Evaluation of DCBT

A Major Qualifying Project

Submitted to the Faculty of

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Bachelor Science

By

____
Brian Conway

____
Charles McAuley

____
Jonathan Yurek

Date February 28, 2002

Approved:

____
Professor Mark Claypool, Advisor

Abstract

This project implemented the Dynamic Class Based Threshold (DCBT) queue management algorithm in addition to a packet requeuing mechanism called Cut-In Packet Scheduling (ChIPS). These algorithms were then compared to other popular queue management schemes. To achieve this goal, code was developed as an extension to the Linux kernel. Extensive testing of DCBT, ChIPS, and other Linux queuing mechanisms took place in a controlled environment. Testing showed that these implementations were effective and fair as queue management schemes.

Table of Contents

1. Introduction 4
2. Related Work 10
   - 2.1 Probability Based Queues 10
   - 2.2 Class Based Scheduling 12
   - 2.3 Packet Requeuing 15
3. Approach 18
   - 3.1 CBT Implementation 18
   - 3.2 DCBT Implementation 19
   - 3.3 ChIPS Implementation 21
4. Experiments 23
   - 4.1 Setup 23
     - 4.1.1 Physical Layout 23
     - 4.1.2 Computer Setup 24
   - 4.2 Testing 26
     - 4.2.1 Tools 26
     - 4.2.2 Tests 27
5. Results and Analysis 30
   - 5.1 Throughput 30
   - 5.2 Jitter 35
   - 5.3 CPU Utilization 36
   - 5.4 Memory Usage 38
6. Conclusions 40
   - 6.1 DCBT 40
   - 6.2 CBT 41
   - 6.3 ChIPS 41
7. Future Work 43
   - 7.1 Implementation 43
   - 7.2 Experiments 44
8. References 46

Chapter 1: Introduction

The explosive growth of the Internet has created an opportunity for a wide range of applications to be quickly deployed over only a few years. The ever-increasing demand for bandwidth requires current traffic flows to be optimized as much as possible. The demand for streaming video and audio, along with the wide spread use of email and sharing of files among many users creates many kinds of traffic present in the Internet. While many algorithms and approaches exist at the application level in order to effectively send multimedia data, many of the bottlenecks involved in transmissions can be found not at the destination, or even the source, but in-between, with the routers and their queue management schemes.

Multimedia data has different requirements compared to most other traffic on the Internet, in that its data is time sensitive and can tolerate various amount of data loss. Multimedia traffic tends to make heavy use of the UDP protocol rather than TCP, as the data is rate-based and does not need guaranteed delivery. Most router queue management schemes do not accommodate for this difference between multimedia traffic and other unresponsive UDP flows, and either punishes both, or treats all flows with an equal discrepancy.

Unnecessary congestion can be easily simulated and explained by the lack of congestion control in the UDP specification. As a protocol, it does not take into consideration the dropping of its packets from a router's queue as a signal of congestion. UDP can be responsive to traffic congestion, but this responsiveness must be implemented at an

application level. Unresponsive UDP traffic, interested in simply transferring the information from source to destination can end up taking bandwidth from other, more "well behaved" traffic. Such UDP flows can effectively fill a simple FIFO queuing mechanism quickly and cause the router to start dropping packets from the top of its queue.

The default best-effort IP packet-forwarding service is typically implemented in routers by a single, fixed-size, First-In First-Out (FIFO) queue shared by all packets transmitted over and outbound link. Router implementations using a simple fixed-size FIFO queue typically just drop any packet that arrives to be queued to an already-full outbound queue. This behavior is often called drop-tail packet discarding. The queue provides a simple capacity for tolerating variability in the load, such as bursty traffic, on the outbound link. A short burst of packet arrivals may exceed the available bandwidth of the link even when the average load is well below the link bandwidth. However, when the average load exceeds the available capacity of the link for sustained periods of time, an unacceptably high amount of traffic is generated, resulting in drastically reduced performance of the network.

Active Queue Management (AQM) techniques involve predicating when a router's queue is under threat of filling up, and dropping packets according to certain probabilities before the queue fills. Such AQM techniques include RED [FJ93] and BLUE [FKSS99], and while some success has come from these approaches, they do not address the problem of identifying unresponsive flows, and instead risk punishing weak or responsive

protocols, such as TCP. RED (Random Early Detection), one of the earliest and most well known AQM techniques, increases its probability of dropping a packet as the router queue grows and notifies senders of dropped packets of router congestion. However, congestion-aware flows are, at best, just as likely to be dropped as non-responsive flows. At worst, these flows can lose their fair share of the router queue because while they curtail their own bandwidth properly, while a non-responsive flow will continue to flood the router with packets. BLUE's approach is similar to RED in that it drops packets according to queue size, but instead of using a set of complex settings adjusted by the router administrator to determine the probability of dropping a packet, it adjusts its probability according to the instantaneous queue size, based on previous changes in traffic. This approach can allow for a much more even limiting effect on the growth of the queue, but still neglects the difference between congestion-aware and non-congestion-aware flows.

A different approach to coping with network congestion involves classifying traffic into per-flow states. These states attempt to control the amount of bandwidth available to each flow traveling through the router, and punish flows that take more than their fair share. One such method is Flow Random Early Drop (FRED) [LM97]. FRED keeps track of each flow and attempts to protect them from each other. If a flow is non-responsive then it is incorporated into the RED algorithm and the flow is dropped with normal drop probability. However, if a flow is marked as unresponsive, then any time that flow takes more than its fair share, its packets are dropped, even if the queue is not under threat of congestion. This means that responsive flows can still be punished

unnecessarily, if they take more than their fair share even if for only a short while.

A problem with most of the queue management schemes we have discussed so far is that they make no differentiation between different types of traffic and the responsiveness of the flows. These different types of flows are often called classes. One class-based approach is Class-Based Queuing (CBQ) [FJ95]. CBQ isolates each different class of traffic and allots each a configurable amount of bandwidth. Each class of traffic is given its own queue which can have other management algorithms laid on top of it, such as priority queuing for real-time multimedia classes, or RED and similar algorithms for other flows instead of the typical drop-tail FIFO. However, the ability to quickly determine and classify all the separate types of traffic becomes very difficult as more classes are added.

Class Based Threshold (CBT) [CC00a] is similar to CBQ, but introduces more stringent rules to protect TCP flows. As opposed to CBQ's division of bandwidth, CBT sets different thresholds for three different types of flows: TCP, Multimedia UDP, and non-responsive UDP. According to the CBT algorithm, TCP traffic is always subject to RED, while flows in other classes have their average number of incoming packets compared against set thresholds, and if the average is above, than the next incoming packet is dropped, while if the average is below, then the flow's queue is subject only to RED. This prevents unresponsive flows from interfering with well-behaved TCP traffic by punishing the well-behaved traffic unnecessarily, while at the same time not punishing congestion-aware UDP flows. Still, like most other methods, CBT has preset thresholds

for its different classes, which may not always be set properly for the local network environment.

Dynamic-CBT (DCBT) [CC00a, CC00b], which we have implemented, has been proposed as a way to address the problem of these preset thresholds not being suitable for the local network environment under all conditions. DCBT functions by dynamically assigning the thresholds for UDP connections in a congested link as the makeup of the traffic through a router changes [CC00a]. This is necessary due to the congestion-unfriendly behavior of UDP traffic. DCBT does this in such a way that all the UDP connections share the available bandwidth that has been given to the UDP class, although there still may be contention within the UDP class. This will allow the router to adapt to congested network conditions as the nature of the connections changes. Before our implementation, this had only been accomplished in simulation.

In addition to DCBT, we have implemented a multimedia-favored scheduling algorithm called Cut-In Packet Scheduling (ChIPS), which is a means to improve the transport of multimedia traffic [CC00b]. ChIPS was designed to be an alternative to the FIFO packet scheduling algorithm, and can be implemented underneath DCBT and as well as any other RED-based AQM schemes. ChIPS functions by monitoring the queue rates of flow-controlled UDP flows, and is called upon whenever these flows are using less bandwidth than the TCP flows. Whenever line congestion is such that queue length of the UDP flows is greater than the average queue length, ChIPS will reward this flow-controlled UDP traffic by allowing their packets to cut ahead of other packets in the

queue, up to the average queue length.

We have implemented both DCBT and ChIPS as modules in the Linux kernel in order to gauge effectiveness and fairness in dividing bandwidth in a non-simulated environment. We created a test bed network and measured bandwidth, jitter, CPU usage, and other statistics used by various TCP and UDP connections across our test router for both DCBT and ChIPS. We found that DCBT is a fair queuing method and can accommodate changes in network traffic easily. We also found that ChIPS is an effective method for allowing a decrease in round trip time for multimedia packets without hurting the overall bandwidth of the class structure.

In Chapter 2, Related Work, we introduce related queuing mechanisms and detail their strengths and weaknesses. Chapter 3, Approach, highlights our methods for implementing CBT, DCBT, and ChIPS in the Linux kernel. Chapter 4, Experiments, details the process we used for evaluating the various queuing methods, including the layout of our network and descriptions of our tools. We present the findings of these tests in Chapter 5, Results and Analysis. In Chapter 6, Conclusions, we draw conclusions regarding the effectiveness of our work based on the results and analysis of the experiments completed. Finally, Chapter 7, Future Work, lists what we believe should be done to expand our work on DCBT and ChIPS.

Chapter 2: Related Work

Before any implementation of a router queue management schemes can be completed, a study of other relevant router algorithms should be pursued. The first part of this chapter details probability-based queues, and demonstrates the original need for them and the drawbacks that the DCBT algorithm aims to correct. The second section explains some of the class-based approaches to solving the different needs flows have when traveling through a router's queue, including the approach used by DCBT. Finally, we conclude this chapter explaining ChIPS and ABE and how they might provide better total service rather than best-effort FIFO queues.

2.1 Probability Based Queues

The first large improvement in Active Queue Management (AQM) was the development of Random Early Detection in 1993 by Sally Floyd [FJ93]. This new algorithm, termed RED, was designed to ease router congestion. The algorithm itself was simple enough so that it became the first widely accepted alternative to a standard FIFO queue. The premise behind the implementation of RED was that as network congestion grew inside the router, the probability of its queue filling up increased. In order to prevent this from happening, it was proposed that as the queue grew in size, the probability for dropping a new packet from the queue should increase, instead of waiting for the queue to become full.

The RED algorithm makes use of one variable and two fixed values: average queue size, maximum queue threshold, and minimum queue threshold. The average queue size for

each new packet is computed. If the average queue size is over the set maximum threshold, then the packet is 'marked'. No marking occurs to a packet if the average queue size is currently below the minimum queue threshold. However, if the average queue size falls between the maximum queue threshold and minimum queue threshold, then the packet is marked according to a probability determined by the average queue size. By using the average size of the queue as part of calculating the probability to mark a packet, RED is also able to handle transient congestion, where the queue quickly fills for a short period of time due to bursty traffic, dropping more packets than necessary.

While RED does encourage responsive flows to back off, unresponsive flows will still continue to send as much information as before. While RED's algorithm more and more aggressively attempts to drop packets to encourage flows to back off, the queue can actually be overflowed with unresponsive flows that transmit at a rate higher than capacity.

Another active queue management algorithm based on probabilistically marking packets is BLUE [FKSS99]. BLUE's algorithm to determine drop probability is much simpler than that of RED. Instead of setting threshold limits and testing the average queue length against them, probability is incremented if the router's queue continuously overflows, and decrements if the router queue is empty or stays idle. The probability is updated by a fixed time interval referred to as a "freeze time." In preliminary tests, BLUE has shown to provide approximately the same level as service as a well-configured RED router, but with much less sensitivity to bursty traffic as it more gradually increases or decreases its

probability of marking packets compared to RED.

2.2 Class-Based Scheduling

The class based approaches to queue management attempt to take a different approach than the queue modifications we have discussed thus far. Schemes such as RED and BLUE attempt to create fairness on one given link for all traffic at any given time. Examining the diversity of traffic on a given network would make it apparent that such a method is not the best possible solution. These types of queue management schemes have a tendency to punish the wrong flows by punishing everyone equally. While, under certain circumstances, this would be a desirable effect, the responses of some flows of traffic are much more detrimental to its overall bandwidth than the responses of others. [LM97].

Class Based Queuing (CBQ) [FJ95] attempts to divide all flows into the type of service they provide to keep link usage fair. A link's bandwidth is divided among the possible types of service that a link will carry and also according to who will be using these services. The division of bandwidth is determined by how much an individual or business needs for their share, hence one agency's share might be a higher percentage of the link than another's. Each of those shares could then be divided again on a per-service basis inside each agency. If a class' bandwidth usage is above its specified limit, then that class and all classes descended from it in the hierarchy are subject to attenuation by the link scheduler.

The problem with such a management scheme is that the overhead on flow management for Class Based Queuing is quite high. This can introduce higher latencies to the link, which affects the router's performance. Class Based Threshold (CBT) [CC00a] attempts to address this shortcoming by making much simpler assumptions and classifications on the flows coming through the link. All CBT traffic is divided into three main categories: TCP, tagged (or multimedia) UDP, and untagged (other) UDP. Identification of each class has a very low overhead compared to CBQ, as no identification of agency or type of data in each flow is needed.

CBT separates all traffic based on class and treats each class as a completely separate entity. It sees that TCP traffic is responsive to congestion and that UDP is not. As such, each flow is handled differently before being placed into a standard RED queue. When a TCP packet arrives, it is not checked against the early drop test and is instead allowed to pass into the RED queue normally. UDP, on the other hand, is checked for multimedia tags and then given to the early drop test. The test used is set at a fixed rate, and if that class threshold is crossed, then the packet is dropped. If tagged UDP enters the router and the queue size is greater than the average queue size according to the RED calculations, then the packet is dropped. Untagged UDP is treated in the same way, with the exception being that it is compared to the RED minimum.

We have implemented DCBT [CC00a], which differs from standard CBT by the addition of dynamic thresholds on a per-class basis. This requires the counting and identification of flows in each of the three classes of traffic so that the thresholds can be adjusted

according to current traffic conditions. Flows are identified by using a combination of the source and destination addresses, the Type of Service (ToS) field in the IP header, and a timeout according to a user definable setting. The reasoning behind this method is that while slow-sending flows (like telnet sessions) will actually be broken up into a number of flows as far as the router is concerned, the low bandwidth and indeterminate (and probable lack of interest concerning) latency of the actual flow is negligible compared to the traffic on the router as a whole.

Another flow-based queuing scheme is Stochastic Fairness Queuing (SFQ) [M91], which tries to be a fair method of queuing insofar as it does so without requiring the excessive overhead of strict fair queuing. Keeping all the flows separate and treated fairly is an expensive operation, and thus would not be good for implementation in a high traffic router. SFQ attempts to create a low overhead method of pseudo-fair queuing based on hash tables. SFQ separates the flows based on a hash key generated from the source and destination of the packets in question. Using these two addresses, it makes it very quick to reference any given flow for fairness checking and queue manipulation. However, it sometimes turns out that two or more flows could end up having the same hash key, which would mean that those two flows have to split a normally fair share of bandwidth.

To combat this problem, the kernel perturbs the hash key generation in such a way that it is highly unlikely that these two pairs will be placed in the same flow again. The kernel does this periodically in an attempt at keeping the flows' bandwidth fair and even. Still, even with such protection from the kernel, SFQ's fairness isn't quite as fair as it could be,

and so is not ideal for a very high load router.

2.3 Packet Requeuing

Cut-In Packet Scheduling (ChIPS) [CC00b] is a means to improve the delivery of multimedia connections. Its goal is to reduce multimedia jitter, or the lack of continuity among successive connections in multimedia applications over a network, due to packet loss. ChIPS is designed to be an alternative to the FIFO packet scheduling algorithm, and can be implemented underneath any other RED-based AQM schemes, CBT, or DCBT. ChIPS functions by monitoring the queue rates of tagged UDP flows, and is called upon whenever these flows are using less bandwidth than the TCP flows. Whenever link congestion is such that the queue length of the UDP flows is greater than the average queue length, ChIPS will reward UDP traffic by allowing their packets to cut ahead of other packets in the queue up to the average queue length.

By inserting these tagged UDP packets at the average queue length into a congested line, ChIPS improves flow-controlled multimedia jitter by ensuring that more packets make it to their destination in enough time to remove jitter without giving unfair bandwidth to the present congestion. However, it is possible that in the process, this insertion could do damage to the TCP flows, or even make them fail if a large enough delay was introduced due to the multimedia traffic taking up a large amount of the available bandwidth and because of an increase in out of order packet delivery.

Some algorithms, such as the RED queue mechanism discussed earlier, have no means to

monitor and control the queue buffer usage in the router among the different types of flows, and because of that, it is important for ChIPS to monitor the average queue rate itself and to turn on its functionality only when this ratio is minimal. The case is different under CBT, however, because the UDP threshold can be manually set to use only a small portion of the available queue. This behavior of CBT makes ChIPS' automatic control feature unnecessary. Similarly, when ChIPS is used in conjunction with the DCBT scheduling algorithm, the self-regulating nature of DCBT allows the ratio that turns off ChIPS' functionality to be set much higher without hurting the fairness of the flows.

Alternative Best-Effort (ABE) [HKLT01] is an enhancement that is built on top of the IP best-effort service. It is designed on the basis that providing low-delay at the expense of potentially less throughput will yield a positive outcome in a congested link. The driving ideas behind ABE are: First, there are many interactive multimedia applications at present which will perform well despite a wide range of loss and bandwidth conditions, but for which delay, which causes jitter, remains the major problem. Second, unlike differentiated services, the ABE service is designed to not require the active tracking of how much traffic is using the low delay capability contained therein.

The characteristics and requirements of ABE are defined as follows:

I. ABE packets are marked as either green or blue.

II. Green packets receive a low, bounded delay at every hop.

III. Green does not hurt blue. If a source decides to mark some of its packets green rather than blue, then the delay and throughput received by sources that mark all their

packets blue remains the same, or becomes better.

IV. All ABE packets belong to a single best-effort class. If the total load is high, then every source will receive little throughput. However, blue sources will experience more throughput than green sources sharing the same network resources. [HKLT01]

These requirements are built on the design decision that green packets are more likely to be dropped during periods of congestion than blue packets. Simply, ABE can be thought of as allowing an application to make the tradeoff of delay for loss, or less throughput. It accomplishes this by marking some packets green. The third requirement, green does not hurt blue, comes from the goal that the color chosen by an application does not need to be actively monitored. When the third requirement is enforced, then an application which decides to mark some packets green must do so because it values the low delay more so than a potential increase in loss or a decrease in throughput. Otherwise, it would mark its packets blue. In all cases, there is no penalty for other applications that could choose to mark all their packets blue.

ABE supports traffic that might be solely TCP-friendly traffic, non-TCP-friendly, or a mixture of the two. The downside to the ABE service is that applications which wish to make use of its advantages must be rewritten to do so, and all the work cannot be accomplished at the router level, like the scheduling algorithms discussed earlier. Since an IP header mark would be required, ABE would also require the modification of the operating system.

Chapter 3: Approach

We implemented CBT, DCBT, which differs from standard CBT by the addition of dynamic thresholds on a per-class basis, and ChIPS on top of DCBT. We implemented the following as loadable kernel modules for use with the Linux kernel. This benefited us by using open source kernel to expand upon the included RED algorithm to fit the needs of the CBT algorithms.

3.1 CBT Implementation

In order to accurately compare the differences between CBT and DCBT we chose to implement the CBT algorithm as well as DCBT. This part of the implementation was primarily focused on calculating the per-class averages. The averages were then compared against fixed thresholds to determine whether the next incoming packet should be dropped.

CBT culls any packets that fall outside of the thresholds set for that class. Only the UDP packet thresholds are subject to this, however, because of the self-regulating properties of TCP. The thresholds for all UDP traffic are used when the RED average queue size is greater than the RED minimum threshold value. The suggested threshold for multimedia-based packets is set at 10 packets. Unresponsive UDP packets have a different threshold value set at 2 packets [CC00a]. If the length of the RED average queue exceeds the RED maximum threshold value, then the router will drop the packet, regardless of the type of traffic used by the packet.

After the thresholds for each of the queues has been determined, the queue behaves as a standard RED queue. As such, the incoming packets that survive the CBT threshold test are subject to RED rules and a packet gets dropped according to the random drop rate. If a packet survives all drop tests, then the total count for packets is incremented, as well as the count of packets for that traffic class. All surviving packets then continue to be dequeued as normal.

3.2 DCBT Implementation

Another implemented queuing algorithm we implemented is DCBT. DCBT differs from CBT in that it keeps track of how many flows of traffic are passing through the router, and what type of traffic they are. This information proves useful in allowing CBT thresholds to dynamically change according to type and quantity of traffic in use. In order to achieve this functionality, a flow counter/classifier, new threshold test, and memory collection routine was implemented.

Flows are identified using a combination of the source addresses, destination addresses, and the protocol field in the IP header. By using simple hash key methods the classifier is able to give each flow a unique key to identify it from others. However, the ability to differentiate between flows coming from the same computer of the same protocol proved beyond the scope of this project, but did not cause difficulty in attaining accurate results. Multimedia traffic, or congestion sensitive UDP, is identified as all UDP traffic originating from the IP address of one of the test machines. The hash keys for each flow are contained in a linked list, the structure of which also contains an expiration time for

the flow (determined by the current time plus a tunable option in the code expressed in centiseconds), the class of traffic, and an "inactive" flag that signifies that the flow had expired. A new flow is identified as one that was not already in the linked list without the inactive flag set. When such a flow is discovered, the count of the total number flows incremented by one, as well as the count of the flows of that class of traffic. Since every flow adds another element to the list of flows, as the number of flows increases, the amount of memory required increases linearly with it.

After the flows have been classified and counted, a similar threshold test to CBT takes place. The threshold for multimedia-based packets is set at the RED average queue size multiplied by the multimedia packets share of the bandwidth. Unresponsive UDP packets have a different threshold value set at a small amount of the impending state queue buffers [C00a].

```
MM UDP        = (mmflows/totalflows ) * RED_avg
Other UDP     = (udpflows/totalflows) * [RED_min - (RED_max - RED_min)*0.1]
```

The old flow collector, or memory collector, runs at a preset interval during normal operation. This interval is configurable in the code, is expressed in centiseconds, and is activated using standard Linux timers. When the collector is activated the current list of flows is traversed and any flows that have passed their expire time have their inactive flag set, and the count of total flows decremented as well as the count of flows of that flow's traffic class. The inactive entry in the list is then available for reuse by new incoming flows. Old flow memory is not deallocated back to the kernel, as it would be more expensive to deallocate memory only to reallocate it in such brief periods of time, causing the kernel to do more work through page faults and interrupts only to achieve a

similar result.

3.3 Implementation of ChIPS

The last queuing mechanism implemented was ChIPS, or Cut-In Packet Scheduling. This algorithm was designed with the intent to lower latency outliers involved in delivery of congestion sensitive multimedia traffic. This functionality is gained by moving some multimedia packets ahead in the queue to the queue average during times when the queue average is greater than the RED queue minimum, and the total number of flows twice or more than that of multimedia flows.

The original implementation of ChIPS, which was implemented in the network simulator, called for the inclusion of a "virtual queue" which was an almost exact copy of the original queue. This second queue contained all the packets received in order, as would have happened in a normal FIFO queue. This second queue was the one that was used in all the threshold calculations, in order to prevent unfair bandwidth allocation from TCP flows due to a faster drain rate of multimedia packets.

We opted to discard this queue. The virtual queue seems to offer negligible advantages compared to the overhead involved in keeping track of every packet over again. We also encountered problems in maintaining packet order. This was due to some packets being queued ahead of an earlier queued packet, thus skewing jitter results that were dependent on time of departure and time of arrival. This packet order problem needed its own queuing mechanism, and instead of using kernel resources to keep a virtual queue we

decided to implement this new queue instead.

This queue is an "in-order" multimedia queue. As multimedia packets arrive in the router, they are first subjected to DCBT culling tests, and then a copy is placed in this in-order queue in FIFO fashion. ChIPS can then move the original, copied packet into RED's queue average pointer if necessary. When any multimedia packet is to be dequeued, a packet from the in-order queue is dequeued in its place and the other packet is discarded, keeping all the packet deliveries in order.

The benefit of this system is that multimedia packets can be inserted anywhere into the queue, and it will not matter what order in which they arrive at the head of the queue, since the proper packet in the multimedia order will be dequeued in its place. This halted all the out of order packet arrivals, as expected.

Chapter 4: Experiments

Our network test bed was built to evaluate our implementations of CBT, DCBT, DCBT/ChIPS, and other queue management schemes on a router using the Linux kernel. We performed a variety of tests using network tools to test the efficiency of these implementations in our Linux router.

4.1 Setup

This section outlines the setup of our network test bed. Points that were taken into account included the physical layout of the network as well as the computer setup of the router and the clients that would be accessing it.

4.1.1 Physical Layout

In order to efficiently test our router, it needed to be in a position to have at least two lines connected to the router for network traffic. On each side of the router, we required a number of client machines either generating or receiving traffic, thereby testing our implementation and its effectiveness both in terms of processor usage on the router and in throughput on the clients' sides. Placing only a single client on each side of the router would have been ineffective for this goal, as there would be no congestion present between the client machines competing for bandwidth through the router. Therefore, our design included at least one client computer on a given side of the router, and two on the opposite side generating traffic that needed to compete for resources when reaching the far client. To prevent traffic collisions on the side of the two client computers, we attached each computer to the router with its own ethernet interface. A detailed diagram

of the network setup is as follows:

4.1.2 Computer Setup

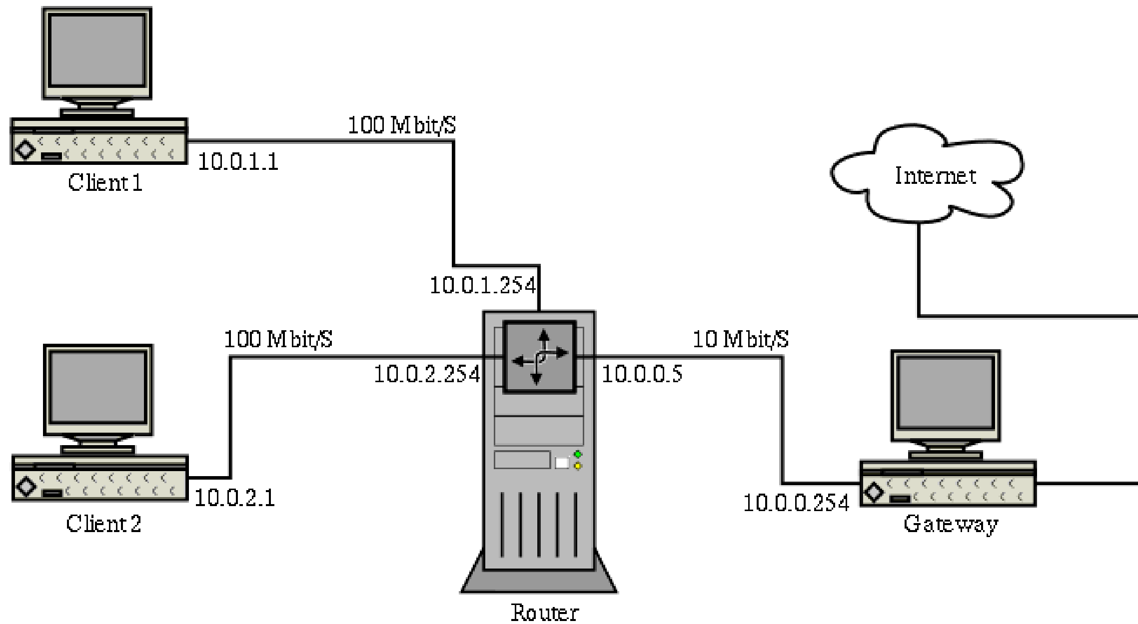

Fig 4.1 – Network Test Bed Diagram

Our Linux router was a dual Pentium II 300 MHz computer with 128 MB memory running Mandrake Linux 8.1 and using version 2.4.13 of the Linux kernel with New Reno TCP. For more accurate CPU measurements without the need to take multiple processors into account, the router kernel was compiled to take advantage of only a single processor. This machine used the advanced router configuration available in the 2.4 Linux kernel using the iproute2 set of tools to route traffic between the three subnets using the Traffic Control application. Iproute2 is a group of configuration utilities that takes advantage of features in the redesigned network subsystem in version 2.2 and later Linux kernels, including new routing, filtering, and classifying code. Our router featured

two PCI 100BaseT ethernet cards, one in the 10.0.1.0/24 IP range connected to client1 and one in the 10.0.2.0/24 range connected to client2. A third ethernet card was a 10BaseT card, located in the 10.0.0.0/24 range, connected on the opposite side of the diagram to the gateway computer.

The client computers were also a vital part of our network test bed. Each of these client computers was a Pentium 133 mhz computer with 48 MB of memory and a PCI network card running FreeBSD 4.4, using New Reno TCP. While not the most powerful machines to date, they were more than sufficient to saturate a 100 Mbps connection without taxing the processor. All of our tests showed that processor usage on the client machines never exceeded 50% of the 133 mhz processor, ensuring that these computers would not be a bottleneck in our experiments.

The stand-alone client machine served as a firewall to connect our network test bed to the Internet. This was a FreeBSD machine running Network Address Translation (NAT) to forward network traffic from the test bed to the local WPI network as necessary for system configuration from outside the test bed. This connection was also necessary in order to create more flows of traffic for testing by sending data to other computers on campus. This computer featured two 10BaseT ethernet cards, one connected to the Linux router, and the other connected to WPI's network.

On the far side of the router, two client computers ran FreeBSD and served only as traffic generators to send data to the lone client computer on the other side of the router. Each

of these clients was connected to the Linux router's 100BaseT ethernet cards with their own 100BaseT ethernet cards. This allowed two high-speed 100 mbps connections between the clients on one side of the router without the need to worry about traffic collisions, and a smaller 10 Mbps connection on the far side of the router between the router and the gateway.

4.2 Testing

Using freely available network monitoring and testing tools we were able to evaluate CBT, DCBT, and DCBT/ChIPS's efficiency as a router in controlled environment. The most valid and important aspect of this testing was to compare the CBT, DCBT, and DCBT/ChIPS implementations against other queue management schemes already implemented in the Linux kernel, including FIFO and RED. We attempted to discover whether the DCBT scheme is a more effective algorithm compared to the others, in terms of fairness in queuing with reasonable cost in overhead. With the DCBT/ChIPS combination we find whether this scheme can lessen multimedia jitter with a small cost of delay to other traffic.

4.2.1 Tools

Three tools were used in experiments for traffic generation and measurement. The first of these was a traffic generation tool called Iperf[1]. Iperf was developed as a modern network tool for measuring TCP and UDP bandwidth performance. Iperf was able to measure maximum TCP and UDP bandwidth and packet loss, and allowed the tuning of

---

[1]http://dast.nlanr.net/Projects/Iperf/

various parameters as well as UDP characteristics. Iperf ran as a client and server for both TCP and UDP traffic generation, and reported bandwidth, delay jitter, and datagram loss.

The second performance testing tool we used to evaluate our implementations was Iostat, a tool included in the Sysstat[2] package. The Sysstat utilities are a collection of performance monitoring tools for Linux. The Iostat command is used for monitoring system input/output device loading, and was used throughout out tests for processor usage reporting and benchmarking.

The final testing tool we used was NcFTP[3], a command line FTP client. This tool was used to generate TCP traffic via FTP to other computers on-campus, allowing us to increase the number of flows passing through our router. By sending data only to on-campus computer, we were able to maintain a small round trip time, on the order of 2 ms. This was important, as it allowed us to conduct tests on a larger group of computers, yet still operate under similar conditions as our network test bed, and therefore not introduce extraneous variables that might have made it difficult to compare our results with those of our network test bed.

4.2.2 Tests

We used three types of tests in evaluating our queue management schemes in our network

---

[2]http://perso.wanadoo.fr/sebastien.godard/

[3]http://www.ncftp.com

test bed: throughput, processor usage (with two- and many-flow tests), and one-way latency. For each test, our packet size was fixed at 1000 bytes. For each queuing mechanism tested, the minimum queue size was set to 5000 bytes, the maximum queue size was 15,000 bytes, the queue size limit was 20,000 bytes. For RED testing, the maximum drop probability was set to 0.01.

The throughput tests consisted of starting a 30-second TCP stream of traffic from the client1 machine to the gateway, followed by starting another 30-second stream of 10 mbps UDP traffic from the client2 machine to the gateway 10 seconds later, allowing for overlapping areas of TCP and UDP, with each type of traffic alone on each end of the test. This test was carried out for a variety of queuing disciplines to measure bandwidth throughput, packet loss, and their efficiencies using Iperf on each client's side.

The processor usage tests consisted of a two-flow test as described above for testing the throughput, as well as a many-flow test consisting of running many flows from client1 and client2 at the same time through the router for an extended period of time. To generate a greater number of flows, data was sent not only to the gateway computer, but also via FTP to computers on-campus. This allowed us to add as many flows as we had computers to send data to. Iostat was used to measure the system CPU usage during these tests on the router-side at 1-second intervals.

Finally, the latency tests consisted of a five-flow test, including one 1 mbps flow of multimedia traffic from client1 to the gateway, as well as 2 TCP flows from each of

client1 and client2 to the gateway and to on-campus computers. Iperf was used to measure the jitter of these tests. To do so, the jitter calculations were continuously computed by the server, as specified by RTP in RFC 1889. The client recorded a 64-bit second/microsecond timestamp in the packet, and the server then computed the relative transit time as the server's receive time minus the client's send time. Jitter was displayed as the mean of differences between consecutive transit times.

Chapter 5: Results and Analysis

After the network test bed and router implementations were completed, we began our testing procedure. First we compared the throughput of different flows of traffic between FIFO, RED, CBT, and DCBT. Then we compared the difference in jitter between DCBT and DCBT with ChIPS to find what difference ChIPS offered multimedia traffic. Following that, the costs of CPU utilization and memory usage were measured to gauge the benefits versus the costs of DCBT as a packet queuing mechanism.

5.1. Throughput

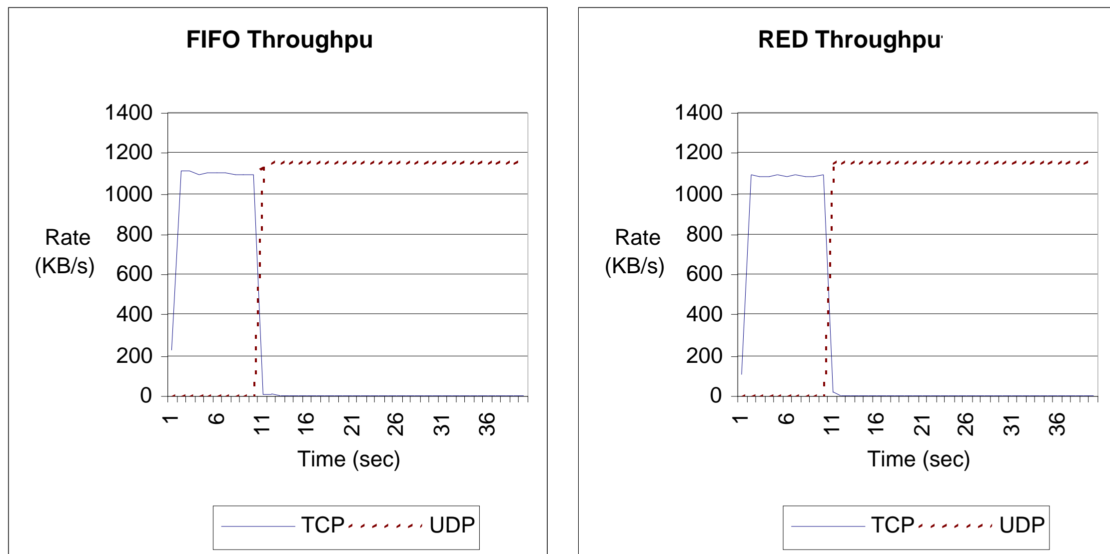

Fig 5.1 - FIFO and RED Throughput

For the throughput tests, we started a TCP flow, shown by the solid line, and let it run for 10 seconds. After that, we introduced a 10 Mbps UDP flow with a 1 KB packet size, shown by the dashed line, and observed the bandwidth of the TCP flow. Since TCP is responsive to congestion and UDP is not, TCP's bandwidth will normally be lost to UDP.

When the tests were run with the standard FIFO drop-tail queue (Fig 5.1, left), we see the results of UDP taking all the bandwidth away from the congestion-responsive TCP flow. The same results occur with a RED queue (Fig 5.1, right). These results are expected since both RED and FIFO queues are ineffective at queuing fairly when there are unresponsive, high bandwidth flows.

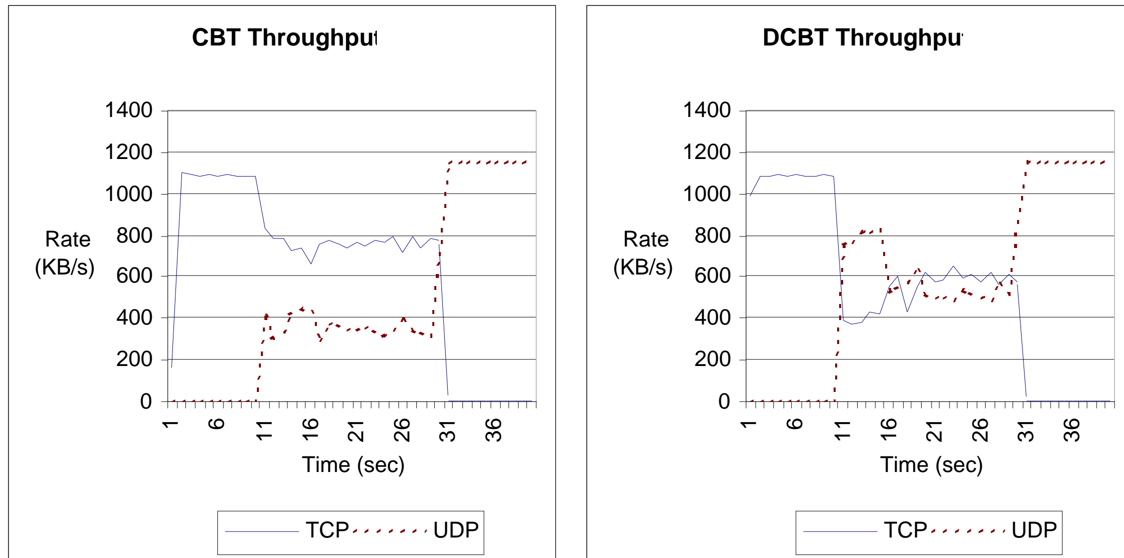

Fig. 5.2 - CBT and DCBT Throughput

Both CBT and DCBT show much more fair throughput results than the RED and FIFO queues. CBT's only failure is that it cannot adapt to conditions the way DCBT can. Even if CBT's parameters were changed to make the results in this graph fairer, there would then be other situations where CBT could not allocate bandwidth fairly. There is a small discrepancy in the above DCBT throughput graph between seconds 12 and 17. UDP takes a clear lead in throughput. Originally, this was attributed to transient flows in the Iperf measurement system. However, after more study it became apparent that this was due to TCP's congestion sensitivity mechanism for low bandwidth situations, where the window

size, or the amount of data sent at one time, collapses to a base rate and then grows back to the maximum size that is supported by the network. An additional experiment was run featuring with a UDP flow first, followed by a TCP flow, illustrating such behavior (Fig 5.3):

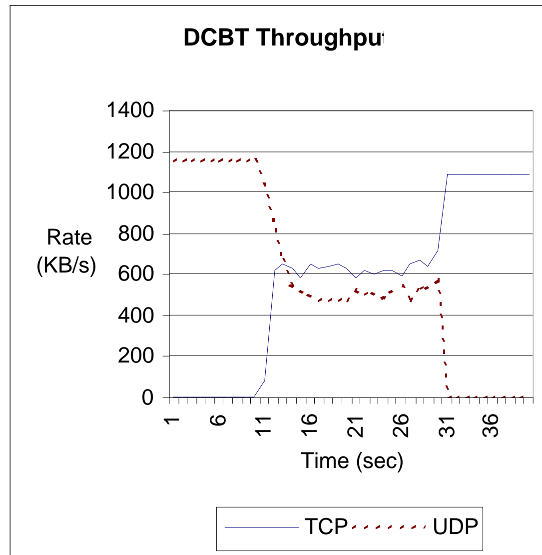

Fig 5.3 - DCBT Throughput tested with UDP first

Since TCP's window size grows until its fair share it does not show an erratic loss in bandwidth when it is introduced to the system nor does it show a lag time in achieving a fair share of bandwidth.

While both DCBT and CBT have shown themselves to be more fair than FIFO and RED in situations where an unresponsive UDP flow is present, we can see from these graphs that DCBT is better at adapting to situations with different traffic mixtures.

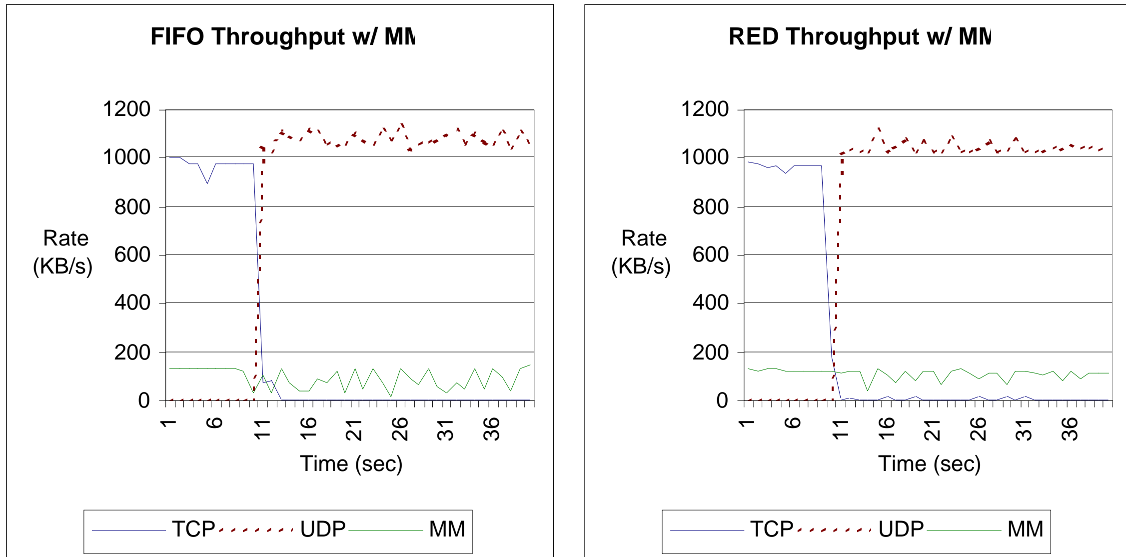

Fig 5.4 - FIFO and RED Throughput with Multimedia

Figure 5.4 shows RED and FIFO in the same testing scheme as before, but with a 1 Mbps multimedia UDP flow added, shown by the green solid line, throughout the course of the test. This flow featured as 1 KB packet size. As shown above, the queues do nothing to protect the multimedia flow. Its bandwidth is very inconsistent, and also reduces TCP's available bandwidth further than with the UDP flow alone.

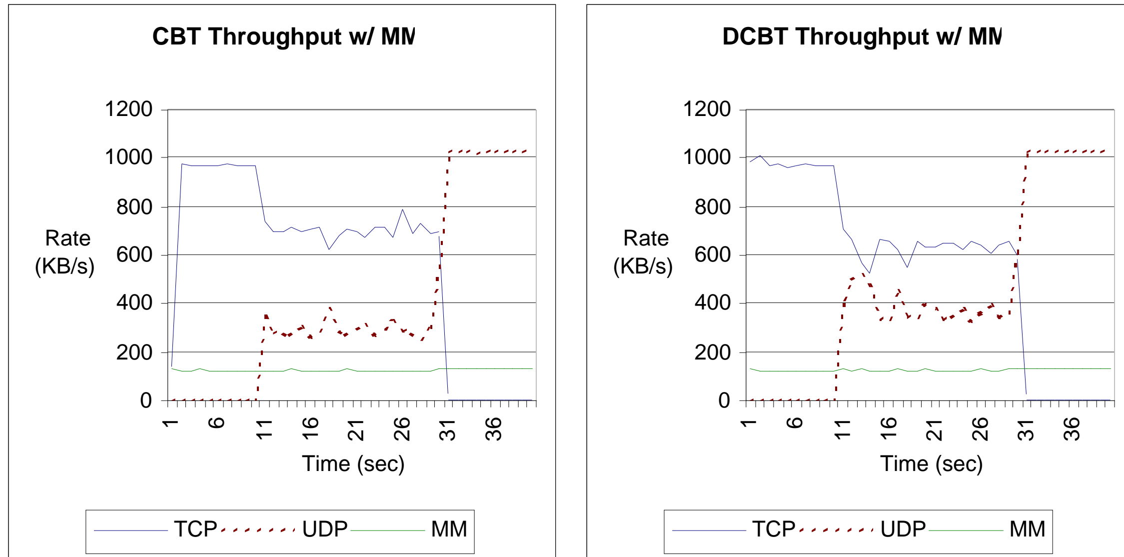

Fig 5.5 - CBT and DCBT Throughput with Multimedia

On the other hand, CBT and DCBT provide a much more consistent throughput for the MM flow. Figure 5.5 looks similar to the graphs without a multimedia flow, but show that the MM flow is getting a fair share of the bandwidth, with the unresponsive UDP flow losing bandwidth to accommodate the new flow. The multimedia flow itself is stable for the length of its run, due to the classification methods.

The results of all of our throughput tests' averages are summarized in the Table 5.1 and 5.2:

Test	**TCP alone**	**TCP while UDP present**	**UDP while TCP present**	**UDP alone**	**Drop Rate for one TCP flow**
FIFO (2 flows)	1110 KB/s	14 KB/s	1138 KB/s	1152 KB/s	No Results
RED (2 flows)	1089 KB/s	16 KB/s	1136 KB/s	1152 KB/s	0.0084
CBT (2 flows)	1090 KB/s	760 KB/s	378 KB/s	1153 KB/s	No Results
DCBT (2 flows)	1089 KB/s	632 KB/s	536 KB/s	1153 KB/s	0.0079

Table 5.1 – Throughput of 2-flow Tests

Test	TCP while MM present	TCP while UDP + MM present	UDP while TCP + MM present	UDP while MM present	MM overall
FIFO (3 flows)	978 KB/s	16 KB/s	1093 KB/s	1069 KB/s	83 KB/s
RED (3 flows)	967 KB/s	16 KB/s	1044 KB/s	1040 KB/s	108 KB/s
CBT (3 flows)	972 KB/s	704 KB/s	309 KB/s	1025 KB/s	126 KB/s
DCBT (3 flows)	976 KB/s	630.3 KB/s	388 KB/s	1025 KB/s	127 KB/s

Table 5.2 – Throughput of 3-flow Tests

5.2. Jitter

ChIPS was designed as a method for multimedia packets to avoid latency outliers without adversely affecting the bandwidth of other flows. We used Iperf's time delay measurement capabilities to test the jitter delay on the packets sent. We ran tests consisting of 3 simultaneous streams, two TCP and one 1 Mbps MM stream. Jitter is shown in Figure 5.6, with DCBT represented by the solid line and DCBT/ChIPS by the dashed line. The results show that ChIPS provides a means to decrease jitter noticeably over DCBT, and guarantee a ceiling rate of packet delivery during most network conditions. Our test bed featured such a small round trip time as to not incur a noticeable penalty latency-wise, however we are unable to test whether ChIPS would produce lower jitter over larger networks.

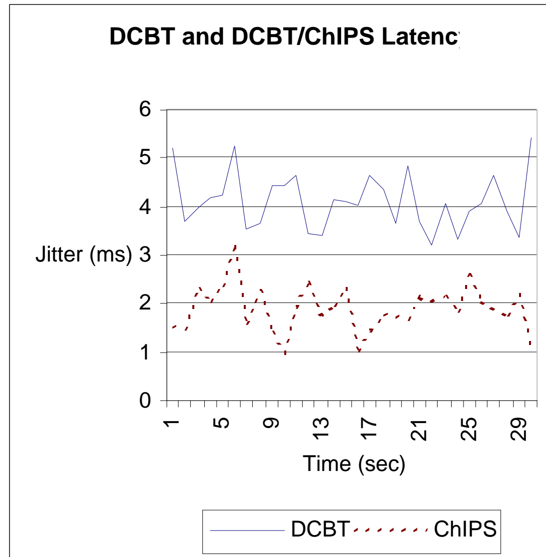

Fig 5.6 - Jitter over time for DCBT and DCBT/ChIPS

5.3. CPU Utilization

One of the concerns in the implementation of DCBT and CBT was the amount of extra load it would place on the CPU. If there was much more than FIFO, than it would be much harder for people to justify using it since they would need more expensive hardware to accommodate the algorithm. The CPU utilization for each of the queuing disciplines tested is shown in Figure 5.7, including standard deviation for each processing usage test:

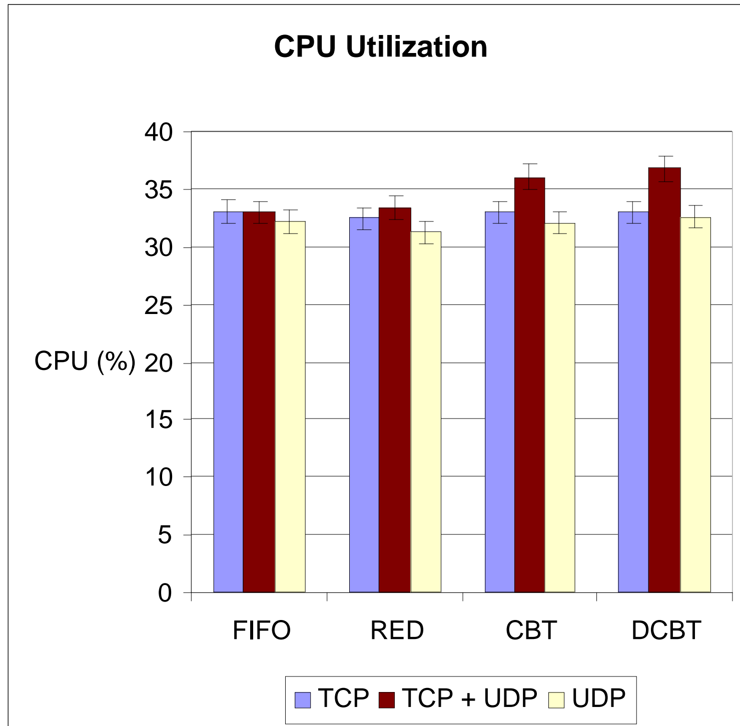

Fig 5.7 - CPU Utilization for all tested queuing disciplines for TCP, UDP, and TCP/UDP together

However, the percentage of the CPU used by CBT and DCBT is not much more than RED and FIFO. In fact, for a TCP-only router, shown by the leftmost bar in each set, the CPU usage is nearly equal, and for UDP-only, shown by rightmost, the utilization is slightly less. When the types of traffic are mixed, however, the utilization goes up appreciably, but is not significant compared to a possible utilization of 100%.

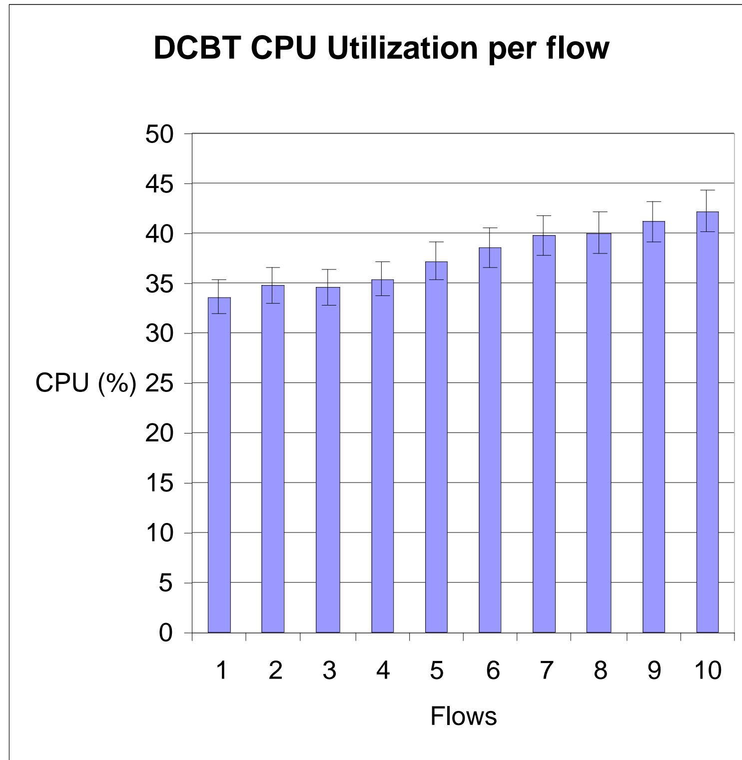

Fig 5.8 - CPU Utilization of DCBT per flow

More importantly is the concern for how CPU utilization scales with the number of flows. When testing an increasing number of flows across the router, the increase in processor usage for flows for DCBT is not significant. Figure 5.8 shows a linear growth, taking into account standard deviation of the average CPU usage. We predict that the 300MHz Pentium II that we used for testing could handle up to 70 flows at once.

5.4 Memory Usage

Similarly to each flow adding an incremental, but noticeable, amount of overhead on the CPU, each flow also uses more of the kernel's available memory. Calculating the memory used in data structures for DCBT yielded a total of 17 bytes per flow. On top of that, a flat overhead of 52 bytes is required over that of RED for the whole DCBT

queuing mechanism. At a maximum of 70 flows, this comes out to be 1242 bytes, or approximately 1.2 kilobytes.

However, memory usage is much more intense with the current implementation of ChIPS, primarily due to the way the in-order queue operates. Since every packet is copied, instead of referenced, to prevent referencing deallocated memory, a page fault occurs every time a multimedia packet is queued. More memory is needed as the amount of multimedia traffic increases. This adds both noticeable memory usage and CPU usage. Due to time constraints of our experimentation period, we were able to conduct multiple throughput and latency experiments on our ChIPS implementation, but were unable to fully conduct CPU utilization experiments.

Chapter 6: Conclusions

The explosive growth of the Internet has created an opportunity for a wide range of applications to be quickly deployed over only a few years. The ever-increasing demand for bandwidth requires current traffic flows to be optimized as much as possible. While many algorithms and approaches exist at the application level in order to effectively send multimedia data, many of the bottlenecks involved in transmissions can be found not at the destination or the source, but in-between, with the routers and their queue management schemes.

This project implemented the Dynamic Class Based Threshold (DCBT) and Class Based Threshold (CBT) queue management algorithms, as well as a packet requeuing mechanism called Cut-In Packet Scheduling (ChIPS). These algorithms were then compared to other popular queue management schemes. To achieve this goal, code was developed as an extension to the Linux kernel. Extensive testing of DCBT, ChIPS, and other Linux queuing mechanisms took place in a controlled environment.

We have produced a number of results by testing DCBT, CBT, and ChIPS. Our implementation of DCBT worked correctly and was fair, while CBT also worked but was not as fair as DCBT. Furthermore, our testing of ChIPS demonstrated a successful decrease of jitter for latency-sensitive multimedia traffic.

6.1 DCBT

DCBT is a fair queuing mechanism due in part to its flow classification. Using the flows as a relative measure of network usage for each protocol, we were able to give each

classification of traffic a fair amount of bandwidth. Because it is able to change the thresholds for each class of traffic, DCBT can adjust to mixes in traffic as is necessary instead of relying on fixed thresholds that may not accurately represent the status of the router's traffic.

DCBT does not place an unreasonable amount of strain on a router in terms of processing power or memory. As mentioned in the previous chapter, using the 300MHz Pentium II found in the test bed environment, we would be able to route 70 flows of traffic simultaneously before hitting the hard limit of the CPU. The memory usage of DCBT is minimal; requiring only 1.2 kilobytes to support the 70 flows.

6.2 CBT

While CBT was constructed to use the same mechanism for dropping packets as DCBT, it lacks the ability to change its own parameters to meet the needs of the traffic at any given moment. Accordingly, while it was fairer than FIFO and RED queuing mechanisms, which lack the ability to distinguish traffic at all, it is not as fair to multiple traffic flows as DCBT. Also, for a limited number of flows there is little to no savings in CPU usage and memory for CBT on any modern processor over that of DCBT.

6.3 ChIPS

ChIPS, designed to improve packet latency by placing multimedia packets at the RED queue average, allows for smaller jitter for multimedia packets. Our tests demonstrated that ChIPS does offer this advantage over DCBT, and effectively guarantees a ceiling on

jitter under normal traffic conditions on a router. However, the current implementation of ChIPS does tax the router's memory and CPU resources by requiring double the amount of memory normally used to queue one multimedia packet, and so a more effective way to maintain packet order is necessary before a smaller delay in jitter will become apparent.

Chapter 7: Future Work

We were able to accomplish a great deal developing DCBT over the course of our project, but there is still room for additional work to be done. There are different directions that our work could be taken in terms of both implementation and experiments.

7.1 Implementation

The first area of future work that we propose would be a general code cleanup and integration. Our code for CBT, DCBT, and ChIPS currently has many similarities with RED, in that the use of the Traffic Control program uses the RED syntax for setting the proper queue values. Our kernel code is added to the RED configuration option in the kernel, and could be integrated as its own configuration option. Finally, future code cleanup would be necessary for future kernel use as the advanced router features change and the code must be update for new API changes. While we do not expect any changes to be necessary for future revisions of the 2.4 Linux kernel, release of new branches could require additional changes.

Another area of future work that we propose is DCBT's flow detection. Currently, the type of flow is detected by the source, destination, and protocol used. While this was very useful for testing the fairness of our code, it may not be practical for traffic on the Internet because a router would not know that all traffic from a single source should be classified as multimedia traffic, nor would this often be the case. Better flow detection of an existing protocol being used as multimedia, or the introduction of a new protocol specifically for multimedia applications, is a crucial piece that would need to be

implemented before wider adoption of this queuing mechanism could be accomplished.

A third aspect of future implementation work that could be pursued would be to make our CBT, DCBT, and ChIPS implementations work under IPv6. With the expanding size of the Internet, the eventual move to IP version 6 is inevitable, and our code will need to take that into account at that time. One advantage of IPv6 over the traditional IPv4 is an expanded packet header, which could be used to store addition information in each packet and possibly aid in flow counting, versus the IPv4 packet, which has insufficient room to store information that would be necessary for more advanced flow counting.

A final addition that we propose for future work is the addition of ChOPS to our queuing mechanisms. ChOPS, Cut-Out Packet Scheduling, functions similarly to ChIPS, but instead of moving packets forwards in the queue, it moves them backwards in the queue. The effect of this would be to give multimedia packets a definite scheduled time of arrival when coupled with ChIPS.

7.2 Experiments

One area open for more extensive experimentation is ChIPS. One method to gather more accurate results on the advantages of ChIPS that we propose is to conduct tests using a larger network that is not as conducive to high-speed traffic. Further testing could be accomplished using cascaded routers, and measuring the affects of ChIPS being used in multiple routers that traffic passes through successively.

Another area that could undergo further testing is the Symmetric Multi Processor (SMP) support for our implementations. We believe that all locking was sufficiently implemented in our code, but all of our testing was performed on a uniprocessor system such that obtaining results was more straightforward. We did not have time to perform extensive SMP testing on our work. It is possible that some cleanup would be required before this could be accomplished.

Chapter 8: References

[BC00] D Bovet, M Ceasti. *Understanding the Linux Kernel*. O'Reilly and Associates. 2000.

[CC00a] J Chung, M Claypool. "Dynamic-CBT - Better Performing Active Queue Management for Multimedia Networking." In *Proceedings of NOSSDAV*, June 2000, Chapel Hill, NC, USA.

[CC00b] J Chung, M Claypool. "Dynamic-CBT and ChIPS - Router Support for Improved Multimedia Performance on the Internet." In *Proceedings of ACM Multimedia*, Los Angeles, CA, USA, November 2000.

[FJ93] S Floyd, V Jacobson. "Random Early Detection Gateway for Congestion Avoidance." In *IEEE/ACM Transactions on Networking*, August 1993.

[FJ95] S Floyd, V Jacobson. "Link-Sharing and Resource Management Models for Packet Networks." In *IEEE/ACM Transactions on Networking*, August 1995.

[FKSS99] W Feng, D Kandlurz, D Sahaz, and K Shiny. "BLUE: A New Class of Active Queue Management Algorithms." U. Michigan CSE-TR-387-99, April 1999.

[HKLT01] P Hurley, M. Kara, J Le Boudec, P Thiran. "ABE: Providing a Low Delay within Best Effort." IEEE Network Magazine. May/June 2001.

[IMD01] G Iannaccone, M May, C Diot. "Aggregate Traffic Performance with Active Queue Management and Drop from Tail." Sprint ATL, Universita` di Pisa, Activia, Februrary, 2001.

[LW00] A Leon-Garcia, I Widjaja. *Communication Networks: Fundamental Concepts and Key Architectures*. McGraw Hill. 2000.

[LM97] D Lin, R Morris. "Dynamics of Random Early Detection." In *Proceedings of SIGCOMM*, 1997.

[M91] P McKenney. "Stochastic Fairness Queuing." Sequent Computer Systems, Inc. January, 1991.

[RC01] A Rubini, J Corbet. *Linux Device Drivers, 2nd Edition*. O'Reilly and Associates. 2001.