A Fair Bandwidth Distribution Algorithm
With A
Bandwidth Reservation Extension

By
Raed J. Sunna

B.A.Sc., Amman University, 1996

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF APPLIED SCIENCE
in
THE FACULTY OF GRADUATE STUDIES
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA
July 2000

© Raed J. Sunna, 2000
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of **ELECTRICAL ENGINEERING**

The University of British Columbia
Vancouver, Canada

Date **12/9/2000**
Abstract:

The algorithm presented in this thesis is designed to provide all flows sharing an output link of a router with a fair share of the bandwidth, while at the same time guaranteeing a high total bandwidth utilization. This Fair Bandwidth Distribution algorithm (FBDA) is designed to work with already existing responsive (e.g. TCP) and nonresponsive (e.g. UDP) transport layer protocols.

The basic idea in the algorithm is to use packet dropping to indicate to a specific flow that it is taking too much bandwidth and that it should slow down. All accepted packets are saved in one output buffer and are scheduled in a first-come first-service manner. Determining which packet should be dropped is done by assigning to each flow in the output queues of the router one per-connection variable. This variable is used to keep track of how much bandwidth the flow is consuming compared to its fair share since the last packet it lost. This same variable is also used with some upper thresholds to control nonresponsive and other aggressive flows. The average queue size is effectively controlled by providing an upper limit to the value of the average flow size (the average number of packets per flow in the queue) used in the different calculations. This is needed to have enough free buffer space for bursty traffic.

An extension to the algorithm provides for allocating certain portions of the bandwidth to specific nonresponsive flows. This is done by using the same variable mentioned earlier as a counter to represent the number of packets that the flow can pass through the gateway in a specific period of time. If this variable reaches zero, all the packets for that flow will be dropped until the end of the current period. A very simple mechanism is designed to allow all the flows with a reserved bandwidth to share the same buffer space.
with all other flows, without overflowing the buffer. This will reduce the overhead and allow for optimal bandwidth utilization.
# Table of Contents

Abstract. ii  
Table of Contents. iv  
Table of Figures. vi  

1. Introduction. 1  
   1.1. Previous Work. 1  
   1.2. The Proposed Algorithm. 8  

2. Overview. 11  
   2.1. The Basic Mechanism In The FBDA Algorithm. 11  
   2.2. Controlling The Queue Size. 12  
   2.3. Reserving Flows. 13  
   2.4. The FBDA Extension. 14  

3. The Basic Algorithm. 16  
   3.1. FBDAs’ Summary. 16  
   3.2. Updating The Per-connection Variable “extraspace”. 18  
      3.2.1. Decreasing “extraspace”. 19  
         3.2.1.1. Controlling The Average Flow Size. 21  
         3.2.1.2. Increasing The Links’ Throughput. 23  
         3.2.1.3. Preventing The Buffer From Overflowing. 25  
      3.2.2. Increasing “extraspace”. 25  
   3.3. Controlling Non-responsive Flows. 27  
   3.4. Keeping Track Of Absent Flows. 29
4. The Extension To FBDA.

5. The Cost Of FBDA.
   5.1. The Cost Of Read/Write Operations.
   5.2. The Cost Of FBDAs’ Arithmetic Operations.
      5.2.1. Cost Of The Scheduled Operations.
      5.2.2. Cost Of The Arithmetic Operations At Packet Departure.
      5.2.3. Cost Of The Arithmetic Operations At Packet Arrival.

6. Tests And Results.

7. Conclusions.

Bibliography.

Appendix A: Implementation of the FBDA algorithm in C++.

Appendix B: The execution of experiment all_A in OTCL, including the
FBDAs’ operations that needs scheduling.

Appendix C: The altered “duplex-link” function found in the “ns” simulator
library, the function creates a two-way link.
**Table of Figures**

1. Figure 1: A summary of the FBDA algorithm.  
2. Figure 2: The packet drop rate as a function of the difference between the number of packets in the queue and the average flow size, at different values of the average flow size.  
3. Figure 3: Updating the queue average, the flow average and the limited flow average.  
4. Figure 4: The packet drop rate as a function of the average flow size.  
5. Figure 5: The packet drop rate as a function of the difference between the number of packets in the queue and the limited average flow size, for different average queue size values.  
6. Figure 6: Controlling nonresponsive flows using two upper thresholds.  
7. Figure 7: Experiments tcp_A and tcp_B. Six tcp connections sharing one link.  
8. Figure 8: Experiments all_A, all_B and all_C. Two udp and three tcp connections sharing one link.  
9. Figure 9: Bandwidth distribution results of experiment tcp_A.  
10. Figure 10: Bandwidth distribution results of experiment tcp_B.  
11. Figure 11: Bandwidth distribution results of experiments all_A and all_B.  
12. Figure 12: Bandwidth distribution results of experiment all_C.  
13. Figure 13: The buffer occupancy for experiment tcp_A, max.
average queue size desired is 60 packets.

14. Figure 14: The buffer occupancy for experiment tcp_A using RED, max. threshold is 100 packets.

15. Figure 15: The buffer occupancy for experiment tcp_A, max. average queue size desired is 40 packets.

16. Figure 16: The buffer occupancy for experiment tcp_A using RED, max. threshold is 75 packets.

17. Figure 17: The buffer occupancy for experiment tcp_A, max. average queue size desired is 24 packets.

18. Figure 18: The buffer occupancy for experiment tcp_A using RED, max. threshold is 50 packets.

19. Figure 19: The buffer occupancy for experiment tcp_A, max. average queue size desired is 12 packets.

20. Figure 20: The buffer occupancy for experiment tcp_A using RED, max. threshold is 25 packets.

21. Figure 21: The buffer occupancy for experiment tcp_B, max. average queue size desired is 40 packets.

22. Figure 22: The buffer occupancy for experiment all_A, max. average queue size desired is 33.3 packets.

23. Figure 23: The buffer occupancy for experiment all_B, max. average queue size desired is 33.3 packets.

24. Figure 24: The buffer occupancy for experiment all_C, max. average queue size desired is 33.3 packets.
Chapter 1: Introduction

A large number of router architectures and scheduling disciplines have been proposed over the years, each trying to provide congestion control, fair resource distribution and good quality of service as needed, with as little complexity and overhead as possible. This thesis presents a new router design that provides the above services with a simple architecture and a small execution overhead, which makes it suitable for high-speed networks.

1.1. Previous Work:

Some scheduling mechanisms like the different Fair Queueing [5, 6] mechanisms use per-connection state variables for every active connection, to control each connection separately. These methods produce very good results, but routers using these methods usually have a high cost. For example, most of these schemes use multiple queues and complex scheduling mechanisms to reach their target of congestion (and delay) control and fair resource allocation. This results in a large overhead in computation and switching time. Also they require complex hardware, and a large buffer space for the large number of per-flow state variables, which will increase the cost of the hardware. The large buffer size and the large number of per-connection variables will also add a lot of overhead because of the time needed for reading and writing the per-flow variables (especially if the buffer and the processor were on separate chips), which makes them unsuitable for high speed networks with a large number of connections. In addition to this, using multiple queues for the different connection classes will result in low buffer utilization, especially if the buffers are to be designed large enough to accommodate very bursty traffic.
For example, one of the most well known per-connection scheduling algorithms, the Weighted Fair Queueing (WFQ) [15], is based on a hypothetical fluid-flow system called the Generalized Processor Sharing (GPS). In this system if N non-empty queues exist, the server serves all of the N queues simultaneously at a rate proportional to their “reserved” weights. The high complexity of the computations in the WFQ algorithm needed for a packet-by-packet emulation of GPS, including time stamping each arriving packet with an index indicating its virtual processing time, makes it impractical for high-speed networks.

In a somewhat simpler scheduling algorithm, Weighted Round-Robin [17], each flow has a separate queue and the scheduler, after dividing the time line into equal time frames, serves each queue for a time period equal to the weight of that flow (which represent the bandwidth reserved to that flow per time frame). To prevent the buffers from overflowing instead of serving each queue with its total bandwidth per time frame before moving on the scheduler can serve one packet at a time from each queue before moving to the next one, while skipping the queues that have already used their bandwidth for the current time frame. Even with this adjustment, WRR still has poor delay performance when it comes to handling bursty flows, in addition to the overhead disadvantages mentioned earlier. In [5], the authors propose an alteration to WRR where at the beginning of each time frame the scheduler arranges the service of the queues so that it starts with the queue with the largest “queue size / weight”, so that to reduce the probability of overflowing the buffer with bursty flows and/or small service time (weight). The additional overhead in this approach is to keep an updated and sorted table, containing the value of “queue size / weight” for each active flow.

Other similar fair queuing scheduling algorithms include Delay Earliest-Due-Date,
Another approach in handling congestions in networks is the hop-by-hop congestion control schemes [7, 9, 10, 11]. The basic idea in these schemes is that congestion control is performed at each router along the path of the flow, instead of using an end-to-end congestion control mechanism (e.g. TCP based mechanisms). Each router will send messages regularly to the routers upstream (towards the source) that indicate the presence, and the degree of congestion. It is assumed that each router has the capability to separately control the transmission rate of each flow going through it, to do this some per-connection scheduling mechanism must be used (e.g. Weighted Round-Robin). In rate-based hop-by-hop flow control, since there is a propagation delay between each two routers the information arriving at a certain router from a router downstream does not represent the current condition, but it represent the condition before a time equal to the propagation delay, and the router receiving it must somehow use it to predict the current condition and react to it. In credit-based hop-by-hop flow control mechanisms suggested for ATM networks, the RTT for a flow must be known to the router before it can assign a buffer space to that flow to give it a certain bandwidth.

The main benefit of a hop-by-hop scheme is faster response to traffic congestions, especially in networks with large propagation-delay bandwidth product, due to the short distance between each two routers compared to the distance between the source and the destination, which will reduce the packet drop rate during overloads and increase the links throughput.

But there are a number of disadvantages that make the use of a hop-by-hop approach undesirable for very large networks, especially the Internet. The most obvious ones are the
overhead in sending and receiving the messages, the possibility of losing some of the mes-
sages, and the bandwidth wasted to send the messages, especially in a network where each
router handles a large number of flows from different sources at the same time. Another
problem is that in a network such as the Internet the propagation delay between the gate-
ways is large enough to make any speed up in the reaction of the network to congestion
due to the use of a rate-based hop-by-hop mechanism too small to justify the overhead of
such mechanism. Also the large propagation delays make it hard to predict the current
condition of the routers sending the messages with sufficient accuracy (because the mes-
sages will be too old). For credit-based flow control mechanisms, the waste in the buffer
space is very large unless a complicated mechanism is added to dynamically change the
buffer space assigned to each flow according to the current total buffer occupancy. Finally,
a hop-by-hop scheme requires that the routers involved must use the same protocol and
must synchronize their work, something that can not be guaranteed in the Internet environ-
ment.

Other simpler mechanisms, such as Random Drop, Early Random Drop, DECbit and
Random Early Detection (RED) [1], use one output queue per link and the simple schedul-
ing scheme of first-come-first-service (FCFS) for congestion avoidance, without the use of
per-flow control. Those schemes have a main goal of increasing the output throughput and
reducing the average queuing delay, while being as fair as possible. DECbit uses explicit
feedback to the traffic source to indicate congestion, which will require a specially
designed transport layer protocol at the sources to cooperate with it. The others use packet
dropping as the main tool to control the congestion, and should work with any general
TCP-like protocol.
Random Drop works by randomly choosing a packet to drop from the queue if the buffer is full. Early Random Drop works by dropping packets with a certain probability if the queue size exceeds a certain level. Those methods have a poor performance regarding providing fairness and controlling misbehaving flows.

RED on the other hand drops each arriving packet (or marks it using the ECN bit) with a certain probability proportional to the average queue size, which will result in all the flows losing the same percentage of their packets arriving at the router. RED also drops (or marks) all the incoming packets if the average queue size exceeds an upper threshold (the sender should use the loss of a packet as a sign of congestion and reduce its sending rate), while accepting all the packets when the average is less than a certain lower threshold. This will allow bursts of packets to pass through the router, while controlling persistent congestions.

One of the disadvantages of RED is that by using the average queue size to determine the probability of dropping any packet, one flow might be “luckier” than another similar flow and lose less packets, and thus take more bandwidth. Also RED is still biased against bursty flows and flows with larger propagation delays, because the average queue size (and the packet drop rate) will increase with higher probability during the time when those flows have some packets in the queue, this is most noticeable if the average value is crossing one of the thresholds repeatedly. A recent modification to RED [21] proposes that instead of dropping all the packets when the average queue size passes the upper threshold, the packet drop probability should increase from the maximum probability (right before the average queue size crosses the upper threshold) to one, as the average queue size increases from the upper threshold to twice the value of the upper threshold. This will
reduce the probability of global synchronization and will help increase the link throughput. This modification is not provided in the simulator used in this thesis, but it is provided in the newer version of the simulator. The effect of this modification on the experiments done in this thesis is insignificant, since in all the experiments (except when the upper threshold is set to 25 packets, where the results might be somewhat better if this modification is used) the queue size never goes above the upper threshold (see the graphs in chapter six).

Even if all flows have the same drop rate, this will not provide a fair distribution of the output bandwidth; this is because the TCP flows with large propagation delay will be affected more by the loss of a single packet than the aggressive flows with smaller delays that cause the congestion. Even if all the flows are sending packets with the same rate, the flows with large window size (and propagation delay) will be affected more by losing a single packet, because it will take them more time to increase the window size back to the original size after reducing it (e.g. by half, as in TCP Reno) as a response to the packet loss. For example, consider two flows A and B sharing an output link. Assume that the two flows are using a TCP transport layer protocol that will reduce its window size by half every time it losses a packet, and will increase its window size by one packet every round trip time until it reaches the window limit if no packets are being dropped. Also assume that flow B has ten times the round trip propagation delay of flow A (ignoring the queueing delay) and ten times the window limit of flow A. Then when there is no congestion (i.e. the window limit of each flow is equal to or less than the propagation delay of that flow multiplied by one half the bandwidth of the shared link), both flows will have a window size equal to their limit, and both will have the same packet arrival rate at the router. If
a congestion happens, due to the arrival of new flows, so that each flow lose just one packet, both flows A and B will reduce their windows by half, and since flow B has ten times the window limit and ten times the round trip delay of flow A, then the time period that flow B will need to increase its window size to the limit again will be one hundred times larger than the time needed by flow A, which means flow A will take more bandwidth during the recovery time. This problem also exists in the Fair Random Early Detection algorithm discussed below.

RED does not have a way to check if dropping the same percentage of packets from the different adaptive flows will result in fair bandwidth distribution. Also RED does not provide a capability to handle non-responsive flows, or time sensitive flows.

In [12, 13] more variations to RED are suggested to provide better link throughput and prevent buffer overflow as the number of TCP flows sharing the link changes (since having a large number of flows requires a higher packet drop rate to reduce the congestion than when having a smaller number of flows, even if the average queue size is the same in both cases). Both methods provide better results than the basic RED, but with extra overhead. [13] tries to provide fair bandwidth distribution between TCP flows with different RTTs with limited success, but the execution overhead and the number of the state variables needed are very large.

Fair Random Early Detection algorithm (FRED) [2, 4] is a variant of RED where also one FCFS output queue is used, but the algorithm keeps per-flow state variables to determine which packets to drop. It calculates the average size of each flow in the queue and drops the packets from that flow with a probability proportional to that value. This algorithm is more fair than RED, but it is complicated and needs a large number of per-connec-
tion state variables. This algorithm also controls non-responsive flows, but with the use of extra per-connection state variables.

One disadvantage in this algorithm is that although each flow has its own average calculations that controls the drop rate of its packets, a drop of any packet will result in one connection reducing its transmission rate, and so the entire queue size will decrease and thus all the averages of all the other connections will also decrease, which is not fair. Also this algorithm does not take the time period in which a certain flow is absent from the queue in consideration; since the lowest average possible is zero regardless of the time the flow spend away (which will hurt fragile flows). FRED also does not provide a capability to handle time sensitive flows.

1.2. The Proposed Algorithm:

Our new fair Bandwidth Distribution Algorithm (FBDA) is designed to be implemented in gateways used in large general-purpose networks like the Internet. It is designed to not only control the average output queue size and prevent the buffer from overflowing, but also to distribute the output link bandwidth fairly between the flows sharing the link and to allow the flows that can not use their fair share, due to the lack of data to send or otherwise, to take all the bandwidth they need. FBDA does this by punishing aggressive flows, but not fragile or bursty flows. FBDA uses one output queue per output link and uses packet dropping to control the size of that queue. It also uses the packet dropping to notify a source that it needs to reduce the rate at which it is sending the packets. Packets accepted in an output queue are scheduled first come first service.

FDBA also does a good job in controlling non-responsive flows and in preventing
them from taking more bandwidth than they deserve with no extra overhead. All this is done with a small number of flow-specific state variables. The FBDA algorithm also provides quality of service when needed by reserving portions of the bandwidth to certain flows; this is done without the need for any extra queues or scheduling mechanisms.

Three goals were in mind when the FBDA algorithm was designed. The first goal, as mentioned before, was to provide fair bandwidth distribution between the different connections sharing the output link, a result where each connection gets +/- 10% of the fair share was assumed to be very accurate and was considered as the desired goal. The simulation results provided in chapter 6, for an environment where the adaptive connections are using TCP Newreno transport layer protocol, shows that if the parameter setup was chosen correctly and if enough buffer space was provided, in reasonably high congestion conditions this goal can be met.

The second goal in designing the algorithm was that it must have a very small execution overhead, so that it can be implemented in high-speed networks. The execution overhead of the RED algorithm was considered to be the minimal value possible, and an effort was made to get the execution overhead of FBDA to be as close to this value as possible. The analysis of the arithmetic operations execution cost for FBDA in chapter 5 shows that this cost is a little more than twice the cost of RED.

The third goal in designing the algorithm was to have as few per-connection state variables as possible, so that the router can handle a large number of connections at the same time, while using a reasonable size memory space to save the related variables. This is important for two reasons, one is to reduce the hardware cost, and the other is to reduce the overhead cost and the number of memory read/write operations needed for each packet
arrival. A small memory is especially needed if one desires to have the memory and the processor on one chip, to reduce the cost. Since one state variable cannot possibly be sufficient in representing the behavior of a certain flow, the goal was to design the algorithm using two state variables per connection. The result was that each flow has two variables associated with it, in addition to its identifier, these variables are the number of packets the flow has in the queue “packets” and a value called “extraspace” that is used to determine whether a packet should be accepted or dropped.

Chapter two of this thesis presents an overview of FBDA and the associated extension. Chapters three and four describe the FBDA algorithm and its extension in more details. Chapter five discusses the different costs of implementing FBDA, while chapter six presents simulation results using the ns network simulator, and when applicable the results are compared with those from using the RED algorithm [1]. Finally, chapter seven concludes.
Chapter 2: Overview

The basic idea in FBDA is that the distribution of the packets between the different flows in the output queue at any time represents the distribution of the output link bandwidth between the flows for the time period that is needed to process the packets in the queue. To distribute the bandwidth equally between the flows, the average queue size must be equally divided between the different flows over a certain time period.

2.1. The Basic Mechanism in the FBDA Algorithm:

FBDA keeps track of the behavior of each flow using a per-connection variable called "extraspace". By increasing and decreasing "extraspace" FBDA memorizes if, and how much, a flow is taking more or less than the fair share of the bandwidth. Using this information FBDA increases or decreases the packet drop rate of each flow separately, thus ending TCPs bias against bursty connections or connections with large propagation delay, so that in the long run all the flows will take their fair share.

This is done by initially setting "extraspace" to a value called "rate" at the arrival of the first packet from a flow. "rate" will determine the rate at which packets will be dropped, given all the other variables being constant.

At every packet arrival, the value of "extraspace" associated with the flow that owns the packet will either be decreased, if the flow is taking more than its fair share of the bandwidth, by a value proportional to the difference between the bandwidth taken by the flow and the fair share, or increased in the same way if the flow is taking less than the fair share. If "extraspace" reaches zero, the next packet from that flow is dropped. The value of "rate" should be large enough to allow for the acceptance of short lived bursts of packets.
without dropping any packets, while small enough to control the average queue size and to punish consistently aggressive flows.

By using this simple credit-based mechanism, in contrast to using probabilities in dropping packets, all equivalent flows under the same circumstances will have the exact same drop rate. Also the packets dropped from each flow are equally distributed over time, assuming all other variables are constant, this will prevent the loss of a number of successive packets in a short period of time which could hurt some adaptive flows (e.g. by switching to a time-out mode).

Also by using this mechanism, a record is kept of the behavior of each flow regardless of time passed. This is in contrast to using an “average size” of a flow calculated by giving the current flow size a certain weight (i.e. time-based exponential decay, where the average is updated using the equation “\( \text{Av} = \text{Av} \times (1 - w) + q \times w \)”, where “\( \text{Av} \)” is the average at that time, “\( q \)” is the instantaneous value, and “\( w \)” is the weight that determines how fast the average will change) to determine when to drop a packet, where more emphasis is given to the most recent behavior.

### 2.2. Controlling the Queue Size:

To control the average queue size (especially needed when there are nonresponsive flows in the queue) a parameter called “per” is used. This parameter represent the percentage of the buffer that the average queue size should not exceed. For example, if “per” is set to 0.25 that means the average queue size should not exceed one quarter of the available buffer. This parameter is used to put a limit on the average flow size used in calculating the value by which “extraspace” is updated, and that will control the packet drop rate which in
turn will control the real average flow size. How close the real average is to the limit specified is determined by the behavior of the transport layer protocol of the source when a packet is lost. FBDA works best if “per” was high enough to allow a substantial average queue size (e.g. more than five times the expected number of connections), while small enough to leave a large portion (e.g. one third) of the buffer empty (the reason will be given later).

By reducing the value by which “extraspace” is decreased as the average queue size decreases, the packet drop rate for all of the flows that are taking more bandwidth than the fair share is also decreased. This is done to have a high link throughput, even though this will hurt the fairness of the algorithm a little.

Other methods used to control heavy congestions are geometrically increasing the packet drop rate, also by adjusting the value by which “extraspace” is decreased, as the average queue size increases. This is done to prevent the buffer from overflowing. For a more immediate response, and for controlling non-responsive flows, upper thresholds are used to determine the maximum number of packets a flow can have in the queue at any point in time. One reason why “per” should not be set too high is to reduce the probability of many flows passing an upper threshold, which would result in a more bursty traffic, and a less cooperative queue with larger oscillations.

### 2.3. Reserving Flows:

The number of packets a certain flow has in the queue can be zero because FBDA does not delete a flow from its flow set whenever the number of packets for that flow drops to zero. Instead FBDA “reserves” that flow for a certain period of time (if the extension to
FBDA is implemented, this period should at least be equal to 1 second; the reason will be discussed later). If no new packets from that flow arrive during this time period, the flow is deleted with its state variable (i.e. “extraspace”). If a packet arrives during that period the flow will be treated as if it is a continuously existing flow. The advantage of doing this is not resetting the value of “extraspace” associated with that flow every time the flow leaves the queue for a very short time, which would otherwise hurt the effectiveness of the algorithm. Also for a flow that leaves the queue for a significant periods of time, when new packets from the flow arrives to the gateway its “extraspace” value is increased to reflect the time the flow was absent. This will result in giving that flow a fairer share of the bandwidth.

The effect of “reserving” flows is most noticeable in the situations where the average queue size and/or the average queueing delay is small, due to a small congestion or due to the setup of the parameter “per” that controls the queue size. Also it is especially helpful to fragile flows that have on average a small number of packets in the queue, and to flows that use protocols that are very sensitive to packet loss as some TCP protocols that drastically reduce their window size due to the loss of one packet. The overhead of implementing this flow reservation is determined by the hardware available and by the accuracy desired in measuring the time, as discussed in chapter three. Still this reservation method is optional, except for the flows that take advantage of the extension to FBDA where it is absolutely needed.

2.4. The FBDA Extension:

The router may decide to give a certain flow more bandwidth than its fair share due to
the time sensitivity of the flow. So an extension to FBDA has been developed for this purpose. The extension provides for allocating a certain portions of the bandwidth to specific nonresponsive flows, such as constant bit rate applications using a UDP protocol, without the need for additional queues or new scheduling mechanisms.

This is done by using the "extraspace" variable as a counter to represent the number of packets that the flow can pass through the gateway in a specific period of time. If "extraspace" reaches zero, all the packets from that flow will be dropped until the end of the current time period. This time period can be decreased to prevent aggressive flows from overflowing the buffer. Three additional per-flow state variables are required, the bandwidth to be allocated to the flow in Mb per second, the time interval between increasing the value of "extraspace" and a timer to measure the time interval (the last two can be shared by all the flows with a reserved bandwidth that are controlled using the same time interval).
Chapter 3: The Basic Algorithm

3.1. A Summary of the FBDA Algorithm:

At each packet arrival, FBDA determines whether to accept the packet or drop it. A packet will always be accepted if the number of packets associated with that flow in the queue (i.e. the value of the per-connection state variable “packets”) is less than the average flow size. If “packets” is also less than the limited average flow size (the limited average equals a certain limit if the real average is larger than this limit, otherwise the limited average and the real average are the same), “extraspace” is increased by a value proportional to the difference between the number of packets the flow has at that moment and the limited average flow size. On the other hand, if “packets” is less than the average flow size, but larger than the limited average (may happen if the real average is larger than the limit), “extraspace” is decreased by a value proportional to the difference between “packets” and the limited average flow size if the result is larger than zero, otherwise “extraspace” is not changed.

If “packets” is larger than the real average flow size (and since the limited average is never larger than the real average, “packets” will also be larger than the limited average flow size), “extraspace” should also be decreased as mentioned above. If “extraspace” is larger than or equal to the value to be subtracted the packet is accepted and “extraspace” is decreased by the appropriate value. On the other hand if the value of “extraspace” is less than the value by which it should be decreased, the packet is dropped. Then if “extraspace” is not negative, and “packets” is not larger than one of the previously determined upper thresholds, “extraspace” is increased by the value of “rate” mentioned earlier.
If the number of packets is larger than any of the upper thresholds, or if "extraspace" is negative, "extraspace" is set to a small negative number. Setting "extraspace" to a negative number means that no packets from that flow will be accepted until "packets" goes below the limited average flow size, where "extraspace" will be increased to a positive value. This mechanism will control the nonresponsive (and other aggressive) flows without the need for extra per-flow state variables. Figure (1) summarize the basic FBDA activities mentioned above:

Figure 1: A summary of the FBDA algorithm
A packet will also be accepted if a minimum threshold was specified and the number of packets is less than this threshold. This minimum threshold will increase the total link throughput when the congestion is small, but at the same time it will hurt the fairness of the algorithm since all packets will be accepted if the minimum threshold is not reached even though the bandwidth is not fairly distributed.

To reduce the negative effect of the minimum threshold it should be very small, and it is better to use exact number of packets than to use an average, in other words it is better to say accept the packet if that flow has less than four packets in the queue (i.e. “packets” < 4) than to say accept the packets if the average flow size is less than two (i.e. “av” < 2). This is because of the fact that some flows may have very small number of packets, even zero, which means a single flow can have a large number of packets in the queue without increasing the average flow size above this threshold.

3.2. Updating the Per-connection Variable “extraspace”:

Two factors where taken in consideration when the function that defines the value by which “extraspace” should be decreased was designed. One factor is that this function must provide a separate packet drop rate for each flow depending on its bandwidth occupancy, so that the output bandwidth can be equally distributed among the different flows. The other factor is that this function must ensure that the average queue size is within the desired range, so that to prevent the buffer from overflowing and to keep the output link throughput high. Those two factors work against each other, for example if the difference in the bandwidth allocation between the different flows was large and the congestion was small, then forcing the aggressive flows to drastically decrease their transmission rate so
that they only get the average may result in low link utilization if the other flows can not fill the newly available bandwidth fast enough. On the other hand if the packet drop rate in general was drastically decreased, to keep the average queue size above a certain level to get high link utilization, the algorithm may not be able to control aggressive flows and the bandwidth distribution might be very unfair. So a balance had to be found that would allow the algorithm to provide good bandwidth distribution while providing a very good link utilization. The function defining the value by which “extraspace” is increased is derived from the function that defines the value by which “extraspace” is decreased, taking the packet arrival rate for each flow in consideration.

3.2.1. Decreasing “extraspace”:

The value by which “extraspace” will be decreased if a packet is accepted while the number of packets for that flow is larger than the limited average flow size has to be proportional to how much of the bandwidth that flow is taking more than its fair share. This is determined by how much “packets” is larger than the limited average flow size, the function used in the algorithm is:

\[
\frac{\text{packets} - \text{max\_av}}{\text{qlim} - \text{queue\_av}} \quad \text{eq}(1)
\]

where “packets” is the number of packets the flow has at that moment in the queue, and “max\_av” is the average flow size with a limit, that is “max\_av” cannot exceed a certain value even if the real average does. “qlim\_” is the buffer size, and “queue\_av” is the average queue size. This function is a linear equation with a slope that increases geometrically
as the average free space of the buffer decreases. Over time the average number of “packets” will determine the average value that is subtracted from “extraspace” at the acceptance of each packet when the flow is using more than its fair share of bandwidth, and this average will determine the packet drop rate. For a flow with a “packets” value that is always above “max_av” (and below the upper thresholds), the packet drop rate depends on the value of “rate”. For example, if “rate” was set to one and for a certain flow the result of equation (1) was 0.1, 0.2, 0.3, 0.4 for the four packets arriving after a packet is dropped, then the fifth packet will be dropped. And the drop rate will be one in five (or 20%). Note that the average result of equation (1) is “rate”/4, which is 1/4. For the same flow if “rate” was set to three instead than one, the result of equation (1) may be 0.1, 0.2, 0.3, 0.4, 0.1..... (for the next twenty packets). This means the drop rate will be one in twenty four (or about 4%). The average of equation (1) is “rate”/24 or 3/24. Note that when “rate” was increased, the results of equation (1) were averaged over a larger number of packets. This is needed so that the packet drop rate does not increase drastically because of the arrival of a burst of packets (which will hurt bursty traffic). But “rate” should not be set too large, so that the packet drop rate would not be too small for FBDA to be affective.

Equation (1) is derived from the basic equation (2) shown below, this equation defines the value to be subtracted to be equal to how much larger than the average flow size the value of “packets” is:

\[
\frac{(packets - av)}{av} \quad \text{eq(2)}
\]

where “av” is the average flow size.
Figure (2) below shows the packet drop rate increase as the value of “packets” increases if equation (2) was used in the algorithm, at different average flow size values for a flow always taking more bandwidth than the average. The slope changes because the effect of an increase of “packets” by a certain value is less important when the average flow size is large than when it is small. The thick line shows the drop rate when the flow is taking twice the fair share of the bandwidth, notice that it is constant.

The next three sub-sections describe the changes done to equation (2) that produced equation (1).

Figure 2: The packet drop rate as a function of the difference between the number of packets in the queue and the average flow size, at different values of the average flow size.

3.2.1.1. Controlling the Average Flow Size:

Non-responsive flows and aggressive responsive flows may keep increasing the average flow size even as they lose packets, this will increase the average queue size and will result in a decrease in the value of equation (2), which in turn will reduce the packet drop
rate for these flows and allow them to increase the average flow size even further. To break this cycle of increasing the average flow size and decreasing the drop rate a limit must be imposed on the average value used in equation (2), which will specify the desired maximum average flow size, this will put a limit on how low the drop rate will go when the real average increases; this in turn will result in controlling the real average and keep it close to the limit. “av” is substituted with “max_av”, where “max_av” is the average flow size with a limit. So equation (2) is changed to:

\[
\frac{\text{packets} - \text{max}_\text{av}}{\text{max}_\text{av}} \quad \text{eq}(3)
\]

The following C code shows the calculation of “av”, “max_av” and “queue_av”:

```c
queue_av = (1 - w) * queue_av + w * Pack_num_ ;
av = (1 - w) * av + w * Pack_num_ / flows_num_ ;
if ( av < max / flows_num_ )
     { max_av = av ; }
else
     { max_av = max / flows_num_ ; }
```

Figure 3: Updating the queue average, the flow average and the limited flow average.

This part of the algorithm is executed each time a packet leaves the queue (to reduce the overhead, it could be executed once for every number of packets leaving the queue).
Also the part that updates “max_av” is executed when “av” is updated after an idle period (where the buffer is empty). “Pack_num_” is the size of the queue at that time, “flows_num_” is the number of flows sharing the queue at that time, and “w” is the weight of the queue size or the flow size. As can be seen the flow size is calculated by dividing the queue size over the number of flows. “max” is the limit we want to impose on the average queue size. In other words “max” is the maximum average queue size that we desire for the queue. The real average queue size may be larger than “max” if the congestion is large enough, still “max” helps in controlling the queue size.

“max” is calculated at the setup phase of the algorithm by multiplying the buffer size by “per”. “per” represents the percentage of the buffer that the average queue size should not exceed. For example, if “per” is set to 0.33 that means “max” has a value equal to one third of the buffer size. From the code above one can see that “max_av” has an upper limit equal to “max” divided by the number of flows sharing the queue at that time.

FBDA use the same procedure described in [1] to adjust the average queue size “queue_av” and the average flow size “av” after a period of time where the queue is empty.

3.2.1.2. Increasing the Throughput of the Link:

Equation (3) represents the value to be subtracted from “extraspace” regardless of the queue size. To prevent the average flow size from going above “max_av” and to increase the throughput of the output link when the congestion is small this equation must be altered to decrease the subtracted value (and the drop rate) when the average queue size decreases, and increase that value as the average flow size gets closer to “max_av”. To do
this equation (3) is multiplied by “max_av” so that the subtracted value increases with the average. The result is equation (4) next:

\[(\text{packets} - \text{max_av})\quad \text{eq}(4)\]

Now the value subtracted will depend on the average flow size and not only on how much a flow is taking more than the average. For example, if a flow is consistently taking twice the average bandwidth allocated for each flow and the average flow size is less than the limit (i.e. equation (3) is always equal to one), then the value dropped will not be constant but will be equal to the value of the average, as shown in figure 4.

Note that as the average flow size decreases the difference in the packet drop rate between two flows with different bandwidth occupancy will also decrease (even though the ratio is fixed), and this will reduce the ability of the FBDA algorithm to control aggressive flows and provide fair bandwidth distribution, but this decrease in drop rate is necessary to provide high link utilization.

\[(\text{packets} - \text{av})\]

For \(\text{av} < \text{max} / \text{flows\_num}\_\)

\[
2y \quad \text{packets} = 2 \times \text{av}, \quad (\text{packets} - \text{av}) / \text{av} = 1
\]

(this is the thick constant line in figure 3)

\[y \quad \text{packets} = 1.5 \times \text{av}, \quad (\text{packets} - \text{av}) / \text{av} = 0.5\]

\[2x \quad \text{packets} = \text{av}, \quad (\text{packets} - \text{av}) / \text{av} = 0.5\]

\[x \quad \text{packets} = \text{av}, \quad (\text{packets} - \text{av}) / \text{av} = 0.5\]

Figure 4: The packet drop rate as a function of the average flow size. The drop rate will decrease with the average flow size to increase the throughput of the link.
3.2.1.3. Preventing the Buffer From Overflowing:

In the extreme cases the average might keep growing even with the previously mentioned constraints. In that case, to prevent the buffer from overflowing consistently one additional change is done to equation (4). Equation (4) is divided by the average size of the empty space of the buffer to produce equation (1). The effect of this change is only noticeable when the average queue size gets close to the buffer size. As shown in Figure (5)

\[
\frac{\text{packets} - \text{max}_\text{av}}{\text{qlim}_\text{-} - \text{queue}_\text{av}}
\]

Figure 5: The packet drop rate as a function of the difference between the number of packets in the queue and the limited average flow size, for different average queue size values.

3.2.2. Increasing “extraspace”:

When the number of packets a flow has is less than “max_av” (the limit is used for the
same reason as before), it means the flow at that time is getting less than its fair share of the bandwidth. In this case its “extraspace” value should be increased so that at a later time that flow can take more than the average flow size without losing any packets.

To determine the value that should be added we assume that the average queue size and the number of flows in the queue are constant. When a flow has on average for a period of time “T” a number of packets less than “max_av” by “X” packets, then we want to add a certain value to “extraspace” so that the flow can spend a period of time equal to “T” with an average number of packets equal to “max_av” plus “X”. This value is shown in equation (5) below, which is the function used in FBDA:

$$\frac{(2*\text{max}_av - \text{packets} + 1) * (\text{max}_av - \text{packets})}{(\text{packets} + 1) * (\text{qlim}_- \text{queue}_av)} \text{ eq}(5)$$

This equation is based on the idea that if the number of packets is less than “max_av” by “p” packets, the value to be added to “extraspace” should be equal to the value that should be subtracted if “packets” was “p” packets larger than “max_av”. Or equal to:

$$\frac{(\text{max}_av - \text{packets})}{(\text{qlim}_- \text{queue}_av)} \text{ eq}(6)$$

But since the value added should reflect the time period the flow spends below the average the arrival rate should be taken in consideration.

The arrival rate of a flow is assumed to be equal to the number of packets that flow has in the queue at that time for each time period needed to process all the packets in the queue.
(or the queueing delay). So at the arrival of a packet, if the flow has at that point “P” packets in the queue, then the arrival rate is assumed to be “P + 1” packets per a time period equal to the queueing delay. This assumption is most accurate when the average queue size is constant.

Assuming the average queue size is constant, the arrival rate when the number of packets is below the average is

\[ \text{packets} + 1 \quad \text{eq}(7) \]

per time period (dropped packets are not included). The arrival rate when the number of packets is larger than the average by the same difference is

\[ 2 \times \text{max_av} - \text{packets} + 1 \quad \text{eq}(8) \]

per time period. Equation (6) is multiplied by equation (8) to get the total value to be subtracted over a period of time equal to the queueing delay. Then the result is divided by equation (7) to get equation (5), which is the value that should be added each time a packet arrives with “packets” less than “max_av”, so that at the end of the same time period the value added equals the value to be subtracted.

If the arriving packet is the first for a flow equation (5) will be:

\[ \frac{(2 \times \text{max_av} + 1) \times \text{max_av}}{\text{qlim} - \text{queue}_\text{av}} \quad \text{eq}(9) \]

### 3.3. Controlling Non-responsive Flows:

As mentioned before if a packet arrives from a flow that has more packets than “max_av” and “extraspace” for that flow is less than the value needed to be subtracted, the packet is dropped. After that “extraspace” is increased by the value “rate” which will
determine how soon another packet is dropped from that flow. To control nonresponsive flows upper thresholds are set so that if the flow exceeds any of them “extraspace” will not be increased by “rate”, instead it will be set to a negative number (e.g. -0.001). This negative number will prevent “extraspace” from increasing no matter how many packets the flow loses until the number of packets for that flow goes below “max_av”. The following C code shows how this is done:

```c
if (packets < 4*max/flows_num_ && packets < qlim_/flows_num_ &&
    extraspace >= 0 )
    {extraspace = extraspace + rate ;}
else
    {extraspace = - 0.001 ;}
```

Figure 6: Controlling nonresponsive flows using two upper thresholds.

This part of the algorithm is executed each time it is decided that a packet will not be accepted and will be dropped. Note that if “extaspace” was large enough no packets will be dropped when the flow crosses the thresholds, this will help bursty flows.

Two upper thresholds are used, one is four times the maximum desired flow size, and the other is the buffer size divided by the number of flows in the queue. The second threshold is chosen to reduce the possibility of the queue overflowing. While the first is used to catch nonresponsive flows when the queue size is very small due to the parameter setting, where the second threshold is not efficient. If “per” is set to less than 0.25, then the first threshold is active, else it is the second. These thresholds should be increased if their
expected values are too small compared with the expected packet bursts size. Note that when a flow gets a negative “extraspace” value, the only way a packet from it will be accepted is if the number of packets for that flow becomes less than the flow average. This will prevent the nonresponsive flows from taking more space than they should.

If the flow average “av” was equal to “max_av” (i.e. the flow average is less than max/flows_num_), then when a packet from a nonresponsive flow that is limited by the average flow size “av” leaves the queue, and until a new packet arrives, the number of packets for that flow will go below “max_av”. This means that “extraspace” will be increased, giving the flow a chance to get more packets accepted until it reaches an upper threshold again. To prevent that, the addition to “extraspace” is altered so that it happens not when “packets” go below “max_av”, but when “packets” go below “max_av” - 1.

3.4. Keeping Track of Absent Flows:

If all the packets of a flow leave the queue, the flow is not immediately deleted. Instead it is held for a period of time in which it is treated as a flow with zero packets. If during this time period no new packets arrive at the gateway from this flow, the flow is deleted (with its state variables), and the number of flows “flows_num_” is updated. To time the period in which a flow is held the state variable “packets” is used, since during the holding period there are no packets for that flow in the queue and the “packets” variable is free to be used in other ways. When the last packet of a flow leaves the queue, “packets” is set to the negative of an integer parameter called “reserve”. FBDA periodically checks the “packets” variable of each flow and increment it by one if it is a negative value. This can be implemented by adding the most significant bit in the “packets” field to the value of
“packets” (the most significant bit represents the sign of “packets”, where 1 stands for negative and 0 stands for positive values), so that a negative number will be increased by one, while a positive “packets” will not be changed. When the value of “packets” reaches zero the flow is deleted.

The time period between each round of incrementing the “packets” variables is called “hold_int”. The product of multiplying “hold_int” by “reserve-1”, should be equal to the desired time period a flow should be held. For example, if the flows should be held for at least one second, “hold_int” could be set to 0.05 seconds and “reserve” set to 21. “reserve” is 21 and not 20 because the time period between a flow leaving the queue and the first time “packets” is increased is less than “hold_int”, and what is needed is at least one second hold time.

Incrementing “packets” for all the flows can be done in parallel, but if not enough adders are available, then the additions will have to be done over a number of cycles equal to the maximum number of flows allowed divided by the number of adders available. The size of each adder needed for this job is related to the size of the “packets” field, which is related to the size of the buffer. For example, if the buffer in the gateway can hold at most 127 packets, then since a flow cannot have more packets than the buffer size “packets” need not be larger than 8 bits including a bit for the sign. So the adders needed should be 8 bit in size, to perform a number of 8-bit add operations every “hold_int” time period. But if the calculations were done using the number of bytes instead of the number of packets, a larger “packets” field (or more correctly “bytes” field) and adders will be needed. Another option is to count variable size packets using units of 10 or 100 bytes (with approximation) to reduce the space needed.
When a flow leaves the queue for a period of time and then new packets from that flow arrive at the gateway, a certain value should be added to "extraspace" to reflect the time in which the flow did not take any portion of the output bandwidth.

When there are packets arriving from a flow and "packets" for that flow is less than "max_av" then a certain value will be added to reflect the fact that the flow is not taking its fair share. Any drop in the arrival rate will be reflected by a drop in the number of packets in the queue relative to the average flow size, and so the value added to "extraspace" will increase. But if the number of "packets" drops to zero, any delay in the arrival of a packet after that will not be reflected by a decrease in the value of "packets" since the number of packets can not go below zero. So another equation should be used to reflect the time a flow spends without any packets in the queue. This is done using equation (10) below. It assumes that during the time a flow was on hold a maximum number of packets have arrived from that flow, each to find that it is the only packet from that flow in the queue at that moment (this resembles the way the average queue size is adjusted after a period in which the queue was empty).

\[
\text{pack}_\text{per}_\text{hold}_\text{int} \times (\text{reserve} + \text{packets}) / \text{queue}_\text{av} \text{ eq(9)}
\]

\[
\text{eq(10)}
\]

To calculate the number of packets mentioned above, first the output link bandwidth is calculated in packets per "hold_int", this is done at the setup time and is represented by the value "pack_per_hold_int". This value is divided by the average queue size "queue_av" to get how many "queues" the link process per "hold_int", assuming that
“queue_av” is constant throughout the period in which the flow was absent. The result is multiplied by the time period in which the flow was absent, this time period is measured in “hold_int” units, and can be calculated by adding the value of “reserve” to the value of “packets” at the time when a new packet arrives. The result represents how many times the gateway can process the entire queue in the time the flow was absent (again, assuming the average queue size is constant).

Since for each arriving packet it should be the only packet in queue, the time interval between packet arrivals should be equal to the time needed to process the entire queue (i.e., the queueing delay). This means the number of queues that can be processed in the absence of the flow, which was calculated above, is equal to the number of packets that the flow has presumably sent during this period. This value is then multiplied by equation number (9), which represents the value to be added when a packet from a flow with zero packets is accepted. The result is the value to be added when a packet arrives from a flow after a hold period.

For a flow that leaves the queue for short periods of time less than “hold_int”, the probability that the value of “packets” will be increased by one and thus its “extraspace” is also increased when a new packet arrives is proportional to the time the flow is on hold. And so, over a long period of time if the time periods that the flow leaves the queue in are uniformly distributed, the “extraspace” for that flow will be increased by a value that reflects the total time the flow spent on hold over that long period of time. For example if “hold_int” was set to 0.1 second and a flow leaves the queue on average for 0.01 second every 0.1 second, with uniform distribution, then there is 10% chance that the flow will be on hold when the “packets” variables are increased by the FBDA. That means it will hap-
pen on average once every second, which represents the flow being on hold for 0.1 second per second, which is the desired result.

If there is no space left to accept a new flow in the gateway, the best way to deal with that is to delete the flow that has been on hold for the longest period of time, since it is the flow that most probably will not return. But if this method is too time consuming, another way to deal with the problem is to delete any flow that is on hold, except the flows that have a reserved bandwidth assigned to them, as discussed in chapter 4.
Chapter 4: The Extension To FBDA

This extension to FBDA enables it to assign to a certain nonresponsive flow a portion of the bandwidth that is different than the fair share, without the need for a separate queue and new scheduling methods. For such flows, the state variable "extraspace" has a different use. FBDA will calculate the bandwidth that the flow should receive in packets per second. Then, every second the "extraspace" variable associated with this flow will be increased by a value equal to the number of packets that the flow should be able to pass through the gateway in the next second. Each time a packet from this flow is accepted, "extraspace" is decreased by one.

In addition to a state variable specifying the bandwidth to be allocated to the flow, each such flow needs a counter to time the one-second time interval. To reduce the state variables number, all the flows that have a reserved portion of the bandwidth can share the same counter, if at the initial arrival of the flow the value by which "extraspace" is initialized is equal to the number of packets to be accepted from now until the end of the current (global) one second interval. If "extraspace" reaches zero or less before the end of the second, all the packets arriving from this flow will be dropped until the end of the second, where "extraspace" will be increased again by the number of the packets to be accepted in the next second (this number does not have to be constant). This will most probably result in that the flow at one point will not have any packets in the queue, and so it will be put on hold as described in the previous chapter. This is essential to prevent the flow from restarting as a new flow before the end of the current one-second period, and retake the bandwidth dedicated to it before the end of the period. This is why the hold period should be at
least one second long, to prevent any possibility of a flow with reserved space from being
deleted before the end of the one second time interval, and then having new packets from
that flow accepted as if they are from a new flow. Notice that the unused portion of
“extraspace” from any one time period is passed on to the next period, and is not lost. This
is needed when some packets get delayed at a previous queue. Note that the flows with no
reserved bandwidth should only see a portion of the average queue size and average flow
size equal to the portion of the bandwidth that is free (e.g. if 30% of the bandwidth was
reserved of certain flows, “av” must equal 70% of the average flow size), also “max_av”
should be reduced the same way.

Sometimes a flow with reserved bandwidth may start sending packets at a higher rate
than what is agreed upon with the gateway. Although this will not give the flow more
bandwidth than it should get, the flow could repeatedly overflow the buffer, and even fill
the entire buffer for a portion of the one-second time period. This will most probably hurt
the adaptive flows due to the loss of large number of packets, and might lead to global syn-
chronization among them and hurt the output bandwidth utilization. To deal with this
issue, whenever a packet is accepted from a flow with reserved bandwidth, FBDA will
check if the buffer is full, if it is full the time interval that FBDA uses to increase
“extraspace” is reduced by half (e.g. from 1 second to 0.5 second to 0.25 second and so
on). This will prevent the aggressive flow from having the time needed to fill the buffer.

In such case these aggressive flows will need a separate counter each to time their
intervals since each has a unique interval, so a number of counters should be set aside for
such circumstances. If not enough counters are available or if the time interval for increas-
ing “extaspace” became too small (that is the overhead of updating “extraspace” became
too large), the flow could be punished by reducing the bandwidth assigned to it (still per second), or it could be cut off completely.

Note that if the time interval mentioned above is allowed to become too small the algorithm will not work (that is in the extreme cases when the time interval is close to the time needed to update “extraspace”). One situation where this could happen is when 100% of the bandwidth is reserved for certain non-adaptive flows and all the other flows are not cut off before the reservation takes effect. In his case any increase in the average queue size due to the arrival of packets from a flow with no reserved bandwidth is permanent, because the total average arrival rate will never go below 100%, and so the buffer may overflow constantly. In this case either the time interval should be fixed, or all the flows with no reserved bandwidth should be cut off before the reservation takes place.

The following is a summary of FBDA. For clarity the equations are substituted by their numbers, and parts of the algorithm is referred to by using a previous figure number.

At packet arrival:

If the buffer is not full
{
    If the flow is new
    {
        accept packet. update “flows_num_”, “Pack_num_”, and “packets”
        if the buffer was empty
            adjust “av”, “max_av” and “queue_av” appropriately
set "extraspace" to rate + eq(9)

}  

If the flow is not new  

{  

If the flow has a reserved bandwidth and "extraspace" is larger than zero. OR if the flow does not have a reserved space, and either "packets" is less than "av" or "extraspace" is larger than or equal to eq(1)  

{  

accept packet . update "Pack_num_"

if the flow does not have a reserved space  

{  

if "packets" is negative  

{  

if the buffer was empty

adjust "av", "max_av" and "queue_av" appropriately  

if "queue_av" is not zero

increase "extraspace" by eq(10)

set "packets" to one  

}  

}  

else

if "packets" is larger than "max_av" and "extraspace" is larger than or equal to eq(1)*

decrease "extraspace" by eq(1)
else if “packets” is less than “max_av” - 1

    increase “extraspace” by eq(5)

    update “packets”

}

if the flow has a reserved space

{

decrease “extraspace” by 1

if “packets” is negative

{

    if the buffer was empty

        adjust “av”, “max_av” and “queue_av” appropriately

        set “packets” to one

}

else update “packets” (i.e. increase “packets” by one)

    if the buffer is now full

        decrease the time interval associated with the flow by half

}

}

}

}

}

}

Else drop the packet, and if the flow does not have a reserved bandwidth

    update “extraspace” as in figure (6)

--------------------------
At packet departure:

update “av”, “queue_av” and “max_av” as in figure (3)
decease “packets” by one
if “packets” equals zero
    set “packets” to negative “reserve”
if “packets” equals zero  (when “reserve” is zero)
    delete the flow and decrease “flows_num” by one
update “Pack_num_”

Recurring events:

Every “hold_int”:
for all flows with a negative “packets” value
    increase “packets” by one
for all flows if “packets” equals zero
    delete the flow and update “flows_num_”

Every second (or the flow specific interval):
for non-adaptive flows with a reserved bandwidth:
    increase “extraspace” by the number of packets the flow can pass in the next time interval

*The second condition is added to prevent “extraspace” from going below zero, this can happen if the packet was accepted because “packets” is less than “av”, but also was larger than “max_av” and “extraspace” was less than eq. (4).
Chapter 5: The Cost Of FBDA

The "cost" of running FBDA can be examined from two points of view, the physical cost of the hardware used, and the overhead "cost" in running the algorithm. Both of these parts are interconnected, and they are both dependent on the accuracy requirements desired. The more hardware available, the shorter the execution time will be. The more accurate the results that are required, the longer it will take to execute the algorithm.

The overhead cost of running the algorithm can be divided into two parts, the time needed to perform the arithmetic operations, and the time spent on read and write operations (ignoring the cost of control signals and logic operations).

5.1. The Cost of the Read/Write Operations:

The basic FBDA use only two per-connection variables, "packets" and "extraspace". As mentioned before in chapter 3, "packets" could be represented by a small field of bits with a size dependent on the maximum expected number of packets each flow may have in the queue. The size of the "extraspace" field depends on how accurate that value is required to be. One can see that these two variables combined do not need a lot of storage space, which means a large number of flows can be served at the same time using a small cache memory, or even using only on-board registers/memory (one possible implementation is to keep active connections in the general purpose registers and saving the "reserved" flows in the cache memory until needed). This will reduce the read/write operations overhead by reducing the number of read/write operations needed (both variables can be read or written at the same time), and by reducing the cost of each such operation,
while keeping the hardware cost at an acceptable level. This also makes the idea of "reserving "flows for a period of time feasible. FBDA also use about a dozen global variables and setup parameters, but if the hardware is built to handle a large number of flows at the same time (i.e. in the hundreds), then the cost of these global variables is negligible.

5.2. The Cost of the Arithmetic Operations:

Looking closer at the cost of performing the arithmetic operations in FBDA, one can divide the algorithm into three parts, operations executed at packet arrival, operations executed at packet departure, and scheduled operations executed regularly.

5.2.1. Cost of the Scheduled Operations:

There are two scheduled operations, one is the "extraspace" incrementing operation for the reserved flows, where as described in chapter 3, the cost depends on the parameters’ settings and the hardware available. Note that the array of small adders that is used for this operation can be shared by a number of output ports, which will reduce their cost (but having them on a separate board will increase the communication cost, and a controller is needed to assign access time for each output port). The other scheduled operation is increasing "extraspace" for the connections with a reserved bandwidth in the FBDA extension, which cost one add operation every one second time interval if the time intervals of all the connections are synchronized, and if enough hardware is available to perform the operations for all connections in parallel (in general, the number of add operations needed is equal to the # of connections with reserved space / # of adders available, per second). To synchronize the time intervals some additional operations are needed at the arrival of the
first packet of the flow, to calculate the bandwidth the flow should have until the end of the current interval. Multiplying the time left by the space reserved for the flow per second does this. These values could be approximated to allow the performing of the multiplication in a number of shift operations. Note that misbehaving flows with reduced time intervals needs to have their add operations done separately, but the number of these flows should be too small to noticeably affect the overall overhead.

From the discussion above one can see that if enough hardware is available the overhead of the scheduled operations is very small compared to the cost of processing tens of thousands of packets per second for high speed links, and so it can be ignored.

5.2.2. Cost Of The Arithmetic Operations At Packet Departure:

At packet departure, other than updating the variables “Pack_num_” and “packets”, which can be done in parallel, the big overhead is adjusting the averages as in figure (3).

Calculating “av” and “queue_av” can be done in parallel, and if “w” was chosen so that the multiplication operations can be done with some shift operations, then the overhead is mainly one division and one add operation. Since these operations are done at the departure of the packets, rather than at their arrival (as in RED and FRED), this overhead is equally distributed over time rather than being concentrated at the times when the arrival rate is high and where the overhead needs to be small. Also adjusting the averages this way gives better average representation since the sampling rate is constant. when the buffer is not empty.

Another method to adjust the averages is to do a sampling of the queue size at time intervals equal to the time needed to process an average-size packet. This method will dis-
tribute some of the overhead over the idle periods, and eliminate the need to adjust the averages after each ideal period, which is inaccurate and adds a lot of overhead [1]. If the division operation is considered too costly, the sampling interval can be adjusted to reduce the cost.

In the other part of Figure (3), the overhead is one add and one division operation. The division operation needs to be done only when the number of flows in the queue changes. And since the flows are reserved for some time and not immediately deleted every time their packet count goes to zero, the change in the number of packets over time is small, and so the overhead of this operation is acceptable. Increasing the reservation time of the flows will reduce this overhead, but one should be careful not to increase it too much as this may reduce the output throughput.

The overhead cost of running FBDA can be cut if the calculations to update the averages are taken out of the critical path of packet processing and instead are run in parallel with the calculations done to determine whether or not to accept a packet. In this case, the only overhead for calculating the averages is the time needed to update the variables with the new values, if that conflicts with a read operation (since one can not read and write a variable at the same time). The fact that FBDA will use the old values of the averages for the packets arriving during the time when the averages are calculated can be ignored since the time interval between updating the averages is still the same, and so the control of the queue size will not be affected.

5.2.3. Cost of the Arithmetic Operations at Packet Arrival:

At packet arrival, in addition to the occasional updates of variables such as
"Pack_num_" and "packets", and in addition to the adjustment of the averages after idle periods (if this method was chosen), there are two basic costs, executing the control instructions, and executing the appropriate response to the result of the control instructions.

Since all the control functions are independent, they can be executed in parallel if the required hardware is available. The control function that takes the longest time, and therefore represents the overhead, is the one that evaluates equation (1) and checks if “extraspace” is larger than equation (1). The overhead for this operation is two subtractions and one division operation (ignoring logic and control signals). The cost of the division operation (as all division and multiplication operations in the algorithm) depends on how accurate the result is required to be. Since “qlim_ - queue_av” is only meant as an indicator of a growing congestion, the division result is not required to be very accurate. One can even approximate the value of “qlim_ - queue_av” so that the division can be done using shift operations, if needed (a table can be used to define the approximation, for example, the integer part of the result of the subtraction can be used to define a location in the table containing the approximation).

The other part of the overhead at packet arrival is the action to be taken according to the state of the flow. If the packet is to be dropped, then the additional overhead is an add operation only if the number of packets is below the threshold (the thresholds needs to be calculated only when the number of flows in the queue changes).

If the flow has a reserved bandwidth, then the additional overhead is no more than decreasing “extraspace” by one if the packet is accepted, and in the cases where the buffer overflows, dividing the flows’ time interval, that controls the increase of “extraspace” by
two (this will not happen frequently, and so can be ignored).

If the flow does not have a reserved bandwidth, and the packet is accepted, then one of the three equations 1, 5, or 10 must be executed. If it is equation (1), which is the most probable result, then no additional overhead exist, since this equation was evaluated in the related control function.

If equation (5) is needed, three adds, one multiplication, and one division operations are needed. This along with the updating of the "av" variable represents the most significant arithmetic operations overhead in the algorithm.

Equation (10) requires three add operations, two multiplications and one division. But since it is only executed when a packet arrives from a reserved flow, the frequency of its execution is very small and so its effect is minimal. Also some other tests can be added to avoid the execution of this equation when it is not necessary, for example it need not be executed when "max_av" is zero, or when (reserve + packets) is zero. Also since this equation is basically an approximation its result is not required to be accurate, so the multiplication and division operations can be approximated through the use of integers instead real numbers and some shift operations.

One can conclude from the discussion above that if the appropriate hardware is available, the read/write overhead for FBDA will be small, and the total arithmetic operations overhead will be at most around nine add or subtract operations and two to three multiplications or divisions (depending on the frequency by which the number of connections in the queue changes, which depends on the average queue size, the flow reserving period, and the how bursty the different flows are) per packet in addition to a number of shift operations. In estimating this value, few assumptions were made. First all packets are consid-
ered to be from a connection with no reserved space, second the packet drop rate is zero, third the averages calculations are not done in parallel with the other operations, fourth equation (5) has to be evaluated for about 50% of the packets, and finally equation (10) and the scheduled operations are ignored. The first four assumptions increase the expected overhead, while the last assumption reduces it. Table (1) below summarize the cost of the arithmetic operations in FBDA.

In comparison, in [1] an implementation of RED is suggested where, after some heavy approximations, the algorithm can be performed with about six add/subtract operations, and some shift operations.

The overhead can be greatly reduced if some operations on the arriving packets are performed in parallel with the execution of the FBDA algorithm. For the vast majority of the packets that will be accepted in the normal circumstances performing some necessary operations on the packets at the same time the FBDA algorithm is running will reduce the execution overhead of FBDA by a value equal to the time needed to execute these operations. On the other hand, for the small minority of the packets that will be dropped, these operations should be undone, and the overhead will not change, but when a packet is to be dropped the execution overhead of FBDA is small anyway since “extraspace” will not be changed.

One such operation that can be done in parallel with FBDA is copying the packet to the output buffer. The overhead of the copying depends on the buffer access time and the packet size, which in most circumstances is substantial. FBDA should have the capability to interrupt the copying if it finds that the packet should be dropped, so that no time is wasted on unnecessary copying. The pointer pointing to the beginning of the free space in
the buffer before the copying begins should be saved, until the results from the FBDA algorithm are obtained, so that the space occupied by the new packet (or a part of the new packet) can be overwritten if the packet is to be dropped. Also, if the buffer was empty at that time the output port should not start processing that packet until it gets a signal from the FBDA circuit indicating that the packet is accepted.

<table>
<thead>
<tr>
<th>Algorithm part</th>
<th>Bottleneck operations (after full parallelism)</th>
<th>Cost (# of op. needed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add/Sub, Mul./Div</td>
<td></td>
<td></td>
</tr>
<tr>
<td>scheduled operations (can be ignored)</td>
<td>“extraspace” incrementing operations for the reserved flows</td>
<td>1* per hold_int 0</td>
</tr>
<tr>
<td></td>
<td>increasing “extraspace” for the connections with reserved bandwidth in the FBDA’s extension</td>
<td>1** per second 0</td>
</tr>
<tr>
<td>Packet departure</td>
<td>updating “Pack_num” and “packets”</td>
<td>1 0</td>
</tr>
<tr>
<td></td>
<td>updating the averages “av” and “queue_av” (not removed from the critical path)</td>
<td>1 1</td>
</tr>
<tr>
<td></td>
<td>updating “max_av”</td>
<td>1 1***</td>
</tr>
<tr>
<td>Packet arrival$</td>
<td>updating “Pack_num” and “packets”</td>
<td>2 0</td>
</tr>
<tr>
<td></td>
<td>evaluating eq (1)</td>
<td>2 0$$</td>
</tr>
<tr>
<td></td>
<td>evaluating eq (5)</td>
<td>3 2</td>
</tr>
<tr>
<td>Total$$</td>
<td></td>
<td>9 2 - 3</td>
</tr>
</tbody>
</table>

Table 1: Cost of the arithmetic operations in FBDA.

* Assuming enough H.W. available (see discussion).

** Assuming enough H.W. available, and ignoring the synchronization cost and misbehaving flows cost.

*** Only when the number of flows in the queue changes. Updating the thresholds can be done in parallel with this operation.
$ The cost of adjusting the averages after an idle period is not mentioned.

$$ After approximation to do a division using shift operations.

$$$ Eq(5) is assumed to be evaluated for 50% of the packets.
Chapter 6: Tests And Results

Five network simulations were run using the ns network simulator "ns v2.1b5" (the VINT project), which is a collaboration between UC Berkeley, LBL, USC/ISI, and Xerox PARC. The simulator is written in C++ and uses OTcl as a command and configuration interface.

In the first two experiments, tcp_A and tcp_B, six FTP connections with six different sources and one destination (sink) shares a gateway, as in figure (7) below.

![Diagram of network simulation]

Figure 7: Experiments tcp_A and tcp_B. Six tcp connections sharing one link.
The six connections use TCP newreno as a transport layer protocol. The connections always have data ready to be send, and the destination (sink) will send an acknowledgment immediately after receiving each data packet.

Each connection has a maximum window size of 50 packets. The packets have a fixed size of 512B each, and the buffer size is set to 120 packets. Each link between a source and the gateway has a bandwidth equal to 100Mb/second. The six connections have a round trip propagation delay of 4ms, 8ms, 16ms, 32ms, 64ms and 128ms.

The output link (between the gateway and the sink) has a bandwidth of 10Mb/second for experiment tcp_A, and 20Mb/second for experiment tcp_B.

Experiments all_A and all_B, use the same network topology as the previous experiments. But there are only five connections sharing the output link. Two of these connections are used by nonresponsive, constant bit rate applications, each using the UDP protocol. The other three connections are FTP connections using the TCP newreno protocol. The FTP connections have a round trip propagation delay of 4ms, 16ms and 32ms. Each has a maximum window size of only 20 packets. Each packet consists of 512B of data, and the buffer size is set to 100 packets. Each link between the sources and the gateway has a bandwidth of 100Mb/second. The link between the gateway and the destination (the output link) has a bandwidth equal to 10Mb/second.

The UDP connections have a propagation delay of 4ms each. One connection sends its data with a rate of 10Mb/second, while the other sends its data with a rate of 4Mb/second for experiment all_A, and 1Mb/second for experiment all_B.

Finally, experiment all_C is exactly identical to experiment all_A, except that the UDP connection that sends data with a rate of 10Mb/second has 3Mb/second of the output link bandwidth reserved for it. Figure (8) next shows the setup for experiments all_A, all_B,
Figure 8: Experiments all_A, all_B and all_C. Two udp and three tcp connections sharing one link.

All experiments were run using four settings of the “per” parameter, 0.1, 0.2, 0.33 and 0.5. For calculating the averages, the queue size weight “w” was set to 0.002. The “rate” parameter was set to one. “hold_int” is 0.05 second, and “reserve” is 21, making the reserve time period equal to at least one second. No minimum threshold was used in the FBDA experiments, except the fact that if the flow has no packets in the queue (i.e. a new flow or a reserved one) the packet is guaranteed to be accepted even if “max_av” is zero, this is because FBDA will accept a packet if the number of packets the flow has in the queue at that time (“packets”) is equal to or less than the limited average (“max_av”). For reserved flows the first packet is always accepted because the result of eq(1) that is compared with “extraspace” to decide whether to accept the packet or not will be negative,
also “packets” at that time will be less than “av” even if “av” is zero. This feature is impor-
tant to keep the reserved flows up to date and to prevent the deletion of active flows.

Experiments tcp_A and tcp_B were also run using the RED algorithm [1] for compari-
son. The RED parameters used are: 5 packets for “minth”, 1/60 for “maxp”, and four set-
tings of “maxth”, 100, 75, 50 and 25 packets. The values of these parameters were chosen
so that the differences in the average queue size (and thus the queueing delay) between the
RED and the FBDA experiments are as small as possible. For the experiments using RED,
“w” is also set to 0.002. Experiments all_A, all_B and all_C were not run using RED
because RED does not support nonresponsive flows (except the fact that all the packets
will be dropped if the average queue size exceeds an upper threshold, which in the experi-
ments done in this thesis will result in the UDP connections taking all the bandwidth and
the TCP connections being completely shut out). All the connections starts sending pack-
ets at the same time, and all of the simulations were run for 90 seconds.

Figures (9) to (12) below show the results of all the experiments, figure (9) is for all
tcp_A experiments (for both FBDA and RED simulations), the top of the figure shows the
different parameter settings. Each column represent one flow in one of the simulations,
where the first column from the left represents the connection with the smallest propaga-
tion delay. How much bandwidth a connection took is shown on the left hand scale (with
an accuracy of plus/minus 0.5% of the output link bandwidth). The hashed columns are for
the FBDA experiments, while the dotted columns are for the RED experiments.

The short, thick lines above or inside each column show the percentage of packets that
were dropped for that flow. The scale on the right hand side represents the percentage of
the packets dropped (with an accuracy of plus/minus 0.05%).

For the FBDA simulations, the connections that did not loss any packets, or lost less
than 0.05% of the packets sent, are hashed with thick lines. For the RED simulations all
the connections had a substantial packet loss.

Figure (10) is the same as figure (9) but for tcp_B.

Figure (11) is for experiments all_A and all_B. The dotted columns represent the non-responsive UDP connections, where the column on the left is for the UDP connection sending at a rate of 10Mb/second, and the hashed columns represent the adaptive TCP connections. The drop rate for the UDP connections is shown in numbers above each column because they are too large to be shown on the scale (the second UDP connection in all_B losses no packets, and so there is no drop rate shown for it). Figure (12) is for all_C experiments.

Figures (13) to (24) show the queue size in packets, as it is sampled once every 0.01 second, for all of the tcp_A experiments, and for tcp_B (using FBDA only), all_A, all_B and all_C experiments when “per” is set to 0.33.

Experiment tcp_A was done to show the behavior of FBDA under heavy congestion conditions and to show the ability of FBDA to control the average size of the queue according to the desired limit.

As can be seen from the figures, although the average queue size is usually larger than the maximum queue size set by “per”, FBDA was able to keep it close to the desired value, and the buffer almost never overflows.

One can see that the average queue size for each pair of FBDA and RED figures is almost the same (figure (13) to figure (20)), which means the effect of any difference in the queuing delay on the difference in the bandwidth distribution between FBDA and RED is minimal. Also from the tcp_A figures one can see that the value of the “per” parameter in FBDA gives a better indication to the average queue size that will result than the value of
Figure 9: Experiment tcp_A. Round trip propagation delay (left to right): 4, 8, 16, 32, 64, and 128 ms.
percentage of B.W. occupied by a flow

using FBDA using RED using RED using FBDA using RED using FBDA using RED using FBDA using RED
per = 0.5 maxth = 100 per = 0.33 maxth = 75 per = 0.2 maxth = 50 per = 0.1 maxth = 25

percentage of packets dropped

Figure 10: Experiment tcp_B. Round trip propagation delay (left to right): 4, 8, 16, 32, 64 and 128 ms.
Figure 11: Experiment all_A. Prop. delay for tcp (hashed) connections (left to right): 4, 16 and 32 ms. Sending rate for udp connections (left first): 10Mb/s, 4Mb/s.

Figure 11: Experiment all_B. Same as experiment all_A, except the sending rate for the udp connection on the right is 1Mb/s.
percentage of B.W.
occupied by a flow

per = 0.5 | per = 0.33 | per = 0.2 | per = 0.1

percentage of packets dropped

70.4% | 70.5% | 70.5% | 70.2%

48.7% | 47.6% | 44.6%

38.2%

20%

15%

10%

5%

Figure 12: Experiment all_C. Round trip propagation delay for tcp (hashed) connections (left to right): 4, 16 and 32 ms sending rate for udp connections (left first): 10Mb/s (with 3Mb/s reserved), 4Mb/s.
Figure 13: experiment: tcp_A, max. average queue size desired: 60 packets

Figure 14: experiment: tcp_A (using RED), max. threshold is 100 packets
Figure 15: experiment: tcp_A, max. average queue size desired: 40 packets

Figure 16: experiment: tcp_A (using RED), max. threshold is 75 packets
Figure 17: experiment:tcp_A, max. average queue size desired: 24 packets

Figure 18: experiment:tcp_A (using RED), max. threshold is 50 packets
Figure 19: experiment:tcp_A, max. average queue size desired: 12 packets

Figure 20: experiment:tcp_A (using RED), max. threshold is 25 packets
Figure 21: experiment:tcp_B, max. average queue size desired: 40 packets

Figure 22: experiment:all_A, max. average queue size desired: 33.3 packets
Figure 23: experiment: all_B, max. average queue size desired: 33.3 packets

Figure 24: experiment: all_C, max. average queue size desired: 33.3 packets
"maxth" or "maxp" (which can also be changed to change the average queue size) in RED.

Experiment tcp_B was run under less congested condition, where as can be seen from figure (21), when using FBDA, the average queue size did not reach the limit specified when the limit is set to a high value. The results in this experiment is largely dependent on the flow reserving mechanism, this is due to the fact that some of the flows (especially the ones with large propagation delay) leave the queue very often, and for a long periods of time, this is due to the small queueing delay in comparison to the propagation delay. The queueing delay is smaller than in experiment tcp_A due to having a smaller average queue sizes (because of the smaller congestion), and a faster output link.

In both tcp_A and tcp_B one can see that the results get better as the value of "per" gets larger (except when "per" is set to 0.5, where the results are worse than the case when "per" is set to 0.33). Part of the reason why this happens is the fact that the average queue size and the queueing delay gets bigger as "per" gets bigger, which will reduce the effect of the propagation delay and helps the connections with a large propagation delay. Also as the average queue size gets smaller, the algorithm will reduce its fairness requirement in order to have a high link utilization (see section 3.2.1.2). But the main reasons are that when "per" is set to a small value, "max_av" will be too small for the flows to get a good representation for the time they spend taking less than the fair share of the output bandwidth, this is because equations (4), (8) and (9) are executed using (and triggered by) the value of "max_av" and not the real average "av" (which means, for example, that equation (8) will never be executed when "max_av" is limited to two), this problem will get worse as the difference between the real average "av" and the limited average "max_av" gets bigger (due to increased congestion). Also having higher "max_av" will provide a larger
range of values that “packets” can have below “max_av”, which in the long run will give a more accurate average estimates (that is, using a scale from one to ten will give a more accurate results than using a scale from one to three). And so, unless a large buffer space is available it is better to have a large value of “per” in order to get a large “max_av” (e.g. five packets or more, taking the expected number of flows in consideration), even if the queueing delay is a little high. But “per” should not be set too high to prevent the empty part of the buffer from becoming too small.

For tcp_A and tcp_B when “per” is set to 0.33, the exact bandwidth distribution is 17.95%, 17.72%, 17.81%, 16.92%, 15.89% and 13.72% for tcp_A, and 20.51%, 19.65%, 19.57%, 18.49%, 13.8% and 7.47% for tcp_B. The fair share bandwidth for tcp_A is 17.26% and for tcp_B it is 19.68%. The fair share is calculated by subtracting the bandwidth taken by the flows that did not lose any packets (i.e. the last flow in tcp_A and the last two flows in tcp_B) from 100, and then dividing the result by the number of the flows that lost a substantial portion of their packets. The fair share is calculated this way because the flows that did not lose a significant number of packets took all the bandwidth that they can take under the current circumstances (e.g. the queueing delay).

Although the exact numbers will change slightly by changing the experiments setup (e.g. the weight “w” used in calculating the averages, the simulation running time,...), they show that all the flows, which had a significant portion of their packets dropped, took almost the exact fair share (plus/minus 10%). The best results when using RED for experiment tcp_A is 28.9%, 26.32%, 18.73%, 12.37%, 9.11% and 4.58%, and for tcp_B the best results are 34.78%, 27.13%, 18.59%, 10.43%, 5.83%, and 3.25%. The reason why the RED results are not as good as the FBDA results is that all the flows have almost the same
packet drop rate, and the fact that the lose of a packet will affect the flows with large propagation delay more than the aggressive flows with small propagation delay.

In the case when “per” is set to 0.5, in the experiments using FBDA, the results are worse than expected because the high congestion will result in an average queue size larger than 70% of the buffer size at some points, when all the flows have some packets in the queue, and at these levels the results from implementing the different equations (4, 5, 8, 9, 10) changes rapidly with small changes in the average queue size (because of the geometrically increasing slop representing the free part of the buffer). This will hurt the connections with large propagation delay more (e.g. tcp 4 in tcp_A), because the arrival of some packets from them causes the average to go up by an amount sufficient, at those high average values, to change the results of the different equations (and therefore the drop rate) significantly, while close connections will always have packets in the queue (this has an effect similar to a very mild case of tail dropping). This problem will diminish if there are a large number of flows using the gateway instead of just six, because the effect of the absence or arrival of one flow will not affect the average values significantly.

Experiments all_A and all_B show the ability of FBDA to control nonresponsive flows in the presence of responsive flows. In all_A, the UDP connection with the lower transmission rate took more bandwidth than the other UDP connection because its “packets” value sometimes fell below “max_av” - 1 (especially when “per” is high, where “max_av” will more likely be equal to “av”, which limits the non-responsive flows), and so “extraspace” will be increased, and the flow will be able to get packets accepted above the flow average until “packets” reach one of the thresholds and “extraspace” is set to a negative value again. In all_B the second udp connection did not lose any packets since it needs less
bandwidth than the fair share.

In experiment all_C, the 10Mb/sec connection has 30% of the bandwidth reserved for it. And in all cases it took almost exactly that amount. Notice how in figure (24) the UDP flow with the reserved space filled the entire buffer a few times at the beginning of the simulation, until the time interval between increasing "extraspace" became small enough to prevent that.

The TCP connections did not do well when "per" is set to a small value, this is because the average queue size is still high due to the fact that the UDP connection with the reserved bandwidth is not affected by the limit on "max_av". The large queue size means that each TCP connection will have a large number of packets in the queue despite the fact that "max_av" is small. This will result in high drop rate for the TCP connections, while the other UDP connection is controlled only by the high "av" value. Thus "per" should not be set to a small value if a large portion of the bandwidth is reserved for specific flows. Another reason why the UDP connection with 4Mb/sec. rate took more bandwidth than its fair share, is that when the connection with the reserved space is cutoff, the queue size drops substantially in very short time, and the number of packets the 4Mb/sec. udp connection has will drop bellow "max_av" -1, and so "extraspace" will be increased as described for experiment all_A, but with more frequency. This problem will also diminish if a larger number of flows share the buffer.
Chapter 7: Conclusions

FBDA is a simple algorithm that provides fair bandwidth distribution during congestion periods between the different adaptive and non-adaptive flows using an output link, regardless of the round trip delay or the transmission rate, while maintaining a high link throughput.

FBDA uses the distribution of packets between the different flows in the per-link output queue as an indicator to the distribution of the output link bandwidth in the next time period needed to process the current queue. It uses a per-connection variable to memorize how much bandwidth each flow has taken over a certain period of time compared to its fair share. The algorithm then drops the incoming packets according to this bandwidth distribution. FBDA provides high link throughput by controlling the average queue size, this is done by dynamically changing the drop rate when the average queue size gets too small or too large (large average queue size may lead to tail dropping and a bad performance). The fairness results are very good, especially if enough buffer space is available.

FBDA is especially useful in the Internet environment since it does not discriminate against bursty flows or adaptive flows with large bandwidth-propagation delay product, FBDA can also handle nonresponsive flows with no extra cost. The algorithm does not need a specially designed transport layer protocol and does not need coordination and communication between the different routers in the network.

With some small modifications, FBDA can be used to reserve a portion of the bandwidth for specific adaptive flows, this will enable the service provider to sell the bandwidth to the users who are willing to pay to get more than their fair share of the
bandwidth. With a small extension to it, FBDA can be used to reserve bandwidth for non-responsive flows that are time sensitive, no extra buffers or scheduling mechanisms are required.

The use of a small number of per-connection state variables, the small execution overhead and the simple FCFS scheduling mechanism makes FBDA suitable for high-speed network gateways with a large and variable number of flows that can afford to have a large average queue size.

FBDA is flexible, scalable and robust against slight mistuning of parameters, the execution overhead of the algorithm can be easily changed by changing some parameters to the desired level, taking into account the availability and speed of hardware, the expected load, and the desired output accuracy. If enough hardware is available to fully make use of the parallelism available in the algorithm, the arithmetic operations overhead per packet will be no more than nine add/subtract operations and two to three multiply/divide operations (one to two multiply operations more will be needed if bandwidth reservation for adaptive flows is implemented). Also the design of FBDA makes it easy to add or remove parts from it if needed at the implementation level.

Simulation results, using TCP newreno traffic, show that if the limit imposed on the flow size value used in the calculations is allowed to be six or more packets, each flow may get to as close as +/- 10% of its desired bandwidth.
Bibliography:


Appendix A

Implementation of the FBDA algorithm in C++.

File new.h:

```
#include "queue.h"
#include "flows_data.h"

class FBDA : public Queue {
public:
    FBDA();

protected:
    int command(int argc, const char*const* argv);
    void enqueue(Packet*);
    Packet* dequeue();
    PacketQueue *q_; /* underlying FIFO queue */

    double per;
    double max;

    int Pack_num_; 
    int flows_num_; 

    double av, max_av, queue_av, w, rate; 

    int reserve, i, temp3, temp4;
    flows_state st;
    double pack_per_hold_int;
    double pack_per_empty_int;
    double temp2;
    int off_ip_;
    Tcl tcl;
```
File new.cc:

#include "new.h"

static class FBDAClass : public TclClass {
    public:
        FBDAClass() : TclClass("Queue/FBDA") {}
        TclObject* create(int, const char*const*) {
            return (new FBDA);
        }
    } class_FBDA;

    int FBDA::command(int argc, const char*const* argv) {

        if ( (argc == 2) && !strcmp(argv[1], "hold_timer")) {
            i = 0 ; temp4 = 0 ;
            while (i < flows_num_) {
                if ((temp3 = st.id(temp4)) != 0) {
                    i++;
                    int pk = st.read(temp3,pk);
                    if (pk <= -1) {
                        pk++;
                        st.write(temp3,pk);
                        if (pk == 0) {
                            flows_num_--;
                            tcl.evalf(" set give(%d) {} ", temp3);
                        }
                    }
                    temp4++;}
            return (TCL_OK); }

        if ( (argc == 4) && !strcmp(argv[1], "add_space")) {
            int n = atoi(argv[2]);
            int n2 = atoi(argv[3]);
            double extraspace = st.read(n, extraspace);
            if (extraspace != -0.5) {
                extraspace = extraspace + n2;
                st.write(n, extraspace);
            }
            return (TCL_OK); }

        return Queue::command(argc, argv);
    }

}
FBDA::FBDA() : Pack_num_(0), flows_num_(0), av(0), max_av(0),
queue_av(0), rate(1)
{
q_ = new PacketQueue();
pq_ = q_; bind("per", &per);
bind("w", &w);
bind("pack_per_hold_int", &pack_per_hold_int);
bind("pack_per_empty_int", &pack_per_empty_int);
bind("off_ip_", &off_ip_);
max = qlim_*per;
reserve = 21;
tcl = Tcl::instance();
}

void FBDA::enqueue(Packet* p)
{
hdr_ip* hdrip = (hdr_ip*)p->access(off_ip_);
int id = hdrip -> flowid();

if ( (Pack_num_ < qlim_) && (st.check_flow(id) == 0))
{
flows_num_++;
Pack_num_++;

if (Pack_num_ == 1)
{ tcl.eval("av_cal stop");
    for (i = 0 ; i < pack_per_empty_int ; i++)
    { av = (1 - w) * av;
      queue_av = (1 - w) * queue_av; }

    if ( av < max/flows_num_ )
    { max_av = av; }
    else
    { max_av = max/flows_num_ ; }
}
if ((temp2 = st.has_space(id)) != 0)
    {
st.write(id, 1, 0);
tcl.evalf(" set give(%d) {reserved %d %f} ", id, id, temp2);
tcl.evalf(" reserved %d %f ", id, temp2);
    }
else
    {
st.write(id, 1, rate + ((2*max_av + 1)*max_av)/
                      (qlim_ - queue_av));
    }
q_->enqueue(p);
}
else
{
    int packets = st.read(id, packets);
double extraspace = st.read(id, extraspace);

if ( (Pack_num_ < qlim_) &&
     ((st.has_space(id) != 0 && extraspace > 0) ||
      (st.has_space(id) == 0 &&
       (packets < av || extraspace >= (packets
- max_av )/(qlim_ - queue_av)))))
{
    Pack_num_++;

if (st.has_space(id) == 0)
{
    if (packets < 0)
    {
        if (Pack_num_ == 1)
        {
            tcl.eval("av_cal stop");
            for (i = 0; i < pack_per_empty_int; i++)
            {
                av = (1 - w) * av;
                queue_av = (1 - w) * queue_av;
            }

            if ( av < max/flows_num_ )
            {
                max_av = av;
            }
            else
            {
                max_av = max/flows_num_;
            }
        }

        if (queue_av != 0)
        {
            extraspace = extraspace + (pack_per_hold_int
* (reserve + packets)/queue_av)*
* ((2*max_av + 1)*max_av)/(qlim_ - queue_av);
        }

        st.write(id, 1, extraspace);
    }
}
else
{
    if ( (packets > max_av) && (extraspace >=
        (packets - max_av)/(qlim_ - queue_av)) )
    {
        extraspace = extraspace - (packets -
        max_av)/(qlim_ - queue_av);
    }
    else
    {
        if (packets < max_av - 1)
        {
            extraspace = extraspace +
            ((2*max_av - packets + 1)*(max_av -
                packets))/(packets + 1)*(qlim_ - queue_av));
        }
    }
    packets++ ;
st.write(id , packets , extraspace) ;
}
else
{
    extraspace-- ;

    if (packets < 0)
    {
        if (Pack_num_ == 1)
        {
            tcl.eval("av_cal stop");
            for (i = 0 ;i < pack_per_empty_int ;i++)
            {
                av = (1 - w) * av ;
                queue_av = (1 - w) * queue_av ;
            }
            if ( av < max/flows_num__)
            {
                max_av = av; }
            else
            {
                max_av = max/flows_num__; }
        }
        st.write(id , 1 , extraspace);
    }
    else
    {
        packets++;
st.write(id , packets , extraspace) ;
    }

    if (Pack_num_ == qlim__)
    {tcl.evalf(" set reduce(%d) 1 ", id) ; }
}
q_-->enqueue(p) ;
else
else
{if (st.has_space(id) == 0)
{if ((packets < 4*max/flows_num_ &&
packets < qlim_/flows_num_ ) && (extraspace >= 0))
{extraspace = extraspace + rate ;}
else
{extraspace = - 0.001 ;}
}

st.write(id, extraspace) ;
drop(p) ;
}
}
}

Packet* FBDA::deque()
{
Packet* pointer = q_->deque() ;
if (pointer != NULL)
{
hdr_ip* hdrip = (hdr_ip*)pointer->access(off_ip_) ;
int id = hdrip -> flowid() ;
int packets = st.read(id , packets) ;

Pack_num_-- ;
packets-- ;
if (Pack_num_ == 0)
{ tcl.eval("av_cal start") ;}

queue_av = (1 - w) * queue_av + w * Pack_num_ ;

av = (1 - w) * av + w * Pack_num_/flows_num_ ;
if ( av < max/flows_num_ )
{ max_av = av; }
else
{ max_av = max/flows_num_ ; }
if (packets == 0)
{packets = -reserve ;
if (packets == 0) { flows_num_--; }
st.write(id , packets) ;
}
return pointer;
}

File flows_data.h:

class flows_state {

private:

    int i, flows_space, flows_with_reserved_bw;

    struct flowdata { int fid;
                        int packi;
                        double extrai;
                    };

    flowdata *array;

    struct reserved { int fid;
                      double space; };

    reserved *reserved_space;

public:

    flows_state();

    int check_flow(int);

    int id(int);

    int read(int, int);

    double read(int, double);

    void write(int, int, double);

    int write(int, int);

    int write(int, double);

    double has_space(int);
}
File flows_data.cc:
#include "flows_data.h"

flows_state::flows_state() : flows_space(3), flows_with_reserved_bw(1) { 
    array = new flowdata [flows_space] ;
    for (i = 0 ; i < flows_space ; i++)
        {array[i].fid = 0 ;
        array[i].packi = 0 ;
        array[i].extrai = 0 ;
        } 
    reserved_space = new reserved [flows_with_reserved_bw] ;
    reserved_space[0].fid = 0 ; /* 1 for all_C */
    reserved_space[0].space = 3 /* Mb/sec */;
}

int flows_state::check_flow(int flowid)
{
    for (i = 0 ; i < flows_space ; i++)
        { if (flowid == array[i].fid)
            return 1 ; }
    return 0 ; }

int flows_state::id(int i)
{ return array[i].fid ; }

int flows_state::read(int flowid , int )
{for (i = 0 ; i < flows_space ; i++)
    { if (flowid == array[i].fid)
        return array[i].packi ; }
    return 0 ; }

double flows_state::read(int flowid , double )
{

for (i = 0 ; i < flows_space ; i++)
    { if (flowid == array[i].fid)
        return array[i].extrai ; }
return -0.5 ; }

void flows_state::write(int flowid , int p , double ext)
{
    for (i = 0 ; i < flows_space ; i++)
        { if (array[i].fid == flowid)
            { array[i].packi = p ;
              array[i].extrai = ext ;
              if (p == 0)
                  {array[i].fid = 0;}
              return ;}
        }
    for (i = 0 ; i < flows_space ; i++)
        { if (array[i].fid == 0)
            { array[i].fid = flowid ;
              array[i].packi = p ;
              array[i].extrai = ext ;
              if (p == 0)
                  {array[i].fid = 0;}
              return ; }
        }
    flowdata* temp = array ;
    flows_space++ ;
    array = new flowdata [flows_space] ;
    for (i = 0 ; i < flows_space-1 ; i++)
        { array[i] = temp[i] ; }
    delete [ ] temp ;
    array[flows_space - 1].fid = flowid ;
    array[flows_space - 1].packi = p ;
    array[flows_space - 1].extrai = ext ;
    if (p == 0)
        { array[flows_space - 1].fid = 0;}
    return ;}
}

int flows_state::write(int flowid , int p )


```c
for (i = 0 ; i < flows_space ; i++)
    if (array[i].fid == flowid)
        
        array[i].packi = p ;
        if (p == 0)
            array[i].fid = 0;
        return 0 ;
    
return 1 ;
```

```c
int flows_state::write(int flowid, double ext)

    for (i = 0 ; i < flows_space ; i++)
        if (array[i].fid == flowid)
            
            array[i].extrai = ext ;
            return 0 ;
        
return 1 ;
```

```c
double flows_state::has_space(int flowid)

    for (i = 0 ; i < flows_with_reserved_bw ; i++)
        if (reserved_space[i].fid == flowid)
            
            return reserved_space[i].space ;
    
return 0 ;
```

```c
```
Appendix B

The execution of experiment all_A in OTCL, including the operations in FBDA that needs scheduling.

#Create a simulator object
set ns [new Simulator]

#variables used in the names of the output directories and files
set exp all_A ; set method FBDA

#the size of any queue in the network
set max_size 100

#maximum percentage of the queue preferably used
set av_queue 0.33

#setup the limit, per and the w parameters
Queue set limit $max_size
Queue/FBDA set per $av_queue
Queue/FBDA set w 0.002

#declare the other class variables
Queue/FBDA set pack_per_hold_int {}
Queue/FBDA set pack_per_empty_int {}

#Define different colors for data flows
$ns color 1 Blue; $ns color 2 Red; $ns color 3 black
$ns color 4 Yellow; $ns color 5 white

#setup the directories for the output files
if ![file isdirectory experiments/experiment$exp/queue_av$av_queue] {
    if [file exists experiments/experiment$exp/queue_av$av_queue] {
        error " experiments/experiment$exp exists and not a directory ."
    }
} else {
    if ![file isdirectory experiments] {
        if [file exists experiments] {
            error " experiments exists and not a directory ."
        }
    } else {

exec mkdir experiments
exec mkdir experiments/experiment$exp
exec mkdir experiments/experiment$exp/queue_av$av_queue

} else {
    if ![file isdirectory experiments/experiment$exp] {
        if [file exists experiments/experiment$exp] {
            error " experiments/experiment$exp exists and not a
directory ."
        } else {
            exec mkdir experiments/experiment$exp
            exec mkdir experiments/experiment$exp/queue_av$av_queue
        } else {
            exec mkdir experiments/experiment$exp/queue_av$av_queue
        }
    }
}

#Open the nam trace file
#set files(nf) [open \
#experiments/experiment$exp/queue_av$av_queue/nam_out.$method w]
#$ns namtrace-all $files(nf)

#Create six nodes
set n0 [$ns node]; set n1 [$ns node]; set n2 [$ns node]; set n3 [$ns node]
set n4 [$ns node]; set n5 [$ns node]; set n6 [$ns node]

#Create links between the nodes
$ns duplex-link $n0 $n5 100Mb 1ms DropTail
$ns duplex-link $n1 $n5 100Mb 1ms DropTail
$ns duplex-link $n2 $n5 100Mb 1ms DropTail
$ns duplex-link $n3 $n5 100Mb 7ms DropTail
$ns duplex-link $n4 $n5 100Mb 15ms DropTail
set new_queue [ $ns duplex-link $n5 $n6 [set link_bw 10]Mb 1ms $method ]

# Function "duplex-link" has been altered to return the handle of the FBDA object

#set the network layout for the nam
$ns duplex-link-op $n0 $n5 orient down
$ns duplex-link-op $n1 $n5 orient right-down
$ns duplex-link-op $n2 $n5 orient right
$ns duplex-link-op $n3 $n5 orient right-up
$ns duplex-link-op $n4 $n5 orient up
$ns duplex-link-op $n5 $n6 orient right

# Create a trace and arrange for all link
# events to be dumped to "out.$method"

#set files(tf) [open experiments/experiment$exp/queue_av$av_queue/out.$method w]
#$ns trace-queue $n5 $n6 $files(tf)

#open the output files

set files(queuesize) [open experiments/experiment$exp/queue_av$av_queue/
queue_size.$method w]
puts $files(queuesize) "TitleText: experiment:$exp, max. average queue size desired: [expr $av_queue*100]% of queue"
puts $files(queuesize) "XUnitText: Time(sec)"
puts $files(queuesize) "YUnitText: Packets"
puts $files(queuesize) "\"queue size\""

set files(percentage) [open experiments/experiment$exp/queue_av$av_queue/percentages.$method w]
set files(alldata) [open experiments/experiment$exp/queue_av$av_queue/alldata.$method w+]

#Monitor the queue for the link between node 5 and node 6 (for nam)
#$ns duplex-link-op $n5 $n6 queuePos 0.5

#create a monitor object (for data collection) and connect an I/O channel to it

set fmon [$ns makeflowmon Fid]
$ns attach-fmon [ $ns link [ $n5 id ] [ $n6 id ] ] $fmon

$fmon attach $files(alldata)

#Create a UDP agent and attach it to node n0
set udp0 [new Agent/UDP]
$ns attach-agent $n0 $udp0
set cbr0 [new Application/Traffic/CBR]
$cbr0 attach-agent $udp0
$cbr0 set packet_size_ 512
$cbr0 set rate_ 10Mb
$udp0 set fid_ 1

#Create a UDP agent and attach it to node n0
set udp1 [new Agent/UDP]
$ns attach-agent $n1 $udp1
set cbrl [new Application/Traffic/CBR]
cbrl attach-agent $udp1
cbrl set packet_size 512
cbrl set rate 4Mb ;# 1Mb for "all_B"
$udp1 set fid 2

#Create a Null agent (a traffic sink) and attach it to node n6
set nullO [new Agent/Null]
$ns attach-agent $n6 $nullO

#Connect the UDP traffic sources with the traffic sink
$ns connect $udp0 $nullO; $ns connect $udp1 $nullO

# Set up Newreno TCP connections.
set srcO [new Agent/TCP/Newreno]
$ns attach-agent $n2 $srcO
$srcO set fid 3
$srcO set packetSize 512
#$srcO set window 100

set src1 [new Agent/TCP/Newreno]
$ns attach-agent $n3 $src1
$src1 set fid 4
$src1 set packetSize 512
#$src1 set window 100

set src2 [new Agent/TCP/Newreno]
$ns attach-agent $n4 $src2
$src2 set fid 5
$src2 set packetSize 512
#$src2 set window 100

set sinkO [new Agent/TCPSink]
$ns attach-agent $n6 $sinkO

set sink1 [new Agent/TCPSink]
$ns attach-agent $n6 $sink1

set sink2 [new Agent/TCPSink]
$ns attach-agent $n6 $sink2

# Create ftp sources at each node
set ftpO [new Application/FTP]; set ftpl [new Application/FTP]
set ftp2 [new Application/FTP]
$ftpO attach-agent $srcO; $ftp1 attach-agent $src1; $ftp2 attach-agent $src2

$ns connect $srcO $sinkO; $ns connect $src1 $sink1; $ns connect $src2 $sink2

#Define the 'finish' procedure
proc finish {} {
    global ns fmon link_bw runtimes exp method files av_queue \ 
         max_size

    $ns flush-trace

    #start at the beginning of the file
    seek $files(alldata) 0 start

    $fmon dump

    seek $files(alldata) 0 start

    puts $files(percentage) "Experiment: $exp\nQueue size: $max_size\nThe\nmax. average queue size desired: [expr \n    $av_queue * 100]\%\n"

    while {
        [gets $files(alldata) data] >= 0} {

        set times [lindex [split $data] 2 ]

        if {!$times != ""} {

            set id [lindex [split $data] 3 ]

            set arr($id) [lindex [split $data] 10 ]
            set drop($id) [lindex [split $data] 20 ]

        } else {

            set times [lindex [split $data] 3 ]

            set id [lindex [split $data] 4 ]

            set arr($id) [lindex [split $data] 11 ]
            set drop($id) [lindex [split $data] 21 ]

        }

        set per($id) [expr ($drop($id) / $arr($id).0)*100]
puts $files(percentage) "flow $id allocated bandwidth is [set \ total_flow_bw [expr (($arr($id)-$drop($id))/$runtimes(f$id))*8/1000000]Mbps\n"

puts $files(percentage) "percentage of the link bandwidth allocated to\nflow $id is [expr ($total_flow_bw /$link_bw)*100]%\n"

puts $files(percentage) "percentage of bytes dropped for flow $id\nis $per($id)% \n\n"

} #Close the trace files
foreach file [array names files] {
close $files($file)
}

#Execute nam on the trace file
exec nam experiments/experiment$exp/queue_av$av_queue/nam_out.$method &
exit 0

#Define a 'results' procedure
proc results {} {
    global ns fmon exp method files av_queue pace
    set time 0.01

    #start at the beginning of the file
    seek $files(alldata) 0 start

    $fmon dump

    #start at the beginning of the file
    seek $files(alldata) 0 start

    #current simulator running time
    set now [$ns now]

    while {[gets $files(alldata) data] >= 0} {
        set times [lindex [split $data] 2 ]
    }
if {$times !=""} {
    set id [lindex [split $data] 3 ]
    set arr($id) [lindex [split $data] 10 ]
    set drop($id) [lindex [split $data] 20 ]
} else {
    set times [lindex [split $data] 3 ]
    set id [lindex [split $data] 4 ]
    set arr($id) [lindex [split $data] 11 ]
    set drop($id) [lindex [split $data] 21 ]
}

if {$arr($id) != "" && $drop($id) != ""} {
    set acc($id) [expr $arr($id) - $drop($id)]
}

if {! [info exists pacc($id)]} {
    set pacc($id) 0
}

if {! [info exists files($id)]} {
    set files($id) [open experiments/experiment$exp/queue_av$av_queue/outbw$id.$method w]
    puts $files($id) "TitleText: experiment:$exp, max. average\n    queue size desired: [expr $av_queue*100]% of queue"
    puts $files($id) "XUnitText: Time(sec.)"
    puts $files($id) "YUnitText: bandwidth alocated(Mb/Sec)"
    puts $files($id) "flow#:$idB.W."
}

if {[info exists acc($id)]} {
    puts $files($id) "$times [expr \n    (($acc($id) - $pacc($id)) / $time) * 8 / 1000000]
    set pacc($id) $acc($id)
}

set queue_size [fmon set pkts_]
puts $files(queuesize) "$now $queue_size \\

#call "results" procedure every "time" simulated seconds
$ns at [expr $now+$time] "results"

}

#Define the 'reserved' procedure
proc reserved { id space } {

    global ns new_queue interval give reduce cbr[expr $id - 1]

    if { ![info exists interval($id)] } {
        set interval($id) 1
    } else {
        if { ![info exists reduce($id)] && $reduce($id) == 1 } {
            set interval($id) [expr $interval($id)/2.0]
            set reduce($id) 0
        } }

    $new_queue add_space $id [expr (($space * $interval($id) * 1000000) / (8 * [set cbr[expr $id -1] set packet_size_]))]

    #current simulator running time
    set now [$ns now]

    $ns at [expr $now + $interval($id)] "$give($id)"

}

set hold_int 0.1

set pack_per_sec [expr ($link_bw * 1000000)/(8 * [set src1 set packetSize_])]

$new_queue set pack_per_hold_int [expr $pack_per_sec * $hold_int]

#Define the 'timer' procedures
proc HOLD_timer { } {

    global ns new_queue hold_int

    $new_queue hold_timer

    set now [$ns now]
$ns at [expr $now + $hold_int] "HOLD_timer"
}

proc av_cal {state} {
    global t1 t2 ns new_queue pack_per_sec
    if {$state == "start"} {
        set t1 [ns now]
    }
    if {$state == "stop" && [info exists t1]} {
        set t2 [ns now]
        $new_queue set pack_per_empty_int [expr $pack_per_sec * ($t2 - $t1)]
    }
}

#the running time for each flow in seconds (needed for the bandwidth calculations)
set runtimes(f1) 90.0; set runtimes(f2) 90.0; set runtimes(f3) 90.0
set runtimes(f4) 90.0; set runtimes(f5) 90.0

#Schedule events for the agents
$ns at 0.0 "results"
if {$method == "FBDA"} {
    $ns at 0.0 "HOLD_timer"
}

$ns at 0.1 "$ftp0 start"; $ns at 0.1 "$ftp1 start"; $ns at 0.1 "$ftp2 start"
$ns at 0.1 "$cbr0 start"; $ns at 0.1 "$cbr1 start"

$ns at 90.1 "$cbr1 stop"; $ns at 90.1 "$cbr0 stop"
$ns at 90.1 "$ftp2 stop"; $ns at 90.1 "$ftp1 stop"; $ns at 90.1 "$ftp0 stop"

#Call the finish procedure after 90 seconds of simulation time
$ns at 91 "finish"

#Run the simulation
$ns run
Appendix C

The altered "duplex-link" function found in the "ns" simulator library, the function creates a two-way link.

Simulator instproc duplex-link { n1 n2 bw delay type args } {
    $self instvar link_
    set i1 [ $n1 id ]
    set i2 [ $n2 id ]
    if [ info exists link_(i1:i2) ] {
        $self remove-nam-linkconfig $i1 $i2
    }
    set q [ eval $self simplex-link $n1 $n2 $bw $delay $type $args ] ; # changed to save what is returned by the simplex-link procedure in the variable "q".
    eval $self simplex-link $n2 $n1 $bw $delay $type $args
    return $q ; # added to return the handler of the created link, the same command was added at the end of the simplex-link procedure.