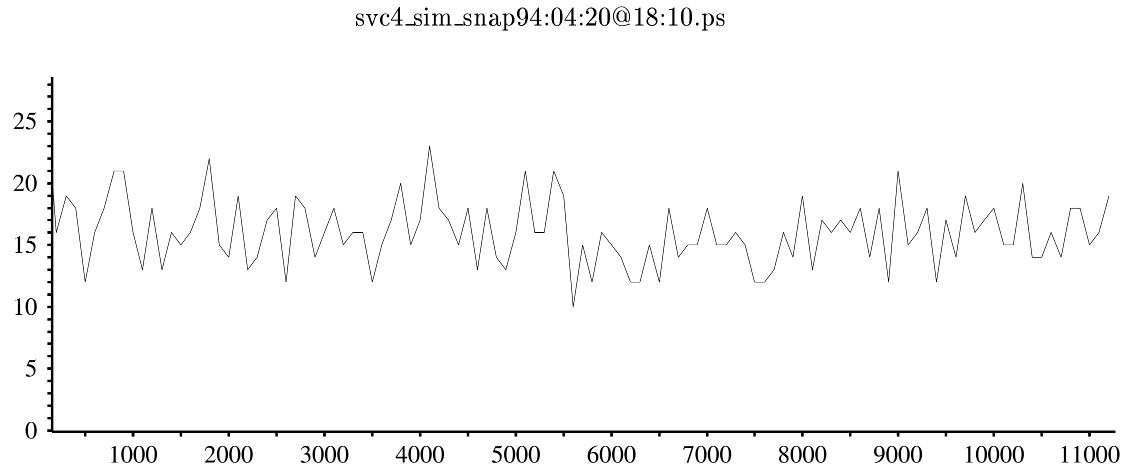


Figure 4.1.6: svc4 share over time; originally 6.3%.

svc4's share fluctuates around 17.5%. The ratio between the share of svc4 and that of svc5 is 2.9 ($= \frac{17.5}{6}$) which is approximately equal to the ratio between the original shares of the two services ($3 = \frac{6.3}{2.1}$). The minimum value of the curve is 10 which is greater than the original share; 6.3%.

svc5_sim_snap94:04:20@18:10.ps

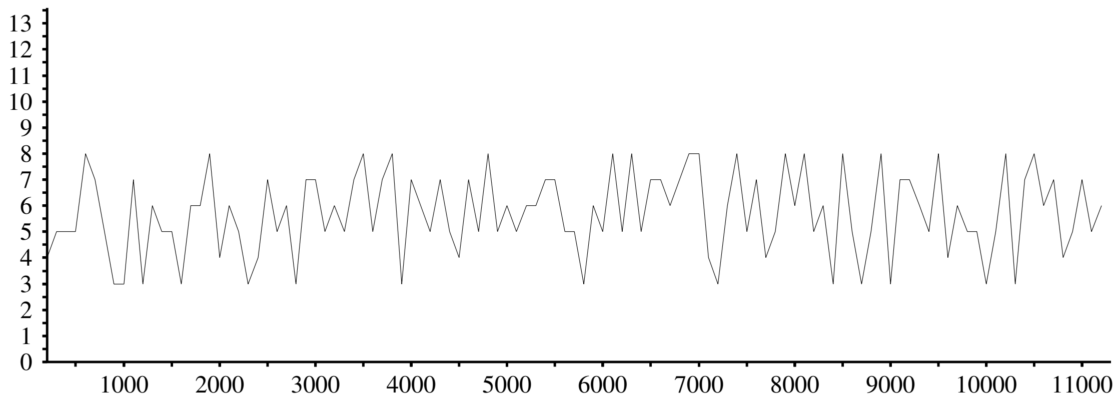


Figure 4.1.7: svc5 share over time; originally 2.1%.

svc5's share fluctuate around 5%. The ratio between the share of svc1 and that of svc5 is $\frac{30}{6} = 6$ which is approximately equal to the ratio between the original share of the two services ($6.2 = \frac{13}{2.1}$). The minimum value of the curve is 3 which is greater than the original share (2.1%).

Because of the inactivity of *node3*[1] (svc1 and svc2) all three preceding curves have minimum values that are greater than the original share of the corresponding services.

The next six graphs of the remaining services; svc6, svc7, svc8, svc9, svc10 and svc11, have a much smaller range of fluctuation. For example svc6's graph takes its values between 14 and 16. Since these services don't have siblings with inactive periods, one might have excepted these graphs to be horizontal lines. The graphs are not horizontal because the WFQ model being used in the scheduling is done on packet basis, it is not the flow model. Furthermore we plotted the number of packet departures against the time axis, therefore the graphs of services with shares of rational value x will tend to take values from $\{\lfloor x \rfloor, \lceil x \rceil\}$. For example, svc6 which own 14.4% of the link share has its graph take its values between 14 and 16.

[1] *node3* is the parent node of svc1 and svc2 in figure 3.3.1. It is inactive if svc1 and svc2 are both inactive and its share is the sum of the shares of svc1 and svc2..

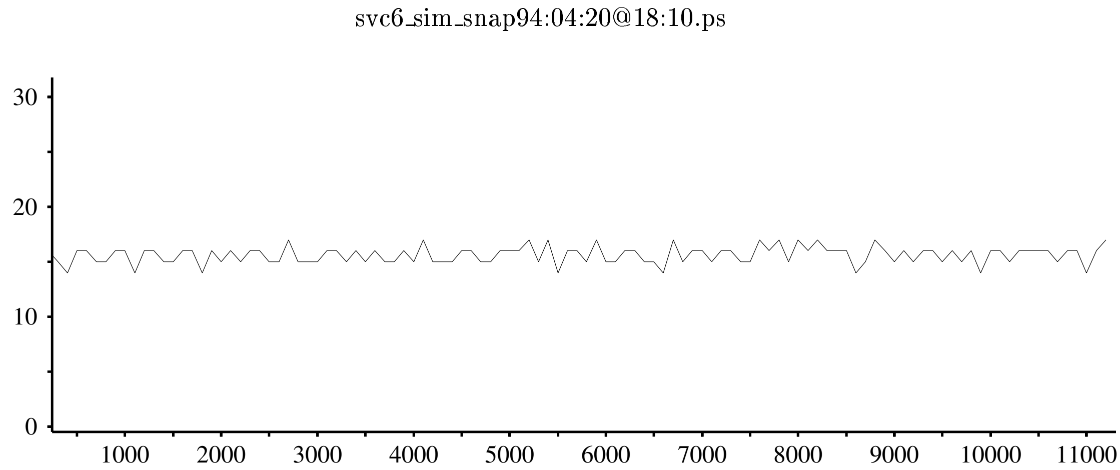


Figure 4.1.8: svc6 share over time; originally 14.4%.

svc6's share is almost constant around 14.4%; its share.

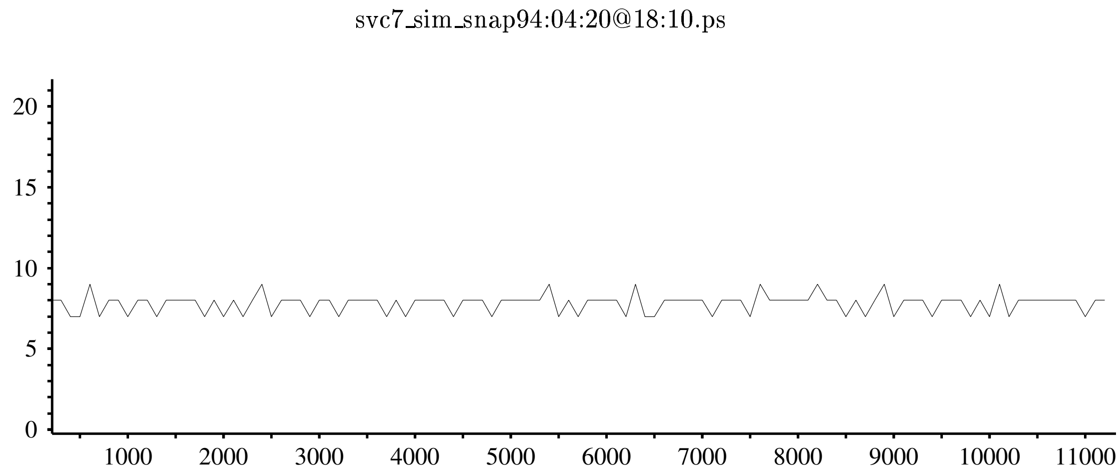


Figure 4.1.9: svc7 share over time; originally 7.2%.

svc7's share is almost constant around 7.2%; its share.

svc8_sim_snap94:04:20@18:10.ps

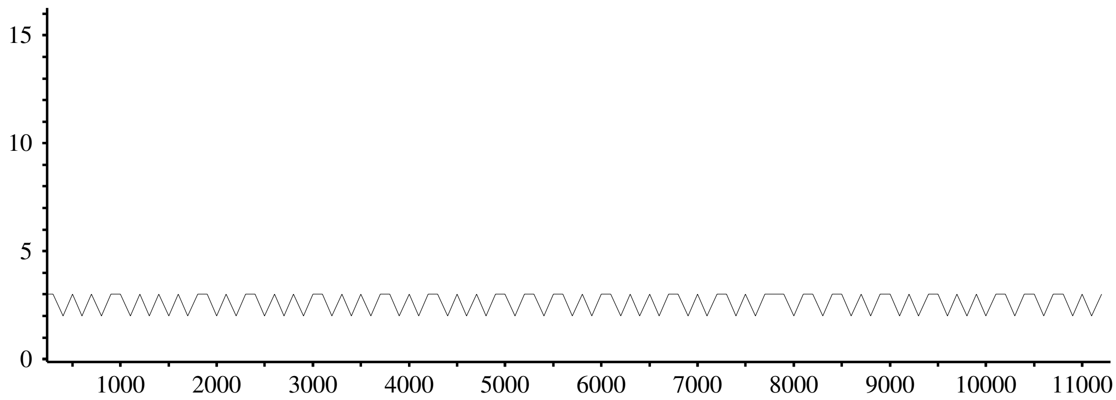


Figure 4.1.10: svc8 share over time; originally 2.4%.

svc8's share is almost constant around 2.4%.

svc9_sim_snap94:04:20@18:10.ps

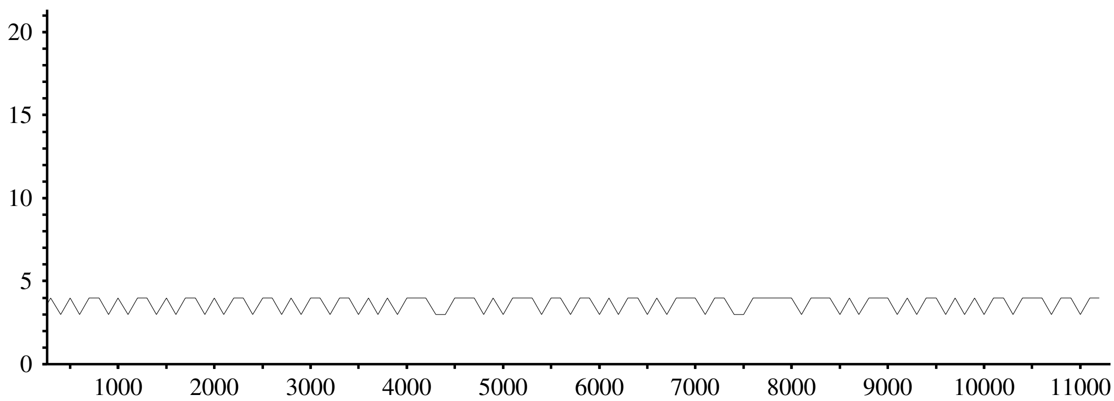


Figure 4.1.11: svc9 share over time; originally 3.4%.

svc9's share is almost constant around 3.4%.

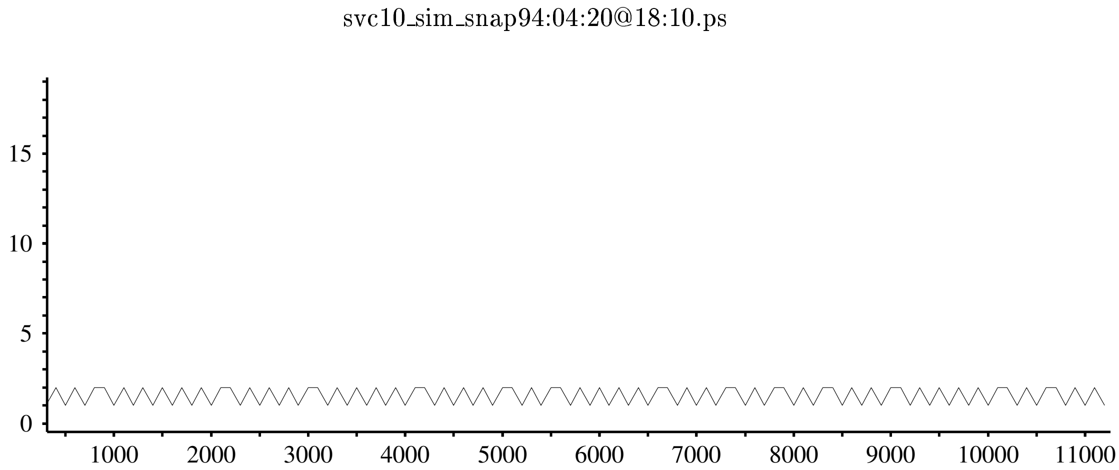


Figure 4.1.12: svc10 share over time; originally 1.4%.

svc10's share is almost constant around 1.4%.

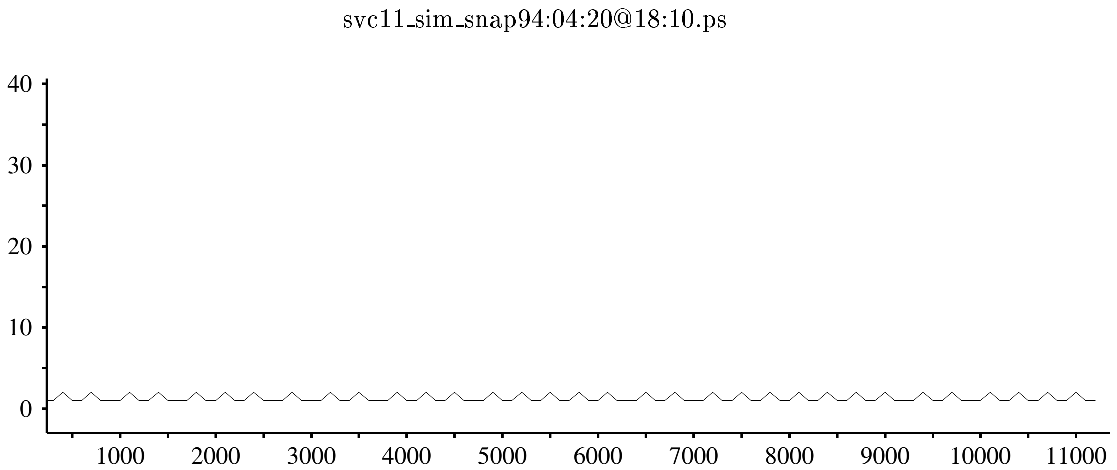


Figure 4.1.13: svc11 share over time; originally 1.2%.

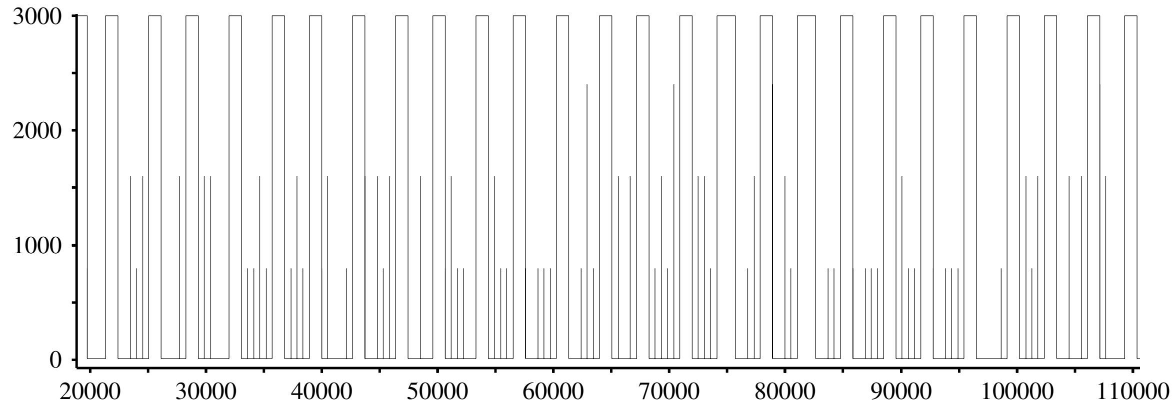
svc11's share is almost constant around 1.2%.

4.2 What happens when the inactive periods are very small

In the previous section, we mentioned the fact that sum of the shares of services svc1, svc2, svc3, svc4, and svc5 (*node1*[2]) was 67.54%. This is due to the small periods during which svc1 and svc2 are simultaneously inactive. Even though the sum of their shares is immediately ($T = 0 \mu sec$) reallocated to services svc1, svc2 and svc3, any of these services will get the chance to use the extra share from svc1 and svc2 only if its Fi gets updated before the period during which svc1 and svc2 are inactive terminates. Thus, although svc3, svc4 and svc5 each continuously have packets to send they won't get to use the extra share if they don't get a packet departure (that is when the new share will be used to compute the new Fi) within the inactive period of svc1 and svc2.

To illustrate this phenomena we sampled in figure 4.2.1 the packets departures of services svc3, svc4 and svc5 during the periods that svc1 and svc2 are both inactive. The data in this figure is based on the simulation configuration of the previous section.

xplot_sim_snap94:01:17@17:50.ps



[2] *node1* is the parent node of *node1* and *node4* in figure 3.3.1 and its share is the sum of the shares of svc1, svc2, scv3, svc4 and svc5. *node4* is the parent node of svc3, svc4 and svc5 and its share is the sum of their shares.

Figure 4.2.1: Number of pkt departures from svc3, svc4 and svc5 within the periods that svc1 and svc2 are both inactive. svc2 generates packets at a rate just enough to fill up its link share. The intervals at which the curve takes the value 0 correspond to the inactive periods of both svc1 and svc2. The horizontal axis is in *ticks* ($1tick = 10\mu sec$). The length of an inactive period is about $25,000\mu sec$ and that of an active one about $10,000\mu sec$. The vertical lines of height 800 inside the inactive interval mark a packet departure from svc3, those of height 1600 and 2400 respectively mark packet departures from svc4 and svc5.

We see that svc4 does not have packet departures in some inactive periods of svc1 and svc2, and svc5 does not have packet departures during many inactive periods of svc1 and svc2. Thus, a good part of the extra share allocated to svc4 and svc5 will be equally shared by all active services in the system. We investigate this phenomena further by repeating the same simulation in the next two cases.

In the first case we reduced svc2's packet generation rate to $7.09698 \times 10^{-6} pkt/\mu sec$ ($= \frac{1}{4} \times 28.38792 \times 10^{-6} pkt/\mu sec$) which is just enough to fill one quarter of its link share. Even though svc2 has less packets to send, the overall fraction of packets of *node1* (svc1, svc2, svc3, svc4, svc5) serviced has improved to 68.7%. The improvement is due, as shown in the figure below, to the wider inactive time intervals of svc1 and svc2. As the intervals get wider more packet departures from the siblings of svc1 and svc2 occur within the intervals. Thus services; svc3, svc4 and svc5 get the chance to use the extra share more often.

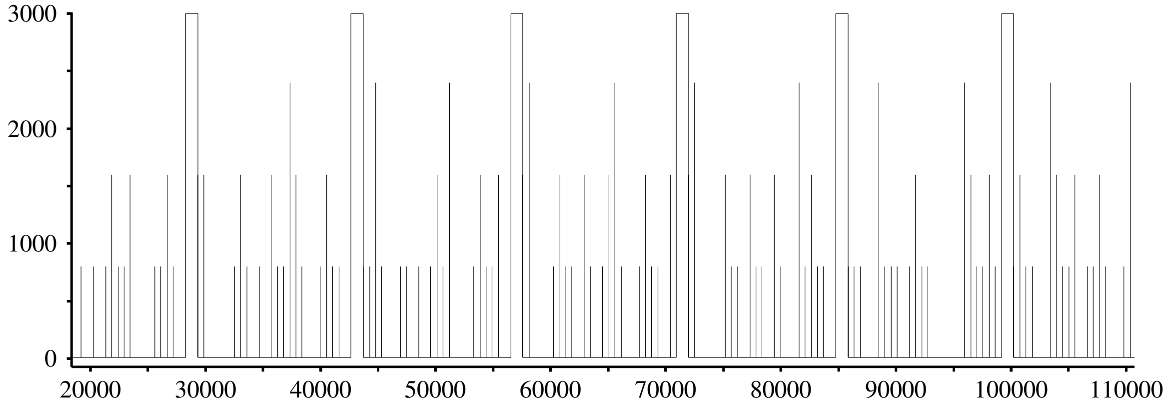


Figure 4.2.2: Number of pkt departures from svc3, svc4 and svc5 within the periods that svc1 and svc2 are both inactive. svc2 generates packets at a rate just enough to fill up one *quarter* of its link share. The intervals at which the curve takes the value 0 correspond to the inactive periods of both svc1 and svc2. The horizontal axis is in *ticks* ($1 tick = 10 \mu sec$). The length of an inactive period is about $120,000 \mu sec$ and that of an active one about $10,000 \mu sec$. The vertical lines of height 800 inside the inactive interval mark a packet departure from svc3, those of height 1600 and 2400 respectively mark packet departures from svc4 and svc5.

In the second case (figure 4.2.3), svc2 generated packets at a rate of $3.54849 \times 10^{-6} pkt/\mu sec$ which just fills up one $\frac{1}{8}$ of its link share. As expected from the previous argument, the total share observed for *node1* improved to 69.5% which is a good approximation to 70%, the target value.

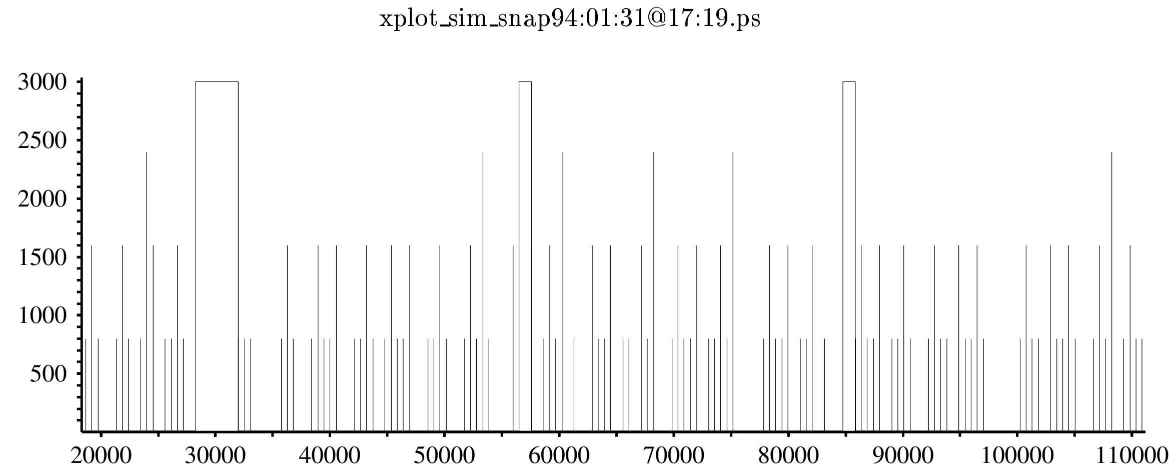


Figure 4.2.3: Number of pkt departures from svc3, svc4 and svc5 within the periods that svc1 and svc2 are both inactive. svc2 generates packets at a rate just enough to fill up one *eighth* of its link share. We note that the total number of packet departures of svc3, svc4 and svc5 in each inactive interval has increased considerably. The length of an inactive period is about $275,000 \mu sec$ and that of an active one about $10,000 \mu sec$.

We conclude the discussion regarding small inactive intervals with two remarks.

The first remark is that packets generation in real networks are pretty much like poisson processes and the effect of small inactive periods is less important if the packets generation of the different services are poisson. Because of the uniform distribution 1^{st} order packet inter-generation of poisson sources the length of the simultaneous inactive periods of svc1 and svc2 are more randomly distributed than in the deterministic case. For example, in a similar simulation to that of figure 4.2.1 with poisson sources *node1* received 68.2% as oppose to 67.54% in figure 4.2.1.

The second remark is that the effect of small inactive periods is even less noticeable if the packet sizes are small. Given a interval t and the same outgoing link speed, there will be more packet departures within t if the packets sizes are small than when they are big. Hence the siblings of an inactive service will get to use the extra share from the service more often. It then follows for the same reason that the effect of small inactive periods will be less noticeable in faster networks. That makes this implementation even more suitable for high speed networks.

4.3 10 pkts-times Update Interval

As already mentioned, the primary goal of the new scheme is to reduce the computation overhead per packet that the old scheme suffered from. For the purpose of analyzing some specific details of the implementation without being distracted by the effect of large update intervals we used small values $(0 - 10,000 \mu sec)$ for T in the preceding simulations. In this section we formally present the results of the simulations for $T = 53,333 \mu sec$ which is equal to 10 packet-departure-times.

$$\frac{53{,}333 \mu sec \times 300 Kbits/sec}{1600 bits} = 10 pkts$$

The simulated configuration is identical to that of section 4.1 (svc2 was generating packets at a rate that just fills up its link share). We chose to use this configuration instead of the ones in the previous section because we already have detailed graphs in section 4.1 that we can compare the results for $T = 53,333 \mu sec$ against.

The average shares over the total length of the simulation of each service is displayed in the table below. We present the shares at different points of the simulations in the next three graphs. The results in this table are approxmiations to those of the table in figure 4.1.1 where $T = 0\mu sec$. For example the share of *node1* averages to 65.9% as oppose to 67.54% in section 4.1 where $T = 0\mu sec$. The approximations are even better for the case where the shares of the potential inactive nodes are not way greater than the average share per service.

sim_snap94:04:23@02:18.T=53333.ps

Services	svc1	svc2	svc3	svc4	svc5	svc6	svc7	svc8	svc9	svc10	svc11
Promised Rate	34%	15%	13%	6.3%	2.1%	14.4%	7.2%	2.4%	3.4%	1.4%	1.2%
Observed Rate	.44%	15.1%	31.4%	14.7%	4.3%	16%	8.2%	2.7%	3.8%	1.6%	1.3%

Figure 4.3.1: Average shares of the different share, $T = 53,333\mu sec$

We present the next three graphs to show that the shares over time of the individual services are approximately the same as those of section 4.1.

svc1.2.3.4.5_sim_snap94:04:23@02:18.ps

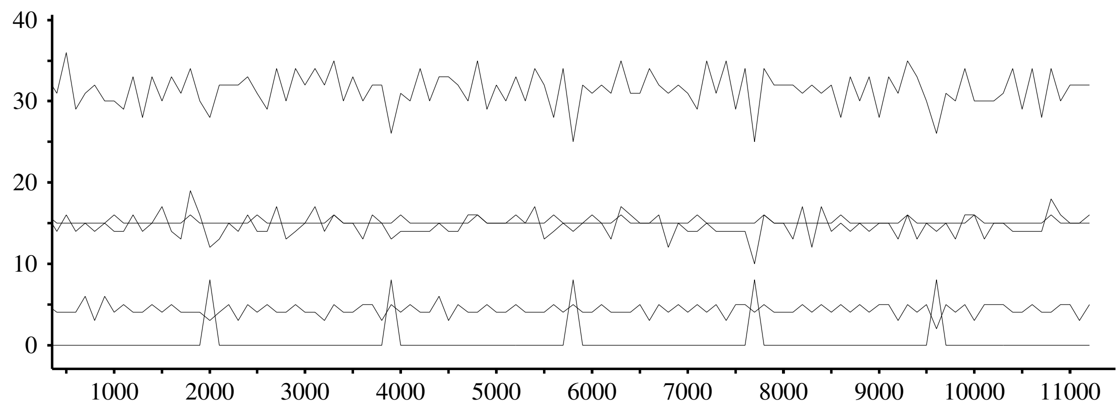


Figure 4.3.2: svc1, svc2, svc3, svc4 and svc5 shares over time; $T = 53,333\mu sec$. The top curve corresponds to svc3. The two curves that take their values in the vicinity of 15 correspond to svc2

and svc4; the one that looks like a horizontal corresponds to svc2. The bottom curve with average value 4.3 corresponds to svc5 and the very bottom one corresponds svc1.

svc6.7.8.11_sim_snap94:04:23@02:18.ps

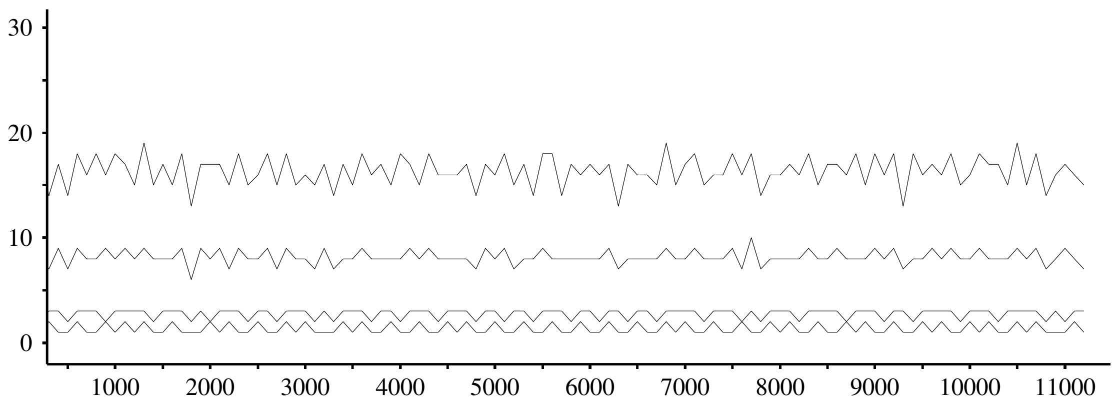


Figure 4.3.3: services 6, 7, 8 and 11 shares over time; $T = 53,333\mu sec$. The top curve corresponds to svc6, the one that has values in the vicinity of 8 corresponds to svc7. svc8 corresponds to the curve with average value 2.7. The very bottom curve corresponds to svc11.

svc9.10_sim_snap94:04:23@02:18.ps

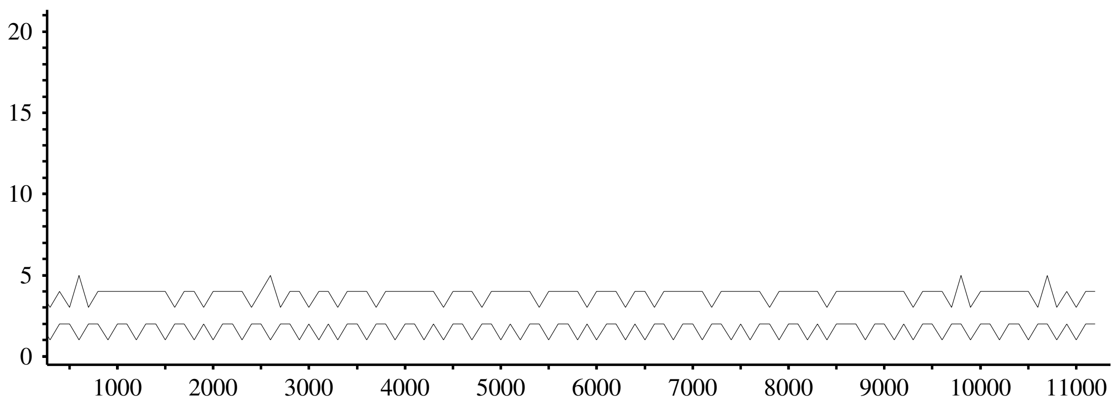


Figure 4.3.4: services 9 and 10 shares over time; $T = 53,333\mu sec$. The upper curve corresponds to svc9 and the bottom one corresponds to svc10.

Comparing the curves displayed in the previous three graphs to their correspondents in section 4.1 we see that the sharing behaved the same way as in section 4.1 expect that *node1* lost 2.4% ($\frac{(67.54-65.9)}{67.54}$) of the share it received in the simulation of section 4.1 where $T = 0\mu sec$.

This loss is due to two facts:

1.) The update interval is larger than one active period or one inactive period of svc1 and svc2 (*node3*).

2.) An inactive period; $25,000\mu sec$, of *node3* is larger than one active period of *node3*; $10,000\mu sec$ (from figure 4.2.1).

Given 1) an active or inactive period of *node3* can occur without being detected at update time. If an inactive period is undetected at update, *node1* will be given less share than it owns (loss) because the unused share of *node3* from its inactive period won't be reallocated to *node4*. And if an active period is undetected at update, *node1* will be given more share than it owns (gain) because the share of *node3* will be allocated twice to *node1*; once through *node1* and a second time through *node4*. Thus, given 2) the average will result to a loss for *node1* which explain the 2.4% loss. If the sizes of the active and inactive periods in 2) are interchanged the average results in a 2.4% gain for *node1*. However, in a real network typically when 1) applies the loss will tend to average out to zero because of the randomness of the lengths of both active and inactive periods. As a result of the randomness active periods will be alternatively smaller (loss) and larger (gain) than active periods.

56

Chapter 5

Conclusions

5.1 A Good Approximation

When optimizing, one has to identify a desired upper bound of the optimization variable (*eg:* instructions count). If this bound is not attainable while being completely faithful to the algorithm, one would have to decide on a tolerance level below which the implementation is allowed to deviate from the specification and still be correct. Following this engineering principal, we found a neat compromise between time constraint and correctness. While maintaining the correctness, we were able to speed up the packet processing time in the sense that the steps that were needed to be taken in the original code at packet arrival and packet departure were broken into different components that could approximate the original scheme while running in parallel.

The parallelism resulted in a cost increase in hardware. However the parallelism also resulted in a speedup of the packet scheduling process which in turn resulted in a greater processing capacity. The processing time of a packet enqueue is now reduced from 679 to 55 instructions yielding an improvement by the factor of 14.7. The time required to process a packet dequeue reduced from 875 to 70 instructions yielding an improvement factor of 12.5. Thus, the resulting capacity is multiplied by a factor of 13.

We showed through different simulations that although the rates are now being updated periodically, the new implementation achieved good results for reasonably large update interval. We got very good approximations to the original implementation for update intervals in the magnitude of 10 packet-times.

5.2 A Remark about the Buffer Size

Through the course of this work the buffer size seemed irrelevant to the performance of the algorithm. Nevertheless, in this section we present a short remark about the buffer size.

In our implementation the buffer is shared among the services on a need basis. When the buffer overflows the packet drop policy being presently used is to drop the last packet of the service with the longest packet queue. If the buffer size is very small the WFQ mechanism used in the packet scheduling will perform poorly. To illustrate what might happen assume it is the case that the buffer size and the number of services are such that the length of the packet queue to drop a packet from is always 1. A trivial scenario in which such case occurs is when the number of simultaneous active services is n and the buffer size is less than n packets. Now assume that there is an active service, svc, whose source is generating packets at a rate that exactly fills its link share. In addition assume that svc's packets arrive at the buffer one shortly before the previous one get to leave the buffer. In this specific case svc will experience packet drops even though it should not have because its source is generating packets according to its link share. Therefore, svc will receives less than its link share.

The phenomena discussed in the previous paragraph can be avoided in two ways:

One solution is to have a buffer size such that any active service is guaranteed at least two packets of buffer space. Guaranteeing each service with a buffer space of 2 packets ensures that each service will always have at least one packet in the buffer ready to be sent as long as the service generates enough packets. The guarantee is achieved by a buffer of size at least $(2N+1)P_{max}$ where N is the number of services simultaneously active in the system and P_{max} is the maximum packet size. This way, the longest packet queue to drop a packet from will always have at least 3 packets.

If the buffer size can not be greater or equal to $(2N+1)P_{max}$, the alternative solution which is a good area of future work is to change the packet drop policy. One possible policy which must be subject to further study would be to drop a packet from the service with the longest packet queue if the length of this queue is more than 2 else drop a packet from the service with the greatest Fi.

5.3 Implementation Issues

5.3.1 Summary of the instruction counts

In the next table we summarize the instruction counts of the different routines for different tree sizes.

ROUTINE NAME	*NUMBER OF INSTRUCTIONS for a tree of p wfq leaf nodes and a total of n wfq nodes*	
	without priority	*with priority*
Tree_enque	47	55
Tree_deque	31	70
Update_rate	* (see blow)	* (same)

$$*=68n-65p+11+(11+18a+7log_a(u))u+(5+37a)log_a(\tfrac{u}{2})$$

where n is the total number of nodes in the tree, p is the number of *wfq* leaf nodes, a is the average number of children per node and u is the number of packet-times per update interval.

Note that *Update_rate* is independent of the number of *priority* nodes. It depends only on the number of *wfq* nodes.

5.3.2 Hardware dependencies

The major advantage of the new scheme relies on the possibility to accomplish the rates update in the background within 10 or 20 packet-times. And since the measure of a packet-time depends on the link speed, the required processor speed to complete the rates update will depend on the size of the tree and the link speed. The table below gives the required processor speed for different tree sizes and link speeds.

For example, given a tree of 40 nodes with a total of 27 *wfq* leaf nodes, average number of children equal to 3 and a 10 packet-times update interval *Update_rate* will require $1,613$ $(=(68\times40)-(65\times27)+725$ $(68\times40-65\times27+11+10(11+(18\times3)+7log_3(10))+(5+37\times3)log_3(5))$ instructions. Thus, given a link speed of $10Mbits$ and packet sizes of 200 $bytes$, a $1Mips$ ($\approx 1,613 \times \frac{10M}{10\times200\times8}$Mips) processor is required to complete *Update_rate* within 10 ($u = 10$)

packet-times ($=1.6msec$)

TREE SIZE	*LINK SPEED (Mbits/sec)*		
	1	*10*	*100*
Full 3–nary tree of depth 3 (n=40, p=27)	.1Mips	1Mips	10Mips
Full 5–nary tree of depth 5 (n=3906, p=3125)	4Mips	40Mips	400Mips
n=10,000 p=9000 a=10	7Mips	70Mips	700Mips

This table is based on packet lengths of 200 bytes.

To use our implementation, one must take into account the table above and that the buffer size should be large enough as discussed in section 5.2 .

5.4 Future work

The implementation of the sharing presented in this thesis also works with priority nodes as long they have *pkt* child type and parent nodes of type *wfq*. In this implementation, the Fi of a *wfq* node of child type *priority* is computed using its current rate as usual and the length of the packet at the head of the packet queue (the first packet) of its active child node with the highest priority. We refer to such a priority node as the current priority (*cur_pri*) of the *wfq* node.

When a child node, x, with a priority greater than that of its parent node's current priority becomes newly active, the parent's *cur_pri* needs to be updated. The parent's *cur_pri* is set to x. Also the parent's Fi which was computed with the length of the first packet of the parent's former *cur_pri* needs to be updated. The newly arrived packet at x, now the *cur_pri*, may be of different size than that of the first packet of the former *cur_pri*. Thus, the parent's Fi is updated as follows:

$$\text{parent}\!\rightarrow Fi = +\frac{-(p_1\!\rightarrow\! len)+(p_2\!\rightarrow\! len)}{(parent\!\rightarrow\! cur\_rate)\times LINK\_SPEED}(5.1)$$

Where p_2 and p_1 are respectively the first packets of x and the former *cur_pri* of *parent*; the parent node.

The reasons behind the update in (5.1) is to first take into account the difference in length of p_1 and p_2 and second to consider the time (value of now) when p_1 was last used in the formula below ((5.2)) to be the arrival time of p_2 at x. To illustrate the second reason behind updating according to (5.1), suppose that instead of updating the parent's Fi using (5.1) when p_2 arrives at x we update Fi as follows: First decrement the parent's Fi by $\frac{p_1\rightarrow length}{parent\rightarrow cur\_rate\times LINK\_SPEED}$ and apply (5.2) with $p = p_2$. If $\max(parent \rightarrow Fi, now)$ happens to be now when (5.2) is applied then the length of time the parent node, par, spent waiting in llq while p_1 was its first packet to be sent won't be taken into account in its new Fi.

$$parent\!\rightarrow Fi = \max(parent \rightarrow Fi, now) + \frac{p\rightarrow len}{(parent\rightarrow cur\_rate)\times LINK\_SPEED}(5.2)$$

This equation is the same as equation (2.1), the usual formula used to compute the Fi's, written in a different form.

Notice that it may be the case that the value of *par*'s *cur_rate* used in (5.1) (current value) may not be the same (old value) when *par*'s Fi was computed using the length of p_1 for the first time. This case may occur when the active states of the *par*'s siblings have changed between the two Fi updates (when p_1 becomes the 1^{st} packet of the former *cur_pri* and when p_2 arrived at x). Even though updating according to (5.1) takes into account the difference in packet sizes and the arrival time of p_1 it does not take into account the fact that the value of *par*'s *cur_rate* may not be the same in both updates. Thus, it is not clear what the effects of the update in (5.1) would be to the over all performance of the implementation.

It would be of interest to conduct simulations that would give an insight to the effects of (5.1) if any. A fix to (5.1), if need be, could be the use of the value of *cur_rate* from the first Fi update in the decrement in (5.1) instead. In this new version of (5.1) it is still unclear the usage of which of the two values of *cur_rate* in the increment in (5.1) will best serve the fairness of the sharing. Should we always use the current value of *cur_rate*, use the current value only when it is greater (smaller) than the old value, or always use the old value?

Always using the current value of *cur_rate* will not take into account the extra share from the siblings of *par* that were inactive during the first update. Using the current value of *cur_rate* only when it is greater (smaller) than the old value will introduce a bias of the sharing in favor of (against) the *wfq* nodes of child type *priority*. Even though always using the old value may seem to be the alternative of choice, it is not evident that it is best to continue to use the old value even when the current value is greater because some siblings of *par* just became inactive. These are issues that we would like to investigate in the future.

Appendix A

Source Codes

We present here the C programming language source codes of the main routines of the implementations. The header file of the tree data structure is in appendix A.1. In appendix A.2 is described a new routine; Tree_add which just creates the tree node structure and initializes it. The source listings of Tree_enq, Tree_deque, Update_rate, Mark_active and Mark_inactive are respectively in appendices A.3, A.4, A.5, A.6, and A.7.

A.1 The Tree Data Structure

```
#define ROOT 0
#define WFQ 1
#define PRIORITY 2
#define PKTBUF 3

#define BEEN_ACTIVE   1
#define BEEN_INACTIVE 0

typedef struct tree{
  struct tree *next;  /* parent child Queue */
  int type;           /* Root, WFQ, Priority */
  int child_type;     /* WFQ, Priority, Pkt  */
  PQ pQ;              /* if child = PKT, pkt queue */
```

```
  struct tree *parent;/* Ptr to the parent node */
  int w;              /* If WFQ, share of link */
  float rate;       /* how many rounds to send a bit*/
  unsigned long Fi;   /* finish of active last packet */
  unsigned lng;       /* Len of first packet below it*/
  int pri;            /* If priority, the priority  */
  struct tree *dropQ; /* pkt dropping queue */
  int npkt;           /* if child = PKT, pkt count  */
  struct tree *tQ;    /* A link to the child nodes. */
  struct tree *prev;  /* pointers to neighbors in llq*/
  struct tree *nxt;   /* for the q of potential change*/
  struct tree *after;  /* state nodes. */
  int in_statechangeq; /* flag to tell if in the q of  */
  struct tree *cur_pri;     /* potential state change.*/
  short int    active;     /* tells if active or not. */
  float  cur_rate;
  float  sum_active_rate; /* sum rate of active children*/
  short int counted_in_sum_active_rate; /*it is only */
        /* needed for nodes of child_type WFQ */
} Tree;
```

A.2 Tree_add

The routine below create and initialize a tree node structure.

```
Tree_add(isps, type, child_type,trep,w)

     int type, child_type;
     Tree *trep;
     int w;

{
  Tree *tree, *ntree, *prev;

  tree = (Tree *) Mem_alloc(isps->isps_mem, TREE,
    sizeof(Tree));
  if (tree) {
    tree->nflw = 0;
    tree->npkt = 0;
    tree->type = type;
```

```
    tree->child_type = child_type;
    tree->parent = trep;
    if(type == ROOT){tree->rate = 1;tree->sum_active_rate=0;
     tree->w=1;tree->cur_rate=1;
     tree->Fi=0;
     tree->active= 1;
   }
    if (type == WFQ) {
      tree->w = w;
      tree->rate =tree->cur_rate= (float)w/100;
      tree->sum_active_rate=0;
      tree->counted_in_sum_active_rate=0;
      tree->in_statechangeq=0;
      for(ntree=trep; (ntree->type!=ROOT);
  ntree=ntree->parent){
tree->cur_rate =(float)(ntree->rate *
tree->cur_rate);
      }
      tree->F_real = Time_now();
    }
    if (type == PRIORITY) {
      tree->pri = w;
    }
    tree->num_child = 0;
    tree->tQ = (Tree *) 0;
    tree->next = (Tree *) 0;
    tree->pQ.first = tree->pQ.last = (Pkt *) 0;
    tree->dropQ = (Tree *) 0;
    tree->next_go = (Tree *) 0;
    tree->prev = (Tree *) 0;
    tree->nxt = (Tree *) 0;
    if (type == PRIORITY)  {
      prev = trep->tQ;
      if (!prev) {trep->tQ = tree; tree->nxt =tree->nxt=
    (Tree *) 0;}
      else
{if (prev->pri > tree->pri) {trep->tQ = tree,
       tree->nxt = prev;}
else {
  for(ntree = prev;
      ntree && ntree->pri < tree->pri;
      ntree = ntree->nxt)  prev = ntree;
```

```
  prev->nxt = tree;
  prev->nxt = tree;
  trep->nxt = ntree;
  trep->nxt = ntree;
}
       }
    }
    if(type == WFQ){
      tree->next= trep->tQ;
      trep->tQ= tree;
      tree->active= BEEN_INACTIVE;
      tree->active_last_interval= FALSE;
    }
    if (child_type == PKTBUF)
      { tree->dropQ = isps->dropQ; isps->dropQ = tree;}
    if ( (type==WFQ) && ((child_type == PKTBUF) ||
 (child_type == PRIORITY)) ){
      if(!isps->isps_low_level_q){
isps->isps_low_level_q= isps->isps_low_level_q_last
  = tree;
      }else{
isps->isps_low_level_q_last= tree;
tree->low_level_next= (Tree *) 0;
      }
    }
  }
  return(tree);
}
```

A.3 Tree_enq

```
void
Tree_enque(isps,pkt,tree)
Isps *isps;
Pkt *pkt;
Tree *tree;

{
  Tree *trep,*ntree,*prev_tree;
```

```
  if(!tree->npkt){
    tree->pQ.last= tree->pQ.first = pkt;
    pkt->next = (Pkt *) 0;
  }else{
    tree->pQ.last->next= pkt;
    tree->pQ.last= pkt;
    tree->npkt++;
    tree->parent->npkt++;
    return;
  }
  if (tree->type==PRIORITY){
    trep= tree->parent;
    if(!trep->cur_pri){
      if(!trep->in_statechangeq){
trep->in_statechangeq= 1;
if(!(isps->statechangefirst)){
  isps->statechangefirst= trep;
  isps->statechangelast= trep;
  trep->after= (Tree *) 0;
}else{
  isps->statechangelast->after= trep;
  isps->statechangelast= trep;
  trep->after= (Tree *) 0;
}
      }
      trep->cur_pri= tree;
      dt= pkt->dt= max(trep->Fi, ev_now()) +
(pkt->pkt_len/(trep->cur_rate*LINK_SPEED));
      trep->Fi= dt;
      if ((trep->type==ROOT)){
isps->isps_wfq_q= trep;
tree->npkt++;
return;
      }
      prev_tree= isps->isps_wfq_q;
      if(!prev_tree){
isps->isps_wfq_q= trep;
trep->prev= (Tree *) 0;
trep->nxt= (Tree *) 0;
      }else{
if(prev_tree->Fi>dt){
  isps->isps_wfq_q= trep;
```

```
  trep->prev= (Tree *) 0;
  trep->nxt= prev_tree;
  prev_tree->prev= trep;
}else{
  for(ntree=isps->isps_wfq_q; (ntree &&
       (ntree->Fi <= dt))
      ; ntree=ntree->nxt)
    prev_tree= ntree;
  prev_tree->nxt= trep;
  trep->prev= prev_tree;
  trep->nxt= ntree;
  if(ntree){
    ntree->prev= trep;
  }
}
      }
    }else{
      if(trep->cur_pri->pri > tree->pri){
trep->Fi += ((pkt->pkt_len -
      trep->cur_pri->pQ.first->pkt_len)/
     (trep->cur_rate*LINK_SPEED));
dt=  trep->Fi;
trep->cur_pri= tree;
if(trep->prev){
  trep->prev->nxt= trep->nxt;
}else{
  isps->isps_wfq_q = trep->nxt;
}
if(trep->nxt){
  trep->nxt->prev = trep->prev;
}
prev_tree= isps->isps_wfq_q;
if(!prev_tree){
  isps->isps_wfq_q= trep;
}else{
  if(prev_tree->Fi >dt){
    isps->isps_wfq_q = trep;
    trep->prev= (Tree *) 0;
    trep->nxt= prev_tree;
    prev_tree->prev= trep;
  }else{
    for(ntree= isps->isps_wfq_q; (ntree &&
```

```
      (ntree->Fi <= dt));
      ntree= ntree->nxt)
      prev_tree= ntree;
    prev_tree->nxt= trep;
    trep->prev= prev_tree;
    trep->nxt= ntree;
    if(ntree){
      ntree->prev = trep;
    }
  }
}
      }
    }
  }else{
    if(!tree->in_statechangeq){
      tree->in_statechangeq=1;
      if(!(isps->statechangefirst)){
isps->statechangefirst= tree;
isps->statechangelast= tree;
tree->after= (Tree *) 0;
      }else{
isps->statechangelast->after= tree;
isps->statechangelast= tree;
tree->after= (Tree *) 0;
      }
    }
    dt= pkt->dt= max(tree->Fi, ev_now()) + (pkt->pkt_len/
     (tree->cur_rate*LINK_SPEED));
    tree->Fi= dt;
    prev_tree= isps->isps_wfq_q;
    if(!prev_tree){
      isps->isps_wfq_q= tree;
      tree->prev= (Tree *) 0;
      tree->nxt= (Tree *) 0;
    }else{
      if(prev_tree->Fi >dt){
isps->isps_wfq_q= tree;
tree->prev = (Tree *) 0;
tree->nxt= prev_tree;
prev_tree->prev= tree;
      }else{
for(ntree= isps->isps_wfq_q; (ntree && (ntree->Fi
```

```
<= dt));
    ntree= ntree->nxt)
  prev_tree= ntree;
prev_tree->nxt= tree;
tree->prev= prev_tree;
tree->nxt= ntree;
if(ntree){
  ntree->prev = tree;
}
      }
    }
  }
  tree->npkt++;
  tree->parent->npkt++;
  return;
}
```

A.4 Tree_deque

```
Pkt *
Tree_deque(isps)

  Isps *isps;
{

  Tree *ntree, *tree_or_trep, *next_pri, *prev_tree,
       *pri_tree=(Tree *) 0;
  Pkt *pkt, *newpkt;
  double dt;
  int k;
  Tree *node;

  tree_or_trep= isps->isps_wfq_q;
  if(!tree_or_trep){
    return((Pkt *) 0);
  }
  isps->isps_wfq_q= tree_or_trep->nxt;
  if(isps->isps_wfq_q){
    isps->isps_wfq_q->prev= (Tree *) 0;
  }
```

```
  if(tree_or_trep->child_type==PRIORITY){
    pri_tree= tree_or_trep->cur_pri;
    pkt= pri_tree->pQ.first;
    newpkt = pkt->next;
    pri_tree->pQ.first= newpkt;
    pri_tree->npkt--;
    tree_or_trep->npkt--;
    pkt->dp_time= ev_now();
    if(!pri_tree->npkt){
      for(next_pri=pri_tree->nxt; (next_pri &&
   !next_pri->npkt);
  next_pri= next_pri->nxt);
      if(next_pri){
tree_or_trep->cur_pri= next_pri;
newpkt= next_pri->pQ.first;
      }else{
tree_or_trep->cur_pri= (Tree *) 0;
if(!tree_or_trep->in_statechangeq){
  tree_or_trep->in_statechangeq=1;
  if(!(isps->statechangefirst)){
    isps->statechangefirst= tree_or_trep;
    isps->statechangelast= tree_or_trep;
    tree_or_trep->after= (Tree *) 0;
  }else{
    isps->statechangelast->after= tree_or_trep;
    isps->statechangelast= tree_or_trep;
    tree_or_trep->after= (Tree *) 0;
  }
}
return(pkt);
      }
    }
    tree_or_trep->Fi= newpkt->dt= tree_or_trep->Fi +
     (newpkt->pkt_len/(tree_or_trep->cur_rate*LINK_SPEED));
    if(!isps->isps_wfq_q){
      isps->isps_wfq_q = tree_or_trep;
      tree_or_trep->nxt= (Tree *) 0;
    }else{
      prev_tree = isps->isps_wfq_q;
      dt= tree_or_trep->Fi;
      if(prev_tree->Fi >= dt){
isps->isps_wfq_q = tree_or_trep;
```

```
tree_or_trep->nxt= prev_tree;
prev_tree->prev= tree_or_trep;
      }else{
for(ntree= prev_tree; (ntree && (ntree->Fi < dt));
    ntree=ntree->nxt)
  prev_tree= ntree;
prev_tree->nxt= tree_or_trep;
tree_or_trep->prev= prev_tree;
tree_or_trep->nxt= ntree;
if(ntree){
  ntree->prev= tree_or_trep;
}
      }
    }
  }else{
    pkt= tree_or_trep->pQ.first;
    tree_or_trep->npkt--;
    tree_or_trep->parent->npkt--;
    tree_or_trep->pQ.first= pkt->next;
    newpkt= pkt->next;
    pkt->dp_time= ev_now();
    if(newpkt){
      tree_or_trep->Fi= newpkt->dt= tree_or_trep->Fi +
(newpkt->pkt_len/(tree_or_trep->cur_rate*
  LINK_SPEED));
    }else{
      if(!tree_or_trep->in_statechangeq){
tree_or_trep->in_statechangeq=1;
if(!(isps->statechangefirst)){
  isps->statechangefirst= tree_or_trep;
  isps->statechangelast= tree_or_trep;
  tree_or_trep->after= (Tree *) 0;
}else{
  isps->statechangelast->after= tree_or_trep;
  isps->statechangelast= tree_or_trep;
  tree_or_trep->after= (Tree *) 0;
}
      }
      return(pkt);
    }
    if(!isps->isps_wfq_q){
      isps->isps_wfq_q= tree_or_trep;
```

```
      tree_or_trep->nxt= (Tree *) 0;
    }else{
      prev_tree= isps->isps_wfq_q;
      dt= tree_or_trep->Fi;
      if(prev_tree->Fi>dt){
isps->isps_wfq_q= tree_or_trep;
prev_tree->prev= tree_or_trep;
      }else{
for(ntree = isps->isps_wfq_q; (ntree &&
       (ntree->Fi <= dt));
    ntree=ntree->nxt)
  prev_tree=  ntree;
prev_tree->nxt= tree_or_trep;
tree_or_trep->prev= prev_tree;
tree_or_trep->nxt= ntree;
if(ntree){
  ntree->prev= tree_or_trep;
}
      }
    }
  }
  return(pkt);
}
```

A.5 Update_rate

```
void
Update_rate(isps,tree)
Tree *tree;
Isps *isps;

{
  Tree *ntree, *ar[20];
  int k,i=-1;
  double sum_active_rate=0;
  static int active_flag=0, inactive_flag=0;

  if(tree->type== ROOT){
    isps->numbernewinactive= -1;
    active_flag= Mark_active_nodes(isps);
```

```
    inactive_flag= Mark_inactive_nodes(isps->ar_newinactive,
isps->numbernewinactive);
    if (!( inactive_flag || active_flag)) {
      return;
    }
  }
  for(ntree= tree->tQ; (ntree && (ntree->active==
  BEEN_ACTIVE)); ntree=
      ntree->next){
    ntree->cur_rate= (double)( (ntree->rate/
tree->sum_active_rate)*
      tree->cur_rate);
  }

  if(tree->child_type==WFQ){
    for(ntree=tree->tQ; ntree; ntree= ntree->next){
      Update_rate(isps,ntree);
    }
  }
  return;
}
```

A.6 Mark_active

```
int
Mark_active_nodes(isps)
Isps *isps;

{
  Tree *tree, *ntree, *prev;
  short int returnvalue=0;

  if( isps->isps_low_level_q->type==ROOT ){return (0);}
  prev=isps->statechangefirst;
  for(tree=isps->statechangefirst;tree;tree=tree->after){
    if(tree->Fi >= ev_now()){
      if (tree->active==BEEN_INACTIVE){
tree->active= BEEN_ACTIVE;
if(!tree->counted_in_sum_active_rate){
  tree->parent->sum_active_rate += tree->rate;
```

```
  tree->counted_in_sum_active_rate=1;
}
returnvalue++;
  for(ntree= tree->parent; (ntree->active==
  BEEN_INACTIVE); ntree=
    ntree->parent){
  ntree->active= BEEN_ACTIVE;
  if(!ntree->counted_in_sum_active_rate){
    ntree->parent->sum_active_rate += ntree->rate;
    ntree->counted_in_sum_active_rate=1;
  }
}
if(tree->npkt){
  if(tree==isps->statechangefirst){
    tree->in_statechangeq=0;
    isps->statechangefirst= tree->after;
  }else{
    tree->in_statechangeq=0;
    prev->after= tree->after;
    if(isps->statechangelast==tree){
      isps->statechangelast=prev;
    }
  }
}
      }
    }else{
      isps->numbernewinactive++;
      isps->ar_newinactive[isps->numbernewinactive]=tree;
      tree->active= BEEN_INACTIVE;
      if(ntree->counted_in_sum_active_rate){
tree->parent->sum_active_rate -= tree->rate;
tree->counted_in_sum_active_rate=0;
      }
      if(tree==isps->statechangefirst){
tree->in_statechangeq=0;
isps->statechangefirst= tree->after;
      }else{
tree->in_statechangeq=0;
prev->after= tree->after;
if(isps->statechangelast==tree){
  isps->statechangelast= prev;
}
```

```
      }
    }
    prev=tree;
  }
  return(returnvalue);
}
```

A.7 Mark_inactive

```
int
Mark_inactive_nodes(ar,ar_len)
Tree **ar;
int ar_len;

{
  Tree *tree,*ntree;
  Tree *next_level_ar[20];
  int  i,next_level_ar_len= -1;

  if( (ar_len<0) || ((ar[0])->type==ROOT) ) {return (0);}
  for(i=0; i<=ar_len; i++){
    tree= ar[i];
    for(ntree=tree->parent->tQ; (ntree && (ntree->active==
   BEEN_INACTIVE));
ntree=ntree->next);
    if(!ntree){
      if((tree->parent->active == BEEN_ACTIVE)&&
 (tree->parent->type !=ROOT)){
next_level_ar_len++;
next_level_ar[next_level_ar_len]= tree->parent;
tree->parent->active = BEEN_INACTIVE;

      }
    }
  }
  if(next_level_ar_len >-1){
    Mark_inactive_nodes(next_level_ar,next_level_ar_len);
  }
  return(1);
}
```

Bibliography

[1] Clark, D. D., Shenker, S., Zhang L.: *Supporting Real-Time Application in an Integrated Packet Network:Architecture and Mechanism.* Proc ACM SIGCOMM'92, August, 1992.

[2] S. Jamin, S. Shenker, L. Zhang, D. D. Clark: *An Admission Control Algorithm for Predictive Real-Time Service (Extended Abstract)*

[3] S. Shenker, L. Zhang, D. D. Clark: *Some Observation on the Dynamics of a Congestion Control Algorithm*, Computer Communication Review Volume 20, Number 5 October1990

[4] D. D. Clark, D. L. Tennenhouse: *Architectural Considerations for a New Generation of Protocols*, presented at the ACM SIGCOMM'90 Symposium, 9/90.

[5] A. Demers, S. Kesav, S. Shenker: *Analysis and Simulation of a Fair Queueing Algorithm* in the ACM SIGCOMM 1889.

[6] H. J. Chao: *Architecture Design for Regulating and Scheduling User's Traffic in ATM Networks* presented in the ACM SIGCOMM'92

[7] A. Parekh: *A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks*, In Technical Report LIDS-TR-2089, Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, 1992.

[8] A. Parekh and R. Gallager: *A Generalized Processor Sharing Approach to Flow Control- The Single Node Case*, In Technical Report LIDS-TR-2040, Laboratory

for Information and Decision Systems, Massachusetts Institute of Technology, 1991.

[9] H. Ahmadi, W. E. Denzel: *A Survey of Modern High-Performance Switching Techniques* IEEE Journal on Selected Areas in Communications, Vol.7,No.7, 1989, pp.1091-1103.

[10] J. Nagle: *On Packet Switches with Infinite Storage,* RFC 896 1985.

[11] J. Nagle: *On Packet Switches with Infinite Storage,* IEEE Transactions on Communication, Volume 35, pp 435-438, 1987.