Class-Based Thresholds: Lightweight Active Router-Queue Management for Multimedia Networking

by

Mark Anthony Parris

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2001

Approved by:

____
Advisor: Kevin Jeffay

____
Reader: F. Donelson Smith

____
Reader: Bert Dempsey

____
Reader: David Stotts

____
Reader: Ketan Mayer-Patel

ABSTRACT

MARK PARRIS: Class-Based Thresholds: Lightweight Active Router-Queue Management for Multimedia Networking
(Under the direction of Kevin Jeffay.)

In the best-effort Internet, congestion can cause severe degradations in the performance of both reliable data transfer flows and multimedia flows. These reliable data transfers are typically based on TCP, a *responsive* protocol. Responsive protocols are those that respond to congestion by reducing the rate at which they transmit data. This behavior is desirable because when all traffic is responsive, the traffic sources cooperate to ameliorate congestion by arriving at an aggregate operating point where the generated load matches the available capacity. In contrast, multimedia applications are typically based on *unresponsive* protocols, such as UDP, where the transmission rate is determined by the application, independent of network conditions. There exists a fundamental tension between these responsive and unresponsive protocols. When both traffic types are present during periods of congestion, unresponsive traffic maintains its load, forcing responsive traffic to reduce its load. Consequently, unresponsive traffic may benefit from this behavior and consume more than its fair share of the available bandwidth while responsive flows receive less than their fair share and suffer poor throughput. In the extreme, congestion collapse is possible. This offers a disincentive for using responsive protocols.

Recent proposals have attempted to address this problem by isolating responsive traffic from the effects of unresponsive traffic. They achieve this goal by identifying and severely constraining all unresponsive traffic. We take the position that some unresponsive traffic, specifically multimedia, is necessarily unresponsive. Maintaining the fidelity of the media stream places bounds on minimum levels of throughput and maximum tolerable latency. Since responsive protocols focus on reducing the transmission rate to match the available capacity independent of application-level concerns, these bounds may be difficult or impossible to meet with responsive protocols. As such, we argue that in addition to isolating responsive traffic, multimedia should not be unduly constrained and must also be isolated from the effects of other unresponsive traffic. In this dissertation we propose and evaluate a novel algorithm to allocate network bandwidth by allocating buffer space in a router's queue. This algorithm is called Class-Based Thresholds (CBT). We define a set of traffic classes: responsive (i.e., TCP), multimedia, and *other*. For each class a threshold is specified that limits the average queue occupancy by that class. In CBT, when a packet arrives at the router it is classified into one of these classes. Next, the packet is enqueued or discarded depending on that class's recent average queue occupancy relative to the class's threshold value. The ratio between the thresholds determines the ratio between the bandwidth available to each class on the outbound link.

CBT and other router queue management algorithms from the literature (FIFO, RED, and FRED) are implemented in a FreeBSD router and evaluated in a laboratory network using several combinations of TCP, multimedia, and generic unresponsive traffic. We show that CBT can be tuned to offer prescribed levels of performance for each traffic class. We present analysis that predicts network performance when using CBT based on initial configuration parameters, explain how this analysis can be reversed to derive optimal parameter settings given desired network performance, and empirically demonstrate the accuracy of this analysis. We present experimental data that contributes to our understanding of how existing queue management schemes perform, articulate relationships between parameters and performance metrics, and determine optimal parameter settings for each algorithm under various traffic mixes. We empirically demonstrate that CBT effectively isolates TCP while providing better-than-best-effort service for multimedia by comparing CBT's performance to the optimal performance for other algorithms. Finally, we show CBT provides better protection for TCP than RED and FIFO and better multimedia performance than RED, FIFO, and FRED.

ACKNOWLEDGMENTS

Completing my dissertation would not have been possible without the support of my colleagues and friends. I gratefully thank:

<u>My committee:</u>
Kevin Jeffay for guidance, motivation, and support,
Don Smith for insightful advice and well-considered opinions as well as ifmon, thruput, and trouble-shooting old hardware, and
Bert Dempsey, Prasun Dewan, Ketan Meyer-Patel, and David Stotts for their feedback.

<u>My colleagues:</u>
Terry Talley, Steve Goddard, Peter Nee, and Don Stone for showing it can be done and breaking Kevin in,
David Ott for the HTTP model, maintaining the lab so I didn’t have to, and taking care of the finishing touches,
Michele Weigle for vidsim,
Mikkel Christiansen for conversations and comments,
Ramkumar Parameswaran for the MPEG tool,
Felix Hernandez for maintaining the DiRT lab and repeatedly delaying upgrades,
Jan Borgersen for the Proshare model,
K. Cho for ALTQ,
Marc Olano for putting together a dissertation style template that works, and
Wu-chi Feng for providing the MPEG frame traces.

<u>And my other friends:</u>
Betsy for understanding, encouragement, and having faith in me,
Dad for asking, “where are my pages?,”
Mom for her love,
Christine for getting tired of me being lazy,
Eileen for commiserating with me,
Stefan for showing the process is survivable,
Max and Cleo for having no idea what a dissertation is and not caring at all, and
Darkside, Pleiades, and ENB for making the past eleven years fun. I have no regrets.

This research was supported in part by funding and equipment donations from the National Science Foundation (grants IRIS 95-08514, CDA-9624662, and ITR 00-82870), the Intel Corporation, Sun Microsystems, Cabletron Systems, Cisco Systems, the IBM Corporation, and MCNC.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

At its most basic level the Internet provides a service to transfer units of data, *packets*, from a sender to a receiver. The service specification promises only a “best-effort” attempt to deliver packets and no guarantee is given. Loss of connectivity, incorrect routing tables, or periods of overload (congestion) can all result in packet loss. The most common cause of loss is congestion. As a result, many transport protocols, such as TCP, use packet loss to infer the presence of congestion and respond by reducing a connection's transmission rate to avoid overload. When all protocols have this agreed upon behavior, they arrive at an aggregate load that matches the capacity of the link. Other protocols, like UDP, either do not have a mechanism for detecting loss or do not respond to it as an indicator of congestion. Two forms of application-level traffic, reliable data transfer and interactive multimedia, rely on, respectively, TCP and UDP. Unfortunately, a fundamental tension exists between these two approaches. During periods of congestion, responsive protocols reduce their generated loads in response to loss while unresponsive protocols maintain their loads, consuming most of the capacity of the link. Consequently, the traffic using protocols that respond to congestion may suffer poor performance while the traffic using unresponsive protocols benefits from maintaining or increasing its load. Most of the work addressing this tension has focused on rewarding the responsive behavior and punishing the unresponsive behavior. We take the position that in some cases this unresponsive behavior is necessary to the successful behavior of the application (e.g. interactive multimedia) and not all unresponsive applications are problematic. In this dissertation we present a router queue management discipline, Class-Based Thresholds (CBT), that protects and encourages responsive behavior, discourages needlessly unresponsive behavior, and still allows unresponsive behavior when necessary.

1. Background

Before considering our router queue management algorithm, it is helpful to review the interaction between congestion, buffering in routers, and the traffic types that share a network. The Internet is a collection of small networks connected by store-and-forward devices called routers. To understand the router's purpose, recall that the Internet is a “network” in a graph-theoretic sense. Routers and end-stations are the nodes and the links that connect them are the graph's edges. Interior nodes with multiple edges are routers while leaf-nodes with single edges are end-systems. Each router has multiple bi-directional links that it could direct the packet along. Using information from a routing table, the router decides the next link the packet must cross to move the packet closer to its ultimate destination. Each of these links has a finite, fixed capacity and the amount of data that needs to be forwarded along a given link may, at times, exceed the capacity of the link. When this happens the link is overloaded and said to be *congested*. This congestion may be transient or persistent. It may be transient because packets happen to arrive in a bursty manner. That is, a large number of packets arrive nearly simultaneously, in a group, followed by a gap during which few packets arrive. Bursty traffic is a common phenomenon in the Internet [Leland94], [Willinger95]. In the transient case the congestion can be remedied by providing a queue of buffers in the router to allow packets for a given outbound link to be stored briefly before being forwarded. Alternatively, the congestion may be persistent because the offered load is consistently exceeding the capacity of the link. In the persistent case, the buffering would have to increase with the length of the congested period in order to avoid discarding, or dropping, packets because of queue overflow.

To better understand the relationship between queue-induced latency and traffic load, consider the concept of *traffic intensity*. Traffic intensity is the ratio between the rate at which bytes arrive and the rate at which they are transmitted. If L is the average number of bytes per packet, a is the packet arrival rate, and R is the transmission rate (in bytes/second) then traffic intensity is La/R. If packets actually arrived periodically, then no buffering would be needed so long as $La/R \leq 1$. However, the actual arrival process is bursty. A burst of N packets may arrive every $(L/R)N$ seconds, averaging a traffic intensity of 1. Moreover, since the N packets arrive as a burst, a buffer is necessary to accommodate this traffic pattern and the average buffer size grows large, (n-1)/2*L. Since an arriving packet has to wait for all of the packets in front of it in the buffer to be transmitted, the longer the buffer grows the more time it takes for arriving packets to transit the router. Therefore, the average time each packet spends in the buffer is the average size of the buffer divided by the transmission rate, (n-1)/2*L/R. The resulting delay is referred to as *queue-induced latency*. With infinite buffers and bursty traffic, queue-induced latency increases rapidly, approaching infinity as traffic intensity approaches 1. When congestion is persistent, traffic intensity averages near 1, thus queue-induced latency can approach infinity with sufficient buffer capacity. Since memory costs money and latency is undesirable, buffering alone does not offer a suitable solution to the problem of persistent congestion. Instead, buffer capacity is allocated to accommodate transient, bursty congestion but when those buffers fill, arriving packets are dropped. Although routers can address transient congestion with sufficient buffering, router buffering is not a viable solution to persistent congestion. Instead, persistent congestion has traditionally been addressed by end-systems recognizing congestion indicators such as lost packets and responding to congestion by reducing the transmission rate. When an end-system responds to indications of congestion by reducing the load it generates to try to match the available capacity of the network it is referred to as *responsive*.

Responsive traffic is one of two major types generated by the end-systems. (The other is unresponsive.) From a network management perspective, responsive traffic is highly-desirable. In principle, when all flows are responsive, they adjust the load they generate to arrive at an aggregate load that stabilizes near the available capacity of the network and the network performs well. As a result, network utilization is high but overload is uncommon. The primary example of, and indeed the de-facto definition of, a responsive transport-level protocol is the Transmission Control Protocol (TCP) [Stevens94]. TCP is the protocol most commonly used for reliable data transfer. We will examine TCP and its responsiveness more closely in Sections 1.1.2 and 1.2. Conversely, an *unresponsive* flow is one that generates network load without any consideration of the current network conditions. The load generated at any time is solely a function of the operation of the application. The primary example of an unresponsive protocol is the unreliable datagram protocol (UDP) [Postel80]. UDP is commonly used by applications such as interactive multimedia where timely delivery of data is more important than reliability. However, when flows are unresponsive the load may continuously exceed the capacity of the network resulting in persistent congestion and poor performance for all flows. Note that the source of the responsive behavior is of no concern. It may be the result of a transport level protocol like TCP, a general application-level protocol, or a protocol unique to a single application. Henceforth, we refer to responsive and unresponsive *protocols*, meaning mechanisms at any of these levels.

There is a fundamental tension between these responsive and unresponsive protocols. Responsive protocols assume that the other flows sharing the network are also working towards the common goal of stabilizing the network load to match the network capacity. But, an unresponsive protocol sends its load into the network regardless of network conditions. When both types of flows are present during persistent congestion, the responsive flows reduce their load on the network but the unresponsive flows potentially maintain their load. Thus, the unresponsive flows benefit at the expense of responsive flows since the unresponsive flows achieve high throughput by using the capacity freed when the responsive flows reduce their load. In the worst case, the unresponsive flows generate a load greater than the available capacity of the link. In that case, the network remains congested while responsive flows reduce their generated load to essentially zero. Meanwhile, some or all of the unresponsive flows also still fail to receive the desired performance because the load they generate still exceeds the capacity of the link resulting in loss for those flows. A somewhat better scenario for unresponsive traffic (i.e., the set of flows using unresponsive protocols) has the aggregate load generated by the unresponsive traffic summing to less than the capacity of the network. In that case, the unresponsive traffic should be able to get good performance while only the responsive traffic’s performance suffers. Unresponsive traffic is able to achieve its desired throughput level while the responsive traffic reduces its load to (potentially) a small fraction of what it would receive if all flows were responsive. However, in this scenario responsive flows are still suffering poor performance due to the interaction of their behavior and unresponsive traffic's greedy approach.

Responsive traffic's vulnerability to the effects of unresponsive flows offers a disincentive to use responsive protocols. Application designers seeking high throughput may be tempted to use unresponsive protocols. However, if all flows are unresponsive, then congestion will be even more common since the end-systems will not adjust their loads to find a stable operating point. Incentives should be found to encourage the use of responsive protocols.

To better appreciate how this tension affects performance in the Internet it is helpful to consider the makeup of traffic in the Internet and the behaviors of the most common protocols at the application and transport levels. It is also useful to consider the current proposals for addressing the tension between responsive and unresponsive flows. In the remainder of this chapter, we first consider the nature of Internet traffic, its composition and the requirements and characteristics of the different types of traffic, particularly reliable data transfer using TCP and interactive multimedia using UDP. We then consider the problem of congestion control and some of the current approaches that have been taken to address both congestion in general and the tension between responsive and unresponsive flows specifically. Next, router queue management approaches are considered, including our approach, CBT. From there we present our thesis and outline our experimental approach to supporting it. Finally, we present an overview of the remainder of this dissertation.

1.1. The Nature of Internet Traffic

Different types of applications present different demands on the Internet. Application designers usually create application-level protocols to meet the specific demands of their application. For example, HTTP has a well-defined format for specifying a requested object, the transfer format to use, and any additional information. However, those application designers also follow the modularity argument that says that they should select a transport level protocol (e.g. TCP or UDP) to provide basic transport-level services when possible. For example, HTTP needs a reliable byte stream to insure files arrive intact so the designers chose to use TCP as the transport protocol because it offers ordered reliable delivery. Conversely, video conferencing applications seek to deliver the most recently recorded frame to the play-out station as soon as possible. If a frame is lost, the emphasis is on making sure the next frame arrives quickly, not on retransmitting the old frame until it arrives successfully. As such, those designers often choose UDP as the transport protocol because it offers an unreliable, datagram service.

In this section we present measurements of typical backbone traffic showing the major traffic types found. Then we identify two major categories of Internet traffic, reliable data transfer and interactive multimedia. For each, we present the transport level protocols they most commonly use and the reasons why.

1.1.1. Types of Traffic and Percentages

Internet traffic can be described at many levels. For the purposes of this discussion we will consider the distribution of Internet traffic by transport-level and application-level protocol. Figure 1.1 shows a typical graph of the traffic monitored on an Internet backbone link classified by transport protocol type [Claffy98]. See also [McCreary00, Charzinski00] for more examples. Table 1.1 shows a subset of the data in a tabular format. [1] The high percentage of TCP traffic is immediately obvious from the graph. Further, UDP is the second most common type of traffic. These two protocols represent 95% and 5% of the bytes, respectively. This shows that TCP clearly is the predominant transport protocol in the Internet. Moreover, UDP is the only other transport protocol using a significant fraction of the link's capacity.

[1] The data shown here comes from monitoring the connection between a core router and the backbone at a junction point including several backbone trunks as well as a connection point for local traffic near a major U.S. East Coast city.

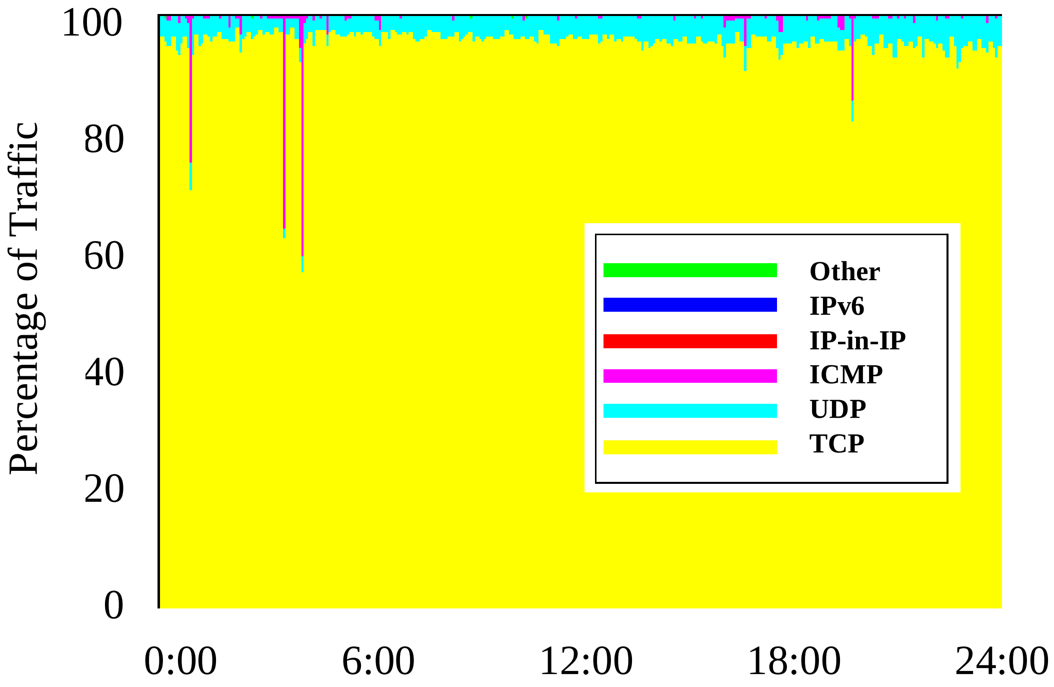


Figure 1.1 Transport-Level Protocols as Percentage of Internet Traffic (figure taken from [Claffy98])

	% of Bytes	**% of Packets**	**% of Flows**
TCP	95%	90%	80%
UDP	5%	10%	18%
ICMP	0.5%	2%	0%

Table 1.1 Transport-Level Protocols as Percentage of Internet Traffic

Figure 1.2 shows a graphical representation of the same traffic, broken down by application-level protocol. Table 1.2 shows the same data in a tabular format. Once again, one type of traffic clearly dominates the link. That traffic type is HTTP, the traffic type used by the world-wide web. HTTP depends on the transport layer protocol, TCP, to achieve reliable data transfer of objects (e.g. HTML documents and images). The second highest category in the chart is other, the combination of miscellaneous application protocols that individually represent less than 1% of the traffic. As noted in [Claffy98], the web may actually be under-represented in these measurements because the most common port numbers in the category of other traffic are associated with various web-related protocols.

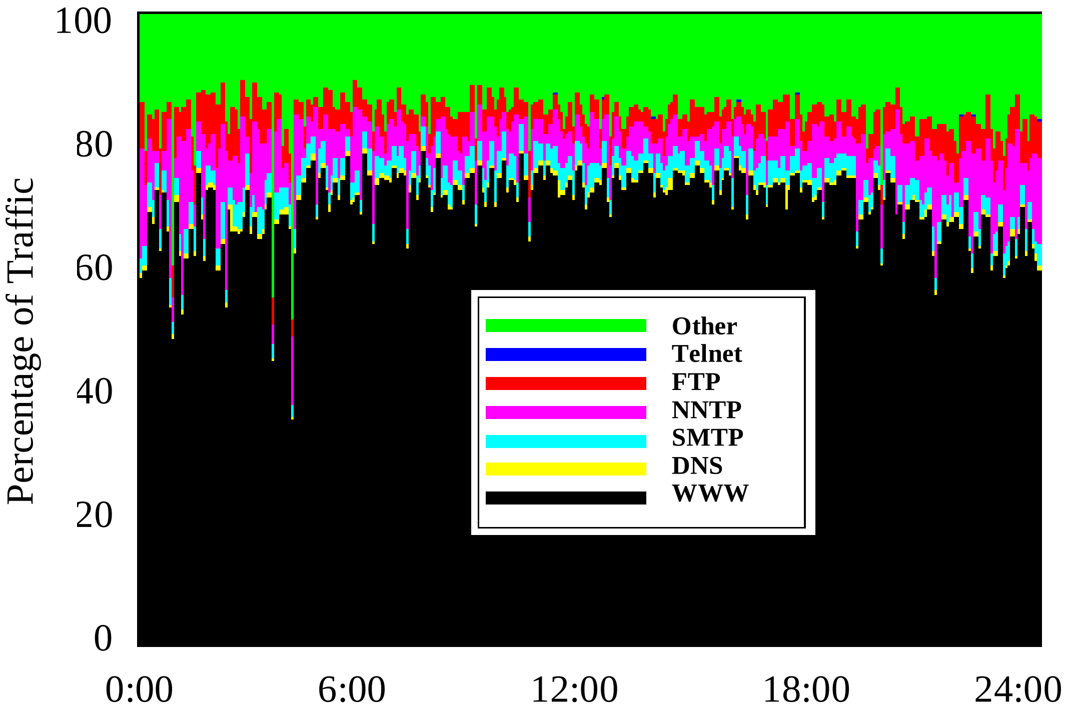


Figure 1.2 Application-level Protocols as Percentage of All Internet Traffic (figure taken from [Claffy98])

Protocol	% of Bytes	% of Packets	% of Flows
WWW (TCP)	75%	70%	75%
DNS (UDP)	1%	3%	18%
SMTP (TCP)	5%	5%	2%
FTP (TCP)	5%	3%	1%
NNTP (TCP)	2%	<1%	<1%
TELNET (TCP)	<1%	1%	<1%

Table 1.2 Application-level Protocols as Percentage of All Internet Traffic

1.1.2. Reliable Data Transfer and TCP

As this data shows, reliable data transfer, specifically HTTP, is the most common type of traffic in the Internet. Here, *reliable data transfer* refers to the reliable transmission of a single, contiguous unit of data from one end-system to another, putting an exact duplicate of the data unit on another end-system. TCP's ordered, reliable byte stream abstraction was designed for this type of task. Below we discuss the properties of and types of applications that use reliable data transfer. We then discuss the abstraction of an ordered reliable byte stream offered by TCP and why TCP is the transport protocol of choice for reliable data transfers.

Reliable data transfer is distinguished by a binary correctness property. If the data does not arrive correctly it is useless. Proper delivery has three properties focused on the integrity of the data: completeness, correctness, and order. The data unit must be complete; reception of only a part of the data holds no value. The data must also be correct; corruption of any part of the data removes all value. And, the data must be properly ordered; These integrity properties are the primary requirements for reliable data transfer. Performance is also a concern, but secondary to the data’s integrity. Applications should transfer their data in a reasonable period of time but the timeliness of the data’s arrival does not effect its usefulness. Consequently, reliable data transfer needs a mechanism that provides a reliable ordered data stream connecting the sender and the receiver.

The predominant choice among transport-level protocols for reliable, ordered delivery of data is the Transmission Control Protocol, TCP. TCP offers the application-level programmer the abstraction of a reliable, ordered byte stream. It is these three properties that make TCP very attractive for services that move data from one location to another.

However, it is a fourth property, added after the initial design of TCP and essentially invisible to the application-level programmer, which makes TCP especially well suited to the shared nature of the Internet. That property is TCP's responsiveness to congestion [Jacobson88]. Below, we discuss how the properties of ordering and reliability are achieved. We will discuss how the initial design of TCP was extended to respond to congestion in Section 1.2.

TCP was initially designed to offer the abstraction of a byte stream connecting two applications on separate machines connected by a network. Data sent from one application will eventually arrive at the other application in the order it was sent. In the relatively rare case of network failures (e.g. physical link failure, or route failure) that break the "pipe" between sender and receiver, the application-level abstraction will report failure of the connection. The network protocols (e.g., IP) neither insure that data arrives nor that it arrives in the order sent. The network layer only provides best-effort, possibly unordered, forwarding of packets to the destination. TCP orders these packets using a sequence number in each message it sends. Contiguous bytes of data are made available to the application-level receiving process. Non-contiguous bytes are not immediately delivered to the application and instead are stored in the protocol stack until the missing bytes arrive. This leads to the issue of retransmissions.

If a packet fails to arrive at the receiver it leaves a gap in the sequence number space. If the packet has been lost, the sender must retransmit the packet necessary to fill such a gap. However, the sender must have some mechanism to recognize when a packet has been lost. TCP achieves this with a feedback mechanism. Data that arrives successfully at the sender is acknowledged in a reply from receiver to sender. Failure to receive acknowledgement of a packet after a reasonable period of time indicates that the packet was lost. This time-out period may be measured either in time as a function of the round-trip time between sender and receiver or by the number of subsequent packets with a higher sequence number that arrive at the receiver. TCP receivers send multiple, duplicate acknowledgements of the highest contiguous byte received. Whether loss is detected via time-out or duplicate acknowledgements, the sender retransmits the lowest numbered unacknowledged message. This cycle is repeated as many times as necessary. As a result, achieving this reliability can introduce significant latency, on the order of round trip times, when a packet has to be retransmitted. This retransmission of data is the key to TCP’s reliability and this reliability makes TCP ideally suited to reliable data transfer.

1.1.3. Interactive Multimedia and UDP

Interactive multimedia's correctness is less discrete than that of reliable data transfer. Loss or out-of-order delivery may be tolerable. However, throughput and latency are important performance factors. We explain these factors below and then explain why UDP is a better choice than TCP for interactive multimedia.

While reliable data transfers have long dominated the Internet, multimedia, such as audio and video, is one of the newest forms of traffic in the Internet. It can be widely used interactively, either to allow two or more individuals to converse or to allow a user at one end-system to interact with a remote environment. This remote interaction may be real, such as controlling some remote robotic device, or virtual, such as interacting with a virtual environment being generated at a remote location.

Interactive multimedia's primary characteristic is that the quality of the content is time-dependent. Data may arrive at the viewing station slightly later than the ideal time it should be played and still be useful. However, data that is delayed significantly may be useless. If the throughput is lower than intended the quality of the play-out may suffer but still be useful. However if the throughput falls below some minimum level the media stream may be useless.

The fundamental generic data units for multimedia are frames that are periodically generated. Depending on the specific media type, the receiver can tolerate some lost frames. In particular, loss may be preferable to the increased latency required to insure reliable delivery. Increased reliability usually comes at the cost of the increased latency to sense the loss and then retransmit the lost data. Because of concerns over smooth playback as well as interactivity, significant latency may degrade the properties of an interactive multimedia interaction. We discuss each of these properties, throughput, loss, latency, and jitter, in greater detail below.

Consider how interactive multimedia works, in a perfect network. In the simplest case, a continuous stream of independent and discrete samples (frames) of analog audio or video are periodically recorded at the recording station, transmitted across the network to the viewing station, and then played back with the same periodicity. The periodicity at the recording station is the recording rate and the periodicity at the viewing station is the display rate. With a sufficiently high recording and display rate these consecutive frames give the illusion of continuity. However, sometimes the display rate may be insufficient to maintain the illusion of perfect continuity. Even in these cases where the discrete nature is perceptible, the quality may be tolerable. The tolerances on display rate vary with the type of media stream and the content. In a video stream with a great deal of motion, such as a football game, the display rate must be very high to be tolerable. In a narrated slide presentation a low video display rate may be fine because the image changes slowly and discretely. If the video is combined with an audio stream with a higher play-out rate the interaction may be quite satisfactory. However, audio media has a more strict minimum acceptable play-out rate than video. While visual media has both an instantaneous (e.g. photograph) and continuous (e.g. motion-picture) dimension, audio lacks the instantaneous dimension. If audio samples are missing at play-out you have noticeable gaps or "pops".

Another key aspect of multimedia lies in the fact that frames are independent. (For the purposes of this introduction we will ignore encoding schemes and features such as reference frames and the considerations they present.) By *independent* we mean that if a single frame fails to reach the playback station, that frame can simply be skipped, essentially reducing the frame-rate for that instant. Loss of one frame does not effect the value of the other frames. Moreover, the loss of one frame may be beyond the human perception threshold. Even moderate loss may only result in a decreased frame-rate that although perceptible, remains above the user's tolerance threshold. Although reliability at some level is desirable, perfect reliability it is not a primary requirement for the transmission of interactive multimedia in the Internet because of frame independence.

Next, consider latency. Minimizing latency is another important property for interactive media. In order for the media to be used interactively (e.g. for human-to-human conversations) the delay between the time that a frame is recorded and the time that it is played must be minimized. We refer to this delay as *end-to-end* latency. The most common example of interactive media is a telephone conversation. Studies have indicated there are definite upper bounds on the order of 250 to 400 ms for tolerable one-way latency for interactive audio [Ferrari90],[Wolf82]. Also, consider the effect of latency on control of a remote device. In order for the control to be natural, the time between taking an action and receiving the feedback based on that action should be nearly instantaneous.

Users expect to see the results of their input adjustments with no apparent delay, on the order of 100 ms [Schneiderman98]. In some cases, delays of 50ms are noticeable and delays of 80ms are irritating [Bhola98]. Interactive multimedia demands low latency for the interaction to be useful.

Even when dealing with non-interactive media, latency, particularly variance in latency, is a serious issue. We refer to the variability in latency as "jitter". For example, jitter results in irregular or random movement that appears in a video playback with variable latency. If the latency is variable, the inter-arrival time between frames will be variable. If frames are played back as soon as they arrive, the display of the frames will lose the illusion of continuity. In this context, jitter applies to all types of media, referring simply to a non-periodic playback rate of the frames. Jitter can be addressed by artificially adding delay to those packets that arrive quickly at the receiver by buffering those packets [Stone95]. This technique causes all packets to have the same, potentially large, delay but avoids variation in latency. Typically the buffer is configured to be sure all packets incur a delay on the order of the maximum end-to-end delay. Thus, it is useful to have an upper bound on the end-to-end latency in order to limit the amount of buffering and latency that must be allocated at the receiver. The variability in latency needs to have a reasonable bound in order for end-system jitter management techniques to be effective.

The protocol commonly used for interactive multimedia is UDP. UDP offers a datagram service that makes a best-effort attempt to deliver data to the destination. UDP is basically a transport-level wrapper for IP.

UDP appeals to designers of multimedia applications because is an extremely lightweight protocol. Because it simply guarantees the transmission and not the delivery of the packet, it offers a more stable transmission rate and less transport-level buffering. This lack of transport-level buffering eliminates one potential source of latency. Datagrams are never retransmitted. Transmission is never delayed because a previous packet is unacknowledged. Datagrams are only buffered if the network interface lacks capacity to transmit them, or, at the receiver, if the receiver is not ready to receive them. Since interactive multimedia applications are more concerned with timely delivery of frames than reliable delivery, UDP is well-suited to this application domain.

In contrast, TCP is less well-suited for multimedia. Since latency and throughput are primary concerns and the data stream is loss tolerant, TCP is less appealing. TCP's primary design feature, reliability, carries some overhead in terms of limited effective bandwidth and latency. Because all data must be buffered at the sender until acknowledged and at the receiver until properly ordered, and because buffer space is limited, TCP's transmission rate for new, unsent data is limited. When the sender's buffer is full, the application is blocked from transmitting new data until older data has been successfully delivered. This can result in significant latency during periods of congestion. During periods of intense congestion, a 10% drop rate may result in latency up to 70 seconds for some TCP flows [Cen98]. Another TCP feature, ordering, may appear desirable since frames should be played in order. However, out-of-order arrival usually occurs because of loss. To correct this problem one must wait for the out-of-order frame to be retransmitted and arrive before processing the later packets, resulting in high latency. Instead, multimedia application designers use UDP and treat the out-of-order frame as lost and discard it. Since multimedia places an emphasis on the most recent data and is tolerant of some loss, this policy works well. Multimedia application designers usually choose the lighter-weight semantics of UDP and rely on application semantics to deal with pathologies.

1.2. The Congestion Control Problem

Having described the TCP and UDP protocols and the nature of the applications that use them, it is helpful to give more detailed consideration to congestion and the way that the two protocols behave in the presence of congestion. The protocols vary widely in the ways that they contribute to and respond to congestion. TCP is responsive; it heuristically recognizes and responds to congestion. Conversely, UDP is unresponsive; it does not recognize congestion so it does not respond to it. To better understand how and why the two most popular protocols take such different approaches to this problem, we must first consider the nature of congestion. Why does it happen, how is it identified, and how can it be avoided?

Congestion on a link occurs when the traffic arriving at a given link (i.e. load) exceeds the maximum capacity of that link. This congestion may occur because the outbound link’s capacity is less than that of a single inbound link and all of the traffic arriving on that inbound link is destined for the lower capacity outbound link. It may also, more commonly, occur because packets arriving from multiple inbound links are all destined for the same outbound link and, in aggregate, those packets exceed the capacity of the outbound link. The congestion may be transient, resulting from the chance simultaneous arrival of packets on multiple inbound links or it may be persistent as the aggregate load consistently exceeds the outbound link’s capacity.

Traditionally, there have been no widely used explicit indicators of congestion in the Internet. There was no widely used mechanism for providing feedback to end-systems to indicate that a particular link was congested. (Although the source quench option of ICMP was designed to allow routers to signal congestion, it has not been widely used in routers or end-systems. For more discussion of source quench, see Chapter III.) Instead, end-systems had to infer the presence of congestion by noting packet loss and assuming that loss was due to overflow in a congested router's queue. In order for loss to be an effective indication of congestion, either the transport or application-level protocol must detect loss by the absence of expected packets at the receiver, and report the loss to the sender. TCP must detect loss in order to implement reliability through retransmissions. It can use this same mechanism to infer congestion and respond accordingly.

There are proposals for explicit congestion indicators as well. Recent proposed extensions to TCP recommend taking advantage of a an explicit congestion notification (ECN) bit in the IP header which would indicate congestion may be imminent [Floyd94]. Those proposals recommend that routers set this ECN bit in the IP header of packets when persistently occupied queues indicate congestion. The receivers of messages with the ECN bit set would then be responsible for informing the senders that congestion may be imminent. Since this mechanism would indicate congestion before queues overflow, the senders would be able to adjust their transmission rate and actually avoid loss, rather than simply respond to it. This would reduce the need for retransmissions since the notification occurs before packet loss actually occurs. Unfortunately, ECN, like many Internet proposals faces the problem of inertia. There are many deployed implementations of the transport protocols that would need to be modified to respond to this ECN bit. Changing all of the deployed implementations is an immense task. Further, until more routers become ECN capable, there is little motivation to use ECN. As a result, we continue to focus on the implicit indicator, loss, as the primary indicator of congestion and next consider the effects of congestion.

Once a link becomes congested, if the congestion persists it may lead to a pathological condition called congestion collapse. Congestion collapse is a phenomenon where the capacity of the network, or of a given network link, is fully utilized but little or no useful data is actually reaching its intended destination. The most well-known form of congestion collapse led to the development of congestion control mechanisms in TCP. When TCP was first deployed it had no congestion control mechanism. It became apparent that multiple TCP streams sharing the same network resources and achieving reliability through retransmissions could quickly lead to one form of congestion collapse called *retransmission collapse* [Nagle84]. In retransmission collapse, flows respond to packet loss by maintaining their transmission rate and retransmitting the lost packets. This maintains the overload situation and causes the congestion to persist or grow. Moreover, the same data may be transmitted over the links that lead to the congested router multiple times before being dropped at the congested router. This wastes capacity on all of the links. The links may be fully utilized but little useful data actually reaches the receivers.

Clearly, recognizing and responding to congestion in a way that does not lead to congestion collapse is a key element of maintaining good performance in today’s best-effort Internet. Congestion may be addressed with congestion avoidance algorithms in each sender or with techniques to manage and allocate link capacity in the routers. Currently, the primary congestion avoidance technique is TCP's congestion control policy [Jacobson88]. TCP interprets loss as an indicator of congestion and adjusts its transmission rate accordingly. When congestion is detected, a given flow decreases its transmission rate geometrically. This is accomplished by halving the number of bytes that a flow may have sent but not yet received acknowledgement for. When losses cease, the flow's transmission rate is increased linearly, adding one segment per round-trip time. Using this technique, individual TCP flows probe for network capacity. The intent is to find equilibrium where the load matches the capacity of the bottleneck link. When all senders address congestion in this fashion, most of the data passing through the network is actually useful data that reaches the senders.

However, many applications use other transport protocols that are not responsive, often with good reason. For example, UDP is selected for multimedia specifically because the protocol doesn't introduce any artificial delays. Moreover, because responsiveness requires a feedback mechanism, UDP can not be changed to be responsive without losing compatibility with previous versions. Alternative means of responding to, or avoiding congestion must be considered to address the demands presented by traffic that does not use TCP.

2. Active Queue Management

Within the infrastructure of today’s best-effort Internet, there are several approaches to providing good congestion management. Most of these approaches involve changes to the end-system at either the application-protocol or transport-protocol levels. Many of these approaches are discussed, in Chapter II. However, end-system reactions to congestion can be improved by having routers identify, manage, and signal congestion within the best-effort Internet. By taking an active approach to deciding which packets to drop (or mark as with ECN) and when to drop them, routers can provide better feedback to responsive flows. They can also identify and regulate unresponsive flows. In contrast, traditionally the buffers in routers were treated strictly as queues with fixed, finite capacity. That is, they were first-in, first-out (FIFO) queues with drop-tail when full behavior.Having end-systems address congestion by responding to loss is not perfect. First, notification by drops is somewhat arbitrary. Only those end-systems that have packets dropped recognized congestion. The distribution of packet drops is determined by the pattern of packet arrivals since packets are only dropped if the queue is full when they arrive. Some flows might send bursts of packets that arrive at a near full queue, forcing most of the packets from that flow to be dropped. In contrast, in the same interval another flow may send a more evenly spaced stream of packets, all of which are fortunate enough to arrive when the queue is not quite full because the queue drains slightly between bursty arrivals. Consequently, some flows may receive multiple implicit notifications (i.e. multiple lost packets) while others receive few or none. Recent efforts have focused on equitable notification for all flows. This was accomplished by extending the options of when to drop packets and which packets to drop beyond simply dropping the newly arriving packet when the queue is full. These alternative approaches that make active decisions on when to drop packets and which packets to drop are referred to *as Active Queue Managemen*t (AQM). We discuss Active Queue Management in detail in chapter III but provide a brief overview here.

The most recent AQM approaches use statistics about the recent *average* queue occupancy as an indicator of congestion. By monitoring the average queue size they recognize situations where congestion is persistent or imminent and differentiate between those situations and ones where the congestion is transient because of bursty arrivals.

Recently, members of the Internet community recommended that some form of AQM be deployed at most routers in the Internet [Braden98]. They specifically suggested the deployment of a mechanism called Random Early Detection (RED) [Floyd93]. RED monitors the average queue occupancy and drops arriving packets probabilistically when the average is too high. Below a minimum threshold the probability is zero and above a maximum threshold the probability is one. Between those thresholds the probability a packet will be dropped increases with the average occupancy. RED thus offers a reasonably accurate mechanism to detect congestion by comparing the average queue length to these maximum and minimum thresholds. Moreover, the algorithm provides an effective mechanism to distribute the notifications of congestion among all flows. Although the basic mechanism of probabilistically dropping arriving packets distributes drops more evenly, it is further enhanced by the use of a mechanism that increases the likelihood an arriving packet will be dropped as the number of packet arrivals since the last packet drop increases. This makes it less likely that multiple packets in the same burst will be dropped.

However, as we will show in Chapter III, RED does nothing to address the tension between responsive and unresponsive flows. During a congested period, RED will drop an equal percentage of the packets on each flow. Responsive flows will interpret these drops as congestion indicators and respond by reducing the load they generate. However, unresponsive flows will continue to generate the same load. If the network remains congested, all the flows will continue to be subjected to drops and the responsive flows will continue to reduce their load, allowing unresponsive flows to dominate the link. In response to this issue a second recommendation was made calling for continued research into mechanisms to identify and constrain unresponsive flows [Braden98].

Most of the approaches to this problem have been TCP-centric. That is, the approaches are based on the observation that TCP's congestion avoidance policy makes it a good network citizen as it achieves equilibrium between load and capacity. As such, other flows should be encouraged to behave in a more TCP-like way and those that don't should be harshly constrained. Floyd and Fall define tests to identify high-bandwidth, unresponsive, or non-TCP friendly flows (i.e. flows that are responsive, but slower to respond than TCP) and propose that such flows should be severely constrained by preferentially discarding packets from those flows [Floyd98]. Similarly, extensions to RED, such as Flow-based RED (FRED), propose that all flows should have access to an equal share of the link's capacity [Lin97]. FRED takes advantage of the fact that packet drops serve a dual role, notification and constraint. Responsive flows are primarily subject only to drops for notification following a RED like algorithm. However, FRED's designers use packet drops specifically to constrain those flows that do not respond to congestion notification to have no more than a fair share of the queue, and thus the outbound link.

2.1. Our Approach - Class Based Thresholds

We take a different perspective on the tension between responsive and unresponsive flows. We recognize that some flows are unresponsive because the nature of the communication does not lend itself to having a feedback channel that can be used to implement a TCP-like detection of congestion. Further, application requirements may constrain the responsiveness. Moreover, we realize that aggressively constraining flows like interactive multimedia may drop the value of the interaction to zero. This actually contributes to inefficiency, as those packets that reach the receiver have no value since the degradation of the multimedia stream because the other losses makes it useless. Instead of punishing all unresponsive flows, we seek to isolate classes of flows from one another, protecting responsive flows from the effects of unresponsive flows, and protecting different types of unresponsive flows from one another. We propose a new algorithm, class based thresholds (CBT), that offers these properties.

The implementation of the CBT algorithm introduced here defines a set of traffic classes. Flows belong to one of several classes and CBT maintains statistics for each class of traffic. Each class's average queue occupancy in a router is monitored and statistics are maintained. The algorithm assures that each class is limited to a fixed average queue occupancy by using an occupancy threshold. Arriving packets of classes that exceed this threshold are discarded. The relative queue occupancy of each class also determines the relative share of the outbound link used by each class. For example, if a given class occupies 10% of the bytes in the queue, then as the queue drains, that class will represent 10% of the traffic on the outbound link. Adjusting these occupancy thresholds allows a network administrator to allocate shares of the outbound link to each class. Further, because the sum of the thresholds determines the maximum average queue occupancy, these thresholds also control the maximum average queue-induced latency. Finally, CBT is light-weight. It does not have the algorithmic or state maintenance overhead associated with packet scheduling techniques that provide similar functions. Instead, CBT maintains a small number of statistics and performs a simple test at each queue's arrival to determine whether or not to enqueue the arriving packet.

3. Thesis Statement:

In this dissertation we will show:

> Our active queue management algorithm, class-based thresholds, can effectively isolate responsive traffic from the effects of unresponsive traffic. Moreover, CBT also isolates unresponsive traffic types from one another. Further, analysis shows that CBT can be configured to allocate bandwidth and manage latency and that the performance under varying loads is pre-

dictable. Finally, empirical analysis confirms that CBT is superior to well-known AQM algorithms and comparable to a packet scheduling approach.

4. Empirical Validation

To determine the effectiveness of our algorithm we compare it to other queue management approaches, specifically FIFO, RED, and FRED. Moreover, we also conduct the same experiments using a packet-scheduling discipline, Class-based queueing (CBQ), as a "gold-standard" for comparison. Performance with packet scheduling will be near optimal in terms of reliable data transfer and multimedia performance. We argue the merits of active queue management over packet scheduling regarding complexity, infrastructure, and state in Chapter VI. The experiments compare the performance of TCP, multimedia, and other traffic as they traverse a congested network with a bottleneck router running one of the algorithms discussed. We use multiple types of TCP and multimedia traffic in different combinations to explore a wide range of parameter settings for each algorithm. We describe the nature of our empirical experiments in more detail below.

4.1. Empirical Comparison

We have implemented CBT and FRED within the FreeBSD kernel with ALTQ extensions [Cho98]. ALTQ is a set of extensions to the default IP-layer packet queueing policies in a FreeBSD router to support development of experimental packet schedulers and active queue management schemes. In addition to our active queue management implementations, we also used the existing ALTQ implementation of RED and CBQ.

To test the implementation we have constructed a simple network consisting of two switched 100 Mbps Ethernet LANs that are interconnected by a 10 Mbps Ethernet. FreeBSD routers route traffic between the 100 Mbps Ethernets across a full-duplex 10 Mbps Ethernet as shown in Figure 1.3. The speed mismatch between the “edge” and “backbone” (i.e. middle) networks exists to ensure the backbone network is congested. A series of machines at the edges of the network establish a number of connections to machines on the opposite side of the network. Connections include a mix of TCP connections and UDP connections. Several of the UDP connections are multimedia.

For the TCP flows, the kernel on each machine introduces a delay in transmitting packets from each connection to simulate a larger round-trip time. This allows us to create conditions that emulate a wide-area backbone link.

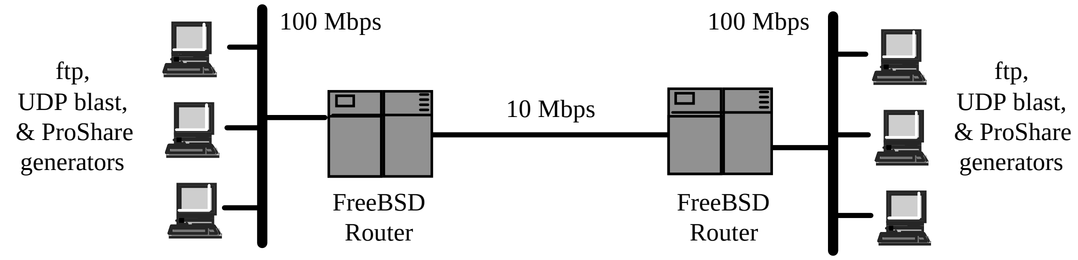


Figure 1.3 Experimental Network Setup

Traffic is generated in all of the experiments using a scripted sequence of flow initiations. This ensures comparable traffic patterns are generated in each experiment and hence the results of experiments may be compared. The timing of the traffic and the specific traffic types used varies between experiment types (but not between instances of the same experiment).

The types of traffic representing TCP and multimedia vary. In some traffic mixes a small number of long-lived reliable data transfers (BULK) are the representative TCP traffic type. In other experiments we used a large number of light-weight short-lived flows that are part of an HTTP model. Multimedia also had two types of traffic, a set of Proshare flows and MPEG flows. Proshare is a proprietary video-conferencing application from Intel. The Proshare flows represented a series of independent frames while the MPEG flows had the common MPEG dependencies between I, B, and P frames. This allowed us examine the effect of congestion on actual frame-rate vs. playable frame-rate. Our *other,* unresponsive, traffic was a small number of UDP flows sending at a fixed rate which in aggregate summed to greater than 10 Mb/s.

Using this methodology we considered each algorithm’s behavior under a variety of traffic conditions. We initially explored the parameter space for each algorithm, determining the effects of adjusting different parameters and selecting the optimal parameter settings for each combination of algorithm and traffic mix. We then compared the performance of the algorithms under the same traffic mixes. In each case, the algorithms where configured with their optimal parameter settings for the given traffic mix.

4.2. Summary of Results

Our experiments revealed several important results:

- We determined optimal parameter values for FIFO, RED, FRED, and CBT under a variety of traffic conditions. Along the way we found several important insights on the relation between different parameters and key performance metrics. For example, the maximum threshold value is linearly related to the maximum average queue-induced latency for RED.
- TCP receives a better share of bandwidth when facing misbehaving or unresponsive multimedia flows with CBT than with RED. Performance with CBT is comparable with FRED.
- Using CBT, the number of drops on low-bandwidth protected flows is substantially lower than when RED or FRED are used.
- Using CBT, we can approach the class isolation and multimedia drop rate of CBQ.
- Moreover, CBT achieves these results with less complexity than FRED or CBQ. FRED maintains state for every flow. CBQ must schedule every packet dequeue.
- CBT also offers flexibility in assigning router resources to different classes of traffic instead of the uniform distribution offered by FRED.

CBT shows promise as an extension to conventional RED to constrain unresponsive traffic and to support self-identified multimedia flows in today’s Internet.

In the remainder of this dissertation, we first consider related work in congestion control in chapter II. Then we focus on active queue management in chapter III. In chapter IV we examine CBT in detail including the algorithm, and empirical verification of analytical results. Chapter V shows the results of our empirical evaluation of those algorithms while chapter VI summarizes and discusses future work. Appendix A covers the experimental methodology and Appendix B reviews the choice of optimal parameters.

II. RELATED WORK

1. Introduction

At a high-level this dissertation addresses the problem of congestion in the best-effort Internet. In this work, we focus on the specific problem of tension between unresponsive flows and network stability when end-to-end congestion control mechanisms like those in TCP are the only mechanism. However, other researchers have considered the more general problem of congestion from different perspectives. We consider the related work here.

We begin by considering the approaches currently used widely in the Internet. These approaches focus on dealing with congestion in the transport-level protocols. The dominant example of this approach is TCP. TCP has been modified from its original deployment to infer the presence of congestion and adjust its transmission rate accordingly. Approaches like TCP's responsive congestion avoidance work well when all other traffic is also responsive. This is not always true. UDP, the second most common transport protocol, is unresponsive. This creates a problem as UDP may starve TCP flows by claiming the capacity that TCP flows free when they reduce their transmission rate in response to congestion.

In response to this issue, the Internet research community is calling for making all traffic more responsive to congestion [Braden98]. There have been proposals to add responsiveness at all levels of the protocol stack. At the transport level, Cen et al., and Sisalem et al. propose protocols that maintain the low overhead associated with unreliable protocols while adding the capability to respond to congestion by adjusting the transmission rate independent of the application [Cen98, Sisalem98]. Others have proposed application-level approaches which detect changes in application level-performance and make application level adjustments, such as change the recording parameters (e.g., frame-rate, resolution), in order to adjust the load placed on the network [Talley97, Delgrossi93]. Additionally, Balakrsihnan, et al., offer an integrated congestion management architecture for end-systems that manages congestion for all flows on the end-system at the transport-level, while providing an API to integrate application level control and feedback [Balakrishnan99]. Finally, others propose integrating network and application level approaches. In these approaches, some routers use application level tags to determine either which packets to drop or the order in which to forward packets [Hoffman93, Delgrossi93].

In contrast to these changes to the end-system protocols, there are also router-based approaches, both for constraining unresponsive traffic and for allocating bandwidth so capacity is available for responsive traffic. These proposals and implementations introduce traffic management into the Internet architecture through designs such as the differentiated service architecture [Nichols97], [Clark97], and the integrated services architecture [Braden94]. However, these architectures have not been widely deployed and may require significant and complex modifications to the network infrastructure. These approaches provide policies for allocating network bandwidth and offering guarantees on latency and throughput. In contrast, there are other network-centric approaches that focus on providing better feedback to responsive flows as well as identifying and restricting unresponsive flows. Those approaches will be discussed in detail in Chapter III. In this chapter we cover transport-level, application-level, and integrated end-system approaches. We then consider the router-based quality of service (QOS) approaches. Finally, the strengths and weaknesses of each are summarized.

2. Transport-level Approaches

At the transport level, many protocols provide mechanisms to detect and respond to congestion by adjusting the rate at which they send. These mechanisms range from the idealized responsiveness of TCP to the complete absence of responsiveness in UDP. In contrast to the well-established UDP, recent work has focused on several responsive forms of unreliable datagram protocols. This work includes the Streaming Control Protocol (SCP) [Cen98] and the Loss-Delay Based Adjustment (LDA) [Sisalem98]. Both the current and proposed protocols are considered at greater length below.

2.1. TCP’s Responsiveness

TCP is the primary example of a congestion avoidance solution. In response to the congestion collapse phenomenon, TCP’s response to packet loss was modified [Jacobson88]. Before considering TCP's response to congestion it is helpful to review how TCP's transmission rate is controlled by a windowing mechanism. The TCP protocol buffers units of data called segments at the sender and the receiver. At the sender these segments are buffered until the receiver has acknowledged them. At the receiver, the segments are buffered until all prior segments of the data stream have been received and passed to a higher protocol layer. At the sender, a mechanism called a *congestion window* determines the number of consecutive segments that may be sent without receiving acknowledgement for the first segment in the window. As segments are acknowledged the congestion window advances to cover new segments (and those segments are then transmitted). Since it takes one round-trip time (RTT) for the data to reach the receiver and for the acknowledgement to arrive, the protocol transmits one window's worth of bytes per RTT. Thus, adjusting the size of this window effectively adjusts the transmission rate for the flow.

To avoid congestion collapse, TCP senders respond to loss not only by retransmitting the data that was lost, but also by decreasing the rate at which they introduce data into the network. In principle, when a packet-loss is suspected at the sender the packet is retransmitted but the transmission rate of new data is decreased geometrically. TCP senders make this adjustment by decreasing the size of the congestion window by half. This geometric back-off is balanced with a linear increase in the size of the congestion window when no packet-loss is detected. Using this control loop, the TCP streams are able to adjust the load they place on the network to attempt to match the available capacity of the network. As a result, most of the data passing through the network is actually useful data that ultimately reaches the receivers.

2.2. UDP’s Unresponsiveness

In contrast to TCP, UDP is a transport level protocol that is unaware of the network’s state. One of UDP’s key characteristics is that it is lightweight, with each datagram sent and then forgotten. At the protocol level, the receiver does not acknowledge receipt of the datagram. There is no feedback from the receiver of any kind. Without a feedback channel the UDP sender cannot detect network congestion, either implicitly or explicitly. As a result, it cannot respond to congestion. Instead, UDP simply passes whatever data the application generates to the network (i.e., IP) layer as fast as possible. Once the packet has been forwarded to the network layer the UDP sender's task is complete.

Making UDP responsive in a backward compatible way would be quite difficult. Like TCP, UDP was also widely deployed before the congestion collapse problem was first identified. However, unlike TCP, UDP had no feedback channel as part of its initial design. TCP designers were able to leverage their existing feedback scheme and only change the behavior of the senders, not the protocol itself nor the application using the protocol. As a result they were able to gradually deploy TCP implementations that were both capable of congestion avoidance and compatible with older implementations of the protocol. In contrast, adding congestion detection to UDP would require a redesign of the protocol that would make new versions of the protocol incompatible with previous versions.

2.3. Responsive Unreliable Protocols

Another approach to addressing the tension between responsive and unresponsive flows is to introduce new transport level protocols to replace UDP. These protocols provide feedback from receiver to sender and use this information to adapt the transmission rate of the sender to the available capacity of the network. However, these protocols can be more light-weight than TCP because they need not include reliability among their features. They can also incorporate other desirable delivery semantics for streaming media. Two examples of this approach are the Streaming Control Protocol (SCP) [Cen98] and the Loss-Delay Based Adjustment Algorithm [Sisalem98]. Both of these protocols propose techniques for adjusting the transmission rate of the sender in a TCP-friendly way based on congestion indicators without the overhead associated with TCP’s reliability.

2.3.1. Streaming Control Protocol

SCP is an extension to UDP designed to support streaming media. The design goals include providing a smoother transmission rate which responds to network conditions while minimizing latency. The authors base their design on TCP’s window-based policy for congestion avoidance. Like TCP, they use acknowledgements to detect congestion via packet loss and they increase the sender’s window linearly during slow-start (i.e. the initialization period when a TCP connection quickly increases its transmission window) and decrease it geometrically during congestion. However, they also include additional feedback and additional protocol states. The acknowledgements not only confirm the receipt of a packet, they also report an approximation of the observed recent arrival rate of packets at the receiver, *r*. This rate is maintained as a running average. When this rate becomes constant, the sender enters a *steady-state* in which it departs from TCP’s technique of probing network capacity and instead maintains a window-size slightly larger than that indicated by the current bandwidth-delay product (where the bandwidth is determined by the receiver’s observed packet-rate and delay is the round-trip time when the network is uncongested). If the bandwidth available to a flow remains constant, the excess will not reach the receiver. However, if the available bandwidth increases those additional packets will reach the receiver, resulting in an increase in the observed packet-rate at the receiver, and the sender’s transmission window will grow. Of course, if the available bandwidth decreases due to congestion, drops will occur and the sender will enter the congested state and decrease its window size geometrically. While TCP's generated load oscillates around the around available network bandwidth, SCP converges to a rate that matches the available network bandwidth. Since the protocol stabilizes, the application can adapt the quality of the media stream in order to generate data at a rate the matches the available capacity, providing the best interaction possible with the available bandwidth.

Further, SCP reduces latency in two ways. First, SCP does not repeatedly overload the network with TCP-like oscillating probing techniques. Because TCP relies solely on the absence of acknowledgements as congestion indicators, it attempts to determine the available network capacity by increasing its window-size until network buffers overflow, causing lost packets. However, filling these network buffers is also a source of latency. The second reason that SCP improves latency is the absence of retransmissions. With TCP, the authors encountered situations in which repeated drops and retransmissions resulted in up to 70 seconds of latency. With TCP, a packet may wait at the sender through multiple round-trip times or worse, through multiple retransmission time-outs on the order of seconds. During this time, the sender's window does not advance so the data that lies outside the current send window can not be transmitted. In contrast, SCP does not re-transmit lost data so while the send window's rate of advancement may be restricted while waiting for acknowledgement or time-outs of acknowledgements, it is never blocked for retransmissions.

SCP offers a transport level alternative to UDP and TCP which combines some of the most desirable features (from a streaming media point of view) of each. It regulates the transmission rate in a TCP-like manner while minimizing latency by adjusting the probing mechanism and by not retransmitting lost data. However, SCP has several shortcomings common to responsive protocols. SCP's shortcomings are discussed in section 2.3.3.

2.3.2. Loss-Delay Based Adjustment Algorithm

LDA is an extension to the Real Time Transport Protocol (RTP) to adjust the transmission rate in a *TCP-friendly* way. Although RTP is rate-based, not window-based, LDA seeks to adjust the rate so that it corresponds to TCP's congestion window behavior. Further, LDA uses a model of the expected behavior of a TCP flow that improves upon previous models. That is, the transmission rate responds to periods of congestion with a geometric decrease in the transmission rate and uses an additive increase during periods without congestion to increase the transmission rate. The goal is to transmit data in a *TCP-friendly* way. That is, transmit at the same rate that a TCP source would, given the same round-trip time and loss-rate.

The original model of TCP behavior is based on average loss and delay over the *lifetime* of a connection [Floyd98]. In this model, throughput is given by equation 2.1.

$$r_{TCP} = \frac{1.22 \times M}{RTT \times \sqrt{l}} \tag{2.1}$$

M is the maximum packet length. RTT is round trip time and *l* is the average loss over the lifetime of the connection. r_{TCP} is the rate a TCP connection would be expected to send at under such conditions. However, using this long-term model to make short-term changes in network conditions leads to very oscillatory adaptations [Sisalem97]. The model also has limited value if there is queueing delay or sharing of the bottleneck link with competing connections [Ott97]. Consequently, although this model is useful for general analysis of TCP, it is of limited value for actually controlling transmission rates in actual applications.

As a result of these shortcomings the authors propose the LDA algorithm. The LDA algorithm relies on the RTP control protocol, RTCP, to report the bandwidth of the bottleneck link and computes the RTT and the propagation delay, τ, based on other values returned by RTCP. After a given sampling interval the LDA algorithm computes a new transmission rate by either using an additive increase or a geometric back-off. The rate of increase or decrease is a function of the current transmission rate and the bandwidth of the bottleneck link. If the current rate is a small share of the bottleneck the additive increase is large and if the current rate is a large share the additive increase is small. The intent is to allow flows needing a small share of the link's capacity to converge faster to their fair share. In all cases, the increment is bounded to be no more than one packet per round-trip time. The current rate is simply incremented by the additive increase rate. If losses are detected, the rate is decreased proportional to the indicated losses based on a reduction factor provided as a configuration parameter.

The authors note that streaming media cannot tolerate lots of oscillation in the bandwidth, so they use a smaller reduction factor to decrease the rate of change. Unfortunately, this also decreases the rate of convergence. In one experiment, the algorithm took 300 seconds for 4 flows to converge. LDA is noteworthy because of its rate-based approach for an unreliable transport protocol. Also, the use of actual measurements of the bottleneck link, loss-rate, and RTT as factors in the rate-adjustment lead to a less oscillatory response to changes in network conditions.

2.3.3. Analysis of Responsive Unreliable Approaches

There are four major concerns with SCP, LDA, and other proposed new protocols. The first is inertia. Convincing application designers to build their applications on a new protocol is a serious challenge. Even as new applications are built using these protocols a large number of applications will continue to use existing protocols. A second concern is that although these protocols are responsive, they may not be as responsive as TCP, allowing applications built on these protocols to receive higher throughput than a corresponding TCP application. This is balanced by the third concern, that if the protocols are responsive in a TCP-like way, they will also be vulnerable to the same aggressive unresponsive flows that TCP and other responsive approaches are vulnerable to. The final concern is that these protocols adjust the transmission rate independent of the application. Unless the application itself is redesigned to adjust the load it generates to match these adjusted transmission rates these approaches simply have effect of moving the bottleneck out of the network and into the transport protocol. Queues will build up in the end-system, resulting in loss or application-level blocking. This need to adjust the transmission rate at the application level leads to the consideration of application level approaches.

3. Application-level Approaches

Moving up one layer in the protocol stack, there are also solutions to consider at the application level. Many applications that use UDP as their underlying transport level protocol also establish a separate feedback path from receiver to sender. In the case of interactive applications, this second path is already present to carry the media stream of the other participant. In those cases, feedback can be piggy-backed on the media stream. However, there are also applications that do not have a feedback stream and are unaware of the state of the network. Application's responses, if any, to network congestion are considered below.

Responsive applications include some application-level feedback method that allows them to infer information about the current state of the network using heuristics. These applications respond to current network conditions by adjusting the load they generate to better match the available capacity of the network. Unlike the transport level approaches, these approaches are able to adjust the transmission rate in a controlled way by adjusting the parameters of the media stream. This is in contrast to transport level approaches that decrease the transmission rate unpredictably. Responsive applications seek to directly control the manner in which the quality of the media stream changes. Instead of allowing random changes in quality due to loss and latency in the network, responsive applications adjust the quality of the media stream in a controlled way at the sender to eliminate network effects. Moreover, they may also probe the network to attempt to draw more accurate conclusions about the specific types of congestion present.

For example, Talley proposed a novel probing technique to determine the current network conditions and address them properly. He defined two different types of constraints that exist at network bottlenecks [Talley97]. The more commonly recognized constraint is a capacity constraint. A capacity constraint is one where the limiting factor is purely the quantity of bytes that can be transmitted on the bottleneck link. The other constraint, the access constraint, is subtler. An access constraint is one where the limiting factor is the number of packets that can be transmitted on the bottleneck link. Access constraints typically occur when the processor speed in the router is too slow or if there is shared media access. For example, the time required by a router to analyze each packet header may be greater than the time necessary to transmit the bytes that make up the packet. For shared media (e.g. FDDI rings), the time required to access the media may dominate the transmission time. In this scenario, although the outbound link is underutilized, the router is still the bottleneck point. This typically occurs when very small packets are being transmitted. Talley points out that applications may address access constraints by sending larger packets while maintaining the same bit-rate. Bundling multiple frames into a single packet decreases the packet-rate while maintaining the byte-rate, but increases latency. However, the increase latency is less noticeable than a degraded frame-rate. The overall result is little change in quality of the media stream. Applications often mistakenly respond to access constraints with techniques intended to address capacity constraints. However, these approaches often work because many efforts to reduce the bit-rate also reduce the packet rate (e.g., decreasing the frame-rate). Talley proposed a novel mechanism for probing the bit-rate and packet-rate dimensions to determine the optimal operating point for applications. Once this point was determined the application could adjust its packet-rate or bit-rate accordingly.

Reducing the bit-rate or packet-rate of a media stream in a controlled way can be accomplished with media scaling. Media scaling refers to the idea of adjusting the quality of the media stream to adjust the load generated by the media stream. Delgrossi presents a taxonomy of scaling techniques [Delgrossi93]. Scaling may be transparent or non-transparent. *Transparent* scaling is decoupled from the media semantics and *non-transparent* scaling requires knowledge of the media semantics. Non-transparent scaling requires some interaction with the upper protocol layers. It may require adjusting the parameters of the coding (or even recoding a stream). Frequency, amplitude, and color space scaling are examples of non-transparent scaling techniques. Respectively, these methods change the number of discrete cosine transform coefficients, reduce color depths for each pixel, and reduce the number of colors available. All non-transparent techniques require knowledge of the coding technique in order to make changes at the coding level. In contrast, transparent techniques usually operate simply by discarding some, possibly prioritized, units of the data stream. There are two transparent media scaling techniques: temporal and spatial.

Temporal media scaling adjusts the load generated by an application by adjusting the frame-rate. If the network appears more congested the number of frames per second recorded and transmitted can be decreased to decrease the load on the network. As a result, the illusion of continuity will suffer, perhaps perceptibly. However, if the network is congested and packets are being dropped, the frame-rate at the playback station decreases anyway, perhaps in unpredictable ways. Further, network links closest to the sender waste bandwidth by relaying data that never reaches the receiver, potentially contributing to congestion collapse. By adjusting the frame-rate at the sender, one avoids wasting network bandwidth and adjusts the quality of the interaction in a controlled and predictable way. Alternatively, if the network appears less congested, the number of frames per second recorded and transmitted can be increased to improve the illusion of continuity.

Alternatively, spatial media scaling adjusts the load generated by an application by adjusting the number of bytes included in each frame. For example, in video if one reduces the number of pixels making up each image, the quality of each individual image decreases. Both images are acceptable but the higher resolution image is preferred. The application can respond to network conditions by decreasing or increasing the number of bytes in each frame to better match the offered load to available capacity.

Although application level adaptation is an effective means of controlling the way a media stream's quality degrades during congestion, it is still based on the assumption that media streams must degrade when the network is congested. One would prefer no degradation. Moreover, these adaptations only occur after a change in performance is detected at the end-systems. This change must be detected at the receiver and then addressed at the sender, requiring at least one round-trip time. During that interval the network conditions will still effect the media quality in an uncontrolled way. Moreover, as with all responsive techniques, this approach is vulnerable to the effects of unresponsive flows. This is particularly noteworthy because there are a great number of legacy applications without adaptations that will continue to be unresponsive even when applications with adaptations are deployed.

While these techniques allow applications to adjust their load, there are also many applications that do not adapt. They simply send frames at the desired rate and display those frames that arrive. In a sense, interactive applications that take this approach, such as video-conferencing, rely on the participants to act as a feedback and recovery mechanism. If the quality degrades slightly the participants may attempt to recover by asking each other to occasionally repeat things. If the quality suffers too much they may choose to reduce the frame-rate to zero by terminating the interaction. The users may also adjust the transmission rate by adjusting media scaling parameters through a user interface. Unresponsive applications are less vulnerable to aggressive flows as they continue to maintain their load, regardless of network conditions. However, this also means their flows help contribute to congestion collapse. Further, when the network is congested these non-adaptive applications have no control over the way in which the media quality degrades.

4. Integrated End-System Congestion Management

The approaches reviewed thus far all suffer due to a lack of integration. Transport-level approaches simply reduce the transmission rates without providing feedback to the applications (which could be used to adapt the quality of the media stream). Application level approaches must be implemented on an application-by-application basis. Balakrishnan, et al., propose an integrated congestion management architecture [Balakrishnan99].

The central feature of this architecture is a congestion manager (CM) which manages the transmission rates of all flows on a given end-system in a TCP-friendly way. Moreover, the CM aggregates flows by source-destination address so that all flows between a given pair of end-systems share information about the state of the path between the end-systems and share that path in a controlled way. Because the transmission rate is managed in aggregate, and shared between flows according to weights and hints, flows benefit from information learned by other flows. Finally, although the sender's congestion manager performs best when paired with a receiver that also has a congestion manager, the congestion manager does work even when no congestion manager is available on the receiver.

This approach does not replace existing layers such as TCP or UDP. Instead, the congestion manager introduces an additional layer between the transport and network layer that manages the rate at which data is transmitted into the Internet in a TCP-friendly way. Further, this architecture exposes an application-level programming interface that makes applications aware of changes in the transmission rate. Using this information, applications can adapt to network conditions by adjusting the rate at which they generate data (e.g. by adjusting frame-rate). Moreover, because callbacks inform the application just before its data will be sent, the application can choose to send the timeliest data.

This approach is attractive because it offers applications seeking a responsive protocol an alternative to TCP without the overhead associated with reliability. Moreover, the API allows applications to adapt their data stream based on the current transmission rate. Finally, this approach can be deployed incrementally because end-systems using the congestion manager will work with end-systems without the congestion manager. However, during the incremental deployment of such an approach, those systems that use the congestion manager will expose all their flows to the effects of unresponsive flows currently suffered by responsive flows. Moreover, this integrated approach requires modifying or replacing all applications as well as the end-system protocol stacks.

5. Integrating Application Level Prioritization with Router Drop Policies

Although application level adaptations are effective because they adjust fidelity in a controlled way, they have the overhead of requiring separate implementations for each application or media class. Moreover, it takes one full RTT to detect and respond to congestion in this way. In contrast, queue management algorithms in routers can detect congestion and respond immediately by discarding packets. However, the resulting degradation in the media fidelity is unpredictable. A few approaches integrate application level approaches with network support to take advantage of the strengths of each.

To understand these approaches, recall that *non-transparent* scaling requires knowledge of the media scaling while *transparent* scaling is decoupled from the media semantics [Delgrossi93]. Integrated approaches rely on transparent media scaling techniques.

Hoffman offers such an integrated approach that combines application level tagging of data combined with prioritized packet discard policies in network elements that have no direct awareness of the media semantics [Hof93]. The approach relies on tagged subflows that represent different scales of quality where the information necessary to generate the highest quality image is tagged with the lowest priority. During periods of congestion filters in the network discard either individual elements of low priority flows or shutdown those flows entirely leaving the end user with a lower quality stream. For example, an application may separate a MPEG [Le Gall91] stream into separate sub-flows for the I, P, and B frames. There are decoding dependencies between frames. Both B-frames and P-frames reference I-frames for decoding. B-frames also reference P-frames but B-frames are never referenced. Consequently, the B-frames would be the first ones dropped since all other frames could still be decoded despite the loss. Similarly, P-frames would be the next choice if dropping B-frames wasn't sufficient. This approach also allows for the possibility of changing the filtering based on congestion.

Integrating application-level tagging with router based decisions of when packets need to be discarded allows network elements to identify and address congestion. Moreover, this congestion is addressed while degrading the quality of the media stream in a controlled way. Moreover, this approach removes some of the delay found in application-level adaptations. Instead of waiting one round trip time for notification that performance has changed, the router can respond to changes in network conditions as the queue grows. However, this approach can also contribute to a form of congestion collapse because the application maintains load on the links leading to the bottleneck router even though packets arrive only to be discarded. Further, as with all responsive techniques, this approach may also be vulnerable to unresponsive flows. However, if the prioritized drop policy treats packets from unresponsive flows as lower in priority, this could offer protection for the higher-priority sub-flows. The major draw back to this approach is that it requires integrated changes to the routers and the applications.   However, this approach of indicating drop-priorities leads to the more general proposals to address drop preference as part of an architecture for realizing differentiated services, described below [Ferguson97].

6. Router Based Quality of Service

Instead of considering ways to adapt the media stream, we now consider how to adapt the network to reduce the need for media adaptations. Historically, the Internet has been a best-effort network with end-systems expected to behave cooperatively to address congestion. However, that perspective is changing. There are proposals to offer services other than best-effort. Below, two approaches are presented that attempt to provide a quality of service better than simply best-effort. They are integrated services and differentiated services. They also attempt to provide better congestion management for best-effort flows when congestion does exist.

6.1. Integrated Services Architecture

The Integrated Services Architecture (or int-serv) [Braden94] specifies an infrastructure to offer specific levels of quality of service (QoS) to flows by explicitly reserving capacity for flows and assuring this allocation by scheduling the forwarding times of packets in each router along the flow's path. This approach requires some mechanism to reserve resources. Using a signaling protocol, a user or administrator can request the level of service desired for a particular class of traffic. This class may simply be a single flow or may be an aggregation of flows. Each router participating in the signaling protocol then applies an admission control algorithm to determine if there are resources available to meet the requested level of service while continuing to service the currently admitted classes.   If all of the routers along the path from sender to receiver have available resources, the class is admitted. Each router is then responsible for insuring that each class receives its negotiated level of service.

However, integrated services is both algorithmically complex and requires high levels of coordination. It also requires significant state in the routers. Link capacity is typically managed with a packet scheduler. In order for packet scheduling to work routers must maintain state for each class of traffic. This state includes the negotiated service level, the definitions of which flows belong to the class, and data for the scheduling algorithm. Additionally, the scheduling itself can be computationally complex. Further, deployment is hampered by lack of viable options for incremental deployment. The guarantees are only effective if every router along the flow's path conforms to the integrated services architecture so the value of partial deployment is limited. Dynamic routing changes, which are common, significantly increase the complexity of the admission control process.

However, this integrated approach does essentially avoid congestion for those flows with the higher service level. Since the approach only allows packets to enter the network if they are within their negotiated resource demands or if there is excess capacity, only the class causing overload (by exceeding their negotiated allowance) ever suffers any performance short-comings. Moreover, if a given class is in a state of overload that class may be able to employ any congestion avoidance techniques that are appropriate and only need be concerned with interaction with other flows of the same class.

6.2. Differentiated Services

The differentiated services architecture also seeks to offer a form of quality of service but largely within the infrastructure of the current best-effort Internet. This approach represents a midway point between the integrated services architecture with its associated guarantees and the traditional best-effort approach. No end-system signaling or other changes to existing protocols are required. Instead, clients or organizations negotiate long-term service agreements with their internet service providers (ISPs) for specific classes of traffic. A service level agreement may deal with all traffic from a particular client network or it may be more restricted, covering only flows associated with particular source and destination addresses. These agreements are realized through some ISP provisioning scheme. As with integrated services, the possibility of dynamic routing changes complicates the provisioning task. Moreover, service level agreements that specify bandwidths from one host or network to any other network are even more complex to provision. As a consequence, the agreements do not specify guarantees, just an agreement to offer a differentiated service. The clients agree to conform to the service level agreements (i.e. the specification of the transmission rate and burstiness of traffic from that client) and the service-provider promises to minimize drops as long as the traffic is within the negotiated profile. If the class deviates from its profile, packets from that class may be dropped preferentially. If the end-system is responsive and adjusts its load to stay within its profile, then the drops will cease. If the end-system is unresponsive, then the prioritized dropping mechanism will serve to constrain and isolate the class. Note that for many classes, even in-profile classes are not guaranteed to receive their bandwidth allocation. They are merely promised preferential treatment over classes that are out-of-profile. If the network is underprovisioned in-profile packets will also be dropped. (The exceptions to this case are the expedited forwarding services that, if properly allocated, can guarantee performance. However, the expedited forward service is not commonly offered due to the inefficiencies of reserving bandwidth and the complexity of arranging service level agreements across ISPs.)

There are two popular models for realizing differentiated service. In one model, the so-called 1-bit differentiated services [Clark97], as a packet enters the provider's autonomous system, it is examined and tagged as *in-profile* or *out-of-profile* based on the class's behavior relative to its profile. In this scheme the router places all packets in the same FIFO queue and uses a RED-based (discussed in Chapter III) queue management policy called RIO to decide which packets to drop. The mechanism of RIO is also described in Chapter III and briefly here. RIO is RED with IN and OUT. The RIO algorithm maintains two sets of RED statistics that apply to the two types of traffic. For in-profile packets, drop decisions are based on threshold setting compared to the average queue occupancy by in-profile packets alone. Out-of-profile packets are subject to comparisons between lower thresholds and the average queue occupancy by all traffic. Essentially, when deciding whether or not to drop an arriving packet, RIO applies significantly more constraining tests to the out-of-profile packets.

Another alternative is the so-called two-bit approach [Nichols97]. In this scheme clients negotiate premium and assured service profiles. Premium service guarantees delivery of packets with negligible queueing delays if they are within the negotiated profile but it also guarantees that if a flow is out of profile, its packets are dropped to limit the flow to its negotiated profile. Essentially premium traffic has a virtual leased line. If the load exceeds the capacity of this virtual line, packets are discarded. This assures that with proper admission control premium traffic also cannot starve best-effort traffic. Alternatively, packets using the assured service incur the same delay as best-effort traffic, and the strength of the bandwidth allocation depends on how well individual links are provisioned for bursts of assured packets. However, this is balanced by the fact that assured traffic is permitted to exceed its allocation without being automatically discarded so long as capacity is available.

Consider how packets are classified. When packets are transmitted (or when they exit their originating autonomous system) they are marked with a premium bit (P), an assured bit (A), or neither. This packet marking is a function of both the packet matching the filter for a given profile and the traffic class conforming to its traffic specification. If premium traffic exceeds its negotiated profile, the excess packets are dropped. If assured traffic exceeds its profile, those packets are forwarded along with the other best-effort traffic, but not marked as "assured". At the subsequent routers, premium traffic is placed in a separate high-priority queue that is serviced before other traffic. Assured traffic is treated as in-profile traffic in a RIO queue shared with all other traffic where the other traffic is treated as out-of-profile traffic and dropped preferentially.

Although the two models are popular alternatives to realizing the differentiated services architecture, neither is part of the architecture itself. Rather, these are mechanisms for realizing per hop behaviors (PHB). PHBs are just one component of the architecture, which specifies the policies and framework without specifying the mechanisms for realizing these policies. Other components include traffic monitors, markers, traffic shapers, classifiers, and droppers as well as the framework that ties them together. The architecture specifies the field in the IP headers that can be used to associate packets with a particular service level, and some suggested service levels and associated PHBs. The architecture defines the mechanism, service level agreements, for agreeing on service levels between domains as well as the expected behaviors of differentiated service capable and compliant domains. The behaviors include, but are not limited to, policing of packets at a domain's ingress routers and shaping of traffic at a domain's ingress routers to be sure traffic conforms to the service level agreements. In fact, one strength of the differentiated services architecture lies in the fact that it loosely defines a framework for providing different levels of service without specifying the specific services or mechanisms for realizing those services. This approach should allow network service providers to innovate and incorporate new technology into their offerings. However, this strength may also be interpreted as a weakness as it is difficult to evaluate the effectiveness of the differentiated services architecture without a concrete implementation to study. Instead, individual proposals for realizing differentiated services such as the one-bit and two-bit schemes above must be evaluated.

Consider the queue management scheme proposed for both schemes: RIO. Although RIO does attempt to provide better service for packets marked as in-profile, it offers few guarantees. Although RIO is more likely to drop out-of-profile traffic, in-profile traffic may still be dropped when the queue's average occupancy by in-profile traffic grows large. Moreover, RIO offers little control over latency for assured traffic. The latency incurred will be a function of the number of packets (including out-of-profile traffic) sharing the queue. The performance improvements for assured traffic are limited.

In contrast, the premium service offered in the two-bit scheme is very attractive as it offers a promise of bandwidth and low latency. However, in order for this service to hold value it must be priced to limit its use. If the link's capacity is fully allocated for premium traffic, that premium traffic, although policed, could in aggregate starve best-effort traffic. This is contrary to one of the stated intents of the premium service, that it not starve best-effort traffic. Therefore for the premium service to have value its use must be limited.

Moreover, not all traffic types will be willing to accept bounds on throughput (even when excess capacity exists).

Also, note that unlike integrated services, the differentiated services approach requires no signaling and only per-class state (and ISPs can control the number of classes). Configuration and resource allocation is part of a long-term configuration. Per-flow state at every router is unnecessary because packets are marked at their network entry points and carry their identification bits in the packet headers. Differentiated-services offers an attractive model offering simple quality of service without extensive overhead.

How well does the differentiated services architecture addresses the tension between responsive and unresponsive flows? The differentiated services architecture alone does not answer this question. The answer lies in the way in which the differentiated services architecture is used and what profiles are established. Consider an example in which streaming media is tagged as a premium service, TCP tagged as assured, and everything else left as best-effort. The streaming media would receive excellent performance as long as the profiles and service allocations were properly configured. However, if the streaming media were under provisioned (because of the difficulty of predicting maximum transmission rate for a stream), it would suffer loss, even when the network was underutilized. Moreover, even though TCP as an assured service should work well, the assumption in the differentiated services community seems to be that TCP would be in the best-effort traffic category. Since TCP probes the network's capacity, flows would periodically exceed their profile limitations in either the premium or assured service. Moreover, TCP flows seldom have specific throughput or latency requirements. They simply seek to share the available capacity. Unless one assumes clients will pay to have all unresponsive flows classified as premium traffic, both of the differentiated service models leave TCP vulnerable to the effects of unresponsive traffic.

7. Summary

The unregulated nature of the Internet lends itself to congestion. Various approaches have been taken to addressing this congestion problem. One set of approaches focuses on end-systems detecting network conditions at either the application or transport level and adjusting transmission rates to find equilibrium between generated load and available capacity. On the other end of the spectrum are proposals to use admission control and resource allocation to eliminate or reduce the possibility of congestion occurring.

Both transport level and application level approaches have their strengths and weaknesses. Transport level approaches can suffer from their detachment from the actual application. Decreasing the transport-level transmission rate while the application continues to generate data at the same rate may result in a queue forming in the end-system's protocol stack. The resulting queue-induced delay and possible loss due to overflow has the same negative effects on this single flow as queues forming in the routers. However transport-level approaches do benefit from the modularity argument as they should be more straight-forward to distribute and use. In contrast, application level approaches are better integrated with the actual behavior of the application. Changes to the transmission rate at that level can be based on changes to the media encoding or frame-rate, offering a more controlled and consistent change in the media quality. However, these application level approaches suffer from the modularity argument. The more that generic mechanisms such as congestion control are integrated into higher layers of the protocol stack, the more specialized the implementations of these mechanisms become as they are bound more closely to particular applications. As a result, each application or group of applications will require its own solution to congestion control.

The modularitry argument is also one motivation for pursuing congestion control from a network-centric perspective. Since congestion is a problem across many communication layers, the modulartity argument implies its solution should be implemented at the lowest possible common layer. The approaches that integrate application level marking with router-level drop prioritization follow this argument. Each application uses a general mechanism to indicate the relative importance of packets or flows. The routers then determine the network conditions and use the priority information to be sure to drop the least important packets first. This combination of application-level marking with a general drop prioritization in the routers can be an effective way to minimize the impact of drops on media streams. These transport-level, application-level, and integrated approaches all focus on making the protocols more responsive. As such, they still suffer from one of the key problems: responsive traffic's vulnerability to unresponsive and aggressive traffic. Of course, if all protocols adopted responsive techniques then unresponsive traffic would cease to exist, and, thus, cease to be a problem. And, the availability of these new protocols encourages more applications to use responsive protocols. However, until all traffic is responsive steps must be taken to isolate and protect responsive traffic. This is the motivation behind the router-based approaches.

The router-based quality of service approaches also follow the modularity argument. They use admission control and service allocation to isolate class of traffic from one another and insure that classes receive their requested throughput. However, these allocation schemes can suffer from the fact that only those who can afford to pay for these guarantees receive them, leaving the best-effort traffic to compete for the left-over link capacity. And, the problem still exists there because there are mixes of responsive and unresponsive traffic. In the integrated-services approach, overhead is also a factor. Packet-scheduling is complex to implement and it requires additional state in the router. Also, the problems associated with admission control and dynamic routing remain open problems.

Moreover, all of the network-centric approaches and transport-level approaches suffer from the problem of inertia. Changing all the network routers to implement a new policy and the associated mechanisms is overwhelming. Although some of the proposed approaches can be deployed incrementally their effectiveness may be limited. For example, a guaranteed service that is only available in some of the routers along the path a flow traverses can not provide any real guarantee to the end-system.

All of these approaches show promise. However, we choose to approach the problem from the perspective of active queue management, asking, "how effectively can an algorithm offer feedback, bandwidth allocation, and controlled latency by only managing the queue occupancy in the router?" Toward that end, the next chapter addresses the related work in active queue management.

III. ROUTER QUEUE MANAGEMENT

This chapter considers router-based mechanisms for controlling congestion and allocating network bandwidth. It is helpful first to consider a reference implementation architecture of a router as shown in Figure 3.1.

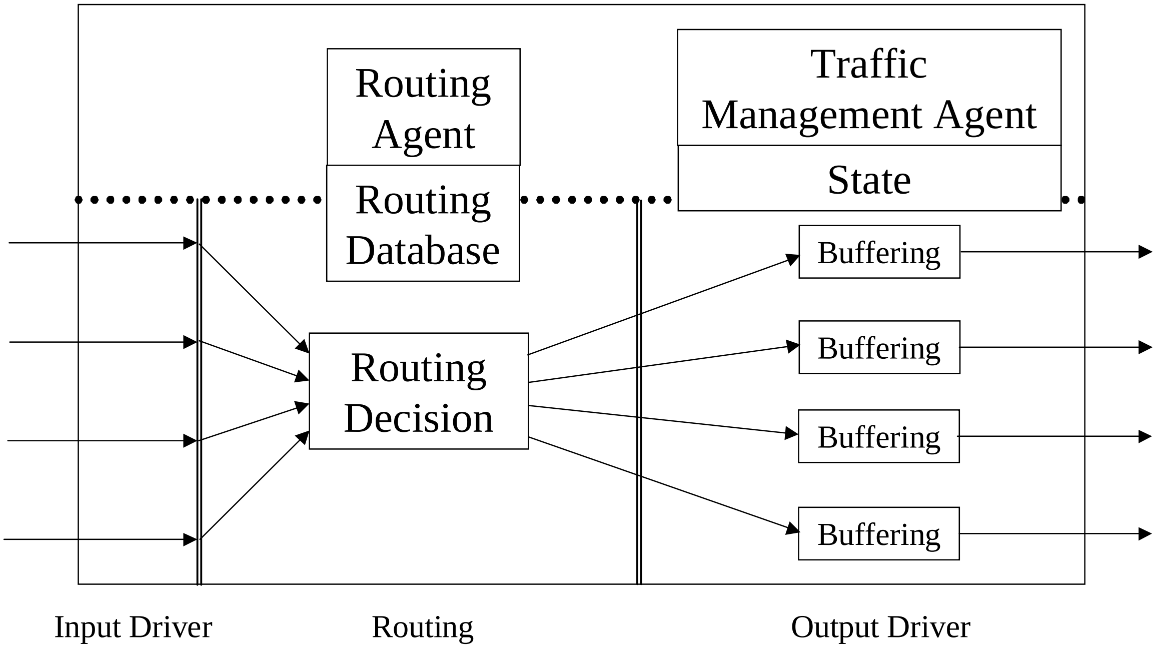


Figure 3.1 Reference Implementation Model for a Router

A router may forward packets between several networks. Although most network links are bidirectional, for the purposes of this diagram and discussion inbound (on the left) and outbound (on the right) links are treated as logically separate entities. A router has two broad functional components, the forwarding engine below the dotted line and the background processing engine above the line. The background processing engine is simply loaded into router memory and executed by a general purpose CPU. Relative to the forwarding functions, the background processing engine rarely executes and is used to maintain and update state used by the forwarding path components. This background processing engine includes the routing agent that maintains the routing database and optional traffic management agents such as admission control or bandwidth management agents.

In contrast, the forwarding path is executed for every packet arriving at the router. The forwarding path can be divided into three major components: the input driver, the routing driver, and the output driver. The function of the input driver is very simple; it assembles a packet from the bits arriving on the inbound link and passes that packet along to the routing component. Routing is the defining function of a router. Using the routing database each arriving packet is directed to the output driver associated with the link for the packet's next hop. This work focuses on the packet forwarding aspects of the router and not the routing decisions. The output driver includes buffers (queues) that are used to temporarily store packets before they are placed on the outbound link. The input driver may also contain buffers but here we only consider the more common case of output buffers. By default, the queues are managed as simple FIFO queues and when these queues are full the arriving packet is dropped. This queueing paradigm is often referred to as "drop-tail" queueing. When the queue fills, arriving packets are discarded until a packet is dequeued from the head of the queue opening space in the queue.

There are two types of algorithms employed in routers to do more sophisticated management of these buffers for congestion control and bandwidth allocation: *queue management* and *packet scheduling*. Ordering of packets is the primary difference between these two approaches. We define queue management approaches as those that deal with when to drop packets and which packets to drop while forwarding packets in the same order in which they arrived. In contrast, we define scheduling as those approaches that reorder the transmission of packets to give some flows or classes of flows priority over others. The recent focus in queue management has been on providing effective feedback to the end-systems while scheduling has focused on issues of resource allocation, specifically link bandwidth allocation, for providing quality of service. However, both approaches are general. Scheduling can certainly provide feedback to the end-systems by discarding packets and, as this work shows, queue management can be used to allocate bandwidth. Further, packet scheduling and active queue management are not mutually exclusive. They can be integrated into a single resource management policy. For example, different active queue management policies may be used to manage each of the individual queues used by a packet scheduler. However, it is helpful to consider the two approaches independently before considering the hybrid approaches.

In this chapter, we discuss queue management at length. First, the general purpose of these queues is discussed along with some of the issues associated with designing a queueing scheme. Next, several active queue management algorithms are considered including their design, and their strengths and weaknesses. Packet scheduling is then briefly reviewed because in later chapters the performance of our active queue management policy will be compared to several other queuing algorithms and that comparison will include a packet scheduling scheme as a "gold standard". Finally, the performance of multimedia is considered for each of the algorithms considered here.

1. Buffering in Routers

Routers contain buffers, implemented as simple, first-in-first-out queues, to avoid unnecessary packet loss and to maintain high link utilization. These queues accommodate transient overloads that are primarily due to bursty packet arrivals. A burst of packets may arrive at a queue when, for example, a collection of packets destined for a common outbound link arrive nearly simultaneously on multiple inbound links. The burstiness could also be due to a burst of packets arriving on a single higher capacity link, all destined to exit the router on a single lower capacity outbound link. In either case, these bursty arrivals result in a load that exceeds the outbound link's capacity. Hence, these two cases are equivalent with respect to the impact on the queue at the outbound link so the source of the burstiness can be ignored, instead focusing on its effect on the allocation of queue capacity in routers. If the average arriving traffic load is less than the capacity of the outbound link, these buffers act as a smoothing filter to allow for significant variance in the load arriving over small time scales (e.g., the time it takes to transmit *k* packets where *k* is the length of the queue.) Network managers attempt to allocate queue capacity sufficient to accommodate the largest burst that can reasonably be expected. This buffering may allow them to keep the outbound link busy with useful work even when there are no packets arriving, thereby increasing the link utilization. Buffering allows routers to both avoid unnecessary packet loss and to maintain high link utilization.

However, when the network is congested, these queues can become a source of latency and delay notification of congestion.

Determining the amount of space to allocate to a link's queue is based on a fundamental trade-off between accommodating burstiness and minimizing queue-induced latency. If the queue is too small, packets will frequently be dropped solely due to the bursty nature of the traffic resulting in lost data and false congestion indicators. If the queue is too large, latency will increase and end-systems will not receive timely notification of network congestion when it happens. The result will be congestion that persists for lengthy periods of time, even when the end-systems are responsive. During periods of true congestion one would like to minimize the amount of time it takes for the buffer to overflow in order to provide notification of congestion (by dropping packets) as soon as possible. Finally, if congestion is persistent, a large buffer causes those packets that do arrive at the receiver successfully to do so with significant delay. Network managers must balance these two conflicting demands when determining the amount of buffering to allocate to each outbound link.

1.1. DropTail When Full (FIFO)

First-in-first-out, drop-tail-when-full was the original queue management scheme used in Internet routers. With this scheme packets are enqueued at the tail of a queue as they arrive and dequeued from the head of queue when there is capacity on the link. Drop-tail is the policy of dropping the arriving packet when the queue is full. (Other alternatives include dropping the packet at the head of the queue.) Drop-tail and FIFO are used interchangeably in this dissertation.

Braden, et al., point out several problems with simple drop-tail-on-full and recommends that Internet routers employ more sophisticated techniques for managing their queues [Braden98]. The two major problems they identify are the problems of lock-out and full-queues.

Lock-out refers to a phenomenon in which the shared resource, link bandwidth, is unfairly consumed exclusively by a small number of flows. The remaining flows are locked-out of (i.e., denied access to) the queue and, consequently, locked out of the outbound link. In this phenomenon the queue is occupied only by the packets from a small number of flows while the packets associated with most flows are consistently discarded. As a result, most flows receive none of the link bandwidth, and starve. This phenomenon occurs because of timing effects which result in some flows' packets always arriving to find the queue full. For example, consider a situation where many sources are periodically sending bursts of packets that in aggregate exceeds the queue's capacity. If these sources become synchronized, all sending nearly simultaneously, the first packets to arrive (e.g. from the source closer to the bottleneck link) will find a queue with some available capacity while the subsequent packets will be discarded. If the same relative order is maintained between the sources, those sources that send first will consistently make progress while the other flows will consistently have all packets discarded and, thus, starve.

Full-queues are queues that are usually occupied to capacity. If bursts of packets arrive to a full queue, many packets are dropped simultaneously. This can lead to large oscillations in the network utilization. If the dropped packets are from different flows there may be synchronized responses (back-off) among multiple flows. Synchronized back-off is a phenomenon in which many sources simultaneously receive congestion notification and reduce their generated load. As a result, the overall load on the network may drop below the capacity of the link and then rise back to exceed the link's capacity resulting in a full queue and once again leading to simultaneous drops. This oscillating behavior is exactly counter to the buffer's intended function, acting as a smoothing filter. Note that *full* queues and *long* queues are not necessarily equivalent. A long queue is one containing a large number of packets, regardless of capacity, while a full-queue is one containing the maximum number of packets it can hold. Long queues cause problems because of queue-induced latency. However, the fact that a queue is long says nothing about its available capacity. A high capacity queue may have many packets enqueued but still have a great deal of unused capacity, leaving it with capacity to accept newly arriving packets and allowing it to perform its intended function of accommodating packet bursts. While full queues may also be long queues, the length of the queue is not the primary problem with full queues. A queue that is usually full is not able to perform its primary function of accommodating bursts.

1.2. Active Queue Management Concept

Active queue management (AQM) is the concept of managing the queue to avoid full queues and lock-out. Most AQM policies take steps to detect congestion based on recent queue behavior. They then drop packets early, before queues overflow, with the intent of providing early feedback to responsive flows and to help maintain lower average queue occupancy. For responsive flows, AQM in principle offers the following advantages:

1. It reduces the number of packets dropped in the router. Limiting the number of drops is important for many reasons and accomplished in several ways. As active queue management maintains lower average queue occupancy and avoids the problem of full queues, it allows bursts of packets to be enqueued without loss. It is also important to reduce the number of drops because any given TCP flow has difficulty dealing with multiple drops and its time-out mechanism may be invoked. Finally, packet drops result in retransmissions by the TCP sender, leading to reduced link efficiency at upstream links. If, instead, a fraction of the arriving packets are dropped before the queue overflows the responsive flows should be able to adjust their load so that average aggregate load approaches the capacity of the link. With most of the AQM techniques individual flows see a smaller number of drops than they otherwise would and there is less need for retransmissions.

2. Provide lower queue-induced latency. AQM maintains average queue occupancy significantly less than with FIFO. As a result, the queue-induced latency is less than that for a full queue with the same capacity. This is important for interactive applications where end-to-end latency is a serious performance consideration.

3. Avoid lock-out. Most arriving packets will be enqueued because the queue is not full so the chances that a single flow will repeatedly have most or all of its packets dropped is low. Instead of a small number of flows having many of their packets dropped when there is congestion, many flows will see a small fraction of their packets dropped, encouraging all responsive flows to adjust their generated load.

2. The Evolution of Active Queue Management Policies

To understand the issues and decisions involved in the design and operation of an AQM policy it is useful to consider the evolution of queue management in routers. This section considers different queue management policies, the issues addressed by each policy, and the issues uncovered by each policy for future consideration.

Early active queue management policies focus on solving the problems of lock-out and full-queues. They seek to avoid lock-out by finding techniques to more evenly distribute drops across all flows. To avoid full queues, the algorithms discard some packets before the queue reaches overflow and, rather than drop the arriving packet when the queue is full, they consider dropping other packets. There were two techniques in this regard, "drop-front-on-full" and "random-drop-on-full".

2.1. Solving the Problems of Lock-out and Full-queues

Drop-front-on-full is very similar to drop-tail-on-full policies. Its behavior can easily be deduced from the name. Like drop-tail, the decision of when to drop is based on queue overflow. However, instead of dropping the arriving packet, drop-front discards the oldest enqueued packet, the packet at the head of the queue. Drop-front addresses the lock-out problem observed with drop-tail [Laksham96]. If some of the packets in a burst of packets arrive to a full queue there is a high probability that the subsequent packets in the burst will also be dropped. Alternatively, if the packets are dropped from the front of the queue there is a good chance that the packets dropped will be a subset of a larger burst, be parts of two different bursts, or a spread of packets from multiple flows that have other packets elsewhere in the queue. So drop-front helps avoid lock-out.

Additionally drop-front gives marginally earlier feedback. This is because the packets that are dropped would have arrived at the receiver earlier than the packets that arrive when the queue is full. Thus, the loss will be detected earlier than loss of later packets. It also decreases latency. Normally packets pass through the queue at a rate based on the speed of the outbound link. With drop-front when the queue is full, packets may pass through the queue at the speed of the greater of the arrival rate or the service rate. This reduces the queue-induced latency for those packets that are ultimately transmitted. Note that data at the head of the queue is older than data at the tail of the queue. So, the decision to drop old data and enqueue new data may be well suited to applications that are delay sensitive and can tolerate drops. Real-time media applications, for example video conferencing, are concerned with timeliness, and delays impede interactivity.

By dropping something other than the arriving packets, drop-front solves the lock-out problem. However, because it waits for the queue to fill before dropping packets, it still has the problems associated with full queues.

Random Drop [Hashem89] is another policy which addresses the lock-out problem by enqueueing the arriving packets and discarding a random packet within the queue. By choosing a packet at random this algorithm avoids dropping consecutive packets which were potentially part of the same burst. Choosing packets at random also helps to distribute drops across all flows, giving some feedback to most flows, rather than giving lots of feedback to a small number of flows. The random dropping also means that the likelihood a given flow will have a packet dropped is in direct proportion to the number of packets that flow has enqueued. A drawback of this approach is complexity. For a traditional implementation of a queue as a linked list, random drop has more overhead than drop-tail or drop-front as it requires an O(N) walk of the queue for every drop compared to the O(1) operation in the other policies. Further, random drop still drops packets only when the queue is full. Thus, both random drop and drop-front solve lock-out but still have problems associated with full-queues.

Early Random Drop [Hashem89] drops packets before the queue fills to address the problems associated with full queues. It also uses randomization to avoid the consecutive drops of arriving packets that can lead to lock-out. Of course, if the decision to drop packets were a binary one based solely on the current queue size, this technique would simply be addressing the problem of *long* queues by reducing the queue's capacity and would fail to address the problem of *full queues*. Ideally an algorithm would allow the queue to fill in response to a burst while trying to avoid allowing the queue to fill too quickly because of congestion. Unfortunately, it is not easy to distinguish between the two cases so a solution must be found that balances the two concerns. To do this, the designers of early random drop make the drop decision probabilistic. When the queue size exceeds a threshold, all arriving packets are considered for dropping with the same, fixed, drop probability. Whenever a packet arrives to a queue that is occupied beyond the threshold value, a random value is selected from a uniform [0,1] distribution and compared to the drop probability to determine whether or not to drop the arriving packet. (Note that the term random is overloaded in active queue management. It may mean selecting a random packet from the queue, as with random drop, or it may mean performing a probabilistic test before dropping the arriving packet.)

By using a probabilistic drop test, the queue can continue to grow up to its allocated capacity, but some packets are discarded before the queue is full. As a result, some of the flows get feedback before the queue fills and if flows are responsive the queue's growth rate is reduced. Thus, probabilistic dropping signals congestion while still maintaining capacity for bursts. Moreover, with the probabilistic approach, all flows should have the same percentage of their packets dropped. This is because the probability of dropping a packet from a given connection is roughly proportional to that connection's share of bandwidth through the gateway. That means with flows transmitting at a higher rate have more of their packets dropped, thus providing more feedback to those flows that consume the greatest bandwidth. Since the drops are more widely distributed in time compared to drop-tail, the probability of consecutively arriving packets being dropped is lower and there is less likelihood of synchronized back-off.

Early random drop does address both lock-out and full-queues. However, the techniques used for identifying congestion (monitoring instantaneous queue occupancy) and providing feedback (dropping a fixed percentage of arriving packets) need refinement. This approach assumes that instantaneous queue occupancy is a good indication of the state of the network. Unfortunately, high occupancy may reflect either a transient overload due to burstiness or the onset of persistent congestion. As a result the instantaneous queue occupancy is not a very strong indication of persistent congestion. The algorithm also assumes dropping a fixed percentage of the arriving packets is an effective indication of congestion. Unfortunately, a constant drop-rate does little to distinguish between slight and severe congestion. In the original work the author noted that adding more dynamic elements to the algorithm, such as a dynamic threshold and/or a dynamic drop probability in response to network conditions was a consideration for future work. Subsequent algorithms followed these suggestions to focus on providing better identification and notification of congestion.

In summary, Drop-front and Random Drop address the lock-out problem by choosing packets other than the arriving packet to drop when the queue overflows. However, because they each wait until the queue is full to act, neither addresses the problem of full queues. In contrast, the early random drop algorithm drops some packets probabilistically before the queue overflows. By dropping packets early, early random drop addresses the problem of full-queues and by only dropping a random sample of the arriving packets it addresses the problem of lock-out. However, none of these algorithms are particularly concerned with accurately detecting congestion or with distributing the feedback evenly among the flows. Their primary focus is simply to manage the queue's size without creating any of the pathological conditions that result from the FIFO queueing policy.

2.2. Providing Notification of Congestion

All of the techniques discussed up to this point have focused as much on managing the queue to avoid ill effects as much as they have with notifying the end-systems of congestion. Moreover, the most common method for notifying the end-systems has been allowing the end-system to infer congestion when packets fail to arrive because the router discards them. However, traditionally packets were not discarded with the intention of signaling congestion but with the intention of managing the queue occupancy. Moreover, packets may be lost due to other reasons including network failures. However, managing congestion with congestion management has been popular because these techniques can be deployed without any changes to the existing end-to-end protocols. However, an alternative approach is to provide explicit notification of congestion to the end-systems. This can be done by sending explicit control messages to the affected hosts. However, the most common technique is packet marking. *Packet marking* refers to the technique of marking a bit in the header of data packets to indicate the presence of congestion. This technique separates congestion notification from data loss. Data is only lost when queues overflow because congestion is persistent. In addition to the use of explicit notification, better designs also address issues of more clearly identifying congestion and distributing this notification to all flows. The active queue management policy, RED, includes these concerns among its many goals. First, we consider an explicit feedback mechanism, ICMP source quench, and two noteworthy packet marking approaches, DECbit and ECN. This is followed by a review of the active queue management algorithm, RED.

2.2.1. ICMP Source Quench

The capability to send explicit feedback to sources has been present in the Internet since 1981 via the Internet Control Message Protocol (ICMP) source quench message [Postel81]. Whenever a router discards a packet because of a full queue, ICMP specifies that the router should send a source quench message to the source that generated the message. The source is then expected to reduce its load. However, in practice this mechanism has been ineffective. Generating the ICMP source quench messages demands router resources during a period of congestion. Moreover, the messages use link bandwidth on the return path. And, in many cases these messages have no effect as they are ignored by end-system protocols. Due to these factors and the fact that TCP's congestion control mechanism has proven effective using loss as an implicit congestion indicator, later specifications of router requirements recommended that routers not generate ICMP source quench messages [Baker95]. Moreover, if the routers do generate source quench messages, they should limit the frequency with which they send them. Subsequent approaches to provide explicit notification avoided the overhead of generating and transmitting feedback packets by choosing to mark packets that were being forwarded to the receiver and expect the receiver to notify the sender.

2.2.2. DECbit

One early proposal for explicitly signaling congestion via packet marking was DECbit [Jain88]. This work took a different approach to dealing with congestion; it integrated the router's detection and notification of congestion with the end-to-end protocol. All of the other queue management techniques examined were detached from specific knowledge of the transport protocol. They simply assume the transport protocols were able to respond to loss as an indicator of congestion, or that discarding packets would be sufficient to constrain the network load. In DECbit the router's behavior is not only changed, but the end-system's transport layer behavior is also changed. Unlike packet dropping techniques where notification must be limited because of the trade-off between notification and data loss, packet-marking techniques can be applied to all packets when congestion is detected. Packets are not discarded as long as space is available. Instead, the central issue becomes one of correctly identifying periods when the network is congested.

DECbit proposes the idea of detecting congestion in a router by monitoring the average queue size over an interval of time. The averaging interval is defined in terms of periods when they queue has been occupied ("busy"). A busy period is one in which the queue continuously has packets enqueued. The averaging interval covers the most recent busy period, the subsequent idle period (i.e., no packets enqueued), and the current busy period. When the weighted average queue size is greater than one over this measurement interval, all packets are marked. When the average queue size is less than one over the interval, no packets are marked.

The transport protocol must be aware of the marking strategy. If a packet arriving at the receiver is marked, the corresponding acknowledgement must be marked. Then, the sender should maintain statistics on how many acknowledgements were marked in the most recent window of data. A window of data includes the data from the earliest unacknowledged packet to the packet most recently transmitted. The size of this window effectively establishes a limit on the maximum amount of data the sender can transmit per round-trip time. Adjusting this window size, regulates the load placed on the network. In this algorithm, if more than half of the packets in the most recent window were marked, the window size is decreased exponentially. Otherwise it is increased linearly.

One of the key innovations of DECbit was the idea of monitoring *average* queue size in a router as an indication of congestion. While instantaneous queue occupancy is highly dependent on the burstiness of arrivals, the average queue occupancy is a much better indicator of the state of congestion. The DECbit technique of computing average queue size based on busy and idle periods means that the period the queue size is averaged over can vary significantly. Another alternative would be to use a weighted running average. However, the designers of DECbit avoided a weighted exponential running average because they believed there would be a bias if the measurement interval was much different from the average round trip time.

2.2.3. ECN – Early Congestion Notification

Early Congestion Notification (ECN) [Floyd94] is a proposed IETF standard for marking IP packets to indicate congestion. Unlike DECbit, ECN is simply a mechanism to signal congestion without a corresponding policy statement of how to respond to the indicator. There are recommendations on how to take advantage of the information provided by the ECN marking but the policy and mechanism are separated [Floyd94], [Ramakrishnan99], [Salim00]. ECN simply specifies a protocol for indicating the end-systems are capable of responding to ECN markings and for marking the packet to indicate congestion has been detected.

It is important to note a key difference between dropping and marking as congestion indicators. Packet drops are an implicit and overloaded indication of congestion while marking is an explicit and unambiguous indicator. Congestion is not the only source of packet loss and networks were not originally designed to use packet drops as a congestion indicator. It was noticed that under FIFO queueing with finite queues congestion could be inferred from packet loss. As routers were not attempting to recognize congestion, it was perfectly reasonable for end-systems to only use hints and implicit indicators to adjust their load. However, when designers began to detect congestion by monitoring the average queue size, the next logical step was to explicitly signal this congestion.

The major benefit of marking packets instead of dropping them is, of course, that congestion can be detected and addressed without losing data. This is particularly important for low-bandwidth flows or flows in slow-start as they may require a time-out to detect loss. As a result, low bandwidth flows would sit idle for multiple round-trip times because of one packet loss while higher bandwidth flows continue to make progress. Avoiding loss is equally important to latency sensitive applications that cannot afford to retransmit data. However, the major shortcoming of marking packets is that unless the infrastructure is in place to interpret and respond to the marking, it has no value. As a result, routers must continue to use the implicit technique of dropping packets unless they know the end-systems are capable of dealing with the markings. That is one of the strengths of ECN; all packets signal whether or not the end-systems are ECN-capable or not. (Issues of fraudulent use of ECN bits are ignored.) ECN's major deployment to date has been as part of Random Early Detection routers.

2.2.4. RED – Random Early Detection

Random Early Detection (RED) was proposed by Floyd and Jacobson. RED combines the ideas of:

- probabilistic drops,
- early notification,
- better congestion detection by monitoring average queue size,
- a drop probability associated with the average queue size, and
- optionally, marking packets instead of discarding them.

RED provides feedback to responsive flows before the queue overflows in an effort to indicate that congestion is imminent instead of waiting until the congestion has become excessive. This scheme also distributes the drops more fairly across all flows.

RED can indicate congestion in one of two ways, by dropping packets or by marking packets. The marking technique depends on the transport protocol but the most common technique uses the Early Congestion Notification (ECN) marking with TCP. However, ECN is not yet widely deployed in end-systems so the common implementations of RED will use ECN only if the packet to be marked/dropped is tagged as ECN capable. Otherwise, it will use implicit notification; the packet will be dropped. For ease of discussion, henceforth only dropping will be considered as the feedback mechanism.

RED monitors congestion by computing the average queue size. It tracks the queue occupancy as a weighted exponential running average. In this way, both the long-term behavior of the queue as well as the recent behavior is factored into the drop decision.

This avoids congestion detection that is overly sensitive to transient increases in queue size caused by the bursty nature of the Internet.

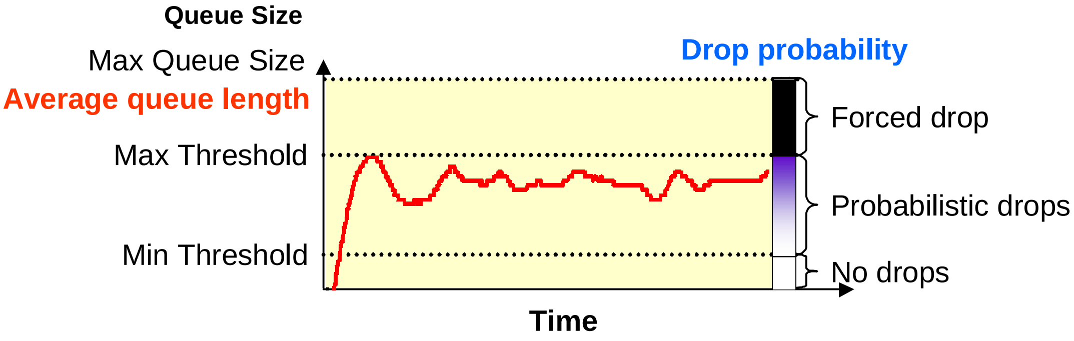


Figure 3.2 RED's Packet Drop Modes

RED's packet dropping decisions are mode-based. Figure 3.2 illustrates the ideas behind the RED algorithm. This figure shows the instantaneous (blue) and weighted average (red) queue size (in packets) over time. The current mode, indicated on the right hand side, is determined by the relation between the average queue size and the minimum and maximum thresholds. When the average queue size is less than the minimum threshold this indicates no congestion so no action is taken. This is *no drop* mode and the probability that an arriving packet will be dropped is zero. In this mode arriving packets are always enqueued. At the other extreme, when the average queue size is greater than the maximum threshold, or if the queue is full, the algorithm infers persistent and severe congestion. All arriving packets are dropped. The probability an arriving packet will be dropped is one. This mode is referred to as *forced drop* mode. Finally, when the average queue size is between the two thresholds the algorithm operates in a *probabilistic* (i.e., random) drop mode. In this mode, arriving packets are dropped at random. The probability that a given packet will be dropped ranges between zero and one as a function of three parameters: max_p, the current average queue size, and *count*. The input parameter, max_p, determines the maximum probability that two consecutive packets will be dropped while in probabilistic drop mode. The variable, *count,* tracks how many packets have been enqueued since the last packet was dropped. Let us consider this function in greater detail.

RED's goal is to give some feedback to flows as congestion occurs and to distribute feedback evenly among flows. The average queue size is an indicator not only that congestion seems to be occurring but also of its severity. Severity here may refer to duration or how extreme the overload is. In either case, associating average queue size with congestion severity is a reasonable heuristic. An average queue size near the minimum threshold indicates congestion may have just begun or that the load is only slightly greater than the capacity. An average queue size near the maximum threshold indicates the congestion may be persistent or the recent load is much higher than the available capacity. The more severe the congestion is, the more feedback is needed. In probabilistic drop mode, the drop probability is linearly related to the average queue size. Additionally, in order to avoid consecutive drops (and resulting phenomenon like global synchronization or lock-out) the function limits the likelihood that two consecutive packets will be dropped to max_p. When the previous packet was dropped and *count* is zero, the maximum probability is max_p. As *count* increases, the drop probability also increases. The effect of count can actually lead the drop probability to reach one hundred percent quickly so some packets will always be dropped while in probabilistic mode. A subsequent variant of RED proposed a more gentle transition from probabilistic drop mode to forced drop mode [Floyd97c]. The original design has a discontinuity in the drop probability, from max_p to 1, when the average queue size exceeds Th_{Max}, Instead, a second linear transition ranges the drop probability from max_p to 1 as the average queue size ranges from Th_{Max} to twice Th_{Max}. This variation has been implemented in [Rosolen99]. This variation avoids severe discontinuities in the feedback delivered to the end-systems but avoids increasing the packet dropping probability too quickly during periods of moderate congestion, yet quickly increases the drop probability during severe congestion in order to constrain the queue size.

In summary, RED's innovations include the use of a weighted running exponential average. The use of the average means that the long-term behavior of the queue is considered but recent behavior counts more. Additionally, RED's choice to relate the probability that an arriving packet will be dropped to the current average queue size allows RED to signal not only congestion, but the severity of congestion. The choice to relate the drop probability to the number of packets that have arrived since the last drop insures both that some feedback is given whenever the algorithm is in probabilistic drop mode and that consecutive drops are minimized when congestion is not severe. As with Early Random Drop, dropping packets early solves the problem of full queues and also allows the queue to continue to accommodate bursts. Since packets are dropped before the queue actually fills, a longer overload is required to fill the queue. Moreover, because most flows are assumed to be responsive, RED can rely on the flows responding to dropped packets to adjust their load to alleviate the congestion, allowing the queue to drain.

RED's probabilistic drop function helps solve the lock-out problem. By making the drop process random instead of binary, RED is able to spread the drops across many flows, giving most flows some feedback while still allowing most flows access to the queue. Over a given interval, all flows should see the same percentage of their packets dropped. If the average drop probability over an interval were 5% packet loss, a flow sending 100 packets/second during that interval should see 5 packets/second dropped while a flow sending 20 packets/second should only see 1 packet/second lost. The designers of RED felt that this was a reasonable notion of fairness. It certainly greatly improved upon the complete lack of fairness in drop-tail where some flows may have 0% packet loss while others may see 100% packet loss.

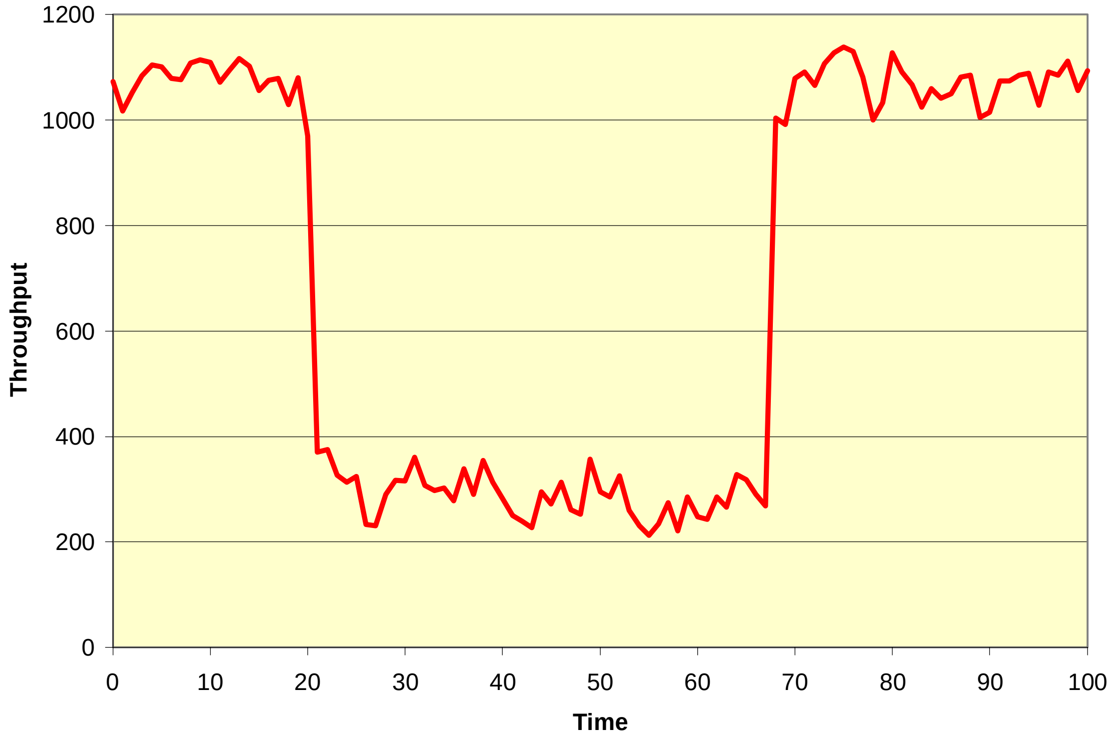


Figure 3.3 Aggregate TCP throughput (KB/s) over Time (seconds) with RED in the Presence of an Unresponsive, High-Bandwidth UDP Flow

However, RED operates on the assumption that all flows are responsive. If a flow has some of its packets dropped, RED assumes that the flow will reduce its load. Unfortunately, unresponsive or misbehaving flows represent a growing fraction of the traffic in the Internet. Under all of the queue management policies considered so far, when these flows are mixed with responsive flows they may steal bandwidth from their responsive counterparts. Figure 3.3, taken from the experimental setup described in Chapter 5, gives an example of this. It shows the performance of TCP in the presence of an unresponsive flow when using the RED algorithm to manage the queue on the bottleneck link. Many TCP flows are active throughout the interval shown, but from time 15-70 seconds, a single, high bandwidth UDP flow is introduced. The UDP flow is able to consume almost all of the network's capacity as the TCP throughput drops to nearly zero.

To understand the starvation effect, consider the feedback loop that occurs with RED. All flows have the same fraction of their packets dropped. The responsive flows respond by reducing their generated load in half. The unresponsive flows continue to send at the same rate. The overall load on the link is reduced so the overall drop-rate decreases. As a result, the unresponsive flows have a smaller fraction of their packets dropped and their overall throughput increases. Meanwhile, the responsive connections still detect loss and the cycle repeats until the unresponsive flows either use almost all of the link's capacity or they have received all of the capacity they need, leaving responsive flows to share the leftover capacity. The TCP throughput does not decrease all the way to zero because the congestion windows only decrease to one segment and some flows are always trying to send data. Moreover, in this experiment there are many TCP flows so in aggregate their load, even with very small transmission windows, is still measurable. The same phenomenon occurs in all of the policies discussed so far. All of these methods provide feedback to the responsive flows encouraging them to reduce their generated load while aggressive flows ignore such feedback, maintaining their load and consuming the freed capacity.

2.2.5. Summary

Better notification of congestion is an important goal of active queue management schemes. It is important that congestion be accurately recognized and end-systems notified in an accurate, timely, and low-impact manner. ECN and DECbit both provide a low impact notification of congestion in terms of packet loss. Both offer a packet marking scheme which allows end-systems to detect congestion explicitly, without requiring that packets be discarded. As a result, data loss and the associated retransmissions are reduced and notification does not have to be limited to limit data loss. However, explicit marking schemes require modification of the end-systems to recognize and respond to this notification. Because of the inertia that must be overcome, these schemes are being deployed slowly and packet-drops continue to be a key implicit notification of congestion.

DECbit and RED focused on the problem of how to accurately identify congestion. Like prior queue management schemes, both identify congestion by monitoring the behavior of the queue. However, while their predecessors examined the instantaneous queue length, RED and DECbit, consider the average queue length. By examining the average, they consider the recent rather than the current behavior of the queue. Further, RED increases its notification (by increasing the drop probability) as the average queue size increases. In this way RED gives some indication of the severity of the congestion in addition to its simple presence. The RED algorithm will be examined in detail in section 3.2. RED, DECbit, and ECN all were important contributions to the accurate identification and notification of congestion.

2.3. Non-TCP-Friendly flows

Any flow that does not respond to drops as an indicator of congestion by employing a congestion control mechanism that is at least as reactive as TCP is considered to be non-TCP-friendly. Non-TCP-friendly protocols are a superset of unresponsive flows. Non-TCP-friendly flows represent an additional vulnerability for AQM techniques. However AQM can also be enhanced to address this problem. Beyond streaming media and multicast bulk data transport [Floyd95b], non-TCP-friendly flows arise from the proliferation of protocols and implementations of protocols. Some TCP implementations do not implement the congestion avoidance mechanisms properly. In other cases, vendors are rumored to intentionally back-off slower and recover faster than specified in an effort to offer a faster TCP than other vendors. Non-TCP-friendly flows can lead to a form of congestion collapse. These flows can dominate the link's capacity because the TCP-friendly flows will reduce their load more aggressively when they detect drops while the other flows may maintain their load, consuming the freed capacity.

Recall that drop-based congestion management techniques have an underlying assumption that the flows transiting the queues and links are responsive. Responsive flows will respond to drops by decreasing their generated load and may respond to periods without drops by increasing their generated load. However, if responsive flows are mixed with unresponsive flows, then when the responsive flows reduce their load the unresponsive flows simply have more capacity available for their use. As a result the congestion may be maintained, forcing the responsive flows to even further reduce their generated load until unresponsive flows dominate the link. The same principle applies when TCP-friendly flows share the link with flows that are responsive, but do not reduce their load as aggressively as TCP.

Recently, the Internet Engineering Task Force has made several recommendations to improve Internet performance [Braden98]. Foremost among their recommendations was a call for widespread deployment of RED or similar AQM policies to address the problems previously discussed. Their second recommendation was for continued research into mechanisms to identify and address problems with non-TCP-friendly flows. Three approaches to dealing with unresponsive flows are outlined below. First, FRED is considered. FRED attempts to insure all flows get roughly equal shares of the link's capacity. Then comes Floyd & Fall's work which focuses on using the RED drop history to identify different types of problematic flows. Finally, a policy called RED with In and Out (RIO) is considered. RIO uses RED to offer differentiated services, focusing on whether or not traffic conforms to service profiles to decide whether to perform strict or relaxed drop tests on the packets.

2.3.1. FRED

Flow Random Early Drop (FRED) was proposed as a mechanism to provide fairness between flows in the network [Lin97]. Previous AQM policies approached queue management as an effort to manage the behavior of the queue and to provide consistent feedback to responsive flows. FRED's developers approached the problem from a different perspective. They begin by dropping RED's implicit assumption that most, if not all, flows are responsive. Then they treat the problem as one of resource management through active queue management. By managing which packets gain access to the queue they can manage how the capacity of the outbound link is allocated.

The designers identify three types of flows: robust, fragile, and non-adaptive. Robust flows are responsive flows that have a significant throughput, such that their response to a small number of drops is simply to reduce their generated load. In contrast, fragile flows are those that are also responsive but are extremely sensitive to dropped packets. For example, a very low bandwidth TCP flow with a small RTT may have a very small congestion window. As a result, a single lost packet is likely to be detected only through a retransmission time-out. While only one packet is actually lost, that flow has no throughput over the time-out interval. Moreover, even when multiple robust flows are involved, if a new flow joins the mix, that flow starts with a small window so it may be trapped in slow-start even if its packets are being dropped at the same rate as other flows. Both fragile flows and robust flows in slow start may suffer much more severely from a given drop-rate than other responsive flows that have their transmission-rate/window-size established. The third type, non-adaptive or unresponsive flows, was not addressed in the design of prior AQM policies. One may assume that since unresponsive flows are subject to the same drop-rate as responsive flows their bandwidth will be constrained. However, as previously illustrated, unresponsive flows may maintain the load they place on the network and if so, cause responsive flows to back out of the way. As a result, unresponsive flows are allowed to consume as much of the link's capacity as they wish.

While RED provided a notion of fairness by dropping an equal percentage of the packets from all flows, the designers of FRED noted that this approach did little to offer fair bandwidth allocation, particularly when the assumption that all flows are responsive was false. The designers of FRED suggest that a congestion management technique should insure that all flows receive a fair share of the network's capacity. A fair share is defined as the minimum of a flow's desired capacity or $1/n^{th}$ of the link capacity when n flows are active. They also propose that fragile flows should be protected from probabilistic drops because their response is much more severe than other flows. Robust flows should receive good feedback but be isolated from the effects of unresponsive flows. Toward that end, unresponsive flows should be identified and strictly constrained to a fair share of the network capacity.

FRED accomplishes these goals by monitoring the performance of all active flows with per flow statistics. By keeping statistics for each flow the algorithm can identify fragile flows (those with a very small number of packets in the queue) and protect them from probabilistic drops. Further, it can constrain each flow to its fair share of the queue during periods of congestion. Finally, by recording the number of times that a flow exceeds its fair share, the algorithm can identify misbehaving flows. They can then be tightly constrained to limit their effect on well-behaved flows.

One of major contributions of FRED was the perspective that assumes that fairness dictates that all flows should have access to an equal share of the link's capacity. This is in contrast to RED's perspective that assumes constraining all flows by an equal percentage of drops is fair. Floyd & Fall and RIO also offer different perspectives on fairness.

FRED is also noteworthy as one of the first Active Queue Management policies to 1) consider unresponsive flows in its design, 2) maintain per flow statistics, and 3) penalize unresponsive flows.

2.3.2. Floyd & Fall

Floyd and Fall also offered an approach to deal with misbehaving and unresponsive flows [Floyd98]. They propose that the router should identify different classes of flows and deal with them by preferentially dropping packets from unresponsive, high-bandwidth, or non-TCP-friendly flows. They propose techniques for identifying the level of responsiveness of a flow based on the recent drop history under the RED algorithm. The authors derive the sending rate of a TCP flow, *T*, expressed in terms of the minimum round trip time, *R*, the drop rate, *p*, and the maximum packet size, *B*, in bytes as:

$$T \leq \frac{1.5\sqrt{2/3} * B}{R * \sqrt{p}} \tag{3.1}$$

TCP-friendly flows are those whose sending rate can be predicted using this equation. By periodically determining the percentage of the total dropped packets that belong to a given flow, the algorithm can deduce the flow's arrival rate both relative to the other flows and relative to that flow's behavior in previous sampling intervals. By examining the way that flows adjust the arrival rate in response to drops the algorithm can determine which flows are responsive, unresponsive, or aggressive. Comparing the number of drops for a given flow relative to other flows can determine whether or not that flow is high bandwidth.

They then propose that routers should take steps to deal with these flows. The standard recommendation is that flows that do not behave in a TCP-friendly manner should be constrained severely. The intent of this work is to encourage the use of end-to-end congestion control and discourage the use of applications that fail to use some congestion control technique. The proposal for accomplishing this is to identify flows that do not respond to congestion in a TCP-like manner and constrain those flows.

2.3.3. RIO – RED with In and Out

RIO is an active queue management technique proposed for use in the differentiated-services initiative of the IETF. The differentiated-services model provides a very simple facility for negotiating and supporting service contracts between service providers and end-users. It allows end-users or network administrators to negotiate profiles specifying the type of traffic load they will place on the network. The service provider then promises to give traffic that conforms to its negotiated profile preferential treatment. RIO is an integrated approach that combines service profiles, policing at network ingress points, and a preferential drop policy based on the RED queue management mechanism [Clark97]. As packets pass through network ingress points, they are policed to be sure they conform to the negotiated profile. Those packets that conform to the profile are marked as in-profile while those that exceed the negotiated profile are marked as out-of-profile. RIO recognizes these two specific categories of packets. By applying the RED algorithm differently to each category of packets, the router can preferentially drop out-of-profile packets before dropping those that are in-profile. In this way, RIO provides feedback to responsive flows and isolates the flows that conform to their profiles from the effects of those flows that misbehave. The service profiles and the queue management approach are considered in more detail below.

Flows are associated with a service profile. The profile constitutes an agreement on the behavior of that flow (or group of flows) between the end-system on one side and the service provider on the other. Upon ingress to the service provider's network, packets are tagged as *In* or *Out* of profile by a policing mechanism, such as a token bucket. If the flow is conforming to its service profile all of its packets will be marked as *in-profile*. If the flow is exceeding its profile, then those packets that represent its excess will be tagged as *out-of-profile* and those packets may be preferentially dropped.

The semantics of the queue management are as follows. All packets share a common queue but two separate sets of statistics and configuration parameters are maintained. One set of statistics is maintained to track the queue occupancy only for *In* packets. The other set of statistics applies to both *In* and *Out* packets. *In* packets are evaluated based only on the behavior of *In* packets. *Out* packets are evaluated based on the behavior of the entire queue. *Out* packets are dropped more aggressively by using smaller thresholds, increased drop probabilities, and by basing the tests on aggregate behavior of all packets. In contrast the decision to drop an *In* packet is based on the average queue occupancy by *In* packets compared to a higher threshold setting and a lower drop probability. As a result, out-of-profile packets are generally dropped before in-profile packets are dropped.

RIO isolates in-profile traffic from the effects of out-of-profile traffic. As long as the in-profile traffic does not exceed the threshold on in-profile queue occupancy, no in-profile packets are discarded. However, the relationship between queue occupancy and throughput does depend on the behavior of out-of-profile traffic. Consider an example where out-of-profile and in-profile traffic are maintaining average queue occupancy equal to their respective thresholds. Then, in-profile traffic has average queue occupancy equal to the threshold for in-profile traffic while out-of-profile traffic has average queue occupancy equal to the difference between the thresholds for out-of-profile and in-profile traffic. The ratio between the average queue occupancies determines the ratio between link capacity available to each class.

RIO is noteworthy as it represents one of the first instances of combining classification with queue management. Instead of classifying packets to assign them to different priority queues to be dealt with by a scheduling mechanism, RIO uses a simple binary classification to determine which set of AQM parameters to apply. All of the packets still go in the same queue so there is no need for the complexity of a scheduling mechanism. Moreover, order is maintained. This has positive effects for both multimedia and TCP. Because multimedia packets are usually played out when they are received, out-of-order packets are often discarded to maintain the illusion of continuity (since subsequent frames have already been displayed). For TCP, out-of-order packets can trigger duplicate acknowledgements and unnecessarily trigger congestion control mechanisms. However, the separate statistics allow the two classes to be managed in different ways.

However, unlike most other active queue management techniques, RIO extends beyond the router itself. RIO requires profiles to be established for each flow or group of flows at the network ingress points and for policy meters to be employed to determine if flows conform to their profiles and tag them accordingly. As a result, RIO is able to isolate out-of-profile traffic from in-profile. This results in constraining non-responsive flows effectively. As long as non-responsive flows like multimedia have enough capacity allocated in the negotiated profile those flows will not be penalized as long as they conform to their profile.

2.3.4. Summary

Each of the three approaches in this section approached the issue of fairness and isolation between different types of flows in a different manner than RED. RED simply applied the same drop-rate to all flows in an effort to give equal feedback and equally constrain all flows. FRED took the approach that fairness means allowing all flows to claim equal shares of the link capacity. Floyd & Fall took the approach that TCP is the defining example of being a good citizen and that all other traffic should be encouraged to follow that model. They encourage constraining unresponsive flows. While the first three approaches all gave a particular definition of fairness, RIO provided the most flexible approach, simply constraining those flows that violate their service profiles. In the case of RIO, fairness can be determined during the profile negotiation and the traffic's level of responsiveness is irrelevant if it conforms to the profile.

2.4. Summary of Evolution of Active Queue Management

Originally, queues were added to routers to provide some buffering for the bursts that are common in Internet traffic. The original queue management algorithm was simply a FIFO queue with drop-tail-when-full semantics. However, these FIFO queues have problems with lock-out and full-queues. Drop-front and Random Drop use different forms of randomization to address lock-out. The Early Random Drop continued to apply the idea of randomness by dropping packets probabilistically before the queue filled. By dropping some packets early, when the queue exceeds a threshold, the Early Random Drop policy avoids full-queues.

After lock-out and full-queues were addressed, researchers began to explore other refinements to manage the queue. These refinements included more accurately detecting congestion and giving more controlled feedback. In both DECbit and RED, accuracy resulted from inferring trends by monitoring the average queue size. The recent behavior of the queue was a better indication of congestion than the instantaneous queue size since a single burst may fill the queue but only sustained congestion would keep the queue occupied and increase the value of the average. Further, in RED changing the percentage of packets dropped in relation to the average queue size did a better job of reflecting the severity of the congestion. The end-systems observed more drops when the congestion was persistent or severe. Research on DECbit and ECN also explored packet-marking techniques to provide feedback without data loss. By marking some packets when congestion is detected, but before queues overflow, these packet-marking techniques allow responsive flows to adjust their load and potentially avoid any data loss due to buffer overflow.

Finally, with these refinements in place the research community turned its attention to problems associated with flows that were not TCP-friendly. Recently, with the advent of streaming media, interactive applications, multicast, and point-cast technology, the amount of non-TCP friendly traffic in the Internet has increased. Previous queue management solutions could not protect responsive protocols, such as TCP, from these less responsive flows. FRED, RIO, and Floyd&Fall present three different alternatives to this problem. The designers of FRED propose that all flows are equal and attempt to insure that all flows can have an equal share of the link's capacity if they need it. FRED accomplishes this goal by maintaining per-flow statistics and penalizing those flows that repeatedly exceed their fair-share. In contrast, Floyd & Fall differentiate between flows based on their responsiveness. They propose that TCP, as a good network citizen, should be protected and other flows, whether unresponsive, not TCP-friendly, or simply high-bandwidth should be constrained. They propose extensions to RED queue management to identify the different types of flows so they can then be preferentially constrained. Finally, RIO, detaches the question of what is fair from the mechanism. RIO, as a mechanism to use in the differentiated-services architecture, simply assures that traffic is allowed to use its allocated bandwidth as specified in a service profile. Maintaining two sets of statistics and thresholds, RIO insures that packets that are out-of-profile are subjected to a more aggressive drop policy that those that are in-profile. This issue, of how to effectively manage different types of flows with Active Queue Management remains a key problem and is the focus of this dissertation.

3. Key Algorithms in Greater Detail

This dissertation compares four different queue management schemes: First In, First Out (FIFO), Random Early Detection (RED), Flow Random Early Detection (FRED), and Class-Based Thresholds (CBT). Each scheme implements a different policy for deciding when to discard packets from the router's outbound queue. The different policies result in different levels of feedback for responsive traffic, different levels of constraint on unresponsive flows, and different end-to-end performance and network utilization.

This section describes each of the four queue management schemes to evaluate. Additinally, a packet scheduling scheme, Class Based Queueing (CBQ) is described. CBQ will be used as a gold standard for comparison with the active queue management schemes.

3.1. Drop-Tail (FIFO)

The default queueing behavior in Internet routers has long been drop-tail-when-full. There was no special rationale in providing these queueing semantics. They are simply the default behavior of a finite capacity queue. The focus when these queues were added to routers was simply to have a queue of some form to provide buffering associated with the outbound link. That buffer is there simply to accommodate transient overloads that result from the bursty nature of the Internet. The general rule of thumb in determining the buffer's capacity was simply to allocate space equivalent to two to four times the delay-bandwidth product [Villamizar94]. This would allow for the simultaneous arrival of a TCP window's worth of data for all of the well-tuned flows to be buffered without loss.

Overall, drop-tail buffering has performed well. The primary evidence in support of this claim can be found in the more than one hundred million hosts currently connected to the Internet. However, drop-tail buffering does have some significant performance flaws. As discussed above, drop-tail can lead to lock-out and full queues. Further, it offers no protection or isolation between flows or classes of flows.

3.1.1. Algorithm Description

The algorithm for implementing a drop-tail FIFO queue is a simple modification of the standard queue as a linked list. The code is shown below in Figure 3.4. When a packet arrives, if the queue is not full, the packet is enqueued. If the queue is full, the packet is dropped. When a packet departs, it is dequeued and sent. The **enqueue** and **dequeue** functions are expected to update the value of **q_len**.

```
Constants:
   int   q_limit;             // Upper limit on queue size

Variables:
   int   q_len = 0;           // Current queue occupancy

General Functions:
   void  enqueue(P);          // Enqueue P and update q_len
   Packet dequeue();          // Dequeue P and update q_len
   void  send(P);             // Transmit P
   void  drop(P);             // Discard P

for each arriving packet P {
   if ( q_len < q_limit  ) {  // If queue is not full
      enqueue(P);
   } else {
      drop(P);
   }
} // End for each arriving

for each departing packet {
   P = dequeue();             // dequeue and send the packet
   send(P);
} // End for each departing
```

Figure 3.4 Algorithm for Drop-Tail (FIFO) Queue Management

3.1.2. Evaluation

FIFO scheduling accomplishes its major design goal: accommodating bursty traffic. However it suffers from problems of lock-out and full queues. Further, FIFO provides no isolation or protection between flows or classes of flows. It is possible for aggressive, unresponsive flows to dominate the queue and, consequently, the link’s capacity just as demonstrated for RED in Figure 3.3. The same phenomenon for FIFO is illustrated in

Figure 3.5. This example shows the result of an experiment designed to illustrate this effect on a 10 Mbps link. The plot shows TCP's utilization of the outbound link on a router employing a FIFO queueing discipline. The aggregate throughput of all TCP connections collapses when a single high-bandwidth UDP flow is introduced between time 15 and 70. The experimental environment in which these data were measured is described in Appendix A.

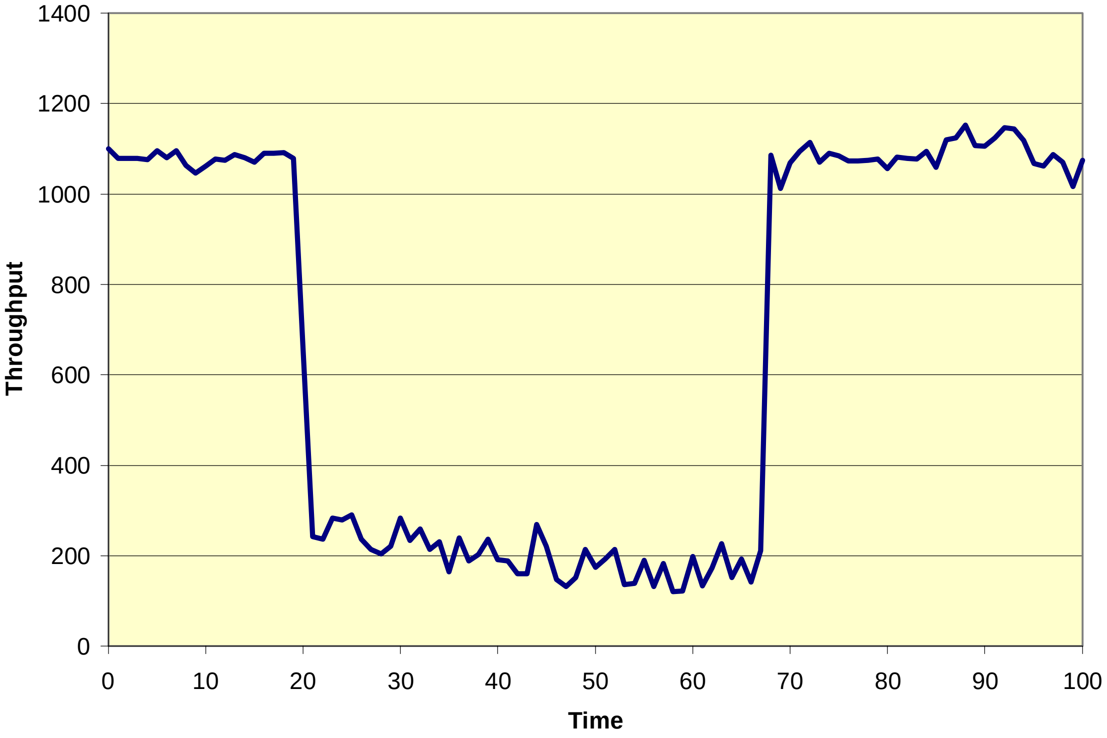


Figure 3.5 Aggregate TCP Throughput (KB/s) over Time (seconds) with FIFO in the Presence of an Unresponsive, High-Bandwidth UDP Flow

While FIFO's wide spread deployment led to the subsequent discovery of these subtle problems, it is important to note that FIFO queueing was never intended to address any of the problems pointed out here. It was simply designed to accommodate transient overloads resulting from the bursty nature of Internet traffic. Its simple design results in very little processing overhead in the router yet helps to avoid a great deal of unnecessary packet loss.

3.2. RED

RED monitors the average queue size and based on its relation to two thresholds, operates in one of three states. These states are *no drop*, *early drop*, and *forced drop* mode. By monitoring the average queue size, RED can react to trends in the queue's behavior and avoid being unduly influenced by the bursty nature of Internet traffic. When the average queue size is small, RED infers that there is little or no congestion and operates in *no drop* mode, simply enqueueing all arriving packets. When the average queue size is moderate, RED operates in *early drop* mode. In this mode, RED probabilistically drops packets based on a random probability that is linearly related to the average queue size. The random factor distributes drops evenly across all flows, avoiding lock-out, and the decision to drop before the queue fills helps to avoid full queues. Relating the drop probability to the queue size allows for some indication of the severity of the congestion to be communicated to the end-systems. Finally, when the average queue size is large or if the queue is full, the algorithm operates in *forced drop* mode. In that mode, all packets are dropped on the assumption congestion is severe and the queue size must be reduced to avoid overflow. The actual algorithm is described below.

```
Constants:

   float w;             // Weight for the running avg
   float max_p;         // Maximum drop probability for RED
   float Th_Max;     // Max threshold
   float Th_Min;     // Min threshold
   float drain_rate;    // The drain-rate for the queue (link capacity)

   int   q_limit;

Variables:

   float q_len = 0;     // Current queue occupancy
   float q_avg = 0;     // Average queue occupancy
   int   count = -1;    // Num packets that have arrived since last drop
   bool  old;           // Indicates transition into early drop mode
   bool  idle;          // Indicates if class has no packets enqueued
   time  last;          // The time when the class went idle

General Functions:
   void  enqueue(P);          // Enqueue P and update q_len
   Packet dequeue();          // Dequeue P and update q_len
   void  send(P);             // Transmit P
   void  drop(P);             // Discard P
   Time  now();               // Return current time
   float random();            // Return random between 0 and 1

Defined functions:
   Bool  drop_early();        // Should we do an early drop
   void  compute_avg()        // update q_avg
```

Figure 3.6 Definitions and Declarations for the RED Queue Management Policy

3.2.1. Algorithm Description

The RED algorithm attempts to detect congestion by tracking the average queue behavior. Figure 3.6 shows the definitions and declarations associated with the algorithm. It maintains statistics including the current queue occupancy, *q_len*, the weighted running average of the queue occupancy, *q_avg*, and the number of packets that have arrived since the last packet was dropped, *count*, and the last time the queue was active, *last*. The algorithm also uses two flags to track the current state. One flag, *idle*, indicates if the queue is currently empty. The other, *in_early_drop*, indicates when the algorithm is in early-drop mode. Parameters also control the precise the behavior of the threshold. These parameters include a maximum and minimum threshold size, Th_{Max} and Th_{Min}, a limit on the queue size, *q_limit*, the weighting factor to assign to each new sample of the queue size, *w*, a maximum drop probability, max_p, and the expected drain-rate of the queue, *drain_rate*.

```
compute_avg() {
   local time  delta_time;
   local int   n;

   if ( q_len ) { // if the queue is still occupied take one sample
      q_avg = ( 1 – w ) * q_avg + w * q_len;
   } else { // if the queue was empty we have to adjust the sampling
      idle = false;
   // if the queue is empty, calculate how long the queue has been idle
      delta_time = now() – last;
   // n packets could have been dequeued during the interval
   // based on drain-rate
      n = delta_time * drain_rate – 1;
   // treat is as if n samples were taken with a qlen of 0
      q_avg = ( 1 – w )^n * q_avg;
   }
} // End compute_avg
```

Figure 3.7 Algorithm for Computing the Average in RED Routers

Computing the Average

A key component of the RED algorithm is its calculation of average queue size. The algorithm used to compute the average can be found in Figure 3.7. The algorithm samples the queue size every time a packet arrives. The average is maintained by giving the current sample a weight of *w* and the previous average a weight of 1-*w*. The intent of the sampling method is for the sample-rate to be, on average, at least as high as the drain-rate. Since every packet that arrives results in a sample, it is obvious how the sample rate matches or exceeds the drain-rate over any interval when the queue is not idle. At the end of an idle period, the averaging algorithm behaves as if a sample were taken at the drain-rate of the queue. To do that the function calculates the number of packets, *N*, that would have been dequeued if the queue had been active. They then update the average with *N* samples of a queue length of zero.

```
for each arriving packet P {
   // count the number of packets since the most recent drop
   count++;
   compute_avg();          // update the average for

   drop_type = NO_DROP;       // Assume we won’t drop this

   // If we’ve exceeded the queue’s capacity
   if ( q_len ≥ q_limit  ) {
      drop_type = FORCED_DROP;
   // If the average is greater than $Th_{Min}$ and there is a queue.
   } else if (( q_avg > $Th_{Min}$) && ( q_len > 1)) {
      // Activate some drop mode
      if ( q_avg > $Th_{Max}$ ) {    // forced drop mode
         drop_type = FORCED_DROP;
      } else if ( not in_early_drop ) {
         // first time exceeding $Th_{Min}$
         // take note (update count & in_early_drop) but don’t drop
         count = 1;          // initalize count as we enter early mode
         old= true;
      } else {       // Not the first time exceeding $Th_{Min}$
      // probabilistic drop mode
         if ( drop_early() ) {
            drop_type = EARLY_DROP;
         }
      }
   } else {
      old = false;      // avg below $Th_{Min}$.
   }

   if ( drop_type == NO_DROP ) {
      enqueue(P);
   } else {
      drop(P);
      count = 0;     //  packet dropped – reinitialize count
   }
} // End for each arriving
```

Figure 3.8 Algorithm for Packet Arrivals in RED Routers

Drop Modes

For every arriving packet, a decision is made whether or not to drop the packet. The drop policy is mode-based. Figure 3.8 illustrates the algorithm for dealing with packet arrivals. The main purpose is to determine which mode to operate in. Initially, the algorithm assumes the packet will be enqueued and the algorithm is in *NO_DROP* mode. If the number of packets exceeds the queue's capacity, then the arriving packet must be dropped. This is a forced drop.   In general, the current mode is determined by the average queue occupancy, *q_avg*. When the average queue size is greater than Th_{Min} and a queue already exists (*q_len* > 1), the queue is in one of its drop modes.   In this situation, if *q_avg* is greater than Th_{Max}, the algorithm operates in forced drop mode and the arriving packet is discarded. Alternatively, *q_avg* must be less than or equal to Th_{Max}, putting the algorithm in early drop mode. In early drop mode, the algorithm performs an early drop test, described below. If the test is positive, the arriving packet is dropped. Otherwise, the packet is enqueued, just as it is if no queue exists or the average is less than the Th_{Min}.

```
drop_early() {
   local float    $p_a$, $p_b$;

   // first approximate drop probability linearly based on q_avg
   // relative to $Th_{Max}$ and $Th_{Min}$
   $p_b$ = $max_p$ * ( q_avg – $Th_{Min}$ )/( $Th_{Max}$ – $Th_{Min}$ );

   // Then adjust the probability based on the number of packets
   // that have arrived since the last drop.
   $p_a$ = $p_b$/(1 – count * $p_b$ );

   if ( random() < $p_a$ ) return true;
   return false;
} // End drop_early
```

Figure 3.9 Algorithm for Making the Early Drop Decision in RED Routers

Early Drop Test

The algorithm for the probabilistic test associated with the random drop mode can be found in Figure 3.9. The parameter, max_p, is an input parameter that defines the upper end of a linear range of packet drop probabilities based on the relation of the average to Th_{Max} and Th_{Min}. It is used to calculate an intermediate value, p_b where

$$p_b = max_p\left(\frac{q\_avg - Th_{Min}}{Th_{Max} - Th_{Min}}\right) \tag{3.1}$$

The value *max_p* is the maximum probability that an arriving packet will be dropped if the immediately previous packet was dropped. In this role, *max_p* contributes to, but does not define, an upper bound on the likelihood of consecutive packets being dropped.

However, the probability that a given packet will be dropped is also a function of the number of packets that have arrived since the last packet was dropped. The variable, *count*, records the number of packets that have arrived since the last packet that was dropped. It is used to compute the final drop-probability, p_a, of the arriving packet as

$$p_a = \frac{p_b}{(1 - count \cdot p_b)} \tag{3.2}$$

Note that while in early drop mode this means that for a given current queue average, a drop is guaranteed at least every ($1/p_b$ – 1) packets. In the most extreme case, where *q_avg* approaches Th_{Max}, p_b approaches max_p. As a result, a drop is guaranteed to occur at least once every ($1/max_p$)-1 arrivals. So for a max_p value of .1, when the average queue occupancy approaches Th_{Max} a drop is guaranteed to occur every 9 packets. In the other extreme, when the average queue occupancy is near Th_{Min}, this guarantee on drop frequency approaches infinity. As the average increases, the number of consecutive packets that can arrive without a drop decreases.

Departing Packets

Figure 3.10 shows the algorithm for departing packets in RED routers. There is simply a bit of extra record keeping that must be done when packets are dequeued. If the packet's departure leaves the queue idle, the time must be recorded and the idle state must be noted for use when the calculating the average.

```
for each departing packet {
   P = dequeue();              // dequeue and send the packet
   send(P);
   if ( q_len == 0 ) {         // If the queue is idle
      if ( not idle ) {
         idle = true;
         last = now();         // record time queue became idle
      }
   } else {
      idle = false;
   }
} // End for each departing
```

Figure 3.10 Algorithm for Packet Departures in RED Routers

3.2.2. Evaluation

The RED algorithm meets its design goals, avoiding the problems of lock-out and full-queues. It also provides effective feedback to responsive flows. It is also worthwhile to note that RED can be deployed without changes to existing protocols or infrastructure beyond the single router that it is being added to. As a result, it can be gradually deployed into the Internet. However, as seen in section 2.2.4, RED is still vulnerable to misbehaving flows. Although a RED router that uses drops as its notification method can constrain unresponsive flows somewhat, the unresponsive flows will still force the responsive flows to reduce their load to near zero.

3.3. FRED

Next, consider the FRED algorithm. Recall that FRED was designed to offer fair access to the link's capacity. By maintaining per-flow statistics and using tests on these statistics in addition to the standard RED statistics and tests, the FRED algorithm seeks to isolate flows from one another. By doing this, the algorithm protects robust flows from unresponsive flows, limits drops for fragile flows, and constrains unresponsive flows. Additionally, robust or unresponsive flows are subject to the standard RED algorithm to provide evenly distributed feedback among the flows. The version of the FRED algorithm used is modified to support many flows (as described in [Lin97]) because in these experiments the TCP flows outnumber the number of buffers available in the router. This version of the algorithm allows every flow to have exactly two packets buffered in order to allow fair bandwidth sharing as the number of active flows approaches the number of packets enqueued. The actual algorithm is considered in detail below.

3.3.1. Algorithm

FRED seeks to provide fairness by assuring that all active flows receive a roughly equal share of the link's capacity. To do this the algorithm attempts to insure that all active flows have equal access to the link by allowing roughly equal occupancy in the queue. Toward that end, the FRED algorithm maintains statistics both on the overall queue behavior and the queue-related behavior of every active flow. In this context, a flow is considered active if it has any packets enqueued. The state that must be maintained in a FRED router is reflected in the variables and constants associated with the algorithm. These definitions can be found in Figure 3.12. There is one instance of each of several constants which are used to configure and fine-tune the algorithm. The values *w*, max_p, Th_{Max}, Th_{Min}, *drain_rate*, and *q_limit*, correspond to their counterparts in the RED algorithm. Every active flow is allowed to have a fixed number of packets in the queue without being subject to drops. This minimum allowable queue occupancy is defined by a new constant, *minq*. As long as a given flow is well-behaved and the algorithm is not in forced drop mode, that flow is allowed to have up to *minq* packets in the queue without being subjected to drops.

Many of the global variables are the same as in RED. They include *q_len*, *q_avg*, *count*, *idle*, and *last*. However, FRED introduces three new global variables. First, the algorithm keeps track of the number of flows that are currently active using the counter *Nactive*. Using that value and the average total queue occupancy it computes *avgcq*, the average number of packets that each flow should have enqueued. Finally, the value *f* is the index of the flow associated with the current packet.

The index *f* is used to access instances of the per-flow variables q_len_f and $strikes_f$. A flow is the set of packets sharing a common source-destination address tuple. The number of packets that are currently in the queue and associated with flow *f* is recorded in q_len_f. The value $strikes_f$ is used to record the number of times flow *f* has exceeded its fair share (as computed by FRED). Note that these statistics are maintained only for active flows.

Since active flows are those which have packets in the queue, the state associated with these per-flow values is O(*q_limit*).

Finally, there are several important subroutines used by the algorithm. The functions *enqueue*, *dequeue*, *send*, *drop*, *now*, *random*, and *drop_early* correspond to their counterparts in RED. The function *flow_classify* returns the flow index for a given packet. The function *compute_avg* is described below.

```
Constants:
   float w;          // Weight for the running avg
   float $max_p$;       // Maximum drop probability for RED
   float $Th_{Max}$;    // Max threshold
   float $Th_{Min}$;    // Min threshold
   float drain_rate; // The drain-rate for the queue (link-capacity)
   int   minq        // Mimimum allowable queue occupancy
   int   q_limit;    // The upper limit on the queue size

Global Variables:
   int   q_len;         // Aggregate queue length
   float q_avg = 0;     // Average queue occupancy
   int   count = -1;    // Num packets that have arrived since last drop
   float avgcq;         // Average per flow queue size
   int   Nactive;       // Number of active flows – initially 0
   boool idle;          // The queue is currently idle
   time  last;          // The time when the queue went idle
   int   f;    // The index of the flow this packet is associated with

Per-flow Variables:
   int   $q\_len_f$ = 0;    // Current queue occupancy for flow f
   int   $strikes_f$ = 0;  // Number of overruns for flow f

General Functions:
   Class flow_classify(P)     // returns the connection id of packet P
   void  enqueue(P,f);        // Enqueue P and update q_len
   Packet dequeue();          // Dequeue P and update q_len
   void  send(P);             // Transmit P
   void  drop(P);             // Discard P
   Time  now();               // Return current time
   float random();            // Return random between 0 and 1

Defined functions:
   Bool  drop_early();        // Should we do an early drop
   void  compute_avg()        // update q_avg for class cl.
```

Figure 3.12 Definitions and Declarations for FRED

```
compute_avg() {
   local time  delta_time;           // BEGIN RED AVERAGE
   local int   n;

   // if the queue is still occupied take one sample
   if ( q_len || not idle ) {
      q_avg = ( 1 – w ) * q_avg + w * q_len;
   } else { // if the queue was empty we have to adjust the sampling
   // if the queue is empty, calculate how long the queue has been idle
      delta_time = now() – last;
   // n packets could have been dequeued during the interval
   // based on drain-rate
      n = delta_time * drain_rate – 1;
   // treat is as if n samples were taken with a qlen of 0
      q_avg = ( 1 – w )^n * q_avg;
   }                                 // END RED AVERAGE

   if ( NActive) {                   // BEGIN FRED AVERAGING
      avgcq = q_avg / Nactive;
   } else {
      avgcq = q_avg;
   }

   avgcq = MAX(avgcq,1);   // any flow may have one packet

   if (( q == 0 ) && ( not idle )) {
      last = now();
      idle = true;
   }                                 // END FRED AVERAGING
} // End compute_avg
```

Figure 3.13 Algorithm for Computing the Average in FRED Routers

Computing the Averages

One of the key steps in the FRED algorithm is the computation of the average per flow queue occupancy. This average, *avgcq*, is computed at the same time the total queue average is maintained. The basic algorithm is shown in Figure 3.13. The computation of the average queue size is exactly the same as that for RED, discussed in 3.2. Only the additions necessary for the FRED algorithm will be considered. Once the average total queue size, *q_avg*, is computed, *avgcq* is computed by dividing *q_avg* by *Nactive*, the number of active flows. If there are no active flows, then the average queue occupancy per flow is the entire average queue size. The algorithm does insure that all flows are entitled to one queue slot. Finally, there is a bit of record keeping to track when the idle period begins.

Deciding When to Drop

Both the average queue occupancy and other queue statistics are used to determine whether or not to drop arriving packets. Like RED, the FRED algorithm is state based and these states largely depend on the current average queue size. However, there are also sub-states and special actions that take place regardless of the current state. In addition, specific flows may be in a state where they are more tightly constrained than other flows because of prior misbehaving. Figure 3.14 shows the section of the FRED algorithm that deals with packets arriving at the router. When a packet arrives it is first classified. If this is a new flow, one with no other packets currently enqueued, its state is initialized.   Next, if the queue had been idle, then the algorithm updates the average. This is necessary to insure the average reflects the recent idle period and not the active period immediately prior to the idle period. After these record keeping steps, the algorithm begins the drop tests. First are the fair-share tests. There are two conditions that lead to a packet drop due to fairness concerns. They are:

$q\_len_f \geq Th_{Min}$ - If the flow's occupancy exceeds Th_{Min} packets in the queue the arriving packet is discarded.

($q\_len_f \geq avgcq$) && ($strikes_f > 1$) – If the flow has already been marked as misbehaving (as indicated by the strikes) it is limited to exactly its fair share of the queue, *avgcq*.

```
for each arriving packet P {
   f = flow_classify(P);

   if f has no state table {  // Initialize this flow
      $qlen_f$, $strike_f$=0;
   }
   if ( q_len == 0 ) compute_avg(); // only update if queue was idle

   // Fair-share test
   if (  ( $q\_len_f \geq Th_{Min}$ ) ||
         (( $q\_len_f \geq$ avgcq ) && ( $strikes_f$ > 1 )) {
      $strikes_f$++;
      drop(P);
      return;
   }

   if ( $Th_{Min} \leq q\_avg < Th_{Max}$ ) {  // probabilistic drop mode
      count++; // count the number of packets since most recent drop
      // Robust Flow Test
      if (( $q\_len_f \geq$ MAX(minq, avgcq) ) && ( drop_early() )) {
         drop(P);
         count = 0;
         return;
      }
   } else if ($q\_avg < Th_{Min}$) ) {  // no drop mode
      count = -1;
   } else { // two packet mode
      if ( $q\_len_f \geq 2$ ){
         count = 0;
         drop(P);
         return;
      }
   }

   if ($q\_len_f$ == 0)  Nactive++;      // If this is a new flow

   compute_avg();
   enqueue(P);
} // End for each arriving
```

Figure 3.14 Algorithm for Packet Arrivals in FRED Routers

If the flow fails any of the tests for fairness, the packet is dropped and a strike is recorded against the flow. Once a flow gets two strikes, it is considered to be a misbehaving flow and is tightly constrained by the second condition of the fairness test. Note that flows do have the opportunity to move out of the misbehaving state. Whenever a flow ceases to have packets in the queue its state table entry is removed. The next time a packet arrives from that flow it is treated as a new and different flow with no strikes. This allows responsive flows to recover from occasional burstiness while maintaining a constraint on unresponsive flows that constantly have packets enqueued.

After the fair share test, FRED essentially applies the standard RED tests to decide whether or not to drop the packet. The major difference involves testing the flow for robustness before applying the probabilistic test. All flows are allowed to buffer up to *minq* packets during probabilistic drop mode without loss to protect fragile flows A flow is considered robust if it has more than *minq* packets in the queue. A very small number of packets in the queue may indicate it is a low bandwidth, fragile flow. In that case dropping a packet may trigger a retransmission time-out, severely impacting the flow's performance. The test for robustness compares q_len_f to the greater of *minq* and *avgcq*. If the flow is exceeding its fair share and the minimum limit it is considered robust and the drop_early test is applied. If the test is true the packet is dropped. (Note: the *drop_early* function is the same as that for RED, shown in Figure 3.9.)

Finally, the algorithm checks to see if the current average indicates the current mode should be no drop mode or forced drop (i.e., two-packet) mode. If $q\_avg \leq Th_{Min}$, then the algorithm is in no drop mode and the *count* is reset to –1. Otherwise, the average must be greater than Th_{Max}. Since this is the version of the FRED algorithm modified to support many flows, the packet is dropped if that flow already has two or more packets in the queue. If the packet is dropped the count of packets since the last drop is reset to zero. If none of the drop tests are positive the packet is enqueued and the average is updated. However, before that happens, the number of active flows is also updated if there are no packets from this flow currently enqueued.

Packet Departures

Figure 3.15 shows the algorithm for dealing with packet departures in FRED. Whenever a packet departs the queue it must be classified. If the q_len_f is now zero, reflecting the absence of flow *f*'s packets in the queue, then the number of active flows is decremented and the state for this flow is deleted. Note that this deletion of state allows flows that respond to congestion slowly to escape from being labeled as misbehaving.

```
for each departing packet {
   P = dequeue();             // dequeue and send the packet
   compute_avg();
   f = classify(P);
   send(P);
   if ( q_lenf == 0 ) {     // If the class is idle
      Nactive--;
      delete state for flow f;
   }
} // End for each departing
```

Figure 3.15 Algorithm for Packet Departures in FRED Routers

Note that FRED computes the average on enqueue and dequeue. In contrast, RED only calculated the average on enqueue, missing the changes in the queue size that happen due to dequeue operation while no packets are arriving. The designers of FRED felt the RED method led to undersampling that led to an inflated average (because many incremental decreases in queue size are not sampled) and, potentially, to unnecessary drops and low link utilization.

3.3.2. Evaluation

FRED seeks to provide fairness between flows. It does this by isolating flows from one another by maintaining per-flow statistics and constraining those flows that consume more than their fair share. The results of this approach can be compared to the results with FIFO and RED using the TCP Throughput graph in Figure 3.16. The traffic load is the same as in the earlier experimental evaluation of RED and FIFO and the results with those algorithms are superimposed for comparison. The misbehaving UDP blast is active during time [15, 70] seconds. While there is some decrease in TCP throughput, the overall performance is much better than that seen when simply using RED or FIFO. FIFO has the lowest throughput during the blast while the throughput for RED is slightly better and FRED's is much better. There is no congestive collapse. The difference in the results illustrated in the RED case and here is that in the FRED case, the unresponsive UDP flow is constrained to consume a fair share of the router's outbound queue. With hundreds of TCP connections (as part of this experimental set-up), there are a large number of active flows (relative to the queue size of 60) at any given time, resulting in every flow being allocated 2 buffers in the queue. Because the UDP flow is unresponsive (and high-bandwidth), it exceeds this share and is constrained to never occupying more than 2 slots in the queue. This results in a significantly higher level of packet loss for the unresponsive UDP flow than under RED (and hence higher throughput for all other well-behaved flows). Under RED, the unresponsive UDP flow could monopolize the queue and achieve significantly higher throughput. Under FRED, each active TCP flow gets at least the same number of buffer slots in the router queue as the unresponsive UDP flow does, resulting in better aggregate throughput for TCP.

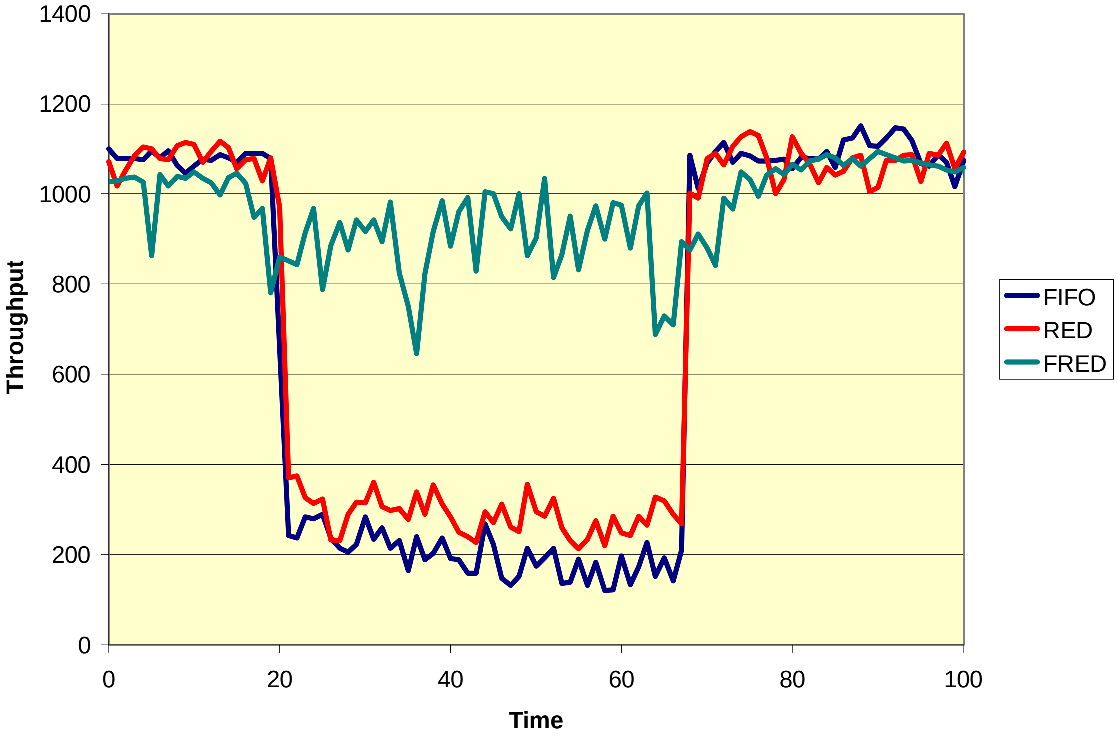


Figure 3.16 Aggregate TCP Throughput (KB/S) vs. Time (seconds) in the Presence of an Unresponsive, High-Bandwidth UDP Flow for FIFO, RED, and FRED.

By using the RED drop policy for robust, well-behaved flows, the FRED algorithm continues to provide well distributed feedback to responsive flows just as RED did. In addition, FRED is able to constrain misbehaving flows, limiting them to a fair share of the queue.

One point of debate with FRED is the definition of fair. RED attempted to be fair by assuring that all flows received feedback and were constrained in proportion to their actual packet arrival rate. RED attempted to insure that the throughput of all flows was degraded by the same percentage by dropping the same percentage of packets for all flows. (The inequities resulting from combinations of responsive and unresponsive traffic will be disregarded for the purposes of this discussion.) However, FRED attempts to insure that all flows are constrained to roughly equal shares of the link's capacity. This fails to consider the fact that flows are associated with applications that may have minimum tolerable throughput values. Perhaps it is more reasonable to assume that all flows can tolerate a 10% degradation in throughput than to assume that the high bandwidth flows can tolerate a 50% degradation. Consider streaming video. Human perception thresholds demand a minimum latency, frame-rate, and image fidelity in order for the interaction to have value. If the initial data flow is degraded slightly it may remain tolerable, but below some minimum packet-rate, the frame-rate and/or image fidelity degrade to uselessness. At that point, the data that does make it through the network is simply wasting network bandwidth. The notion of fairness is a worthwhile concept to explore. However, it may be necessary to introduce flexibility by making the thresholds configurable instead of simply trying to offer equal shares.

In summary, FRED offers good TCP performance and constrains misbehaving flows well. However its notion of fairness by allocating all flows an equal share may be too inflexible. FRED is noteworthy for its application of the queue management on a per-flow basis, for its identification of robust and misbehaving flows, and for its introduction of the concept of fairness to queue management.

4. Packet Scheduling

The techniques considered up to this point were various forms of queue management. Now, consider packet scheduling, another router-based mechanism for managing congestion and bandwidth allocation. With all of the queue management mechanisms considered so far, the packets that successfully transit the queue are transmitted on the outbound link in the same order they arrived at the queue. It is this ordering that distinguishes queue management from packet scheduling. With scheduling, packets are not necessarily transmitted in the order in which they are received. However, order is maintained within a given class. Packet scheduling algorithms may reorder packets to insure that performance objectives are met. This technique is explained below.

Scheduling is used to insure that resources are effectively utilized. That means both that the resource is fully utilized whenever possible and that clients receive the resources they need when they need them. In the case of packet scheduling, the resource is the outbound link’s capacity. Different flows or classes of flows negotiate a contract with the router for their desired performance characteristics. The packet scheduler attempts to be sure that all of the clients receive their negotiated share of the link's capacity. The router then associates that class of traffic with a queue that will be serviced at a rate sufficient to insure the desired performance. When packets arrive at the router, they are classified to determine which queue they are associated with and enqueued. When the link is ready to transmit a packet, the scheduler then selects the queue to service next. Because of differences in queue length, service rate, and arrival rate, packets in one queue may effectively pass packets in other queues that arrived earlier.

4.1. Class-Based Queueing (CBQ)

In our work, CBQ is used as a representative example of a packet scheduling technique. While the AQM policies simply make drop decision on a single FIFO queue, CBQ maintains multiple queues for each class of traffic, servicing the different queues at different rates to provide guarantees on the throughput for each class. When some class of traffic does not consume its allocated bandwidth, the excess can be reallocated to other classes by servicing those queues during the time that would have been spent servicing the idle queue. Because CBQ is able to offer guaranteed performance for different classes of traffic it is used here as a "gold standard" measurement to compare with the active queue management schemes. In the experiments presented later on in this dissertation we will run each traffic pattern through a CBQ router and then compare how close AQM techniques can come to realizing the performance of CBQ.

4.1.1. Algorithm

The CBQ algorithm can be broken into four parts. One component, adding a class, is part of the background processing engine, while the other three parts are part of the forwarding path. Every packet must be classified and enqueued, queues must be scheduled for service, and the appropriate queue must have a packet dequeued and transmitted. The basic function of each of these components is described with pseudo-code below.**Adding a New Class**

The mechanism for adding a new class to a router may range from an end-system requesting a reservation via a network control message to configuration performed locally by a network administrator. How the request to setup a new class arrives is not a concern. The focus is on what happens once the request is made. Figure 3.17 shows pseudo-code for processing a request to add a new class. The setup method is passed a data structure, *class_info*, that contains information about the new class. First, the routine performs *admission control*, checking if there is sufficient capacity available to service the new class in addition to all of the currently supported classes. If not, the request is rejected. If there is capacity, a new queue is created and the class's requested service rate and pattern matching rules are recorded for that queue. The service rate will be used by the scheduler to determine when to dequeue a packet from this queue. The pattern matching rule will be used by the classifier as packets arrive. The new queue is added to the list of queues (*Q)* serviced by the scheduler. Finally, the method either responds affirmatively or negatively to the request. This may require sending a message if the request arrived over the network or simply displaying a notification to an administrator on a console.

```
setup(class_info) {

   // If there is sufficient capacity to meet the class's requested
   // allocation
   if ( capacity_available(Q, class_info) ) {

      // Allocate a new queue with the desired service rate.
      // Also record the pattern matching rule to use for classifying
      // packets.
      queue = new queue(class_info.service_rate,
                        class_info.pattern_matching_rule);

      add_queue(Q,queue);     // Add this queue to the list of queues
      // Confirm that the new class can be supported
      reply_affirmatively();

   } else {
      // There isn't enough capacity so reply
      reply_negatively();
   }
}                       // End setup
```

Figure 3.17 Adding a New Class for CBQ

Scheduling

Although the concept of scheduling the queues is straight forward, the actual implementation is quite complex. In concept, the scheduler should insure that over any given time interval each non-idle queue transmits data in proportion to the allocation associated with that queue's class. However, the fact that the queues must be serviced in discrete but variable size units (packets) complicates the task. The scheduler cannot transmit half of a packet and the packets from one queue may be twice the size of the packets from another queue. As a result the scheduler must, for each queue, keep track of the allocation, the time at which the last packet was sent, the size of the last packet, and the size of the packet at the head of each queue. Using this information the scheduler must compute the next time that each queue should be serviced. Maintaining this state increases the amount of memory required in the router. Each time a packet is transmitted, the scheduler must update the next service time for each queue and perform a sort to determine the queue that must be serviced next. This sorting increases the computational complexity of servicing a packet.

Figure 3.18 shows pseudo-code for the actual selection of the next queue to service. Other functions, such as updating the service times based on the size of the packet sent are incorporated into other components, such as the dequeue routine. In the scheduling routine, it first sorts the queues by the next service times and returns a list of the indices sorted with the earliest service time first. Next, this sorted list is traversed until a queue that has a packet enqueued is found (i.e., Q[i].len > 0). The index of that queue is returned to the calling routine. Since a class without packets enqueued is skipped, when some class is not using its allocation the interval required to service all of the queues decreases. Consequently, the services rate (servicings/time) for all of the other classes increases in proportion to their initial bandwidth allocation.

```
queue_index schedule() {
   // get a list of queues sorted by service time
   indices_sorted = sort_times(Q);

   // Find the earliest service time for a queue that has a packet
   // in the queue.
   for ( i = 1; Q[indicies_sorted[i]].len == 0; i++ );

   return(indices_sorted[i]);       // return the selected queue index.
}                             // End schedule
```

Figure 3.18 Scheduling for CBQ

Packet Classification and Enqueue

Figure 3.19 shows the pseudo-code for handling a packet arrival. An arriving packet is examined and classified to determine which queue it should be assigned to. Classification can be performed using the pattern matching rules for each queue. The classification process is not specified as part of the algorithm but it is commonly based on pattern matching using the source and destination address, port, and protocol fields in the packet header. Techniques exist to classify packets in O(log$_2$ *address_bits*) time. [Waldvogel97]. Although different queue management policies (such as RED) can be used to manage individual queues, in this example all queues are simple drop-tail FIFO queues. As long as there is room in the queue the packet is enqueued. Otherwise it is discarded.

```
for each arriving packet P {
   cl = classify(P);             // Determine the class of the packet

   // If there is room in the class of that queue.
   if ( Q[cl].qlen < qlimit ) {
      Q[cl].enqueue(P);
   } else {                      // If there's no room
      drop(P);                   // discard the packet
   }
}                             // End for each arriving
```

Figure 3.19 Processing an Arriving Packet in CBQ

Dequeueing Packets

The pseudo-code for servicing the outbound link is shown in Figure 3.20. When the outbound link is ready for another packet, the CBQ algorithm checks the schedule to determine which queue should be serviced next. After selecting the queue, a packet is dequeued and transmitted. Once the packet is transmitted the schedule must be updated based on the size of the packet transmitted.

```
when outbound link is ready {
   cl = schedule(Q);          // Select a queue to dequeue from
   P = Q[cl].dequeue();       // Dequeue a packet
   transmit(P);               // Transmit the packet

   // Based on the size of the packet dequeued, update the next service
   // time for this queue.
   update_service_time(Q[cl],P);
}
```

Figure 3.20 Transmitting a Packet in CBQ

4.1.2. Evaluation

CBQ provides accurate allocations of bandwidth between classes. The bandwidth allocation experiment discussed for FIFO, RED and FRED was repeated with CBQ. CBQ's results are compared with those for the other algorithms in Figure 3.21. During the period of the misbehaving UDP blast (15-70), the TCP throughput under CBQ drops off slightly, but only to its allocated bandwidth (79% of the link - 960 KB/s). CBQ has high throughput during the blast. It does not have the severe degradation observed for FIFO, RED, and it is comparable to FRED in throughput while being less variable.

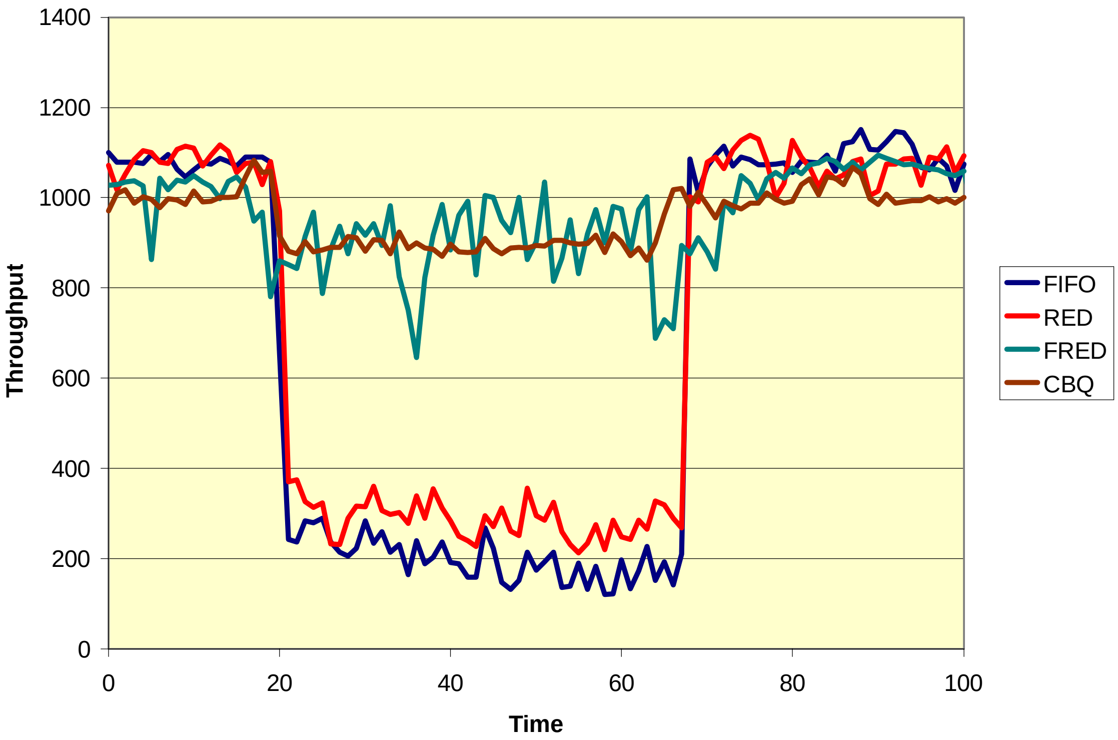


Figure 3.21 Comparing Aggregate TCP Throughput (KB/s) over Time (seconds).

Packet scheduling is an attractive option because it offers very precise control over network resource. A flow or class of flows can be assigned a share of the outbound link and be assured of receiving that throughput level. However, this service comes with a price. First, the algorithmic complexity is significantly greater than in any of the queue management mechanisms. The scheduling algorithm must maintain a concept of virtual time (or one of the equivalent alternatives) and compute deadlines for the next time each queue must be serviced and then sort the queues by their next service time. Second, the algorithm must be configured to indicate the desired allocation of network bandwidth. This may require manual configuration at any router using CBQ or it may be part of a reservation protocol that would need to be deployed for use by end-users and/or network administrators. Third, CBQ is not widely used due to its complexity. Strengths of CBQ include that it can be configured locally in a single router with beneficial effects. It is effective in its allocation between classes. It is flexible in its design as the number of classes and their composition is highly configurable.

5. Evaluation of Router Queue Management

Historically, most active queue management approaches have operated on the assumption that most flows are responsive. As such, one of their goals was to give better, earlier feedback to responsive flows. Although this early feedback did have some constraining effects, it generally had little effect on unresponsive flows.

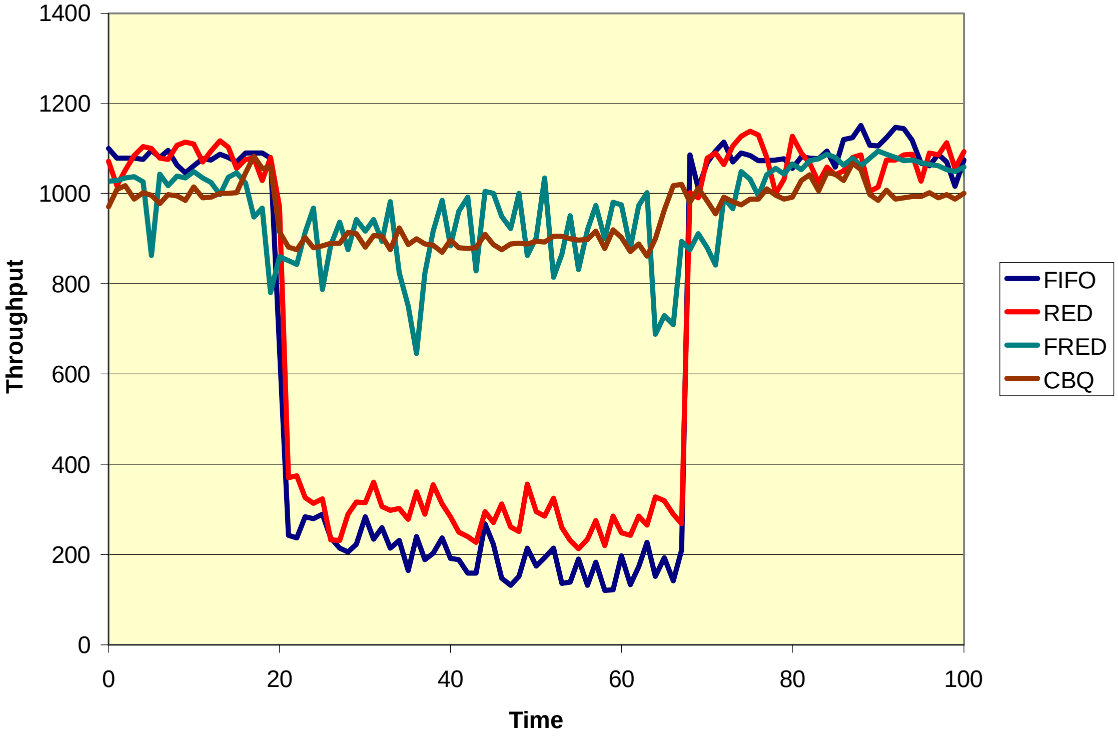


Figure 3.22 Aggregate TCP Throughput (KB/s) vs. Time (seconds) under FIFO, RED, FRED, and CBQ.

Figure 3.22 shows the throughput of TCP in the presence of aggressive UDP flows under the different router queue management and scheduling algorithms. (The experimental infrastructure used is described in Appendix A.) Note that in the case of the queue management algorithms, FIFO and RED, the TCP throughput collapses when the aggressive flows are introduced (between time 15 and 70). That is because the TCP flows respond to the congestion by decreasing their load while the aggressive flows continue to transmit at the same rate. Since the TCP load is reduced, the aggressive flows are able to consume the unused bandwidth.

Other queue management algorithms, like FRED were designed with unresponsive flows in mind. As discussed in Section 3.3, FRED maintains per-flow statistics, constrains all flows to a loosely equal share, and tightly constrains any misbehaving flows to a strictly fair-share of the queue. As a result, aggressive, unresponsive flows are constrained and unlikely to dominate the link. As shown in Figure 3.22, the throughput for TCP is superior in the case of FRED to the other queue management policies.

Finally, the figure shows the results with CBQ, a packet scheduling discipline. Packet-scheduling results are presented for comparison of the best possible approach. Since CBQ provides a guaranteed service it should give optimal performance for those classes that operate within their allocations. In this experiment, CBQ was configured to allocate 79% of the link's capacity to TCP. Before the aggressive flows are introduced TCP is able to consume more than that share, but its throughput reduces to its allocated level when aggressive flows are present. Further, because scheduling is fine grained, the throughput is much less variable with CBQ than with the other approaches.

Recent AQM proposals do improve TCP performance in the presence of unresponsive flows. However, in addition to this we also seek to offer good performance for multimedia. Therefore, multimedia performance must also be considered. None of these active queue management approaches consider multimedia or other unresponsive flows that have minimum acceptable performance levels. In the case of RED, there is no effort to distinguish between flows so multimedia is as likely to receive drops as any other traffic type. Table 3.1 shows the multimedia drop-rate during a congested period and in the presence of a combination of TCP and aggressive flows. These results are from the experiments shown in Figure 3.22. Clearly, multimedia suffers a large number of drops under RED, with a 30% drop-rate. Moreover, FRED's policy of restricting unresponsive flows results in an even higher drop-rate for multimedia, almost 36%. These drop-rates are unacceptable if the multimedia interaction is to have any value [Talley97]. In contrast, the packet scheduling discipline, CBQ, guarantees no drops as long as the class's offered load remains within its allocated bandwidth and the allocation is sufficient for the load.

Algorithm	**Drop Rate for Multimedia**	**Latency**	**Queue Size**	**Th_{Max}**
FIFO	32.4%	~ 60 ms	60	n/a
RED	30.0%	~ 33 ms	60	30
FRED	35.7%	~ 30 ms	60	30
CBQ	0.0%	~ 7 ms	n/a	n/a

Table 3.1 Average Packet Drop Rate and Latency for Multimedia Packets

Latency is also disappointing. FIFO allows the queue to remain full and hence latency is high since the average latency is directly related to the queue size. The 10 Mb/s capacity of the outbound link results in a drain rate of approximately one 1,000 byte packet per millisecond. With average packet sizes of 1,000 bytes, this accounts for the 60 ms latency for FIFO. In the case of RED and FRED, the maximum threshold limits the average queue occupancy and is the primary constraint on latency. A maximum threshold of 30 packets results in latency on the order of 30 ms in each case. Finally, since CBQ schedules packet transmissions, any class that is within its bandwidth allocation should be serviced almost immediately. Multimedia's load is within its bandwidth allocation so in this case the overall latency is 7ms for multimedia.

Two other AQM proposals merit discussion. Floyd & Fall propose router-based mechanisms to encourage end to end congestion control and are based on identifying and constraining misbehaving flows [Floyd98]. Although no implementation of this proposal was available to experiment with, it is clear that this approach would improve TCP performance in the scenarios above. However, it is equally clear that, like FRED, this approach would result in poor performance for multimedia flows because it would identify the multimedia flows as unresponsive and severely constrain them. On the other hand, in RIO service profiles could allocate the desired amount of TCP bandwidth and the desired amount of multimedia bandwidth [Clark97]. Once again, no implementation was available for study, but it seems that RIO would provide good throughput performance for multimedia and for TCP if properly configured. However, with only one queue, latency could grow quite high if there is a significant volume of out-of-profile traffic. Although reducing the threshold setting for out-of-profile traffic could help to address this problem, it would be at the expense of decreasing the fraction of the link capacity available to out-of-profile traffic. (In Chapter IV the relationship between queue occupancy and link capacity will be explained in more detail.) The current proposals to address the tension between responsive and unresponsive flows through router queue management do not provide support for multimedia.

6. Summary

This chapter considered the evolution of router queue management. Originally, queues were used to service the outbound links of routers in order to buffer bursty packet arrivals. When these queues filled, arriving packets were discarded and responsive end-system protocols inferred the presence of congestion from the resulting packet loss. With simple drop-tail semantics, designers had to balance the need to have sufficient buffering to accommodate bursts with the need to limit the size of these buffers to minimize latency and provide early feedback during periods of persistent congestion. Techniques such as early random drop explored the idea of providing some feedback before the queue overflowed by randomly dropping some arriving packets when the queue size exceeded a fixed threshold. Alternatively, other researchers considered techniques such as DECbit and ECN which provided explicit notification of congestion by marking packets during periods of congestion. Deployment of these packet marking algorithms has been limited because they require changes to the end-systems protocol stacks to be effective.

Meanwhile, research has continued in ways to manage congestion by altering the packet dropping policies in the routers. These techniques are referred to as Active Queue Management (AQM). Random Early Detection (RED) probabilistically drops packets based on the average queue occupancy. Monitoring the average queue occupancy allows RED to differentiate between persistent and transient congestion. The probabilistic dropping policy allows RED to distribute packet loss across all flows in proportion to their load. This helps to avoid the problems of lock-out and full queues which existed with simple drop-tail semantics. However, both RED and drop-tail are vulnerable to the effects of high-bandwidth unresponsive flows. When responsive flows respond to packet loss by reducing their load these unresponsive flows are able to consume most of the network bandwidth because they maintain their load regardless of loss. Flow-based Random Early Detection (FRED) addresses this problem by maintaining per-flow statistics on queue occupancy and recent drop history. Using these mechanisms FRED can identify and restrain aggressive and unresponsive flows to insure that all flows get a loosely fair share of the link's capacity. However, it is not clear if FRED's concept of fairness through equal shares is ideal given the heterogeneous performance demands and limits of different classes of traffic. It also has undersirable effects (drops) for multimedia. In addition to the AQM techniques, packet scheduling, the alternative technique for router queue management, was also reviewed. Because packet scheduling can offer performance guarantees it serves as a gold standard to compare with AQM techniques.

However, none of the router queue management approaches provide good performance for multimedia. RED and FIFO treat multimedia like all other traffic, subjecting multimedia to the same high drop-rates as TCP during periods of congestion. FRED and Floyd & Fall actively constrain unresponsive in order to improve TCP performance. Consequently, unresponsive traffic such as multimedia suffers poor performance when these algorithms are used.

In the next chapter we present a new AQM algorithm, class based thresholds (CBT) which protects responsive traffic from unresponsive traffic. It also provides better performance for multimedia by managing router queue occupancy to allocate bandwidth to different classes of traffic. Then, in Chapter 5 we will present an empirical evaluation of CBT with respect to RED, FRED, FIFO, and CBQ. RED is the current state of the art for providing feedback to responsive flows based on aggregate queue behavior. As such, RED demonstrates the effectiveness of aggregate queue management. In contrast, FRED is the extreme in per-flow queue management. Thus, FRED allows us to consider if allocating each flow an equal share of the queue is sufficient for meeting our goal of protection for TCP and multimedia. FIFO and CBQ serve, respectively, as the baseline and gold standard for comparison.

IV. CLASS-BASED THRESHOLDS

1. Problem and Motivation

The Class-Based Thresholds mechanism provides a better-than-best-effort service for multimedia while also isolating TCP and other responsive flows from the effects of unresponsive flows. The tension between responsive and unresponsive flows is a well-recognized problem [Braden98]. However, most of the proposed solutions have taken a very TCP-centric view. The approaches are TCP-centric in the sense that, in addition to protecting and isolating TCP from unresponsive flows, they have also sought to constrain or punish unresponsive flows in an effort to encourage them to be more TCP-like. We take a multimedia-centric view. Although the majority of the Internet’s traffic is TCP and, as a good network citizen TCP flows should be protected, multimedia has legitimate reasons not to use TCP as its transport mechanism. Further, multimedia has very strict performance requirements and the proposed constraints do not simply reduce the efficiency of the flow. They may render the interaction valueless.

Chapter II reviewed many application and transport level approaches that seek to make multimedia flows more responsive. Some focused specifically on providing TCP-like response to congestion. Others focused more heavily on temporal and spatial-scaling techniques to adjust the quality of the media to match the performance of the network in a controlled way. All of these approaches show promise and research and deployment of such techniques should continue. However, even these techniques demand minimum levels of network performance in order to be effective. This is because applications using these techniques have limits on acceptable latency and throughput. Further, there already exists a large number of multimedia applications deployed in the Internet that are not responsive. These applications will not be replaced instantaneously so their presence must be addressed. For these reasons, we seek to insure that the network offers at least minimal performance levels for multimedia.

Chapter III considered some of the network-based approaches to dealing with congestion, specifically active queue management and packet scheduling. Historically, active queue management approaches have operated on the assumption that most flows are responsive. One of their goals was to give better, earlier feedback to responsive flows. While this early feedback did have some constraining effects, it generally had little effect on unresponsive flows. As a result, tension still exists between responsive and unresponsive flows. To address this problem, subsequent proposals such as FRED, RIO, and the work by Floyd & Fall have worked on identification and/or constraint of unresponsive traffic.

However, because active queue management generally focuses on effective notification of congestion, fairness between flows, or constraining or punishing unresponsive flows, these algorithms lead to poor performance for multimedia flows. We approach the problem of network congestion and the tension between responsive and unresponsive flows from another perspective, focusing on support for multimedia. The goal is to provide better-than-best-effort performance for multimedia that also isolate TCP from effects of multimedia flows and isolates both from the effects of *other* traffic. We propose a new active queue management policy, Class-Based Thresholds (CBT), which meets these goals. This chapter begins by explaining the goals of the algorithm and the general approach to achieve these goals. Next, a pseudo-code description of the algorithm is reviewed. Next we explain how to configure CBT, including the equations that explain the relationship between the desired bandwidth and latency and the actual threshold settings. To further understand the algorithm, equations are presented that predict its behavior under different network loads. Specifically these equations explain how classes borrow bandwidth and how latency changes. Finally, experimental results are presented that show the control of different performance metrics that CBT affords.

2. Goals

CBT offers an alternative approach to addressing the tension between responsive and unresponsive flows in today's Internet. General Internet traffic is made up of different types of traffic with different responses to congestion and different performance requirements. To provide each class acceptable performance while limiting the effects between classes, CBT differentiates between packets based on which configurable traffic class the packet belongs to. The algorithm then attempts to isolate the performance of one class from the effects of other classes. CBT is an active queue management technique that manages the average queue-induced latency in a network router and allocates bandwidth to each class. Moreover, the approach has less algorithmic and computational complexity than packet scheduling. To do this, CBT uses limits on average queue occupancy to control the proportions of the outbound link's capacity available to each class of traffic. Latency is also managed with this technique. Further, CBT's design leads to predictable, controllable behavior when some classes do not use their full resource allocation.

2.1. Traffic Types and Isolation

One of CBT's major goals is to provide isolation between three major classes of traffic: TCP, multimedia, and everything else (*other*). TCP should be isolated from any negative effects of unresponsive traffic flows, either multimedia or *other*. Further, multimedia should also be protected from the effects of both TCP and *other* traffic so that it can receive a level of network service sufficient for realizing the minimum requirements of applications generating the traffic. Finally, though there is no intent to give *other* traffic preferential treatment, *other* traffic should be allowed to make progress, though at a controllable and policed service level. A classifier identifies each packet as belonging to one of these three classes. We begin here by describing each class and the intent of isolating each.

2.1.1. TCP

The TCP traffic class can be identified very easily by the protocol field in the packet header. All packets with a protocol field set for TCP are assumed to belong to true TCP connections that properly implement the TCP congestion control algorithm. The issue of dealing with misbehaving TCP implementations and non-TCP protocols with TCP-like congestion response is ignored in this work. However, since queue management is detached from the classification mechanism, it would be reasonable to extend the classification mechanism to use more sophisticated techniques, such as drop-history, to identify TCP-like flows [Floyd98]. In the experiments performed in this dissertation, the TCP flows present were proper implementations of the protocol. Since TCP flows are responsive, if they are allocated a share of the link's capacity and isolated from other classes the individual flows should be able to arrive at equilibrium between their generated load and the allocated share of the link.

2.1.2. Multimedia

The definition of what constitutes multimedia is detached from the definition of the CBT mechanism. Multimedia is not defined in terms of particular applications or media encodings. Multimedia packets to be self-identified. While misrepresentation could be a serious problem and allow other flows to steal bandwidth from multimedia, the issue is beyond the scope of this work. However, there is a substantial body of policing and classification work [Waldvogel97, Floyd98]. However, this work assumes the presence of properly identified multimedia flows and focuses on the effectiveness of the algorithm in that situation. In the current implementation, packets are simply classified by their IP address-tuple, based on well-known ports associated with multimedia.

2.1.3. Other

Other traffic is simply all traffic that does not match the criteria for the TCP or multimedia classes above. By isolating *other* traffic into its own class and limiting the bandwidth allocated to this class, these unresponsive and possibly aggressive flows are left to compete with one another for a controllable share of link bandwidth. Although, this is not the focus of this work, this competitive isolation may motivate application designers to design their applications to behave in a TCP-like way.

2.2. Predictable Performance for Traffic Classes

Once classes of traffic are isolated from one another, the next step is to control the performance each receives. That means controlling the way they use network resources.

One of the goals of this work is to explore how effectively network resources can be managed using a lightweight approach like active queue management instead of a more complex approach like packet scheduling. These resources include the capacity of the outbound link and the buffering capacity of the router. CBT allows shares of the link's capacity to be allocated to each class, resulting in a loose guarantee of average throughput for the class. CBT also manages the average length of the queue associated with the outbound link in order to manage the queue-induced latency.

2.2.1. Bandwidth Allocation

Once CBT classifies packets by their traffic type, it needs to manage the share of the outbound link's capacity that each class receives. The bandwidth allocations should insure that a class like TCP, representing the majority of the network traffic, receives the majority of the outbound link's capacity during a period of congestion. These bandwidth allocations are not strict guarantees but take the form, "class X should, on average, have access to at least *B* Kb/s". Moreover, this specification of "at least *B* Kb/s" refers to the desire to maintain a resource allocation scheme that approximates a weighted max-min fair allocation [Keshav97]. This means that the class with the minimal fractional overrun of its allocation will have its overrun met (i.e., max-ed out) before any consumers with larger fractional overruns have theirs met. When one class is not using its full allocation of bandwidth, other classes should be allowed to use that class's unused bandwidth in proportion to their initial bandwidth allocations. The actual mechanism for bandwidth allocation using active queue management and max-min fair sharing are discussed in Section 3.1.2 below.

2.2.2. Manage latency

CBT seeks to minimize queue-induced latency. Queue-induced latency is a direct function of the number of packets in the queue when a packet arrives. The packet will be delayed until all of the packets ahead of it in the queue are sent. Therefore, to minimize the average queue-induced latency the algorithm must minimize the average queue size. In CBT one can specify the desired worst-case average latency as an upper bound on the average queue-induced latency that might be incurred by a packet transiting a CBT router.

Unlike bandwidth allocations which are per class, the worst-case latency applies across all classes. Section 4.1 provides a set of equations that convert latency and bandwidth requirements to thresholds on queue occupancy.

There is also a small component of latency based on the amount of time it takes a router to process a packet, aside from queueing delay. This is the time required to examine the packet’s headers for routing and classification purposes. The algorithm's design minimizes processing overhead by limiting the algorithmic complexity and the amount of state maintained.

2.2.3. Resource Allocation via Queue Management

In addition to the goals above, we also pursued this research to determine the strengths and limitations of using active queue management as a mechanism for providing resource management. As we discuss in Section 3.1.2, queue occupancy is managed in order to exercise proportional control over the bandwidth available to each class of traffic. Further, this same mechanism can also be used to control the queue-induced latency. In addition to confirming the algorithms ability to isolate and protect the different classes of traffic, we also explore the accuracy and predictability of this general resource management approach.

2.3. Minimize Complexity

CBT's bandwidth allocation mechanism is less complex than packet scheduling approaches. Packet scheduling approaches must maintain a concept of virtual time and the next expected service time for each queue. On each packet arrival and departure, this clock must be updated and the expected service times recomputed for each class. When a packet can be transmitted, the expected service times must be sorted to determine which queue to service. In contrast, when a packet arrives CBT only updates the average queue occupancy for a single class (that of the arriving packet) and compares that average to the threshold for that class. When a packet can be transmitted, CBT simply transmits the packet at the head of the single queue.

3. The Algorithm

First, the general design of the CBT algorithm is discussed. Then the specifics of the CBT algorithm are considered in more detail by examining pseudo-code.

3.1. Design

CBT offers isolation and protection between different classes of traffic. A high-level description of the algorithm is found in Figure 4.1. To provide isolation and protection, CBT classifies each arriving packet into one of a small number of classes and makes the drop decision based on statistics and parameters unique to that class. The statistics and drop tests for each class are like those of RED. The average queue size for the class, q_avg_{cl}, is compared to the thresholds for that class. By tuning selected parameters for a given class (e.g., $max_{p,cl}$, and the relationship between $Th_{Max,cl}$ and $Th_{Min,cl}$.) one can fine-tune the amount of feedback that class receives. Moreover, by adjusting the ratios between the maximum thresholds for each class, the administrator can control the relative bandwidth allocations for each class of traffic.

```
for each arriving packet P {
   cl = classify(P);
   $q\_avg_{cl}$ = compute_avg(cl);        // update the average for this class

   if ( $Th_{Max,cl}$ > $q\_avg_{cl}$ ) {
      drop(P);                             // Forced drop mode
   } else if ( $q\_avg_{cl}$ > $Th_{Min,cl}$ ) {  // probabilistic drop mode
      if ( drop_early(cl) )
         drop(P);
   } else
      enqueue(P,cl);

} // End for each arriving
```

Figure 4.1 High-level Pseudo-code for CBT

3.1.1. Classification

The first step in processing an arriving packet in a CBT router is classification. The algorithm does not specify a classification method. Instead we refer the reader to the differentiated-services concept of tagged flows [Clark97] and other work in packet classification [Waldvogel97]. The algorithm simply assumes that all arriving packets are classified into one of the available classes. The only scenarios considered are those involving only three classes, TCP, multimedia, and *other*, as discussed above. However, the design of the CBT algorithm is general and can have any number of classes. Of course, increasing the number of classes would impact the choice of classification method and increase the amount of state that must be maintained. However, the state would be proportional to the number of classes.

3.1.2. Managing Queue Occupancy

One of the keys to the approach in CBT is the observation that the ratio of bytes occupied by each class in the queue is also the ratio of bytes used by each class on the outbound link. Therefore, if the share of the queue occupied by each class is controlled so is the share of the link used by each class. The threshold and averaging mechanisms from RED manage each class's queue occupancy.

The CBT algorithm tracks the queue occupancy for each class as well as the overall queue occupancy. Whenever a packet associated with a given class arrives, the class's average queue occupancy is updated. That average is then compared to the threshold(s) for the class and the drop decision is made as in RED. Using a RED style weighted average mechanism allows for burstiness in each traffic class while assuring that, on average, each class's average queue occupancy does not exceed its allocated share of the queue or the link. These allocations and their relations with the threshold settings are discussed more formally in Section 4.1.

One fortuitous side-effect of relying on ratios of queue occupancy to control the ratios of link capacity for each class is that another feature, the borrowing of unused capacity, falls out naturally. Borrowing refers to the idea of classes exceeding their maximum bandwidth allocation being allowed to borrow the unused capacity of classes that are not using their allocated share. Further, the borrowing is max-min fair. *Max-min fair* is the idea that classes are able to borrow excess capacity in relation to their shares. For example, assume 100Kb/s of link capacity is allocated to class A, 200 KB/s of link capacity to class B, and there is 150 KB/s of excess capacity because of under-utilization by other classes. The ratio between class A and class B is 1:2. As a result, if A and B need more bandwidth and are the only classes that do, class A would be able to borrow 50 KB/s from the unused 150 KB/s pool and class B would be able to borrow 100 KB/s. This would result in respective throughputs of 150 KB/s and 300 KB/s, maintaining the relative shares between the classes that are operating at capacity. This borrowing is explained more formally in Section 5.2.1.

While the ratios between the classes' thresholds control the shares of the link's capacity that they receive, it is the sum of these thresholds that controls the queue-induced latency. As each class is limited, on average, to have a queue occupancy of no more than their maximum threshold, so too is the aggregate queue occupancy limited on average to no more than the sum of the maximum thresholds. Since the queue occupancy when a packet arrives determines its queue-induced latency, then the sum of the maximum thresholds determines the maximum average queue-induced latency that might be incurred by an arriving packet. Using the set of equations in 4.1, one can adjust both the ratios between the class thresholds (to control their bandwidth allocation) and the sum of the thresholds (to control the maximum average latency).

3.1.3. Different Drop Policies for Different Types of Traffic

In addition to isolating and provisioning for each class, the thresholds and drop policies also give feedback to responsive flows. The drop policy can be tuned on a per class basis, depending upon the degree of responsiveness expected for a particular class. In the case of the TCP class, a full RED drop policy is applied. The algorithm applies probabilistic drops when the average number of TCP packets in the queue is greater than the minimum threshold, but less than the maximum threshold. In the other extreme, for *other* or multimedia the drop-policy is deterministic. By setting Th_{Max} equal to Th_{Min} for those classes one can effectively deactivate the probabilistic, early drop, component of the drop policy. As a result, depending on the average queue size, either all arriving packets are dropped, or all arriving packets are enqueued. This can be useful for an unresponsive class where the concern is constraint, not feedback. The generality of the drop policy is discussed in greater detail in Section 6.

3.2. Algorithm Description

To better understand the mechanics of the CBT algorithm and how it differs from its counterpart, RED, consider a pseudo-code description. Beginning with the relevant declarations and definitions, the analysis proceeds to examine the logic used to decide when to drop a packet. Then the probabilistic drop test is considered. Finally, the way that the average is computed is presented.

3.2.1. Declarations and Definitions

The declarations and definitions for CBT are shown in Figure 4.2. CBT maintains state for every class. These per-class variables and constants are indexed by the class index, *cl*. In addition to the standard constants that only occur once (notably the limit on queue occupancy, *q_limit*) there are also many per-class constants. Each class has its own weight (for computing the weighted average), maximum drop probability, maximum and minimum thresholds, and drain rate (w_{cl}, $max_{p,cl}$, $Th_{Max,cl}$, $Th_{Min,cl}$, and $drain\_rate_{cl}$, respectively). Drain rate refers to the rate at which packets from this class can be expected to drain from the queue and is a function of that class's bandwidth allocation. This expected drain-rate is a function of the expected average packet size and the speed of the outbound link. The drain rate is notable because it is a factor in the computation of the average queue occupancy. The significance of the drain rate and the weights are discussed in more detail in Section 3.2.5. The procedure for selecting the threshold values is discussed formally in Section 4.1. In addition to these constants, there are also many per-class variables, including the instantaneous and average queue occupancy (q_len_{cl}, q_avg_{cl}), as well as several important counters and flags. The algorithm must track the number of packets for a given class that have arrived since the last drop, $count_{cl}$. It must also note if the class is idle, $idle_{cl}$, and record the time, $last_{cl}$, that the idle state was entered.

There are also a number of general purpose functions that the algorithm relies on. The algorithm needs to be able to *classify*, *enqueue*, and *dequeue* packets. In addition to processing the packets, the enqueue and dequeue operations also update the state variables associated with the packet’s class. Finally, the algorithm must be able to get the current time, *now*, and sample a random number, *random*. In addition to these general functions, there are two functions specific to this algorithm, *drop_early* and *compute_avg*, described below.

```
Constants:
   int   q_limit;       // Limit on maximum queue occupancy
Per-class Constants:
   // There is an instance of these values for each class, cl
   // Classes include the defined classes (e.g. TCP, Other, MM)
   // plus “aggregate”
   float $w_{cl}$;            // Weight for the running avg
   float $max_{p,cl}$;        // Maximum drop probability for RED
   // For most classes $Th_{Max}$ == $Th_{Min}$ to deactivate probabilistic
   // drops for non-responsive flows.
   float $Th_{Max,cl}$;       // Max threshold
   float $Th_{Min,cl}$;       // Min threshold
   float $drain\_rate_{cl}$;  // The drain-rate for the class

Global Variables:
   Class cl;            // Class index
   enum  drop_type {NO_DROP, EARLY_DROP, FORCED_DROP}
Per-class Global Variables:
   // There is an instance of these values for each class, cl
   // Classes include the defined classes (e.g. TCP, Other, MM)
   // plus “aggregate”
   float $q\_len_{cl}$ = 0;    // Current queue occupancy by class cl
   float $q\_avg_{cl}$ = 0;    // Average queue occupancy by class cl
   int   $count_{cl}$ = -1;    // Num packets that have arrived since last drop
   bool  $in\_early\_drop_{cl}$;  // Indicates transition into early drop mode
   bool  $idle_{cl}$;          // Indicates if class has no packets enqueued
   time  $last_{cl}$;          // The time when the class went idle

General Functions:
   Class classify(P)          // returns the class of packet P
   void  enqueue(P,cl);       // Enqueue P and update $q\_len_{cl}$, q_len
   Packet dequeue();          // Dequeue P and update $q\_len_{cl}$, q_len
   void  send(P);             // Transmit P
   void  drop(P);             // Discard P
   Time  now();               // Return current time
   float random();            // Return random between 0 and 1

Defined functions:
   Bool  drop_early(cl);      // Should we do an early drop
   void  compute_avg(cl)      // update q_avg for class cl.
```

Figure 4.2 Definitions and Declarations for CBT

```
for each arriving packet P {
   cl = classify(P);
   // count the number of packets since the most recent drop
   count$_{cl}$++;
   compute_avg(cl);           // update the average for this class

   drop_type = NO_DROP;       // Assume we won't drop this

   // If we've exceeded the queue's capacity
   if ( q_len$_{aggregate}$ >= q_limit  ) {
      drop_type = FORCED_DROP;
   } else if ( q_len$_{cl}$ > 1 ) && (q_avg$_{cl}$ > Th$_{Min,cl}$) ) {
      // Any class is allowed to have 1 packet enqueued.
      if ( q_avg$_{cl}$ > Th$_{Max,cl}$ ) {
         // forced drop mode
         drop_type = FORCED_DROP;
      } else if ( not in_early_drop$_{cl}$ ) {
         // first time exceeding Th$_{Min,cl}$
         // take note (update count & in_early_drop) but don't drop
         count$_{cl}$ = 1;        // initalize count as we enter early mode
         in_early_drop$_{cl}$= true;
      } else {       // Not the first time exceeding Th$_{Min,Cl}$
         // probabilistic drop mode
         // note – we won't reach this state with mm and other
         // as we always set Th$_{Max}$ == Th$_{Min}$ for those classes.
         if ( drop_early(cl) ) {
            drop_type = EARLY_DROP;
         }
      }
   } else {
      in_early_drop$_{cl}$ = false; // avg below Th$_{Min}$.
   }

   if ( drop_type == NO_DROP ) {
      enqueue(P,cl);
   } else {
      drop(P);
      count$_{cl}$ = 0;    //  packet dropped – reinitialize count
   }
} // End for each arriving
```

Figure 4.3 Algorithm for Packet Arrivals in CBT Routers

3.2.2. Drop Modes

When a packet arrives at the router, the algorithm must examine the current conditions and decide whether to enqueue or drop the packet. The pseudo-code for this decision process can be found in Figure 4.3. First, the packet is classified and the class index is returned. Next, the $count_{cl}$ of packets of this class that have arrived since the last drop for this class is incremented. Then the current average is computed for the class. The initial assumption is that the arriving packet will not be dropped so the state variable is set appropriately. Then the actual drop tests begin.

If the queue is full, the packet is dropped. In a well-deployed CBT router, this condition should only happen during periods of sudden and extreme overload. However, if the queue is not full, every class is allowed to have one packet in the queue at all times, to avoid starvation. So, if the class already has a packet in the queue and the average queue size, q_avg_{cl}, for the class exceeds its minimum threshold, $Th_{Min,cl}$, one of the drop modes takes effect. If the average for the class is greater than the maximum threshold, $Th_{Max,cl}$, then the packet must be deterministically dropped. This is a forced drop. Otherwise, if the average is between the minimum and maximum thresholds, the algorithm is executed for a probabilistic drop, *drop_early*. The only exception comes during the transition from the no-drop mode ($q\_avg_{cl} < Th_{Min,cl}$). In that instance, the arriving packet is enqueued, counters ($count_{cl}$) are initialized and flags ($in\_early\_drop_{cl}$) are set to record that the algorithm is in one of the drop modes. The decision not to subject that packet to the drop test is arbitrary. The alternative to being in the forced or early drop modes (i.e., $Th_{Min,cl} > q\_avg_{cl}$) is to be in no-drop mode. The only action for no-drop mode is resetting the drop mode flag, $in\_early\_drop_{cl}$. The final operation is the actual enqueue or dequeue of the packet based on the result of the drop tests.

3.2.3. Departing Packets

While the drop decision occurs when a packet arrives, there is some simple record keeping that also occurs when the packet departs the queue. Figure 4.4 shows the algorithm for packet departures. First a packet is removed from the head of the queue and the class of the packet must be determined. While, the packet classification function is called again for simplicity, optimizations do exist to avoid repeating the packet-identification. The packet is sent to the outbound interface device immediately and then the algorithm performs some record keeping. Since the dequeue operation updates the count of packets of this class in the queue (q_len_{cl}) an examination of that value reveals if the class is idle. A class is *idle* if there are no packets of that class in the queue. If the class was not idle before, then the idle flag, $idle_{cl}$, is set and the current time is recorded. This timestamp, $last_{cl}$, is important as it is the time the last packet was sent before the idle period began. The value and the flag will be used in the calculation of average queue occupancy. Of course, if the class is not idle, the flag is set to false.

```
for each departing packet {
   P = dequeue();               // dequeue and send the packet
   cl = classify(P);
   send(P);
   if ( $q\_len_{cl}$ == 0 ) {         // If the class is idle
      if ( not $idle_{cl}$ ) {
         $idle_{cl}$ = true;
         $last_{cl}$ = now();          // record time class became idle
      }
   } else
      $idle_{cl}$ = false;
} // End for each departing
```

Figure 4.4 Algorithm for Packet Departures in CBT Routers

```
// Note this routine is never called if $Th_{Max}$ equals $Th_{Min}$.
drop_early(cl) {
   local float    $p_a$, $p_b$;
   // first approximate drop probability linearly based on q_avg
   // relative to $Th_{Max}$ and $Th_{Min}$
   $p_b$ = $max_{p,cl}$ * ( $q\_avg_{cl}$ – $Th_{Min,cl}$ )/( $Th_{Max,cl}$ – $Th_{Min,Cl}$ );

   // Then adjust the probability based on the number of packets
   // that have arrived since the last drop.
   $p_a$ = $p_b$/(1 – count * $p_b$ );

   if ( random() < $p_a$ ) return true;
   return false;
} // End drop_early
```

Figure 4.5 Algorithm for Probabilistic Drops in CBT Routers

3.2.4. Early Drop Test

The probabilistic test for early drops, shown in Figure 4.5, is logically exactly the same as the early drop test for RED, shown in Figure 3.9. The key difference is that the metrics and parameters are calculated based on the values for the current packet's class. The primary probability, p_b, is a linear interpolation between zero and the maximum drop probability, $max_{p,cl}$, based on the average queue occupancy's, q_avg_{cl}'s, relative position between the thresholds, $Th_{Max,cl}$ and $Th_{Min,cl}$. The primary probability is then adjusted based on the number of packets that have arrived for this class since the last packet drop, $count_{cl}$, to determine the final drop probability, p_a. Finally, that probability is compared to a random number to decide whether or not to drop the arriving packet.

```
compute_avg(cl) {
   local time  delta_time;
   local int   n;

   // if the queue is still occupied take one sample
   if ( $q\_len_{cl}$ > 0 ) {
      $q\_avg_{cl}$ = ( 1 – $w_{cl}$ ) * $q\_avg_{cl}$ + $w_{cl}$ * $q\_len_{cl}$;
   } else { // if the queue was empty we have to adjust the sampling
      $idle_{cl}$ = false;
   // if the queue is empty, calculate how long the queue has been idle
      delta_time = now() – $last_{cl}$;
   // n packets could have been dequeued during the interval
   // based on drain-rate
      n = delta_time * $drain\_rate_{cl}$ – 1;
   // treat is as if n samples were taken with a qlen of 0
      $q\_avg_{cl}$ = ( 1 – $w_{cl}$ )^n * $q\_avg_{cl}$;
   }
} // End compute_avg
```

Figure 4.6 Algorithm for Computing the Average in CBT Routers

3.2.5. Computing the Average

The computation of the average, shown in Figure 4.6, like the early drop test, is logically the same as in RED (Figure 3.7). The average calculation is done in one of two ways. If the class is currently active (i.e., q_len_{cl} > zero) then the current queue occupancy is sampled and the factored into an updated average, q_avg_{cl}, according to the weight factor for the class, w_{cl}. However, if the class has been idle (q_len_{cl} is zero) the algorithm must approximate the number of queue length samples for that class that would have typically been taken during the idle interval if the class had remained active, but in no-drop mode. Before beginning the computation, the algorithm records the transition from the idle state to the active state by setting the flag, $idle_{cl}$, to false. Then it calculates the duration of the idle interval, *delta_time*, by taking the difference between the current time, *now(),* and the time the idle interval began for the class, $last_{cl}$. Since the class was idle, all of the queue length samples during the idle interval would have been zero values. To determine how many zero samples would have occurred, the drain rate of the class, $drain\_rate_{cl}$, is multiplied by the idle interval. Note that drain rate is determined by the class's bandwidth allocation and packet size so this sampling rate is the same as that used when the class is maintaining load equal to its bandwidth allocation. This value, *n*, indicates the number of packets that could have been dequeued in the idle interval. If packets had been arriving in no-drop mode, every arriving packet would have been enqueued and subsequently dequeued so the number of dequeue operations would have equaled the number of enqueue operations, and thus the arrival rate. Thus, the number of queue length samples that should have been taken is the number of packets that could have been dequeued during the idle interval. Finally, the new average is computed based on *n* samples of queue occupancy of zero.

Note that drain rate of the current class is used to compute the average queue size, not the drain rate of the queue. The drain rate of the queue is simply the capacity of the outbound link expressed in units of an average sized packet. However, since the concern is with the number of packets for a given class that may have been successfully enqueued over the interval, the drain-rate for the class is used. Although all packets share the same queue, the effective drain rate varies from class to class. That drain rate is a function of the class's average share of the outbound link's capacity. The value is computed at the same time as the threshold settings and specified as one of the configuration parameters. The CBT algorithm is straight-forward, essentially a multi-dimensional RED. It is the configuration and use of CBT parameters that makes it an effective resource management mechanism.

4. Configuring CBT

The key to obtaining good performance from CBT lies in accurately setting the threshold values for each class. These threshold settings vary with the expected traffic load. To determine the correct threshold settings one must consider the amount of bandwidth to allocate for each class. Naively, one may assume that setting the ratio of the thresholds between the classes equal to the desired ratio of their bandwidths would give the desired result. Unfortunately, the thresholds are allocated in terms of packets and the classes may have widely varying average packet sizes. For example, Table 4.1 shows the packet sizes and maximum load generated for each class of traffic used in these experiments. For example, the BULK traffic has an average packet size of 1, 499 bytes, while Proshare has an average packet size of 728 bytes. (We must also include the packet headers in the total packet size. In these experiments the TCP headers are 60 bytes and the UDP headers are 28 bytes.) With the packet sizes of BULK and Proshare, setting a 1:1 ratio between the thresholds would result in a 2:1 ratio in bandwidth on the outbound link.

Traffic Type	**Packet Size**
BULK	1439 + *60* = 1499
HTTP	1062 + *60* = 1122
UDP Blast	1047 + *28* = 1075
Proshare	700 + *28* = 728
MPEG	811 + *28* = 839

Table 4.1 Packet Sizes and Desired Bandwidth by Traffic Type

So, to allocate bandwidth one must consider the ratio in terms of the number of bytes represented by the threshold settings between the classes. Determining the correct threshold settings for a given traffic mix requires using a set of formulas based on the bandwidth allocations, the average packet size for each class, the capacity of the outbound link, and the desired total latency.

4.1. Equations for Bandwidth Allocation

Name	Symbol	Units
Outbound Link Capacity	$C_{outbound}$	KB/s
Number of Classes	N	classes
Queue induced latency	L	milliseconds
Threshold for class i	Th_i	packets
Bandwidth allocation for class i	B_i	KB/s
Average packet size for class i	P_i	KB

Table 4.2 Variables and Constants for Setting CBT Thresholds

Table 4.2 shows the symbols corresponding to the parameters used to calculate the proper threshold settings. Given the constants and configuration inputs the threshold for a given class can be defined with with equation 4.1.

$$Th_j = \frac{B_j L}{P_j} \tag{4.1}$$

The threshold for a given class, *j*, is simply a function of the desired bandwidth allocation, B_j in bytes per second, the average packet size for the class, P_j in bytes, and the expected average worst-case latency, *L*. Expected average worst-case latency refers to the acceptable queue-induced latency on the CBT router. It specifies an upper limit on the average latency that is acceptable. Note that *L* is the same for all classes since all packets share the same queue. There may be times where the latency exceeds this bound. For example, if a large burst of traffic arrives when the average is small the queue occupancy, and thus the queue-induced latency, can grow quite large before the average activates the drop policy. But on average latency should never exceed the specified value. This equation also assumes that the sum of the bandwidths for all classes sums to the capacity of the link. The rest of this section explains the derivation of this equation.

The expected average worst-case latency can be defined as shown in Equation 4.2. This expression represents the sum of the worst-case average number of bytes enqueued for each class divided by the speed of the outbound link. Any packet placed in the outbound queue should, on average, spend no longer in the queue than the time it takes for the queue to drain. In the worst case, all classes are maintaining an average queue occupancy equal to their threshold value. (This analysis measures occupancy in packets.

However, it could be measured in bytes.) The amount of time it takes for the queue to drain is *L*.

$$L = \left(\sum_{i=1}^{N} P_i Th_i \right) \frac{1}{C_{outbound}} \tag{4.2}$$

The bandwidth allocated for each class can be expressed as shown in Equation 4.3. Each class's minimum average bandwidth during congestion is a share of the outbound link equal to its share of the total bytes enqueued. This assumes that since a period of congestion is being considered, the queue always contains at least one packet (i.e., the link is fully utilized).

$$B_i = \frac{P_i Th_i}{\sum_{j=1}^{N} \left(P_j Th_j \right)} C_{outbound} \tag{4.3}$$

Equation 4.3 is based on the assumption that the sum of all of the bandwidth allocations should equal the capacity of the link, as shown in Equation 4.4. It is this assumption that allows us to equate shares of the queue to shares of the outbound link capacity.

$$C_{outbound} \Leftrightarrow \sum_{i=1}^{N} B_i \tag{4.4}$$

Given the desired bandwidth allocations and average packet sizes for two classes on the outbound link,one can determine the ratio between the threshold settings. This is the key to using allocation of queue space to manage allocation of bandwidth on the outbound link. The outbound queue can be thought of as an extension of the outbound link, with the ratios of bytes in the queue matching the ratios of bytes on the link. The relationship between the thresholds for two classes can be expressed as shown in Equation 4.5. The ratio between the bandwidth allocated to each class is also the ratio between the maximum average number of bytes that can be enqueued.

$$\frac{B_i}{B_j} = \frac{P_i Th_i}{P_j Th_j}$$

$$\frac{B_i P_j}{B_j P_i} = \frac{Th_i}{Th_j} \tag{4.5}$$

Starting with the second half of this expression, the threshold requirement for a given class (Th_i) in the queue can be expressed in terms of the threshold for another class (Th_j) as shown in Equation 4.6.

$$Th_i = \frac{B_i P_j Th_j}{B_j P_i} \tag{4.6}$$

Substituting 4.6 into 4.2, the latency in terms of the threshold of a single class, *j*, is shown in Equation 4.7.

$$L = \left(\sum_{i \neq j}^{N} \frac{B_i P_j Th_j}{B_j} + P_j Th_j \right) \frac{1}{C_{outbound}} \tag{4.7}$$

Using this equation, Th_j's value can be derived as shown in Equation 4.8 and then, using the relationship from Equation 4.6, the other threshold values can be derived.

$$L = \left(\sum_{i \neq j}^{N} \frac{B_i P_j Th_j}{B_j} + P_j Th_j \right) \frac{1}{C_{outbound}}$$

$$L = \left(\sum_{i=1}^{N} \frac{B_i P_j Th_j}{B_j} \right) \frac{1}{C_{outbound}}$$

$$L = \frac{\left(\sum_{i=1}^{N} \frac{B_i}{B_j} \right) P_j Th_j}{C_{outbound}}$$

$$Th_j = \frac{L \cdot C_{outbound}}{P_j \left(\sum_{i=1}^{N} \frac{B_i}{B_j} \right)} \tag{4.8}$$

Since the bandwidth of all of the classes sums to the total outbound link capacity (Equation 4.4) Equation 4.8 can be simplified to that shown in Equation 4.9. This allows the threshold for a given class to be expressed in terms independent of the other classes.

$$Th_j = \frac{L \cdot C_{outbound}}{\frac{P_j}{B_j} \sum_{i=1}^{N} B_i} = \frac{B_j L \cdot C_{outbound}}{P_j C_{outbound}} = \frac{B_j L}{P_j} \tag{4.9}$$

This value, Th_j, is used as the maximum threshold for a given class. For unresponsive classes Th_{Max} should be set equal to Th_{Min} and used to constrain those classes. For responsive flows, like TCP, Th_{Min} should generally be set according to the rules of thumb used for RED. The intention of the minimum threshold is to allow for some small amount of buffering without triggering the packet dropping mechanism, while still dropping some packets early to give feedback before congestion becomes severe. The authors of RED recommend a minimum threshold setting of 5 packets and we accept their recommendation. Determining the correct threshold settings was the emphasis of the selection of optimal parameters. Those experiments conducted to determine those settings are found in Appendix B.

4.2. Setting the Other CBT Parameters

Thresholds have the most obvious effect on the isolation and allocation in CBT. However, other parameters perform as their RED counter-parts, allowing one to fine-tune the behavior ofthe algorithm. Each class has a weighting value and a maximum drop probability. And, of course the queue itself is of finite size. We discuss each of these parameters below and recommend default settings for each.

4.2.1. Weights for Computing Average Queue Occupancy

As with RED, weights control how sensitive each class is to bursts. For example, with multimedia the algorithm must tolerate the arrival of large data units (e.g., I Frames) that are fragmented over many packets and arrive nearly simultaneously without triggering the drop mechanism. However, an aggressive flow should be strictly constrained to avoid it stealing more than its fair share by sending large bursts. Heuristics determine the weight values for the different classes. In addition to desired sensitivity to bursts, the relationship between threshold size and the relative weighting are also considered. If a class has a small threshold value, each sample must be heavily weighted to avoid requiring extreme over-runs to trigger the drop mechanism. Taking this approach helps to avoid oscillations in queue occupancy when the drop mechanism allows a large number of packets to be enqueued and then forces the class to remain nearly idle for a lengthy period while the queue drains to have only one packet enqueued. While there is room for refinement and formalization in this area, the focus here is on accurately selecting the threshold settings.

4.2.2. Maximum Drop Probability

The guidelines for setting the drop probability in CBT are the same as those for RED. For many classes, the thresholds are set to be equal, thereby eliminating the probabilistic drop mode, the maximum drop probability does not come into play. However for classes where it does come into play, the current recommendations for RED are followed and a value on the order of 0.1 [Floyd97a] used. This value is selected to result in a small number of drops when the average queue size is slightly higher than the minimum thresholds. The recommended setting of 0.1 is used throughout.

4.2.3. Maximum Queue Size

The sum of the thresholds defines the maximum average queue occupancy while the maximum queue size, *q_limit*, sets the limit on the maximum instantaneous queue size. There is an upper limit on the queue occupancy that can be reached without activating the forced drop mode (due to the average queue occupancy exceeding the maximum threshold) and one can calculate that value for a given set of thresholds and weights. However, in these experiments the maximum queue size is simply set to 240. This size is sufficient to accommodate bursts that wouldn't otherwise increase the average beyond the maximum threshold.

5. Demonstrating the Effectiveness of CBT

To demonstrate CBT’s capability to control bandwidth allocations and control latency, we present empirical results from the use of CBT in a controlled environment. We conducted experiments to determine how the bandwidth allocations and latency settings compare to the observed values in a laboratory setting. These experiments were conducted in the experimental configuration discussed in Appendix A. A full comparison of function and performance of CBT with other AQM schemes is presented in Chapter V.

Consider CBT's behavior in two situations. In the first case each traffic class generates a load sufficient to fully consume its allocated bandwidth. In that case, the throughput of each class should correspond to that class's bandwidth allocation. Similarly the measured end-to-end latency should closely follow the desired latency setting. The second case is subtler and highlights another facet of CBT, controllable re-allocation of unused bandwidth. The second case is one where some classes do not use all of their bandwidth allocation. To understand CBT's behavior, equations are presented that explain how bandwidth will be reallocated between the other classes based on the initial allocations and the loads generated by the classes of traffic. Additional equations explain how the average latency will be effected by the traffic loads. These experiments are then verified with experimental results.

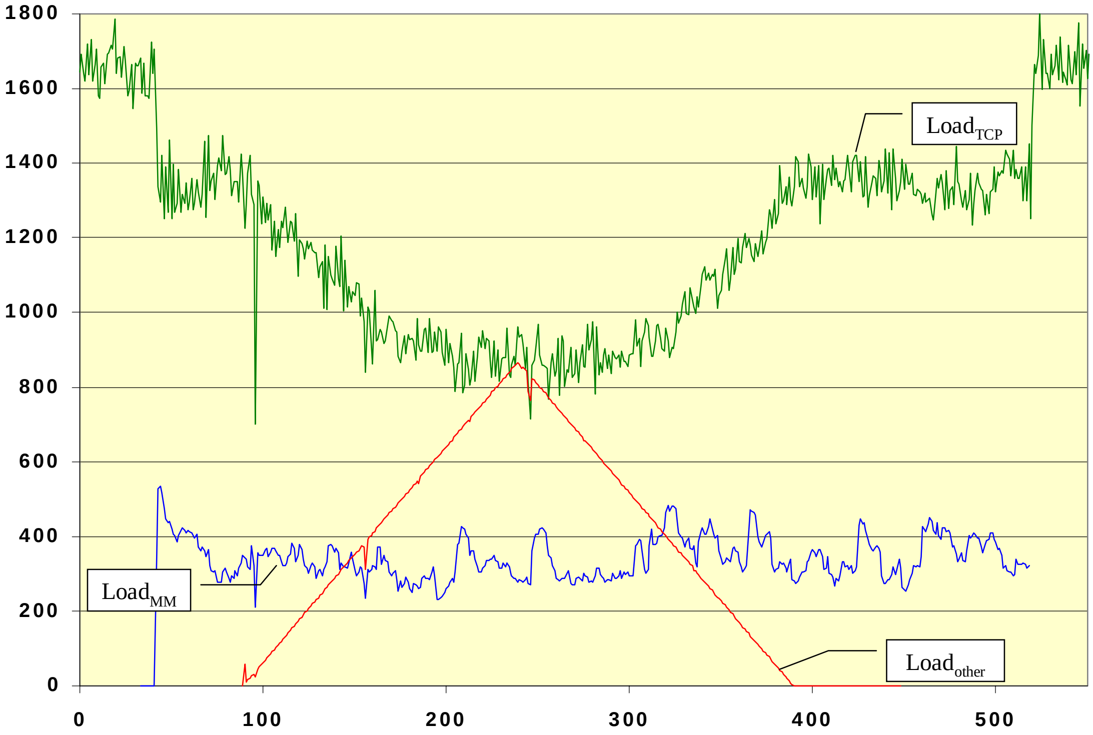


Figure 4.7 Traffic Loads (KB/s) over Time (seconds)

To illustrate CBT's behavior, three traffic classes are used: TCP, multimedia, and *other* passing through a router servicing a bottleneck link. In this experiment the router has an inbound link with a capacity of 100 Mb/s. However, the outbound link's capacity is only 10 Mb/s. The loads for each of these classes are show in Figure 4.7. These loads are measured on the inbound link of the router servicing the bottleneck link. The solid lines indicate the load generated by each class of traffic on the router's inbound link. In all of the plots in this section each data point represents the average value of the metric over a 1 second interval. The TCP load (Load$_{TCP}$) consists of bulk transfers of data capable of generating a load in excess of 1,600 KB/s. Note that Load$_{TCP}$ does decrease in response to the introduction of other traffic types because of downstream congestion that is caused by other traffic and TCP's responsive nature. However, the TCP load is still sufficient to exceed the bandwidth allocations and demonstrate CBT's effectiveness. The multimedia traffic consists of 8 MPEG streams capable of generating an average load ($Load_{MM}$) of 340 KB/s. In order to illustrate how CBT manages changes in traffic load, the load ($Load_{other}$) generated for the class *other* ranged linearly from 0 to 800 KB/s and back down to 0 KB/s. The classes were allocated bandwidth as show in Table 4.3. This table also shows the threshold in packets for each class. Note that these allocations were selected for their illustrative value, not to insure good performance for multimedia or TCP. The thresholds for each class are computed using equation 4.1, a desired latency of 100ms, and the average packet sizes for each class as shown.

	Load (KB/s)	**Bandwidth Allocation (KB/s)**	**Packet Size (KB)**	**Threshold (Packets)**
TCP (BULK)	> 1600	575	1507	39.07
Multimedia(MPEG)	340	150	807	19.04
Other	0-800	500	1075	47.63

Table 4.3 Loads and Allocations for CBT with Latency of 100ms

Figure 4.8 adds the bandwidth allocations (horizontal dashed lines) from Table 4.3 to the graph of each class's generated load. Note that vertical dashed lines at times 176 and 304 indicate an interval wherein all classes's loads exceed their bandwidth allocation. The load for TCP and multimedia always exceed their respective allocations; however the traffic class *other* only exceeds the allocation between time 176 and 304. Table 4.4 lists the times of key events in this experiment.

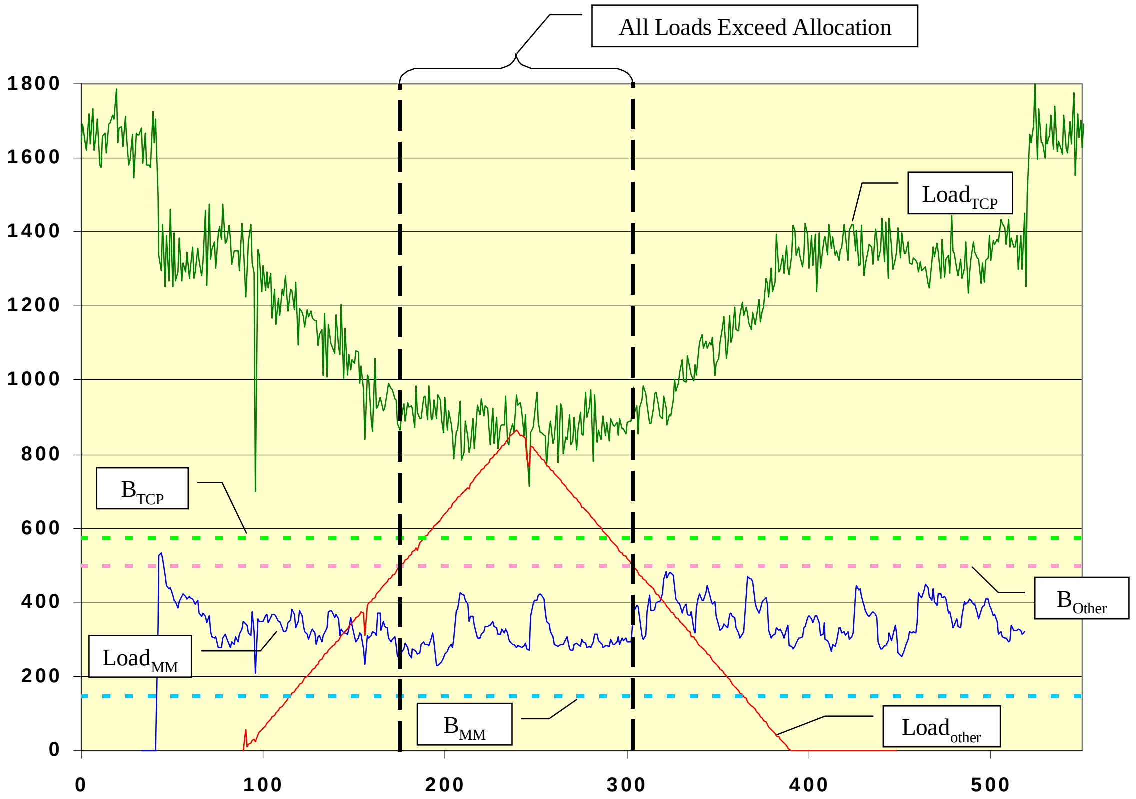


Figure 4.8 Traffic Loads and Bandwidth Allocations (KB/s) over Time (seconds)

Time	Event
0	Begin recording, TCP present.
42	MPEG traffic starts.
90	*Other* traffic starts.
176	$Load_{Other}$ exceeds B_{other}
304	$Load_{Other}$ drops below B_{other}
390	*Other* traffic stops.
515	MPEG traffic stops.
607	Trace ends.

Table 4.4 Timestamps for Key Events

5.1. CBT Behavior When All Classes Consume Their Bandwidth Allocation

We first focus on the period when the loads for all classes exceed the bandwidth allocations for those classes. That is the period from time 176 to 304. Figure 4.9 zooms in on this interval. The figure shows the inbound load and bandwidth allocations for each class. In this scenario, CBT should constrain all classes to their allocated bandwidth while managing the total queue occupancy to keep the queue-induced latency to the desired value.

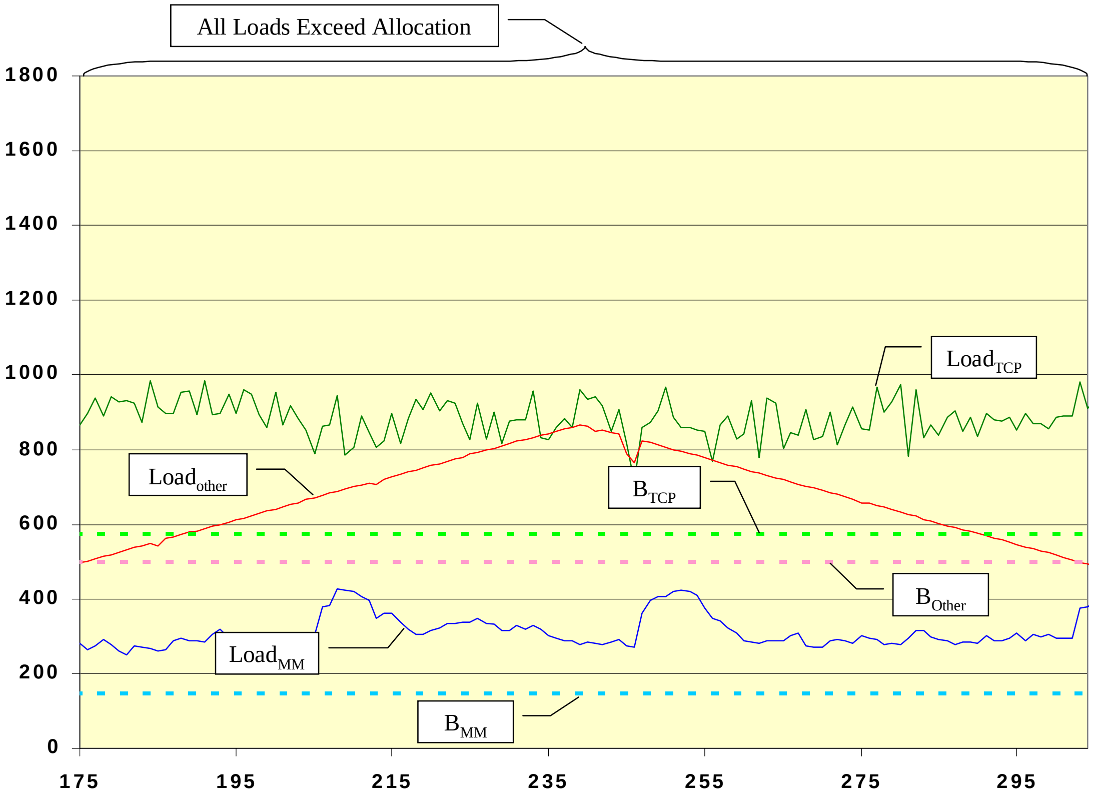


Figure 4.9 Traffic Loads (KB/s) over Time (seconds) with All Loads Exceeding Allocation

5.1.1. Throughput

Figure 4.10 compares the throughput for each traffic class, plotted as individual points, to the bandwidth allocation (B_{class}), plotted as a dashed line. In all cases the throughput corresponds to the desired bandwidth allocation. However, note that at times the throughput for a class may exceed the intended allocation. This is because allocation in CBT is a coarse grained mechanism and hence the constraints that CBT offers are not precise. Moreover, these slight fluctuations may be due to borrowing as some class's load briefly drops below its allocations.

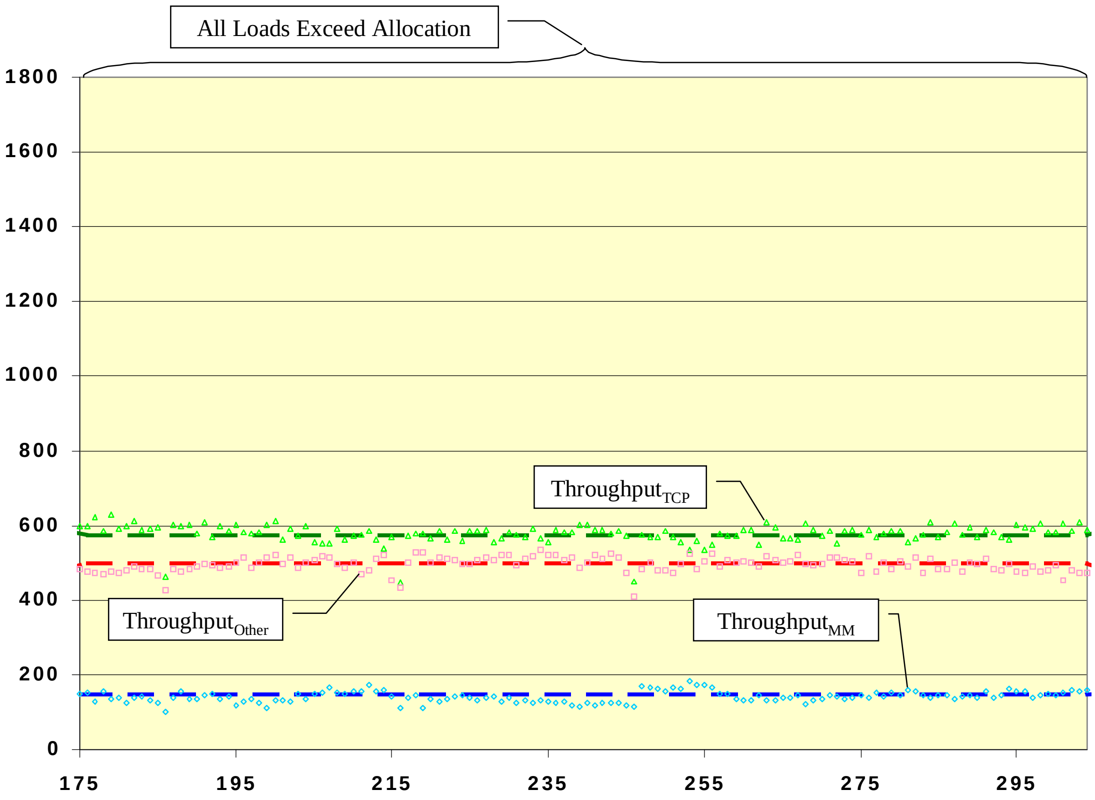


Figure 4.10 Throughput (KB/s) over Time (seconds) for each Traffic Class

5.1.2. Latency

CBT manages latency by adjusting the thresholds to constrain the maximum average queue occupancy. Limiting the average queue occupancy also limits the queue-induced latency. CBT's effectiveness at managing latency when the loads for all classes exceed their bandwidth allocations can be demonstrated in two ways. First, consider the latency during the experiment above. Figure 4.12 shows the latency setting ($L_{intended}$) of 100 milli-seconds as a dashed line. The latency values observed are shown as individual data points. If CBT is effective at managing latency all of the data points should be below the dashed line. However, the average latency slightly exceeds the intended latency for most data points. This difference can be accounted for by reviewing the experimental technique. While the intended latency is a setting for the queue-induced latency, the intend latency shown here is end-to-end latency because latency was monitored at the end-systems. Forwarding, propagation, and end-system delays all contribute to the ~5-10 ms error shown. Further, since allocation in CBT is a coarse grained mechanism it is not expected to offer a precise limit on latency. Rather, it should limit latency to approximately the desired value. Thus, this data does indicate that CBT is effective in managing latency.

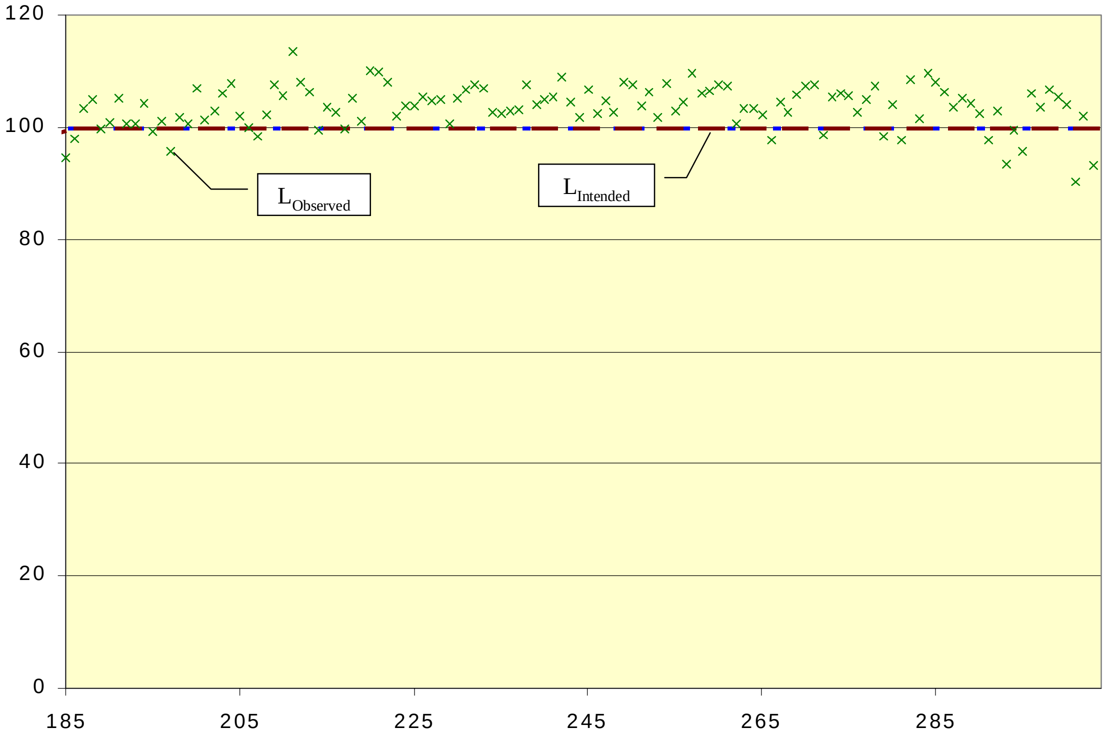


Figure 4.12 Intended and Observed Multimedia Latency (ms) vs. Time (seconds)

While Figure 4.12 shows that CBT is effective in constraining the latency for one experiment, we also want to demonstrate CBT's effectiveness a second way, as we range the desired latency value. To do this, multiple experiments were conducted, each with a different desired latency setting. The latency for each packet of a multimedia stream was recorded and the average latency observed during the entire measurement period was computed. Table 4.5 shows the intended CBT latency vs. the latency actually observed. The data shown here was collected during the blast measurement period using a traffic mix of BULK-MPEG. (See Appendix A for an explanation of the details of this experimental setup.) Throughout the blast measurement period the aggregate load exceeds the capacity of the outbound link so congestion is persistent and latency is directly related to the queue occupancy in the router. In this experiment all traffic types fully utilize their allocated capacity. If CBT performs as expected the observed average latency should be less than or equal to the intended latency setting. Again, the observed latency exceeds the intended value because the latency measured was end-to-end latency which includes propagation and end-system delays and accounts for the approximately 5 ms differences observed.

Latency	
Intended	**Observed**
20	23
40	44
60	64
80	85
100	103

Table 4.5 CBT Intended Latency vs. Observed Latency (ms)

CBT is effective at managing bandwidth allocation and latency when when the load for each class exceeds that class's bandwidth allocation. Moreover, in this scenario it is easy to understand the relationship between CBT's parameters and the resulting performance for the different traffic classes. However, the behavior when some class fails to fully utilize its bandwidth allocation is more complex. This scenario is discussed below.

5.2. CBT Behavior When Overprovisioned

While CBT’s bandwidth allocation is straight-forward, it is important to realize that the actual relationships being defined are not strict limits on bandwidth, but rather ratios between the classes. When all classes are using their full allocations they are limited to the specified bandwidth allocation. But these ratios apply even when each class's offered load does not correspond to its allocated bandwidth. For example, if some class is idle, the unused link capacity allocated to that class is available to the other classes in *proportion to their bandwidth allocations.* This borrowing is max-min fair and is explained formally below. Moreover, when some class is not fully utilizing its bandwidth allocation it will not maintain a queue occupancy equal to its threshold setting so the queue-induced latency will be less than the maximum setting. Recall that the thresholds represent upper bounds on the average queue occupancy for a class and that the ratios between the average queue occupancies for the classes is also the ratio between the throughput for the classes. Consider an example. Suppose some class, *i*, has average queue occupancy of 1000 bytes while some other class, *j*, has average queue occupancy of 2000 bytes. Then over the interval it takes the average queue to drain, class *i* will transmit 1000 bytes to 2000 bytes for class *j*. Class *j's* throughput is twice that of class *i*, in ratio to the queue occupancy. This ratio is independent of the queue occupancy by any other classes. Assume only three classes share the queue and this third class, *k*, has 7000 bytes in the queue. Then over the interval necessary to transmit 10,000 bytes, class *i*, *j*, and *k* will respectively receive 10%, 20%, and 70% of the link's capacity. Now consider the case in which class *k* generates no load and thus has no packets enqueued. Classes *i* and *j* are still constrained to their average queue occupancies so they have 1000 and 2000 bytes enqueue on average. However, because class *k* is not present, the average aggregate queue occupancy is only 3000 bytes. However, during the interval necessary to transmit 3000 bytes, class *i* will transmit 1000 bytes to the 2000 bytes for class *j*. The ratio between the classes maintain their full queue occupancy remains the same. However, the shares of the link capacity increase. Class *i* represents 1000 of the 3000 bytes transmitted, 33%, while class *j* represents 2000 of the 3000 bytes, 66%. Although the average queue occupancy for each class does not change, the resulting throughput increases for each.

We present equations below that explain how bandwidth is reallocated when one or more classes do not use their allocated bandwidth. It is important to realize that the *reallocations* described below are conceptual, simply to elucidate the type of borrowing that occurs. There is no explicit mechanism for borrowing; it is simply a side effect of the queue management. The actual algorithm only enforces the limits on queue occupancy for each class. This section simply offers insight into the resulting behavior.

5.2.1. Equations for Predicting Reallocation of Bandwidth

One may consider the resource allocation in terms of shares for each class. The share, s_i , assigned to given class, *i*, is defined as the fraction of the link capacity (*C*) allocated to that class (B_i) or as the fraction of the queue's maximum occupancy allocated to that class. Both are shown in Equation 4.10.

$$s_i = \frac{B_i}{C_{outbound}} = \frac{P_i Th_i}{\sum_{j=1}^{N} \left(P_j Th_j \right)} \tag{4.10}$$

Since each class's share is its bandwidth allocation as a fraction of the outbound link's capacity it is straight-forward to show that the shares sum to one, as shown in equation 4.11 using equation 4.4.

$$\sum_{i=1}^{N} s_i = \sum_{i=1}^{N} \frac{B_i}{C_{outbound}} = \frac{C_{outbound}}{C_{outbound}} = 1 \tag{4.11}$$

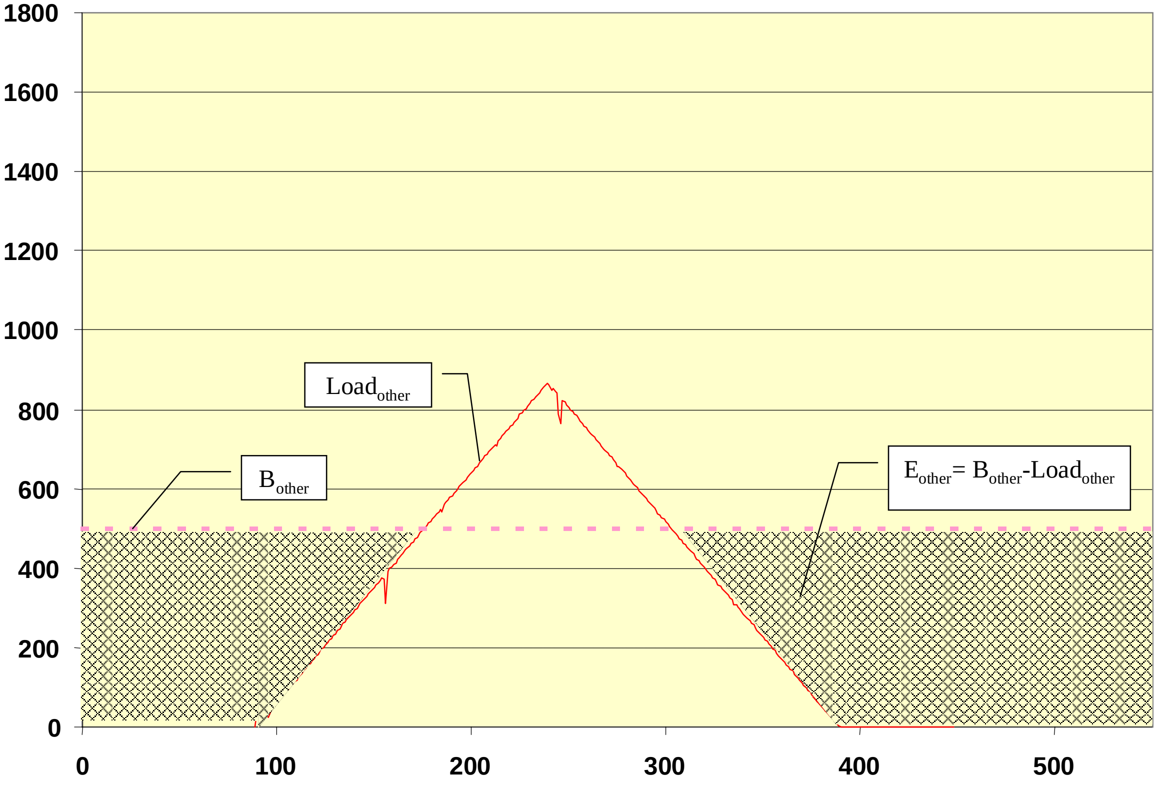


Figure 4.16 Relation Between B_{other}, $Load_{other}$, and E_{other} (KB/s) over Time (seconds) for Multimedia

To illustrate the borrowing in CBT, first consider Figure 4.16. This figure shows the load (L_{other}) and bandwidth allocation (B_{other}) for the class *other* during the experiment. It also shows the excess capacity (E_{other}) that is unused by class *other*. The excess is indicated by the cross-hatched region. The excess at any given time is the difference between the bandwidth allocation and the actual load at that instant. This excess link capacity may be borrowed by the other classes. The reallocated bandwidth for a class, *i*, is referred to as $B'_{i.}$

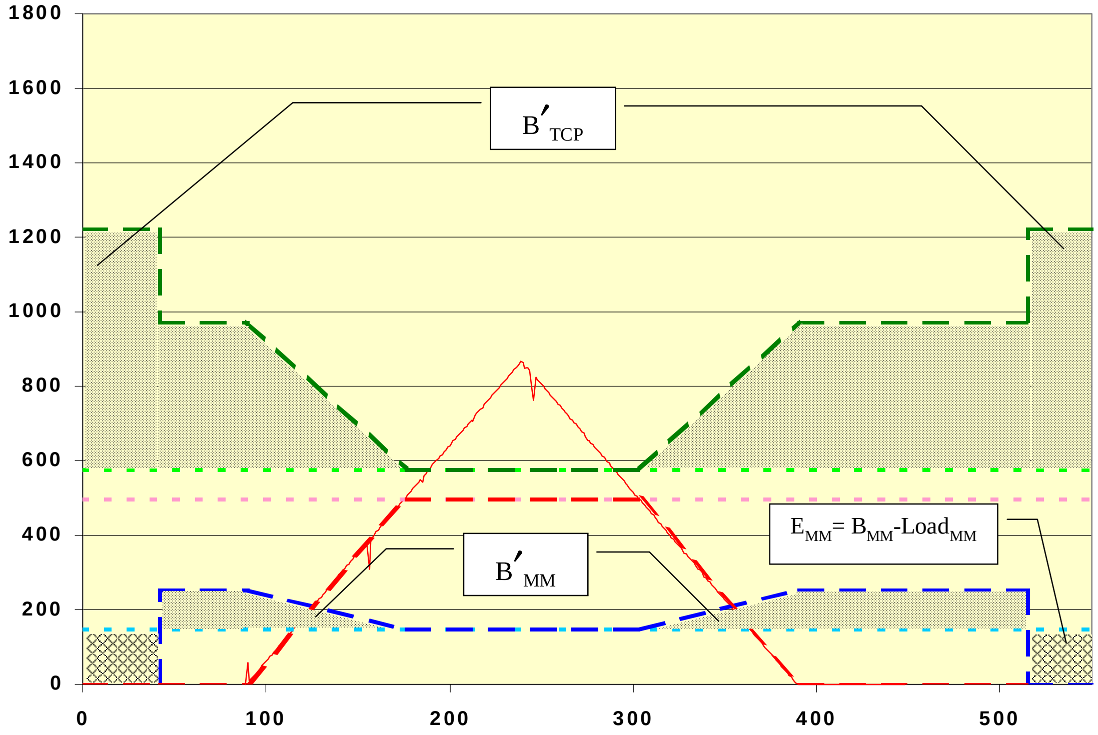


Figure 4.17 B′ (KB/s) over Time (seconds)

Figure 4.17 illustrates the borrowing that takes place in CBT. This figure presents the bandwidth allocations (horizontal short dashed lines) and bandwidth reallocations (long dashed lines) for each class. It also shows the load for the class *other* as a solid line. To simplify the figure the loads for TCP and multimedia are not shown. However, those loads exceed both *B* and *B′* for both classes from time 42 to 515. Recall from Figure 4.7 or Table 4.4 that during the periods from time 0 to time 42 and 515 to 607 only TCP traffic is present in the experiment. During these periods the allocations of both multimedia and *other* go unused. Since the queue is serviced in FIFO fashion and TCP is the only class with packets enqueued, TCP is able to consume this excess capacity. This excess (E_{mm} + E_{other}) is effectively reallocated to the classes that have sufficient load to use this excess. Only TCP has load greater than its allocation during the period (0,42). So B'_{TCP} is equal to B_{TCP} + E_{mm} + E_{other}. However, once multimedia begins generating load at time 42, both TCP and multimedia exceed their allocations and borrow from E_{other} in proportion to their original allocations. At time 90, the load for *other* increases so the borrowing by TCP and multimedia decrease correspondingly until at time 176, $Load_{other}$ exceeds B_{other}.

At that point all classes have loads greater than their bandwidth allocations so borrowing ceases until Load$_{\text{other}}$ drops below B$_{\text{other}}$ at time 304. In this under-provisioned case, no borrowing is possible so *B′* is equal to *B* for all classes.

Whenever some class is over-provisioned, *B′* diverges from *B* for those classes where *Load* is not equal to *B*. *B′* may be greater than, less than, or equal to B. As previously noted, *B′* may exceed B as in the case of TCP and multimedia. Notice that in the case of *other* B'_{other} is *less than* B_{other}. The reallocation for *other* is reduced to the actual load. The difference (shown in cross-hatch) between the original allocation, B_{mm}, and the recalculated allocation, B'_{mm}, is available for the other classes to borrow. The excess, E_i, for a given class is the difference between the allocation and the load as shown in Equation 4.12.

$$E_i = \max(B_i - Load_i, 0) \tag{4.12}$$

The sum of the excess across all of the classes is divided among the other under-provisioned classes according to their shares as calculated with Equation 4.10. The recalculated allocation (*B′*) is simply the original allocation (*B*) plus the appropriate share of the over-provisioned classes' excess.

Now let us consider the equations used to compute these reallocations. Because of the iterative nature of this calculation the notation is amended to include an indication of the current iteration. For example, $B'_{\text{j,k}}$ is the k-th iteration of *B′* for class j. So $B'_{\text{j,0}}$ is B$_{\text{j}}$. When only one class, *i*, has excess $B'_{\text{j,1}}$, is defined in Equation 4.13.

$$B'_{j,1} = B_j + \frac{s_j}{(1-s_i)}(B_i - Load_i) \tag{4.13}$$

When more than one class's load is less than its allocated bandwidth, all of the unused provisioned bandwidth can be expressed by the term shown in Equation 4.14. It is simply the sum of the difference between the provisioning and the actual load across all of the classes that are not fully using their allocation.

$$E_{all,k} = \sum_{\exists i, B_i > Load_i} (B_i - Load_i) \tag{4.14}$$

Each class that is exceeding its provisioned bandwidth will be able to borrow capacity from the unused provisioned bandwidth based on its share in ratio to the shares of the other classes that are exceeding their allocation. This ratio, for class *j*, is expressed by the term shown in Equation 4.15.

$$\frac{s_j}{\sum\limits_{\exists i, B_i < Load_i} s_i} \tag{4.15}$$

More generally, the first iteration of the expected bandwidth $B'_{\mathrm{j,1}}$ for a given class, including borrowing from classes that aren't using their full allocation, can be expressed as shown in Equation 4.16.

$$B'_{j,1} = B_j + \frac{s_j}{\sum\limits_{\exists i, B_i < Load_i} s_i} \sum_{\exists i, B_i > Load_i} \left(B_i - Load_i \right) \tag{4.16}$$

While Equation 4.16 expresses the bandwidth available to each class it is possible that after reallocation, for a given class *l*, $B'_{l,k} > Load_l$. As a result, that difference may, once again be divided among the classes where $B_{j,k}<Load_j$. So, ultimately, the approach is iterative until $B'_{j,k}$ is less than or equal to $Load_j$ for all classes. Equation 4.17 illustrates the iterative solution.

$$B'_{j,k} = \min\left(Load_j, B'_{j,k-1} + \frac{s_j}{\sum\limits_{\exists i, B'_{i,k-1} < Load_i} s_i} \sum_{\exists i, B'_{i,k-1} > Load_i} \left(B'_{i,k-1} - Load_i \right) \right) \tag{4.17}$$

We refer to the final converged bandwidth allocation with the short-hand B'_{j}. B'_{j} is the n-th iteration, $B'_{j,n}$ where n is defined as shown in Equation 4.18. That is, the final iteration is the one where for every class, *j*, the n-th reallocation of bandwidth is less than or equal to the load for that class.

$$n \middle| \forall j, B'_{j,n} \le Load_j \tag{4.18}$$

Recall that this iteration is only necessary to predict and understand how the bandwidth is allocated between the classes when classes are overprovisioned. As explained above, the actual queueing mechanism does none of these calculations; it simply compares a class's average queue occupancy to the threshold for that class.

Finally, note that this calculation converges within N iterations, where N is the number of classes. It will never be necessary to iterate more than the number of classes. In order to continue iterating some class must be underprovisioned and some other class must be newly overprovisioned. A class can only be newly overprovisioned once because once a class is overprovisioned its reallocation is adjusted to match the actual load for that class and the computation never changes the reallocation for a class that is exactly provisioned. Thus, there can only be N separate instances of a newly overprovisioned class, where N is the number of classes. In the worst case, each of these new overprovisionings occurs on a different iteration, requiring N iterations.

Consider an exampleof determining the value of B' through this iterative process. It is possible that when the excess from the initially over-provisioned class(es) is reallocated among the other classes one or more of the borrowing classes will have a reallocation greater than that class's load. As a result, this new excess will have to be divided among the other classes that are still under-provisioned. Table 4.6 shows some starting bandwidth allocations for TCP, multimedia, and other traffic classes along with the loads generated by those traffic classes for the purposes of this example.

***B* (KB/s)**			**Load (KB/s)**		
TCP	**MM**	**Other**	**TCP**	**MM**	**Other**
935	140	150	700	160	1000

Table 4.6 Sample Bandwidth Allocations and Loads for CBT with link capacity of 10 Mb/s.

Table 4.7 shows the resulting excess and resulting reallocations of that excess bandwidth as analysis iterates to predict the reallocation of excess bandwidth. In the first iteration, no bandwidth has been reallocated so the B' is equal to B for all classes. Since TCP's load is only 700 KB/s, this leaves an excess (E) of 235 KB/s. The other two classes have no excess allocation and both have their load greater than their allocation so TCP's excess is distributed between the multimedia and *other* in proportion to their initial allocations. In addition, TCP's reallocation is adjusted to match its actual load. As a result, in iteration 1, TCP is allocated 700 KB/s, multimedia is allocated 253 KB/s, and *other* is allocated 272 KB/s. Now multimedia's reallocation exceeds its actual load, leaving an excess of 93 KB/s. Since *other* is the only class which is still underprovisioned, all of the excess is reallocated to that class in iteration 2, leaving us with final reallocations of B'_{tcp} of 700 KB/s, B'_{mm} of 160 KB/s, and B'_{other} of 365 KB/s. The iterations cease here because no class has excess available to reallocate.

Iteration	***B′* (KB/s)**			***E* (KB/s)**		
	TCP	**MM**	**Other**	**TCP**	**MM**	**Other**
0	935	140	150	235	0	0
1	700	253	272	0	93	0
2	700	160	365	0	0	0

Table 4.7 Sample Iterative Reallocations and Excesses for CBT

This calculation of B' points out an important aspect of CBT, *predictability*. The algorithm not only controls the allocation of the bandwidth when all classes are running at capacity, its behavior can also be predicted when some classes are generating loads less than their allocated capacity. This borrowing and predictability apply over time scales on order of 10ths of seconds (e.g. the amount of time it takes the average queue occupancy to change). If there is little variation in the loads generated by the classes, the borrowing and predictability can extend to larger time scales.

This predictability is an added bonus in this scheme. The same predictability applies to the latency calculation.

5.2.2. Predictability of Throughput Reallocation

To confirm the accuracy of the equations for reallocation of bandwidth, B'_{TCP}, B'_{MM}, and B'_{other} were computed for the experimental bandwidth allocations and observed loads. These computed values were then compared to the actual throughput observed for each class. Recall the loads and bandwidth allocations in this illustrative experiment, shown in Figure 4.18.

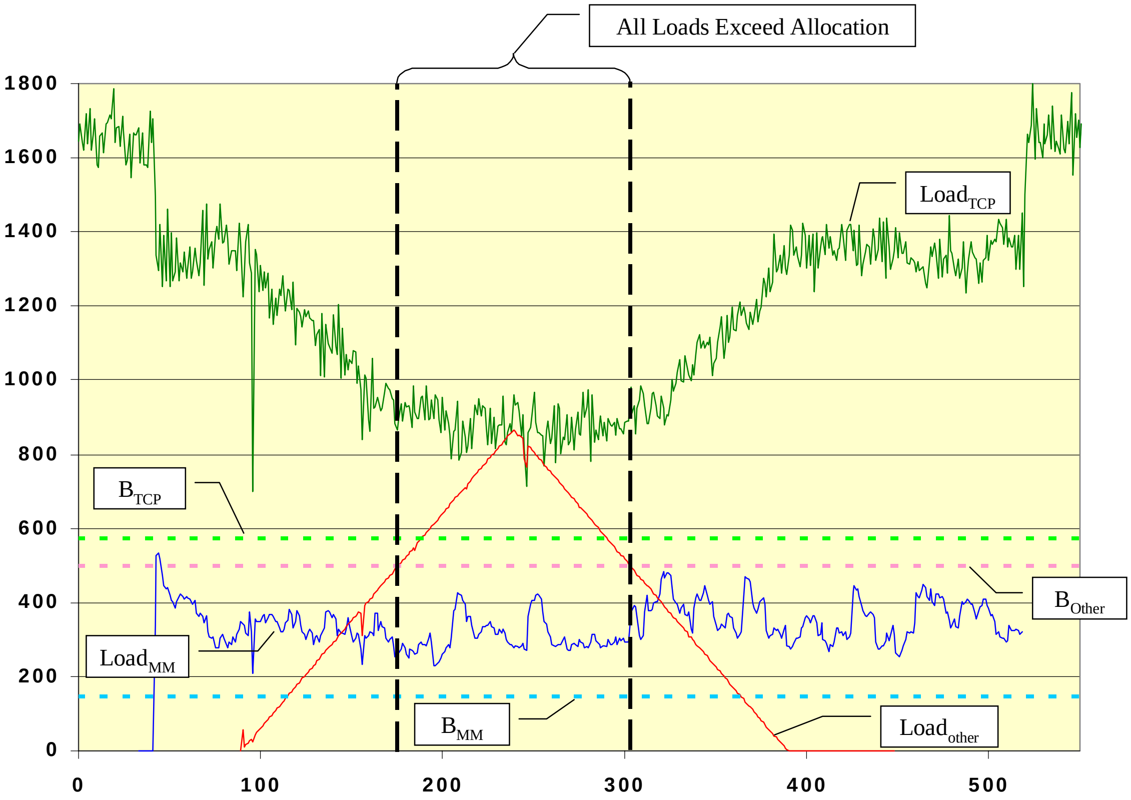


Figure 4.18 Loads and Bandwidth Allocations (KB/s) over Time

The bandwidth allocations remain constant throughout the experiment. The Load_{TCP} exceeds B_{TCP} and B'_{TCP} throughout the experiment. The reallocations to each class change when the loads generated by the classes change in significant ways. These changes occur at the times shown in Table 4.8.

Start	**Stop**	**Event**	**E_{mm}**	**E_{other}**	**B'_{TCP}**	**B'_{MM}**	**B'_{other}**
0	42	TCP only	150	500	1225	0	0
42	90	MM and TCP	0	500	972	253	0
90	176	*Other* traffic increasing	0	500 to 0	972 to 575	253 to 150	0
176	304	Load_{Other} exceeds B_{other}	0	0	575	150	500
304	390	*Other* traffic decreasing	0	0 to 500	575 to 972	150 to 253	0
390	515	MM and TCP	0	500	972	253	0
515	607	TCP only	150	500	1225	0	0

Table 4.8 Times for Changes in Generated Load

From time 0 to 42 and 515 to 607 TCP is the only traffic type present. As a result TCP is able to borrow all of the excess capacity (650KB/s) to combine with TCP's initial allocation of 575 leaving it with a reallocation, B'_{TCP}, of 1225 KB/s. Conversely, multimedia and *other* are reallocated to match their load of zero. When both multimedia and TCP are present (42,515) they share E_{other} in proportion to their initial bandwidth allocations. Multimedia gets 21% of the excess and TCP gets 79%. When the traffic class *other* is idle (time 42-90 and 390-515) multimedia combines 102 KB/s to its original allocation of 150 KB/s for B'_{MM} of 253 KB/s. TCP's reallocation is 972 KB/s. At time 90 *other* begins to generate traffic, linearly increasing the generated load from 0 KB/s at time 90 to 500 KB/s at time 176. We focus on this point because this is the point at which Load_{other} exceeds B_{other} and the total excess capacity is 0 KB/s. At that point the reallocated values for all classes exactly match their initial allocations. Since Load_{other} increases linearly from time 90 to time 176 the reallocations, B'_{TCP} and B'_{MM}, will have a corresponding linear decrease over the same period. The traffic generator for class *other* continues to increase its load up to 800 KB/s at time 240. The behavior after time 240 is symmetric with the period before 240 so we do not address those time periods here.

Using the values of B' calculated for these key times, the expected throughput for each traffic class during the experiment can be plotted as shown in Figure 4.19. The expected throughput should match the expected reallocations, B'_{TCP}, B'_{MM}, and B'_{other}.

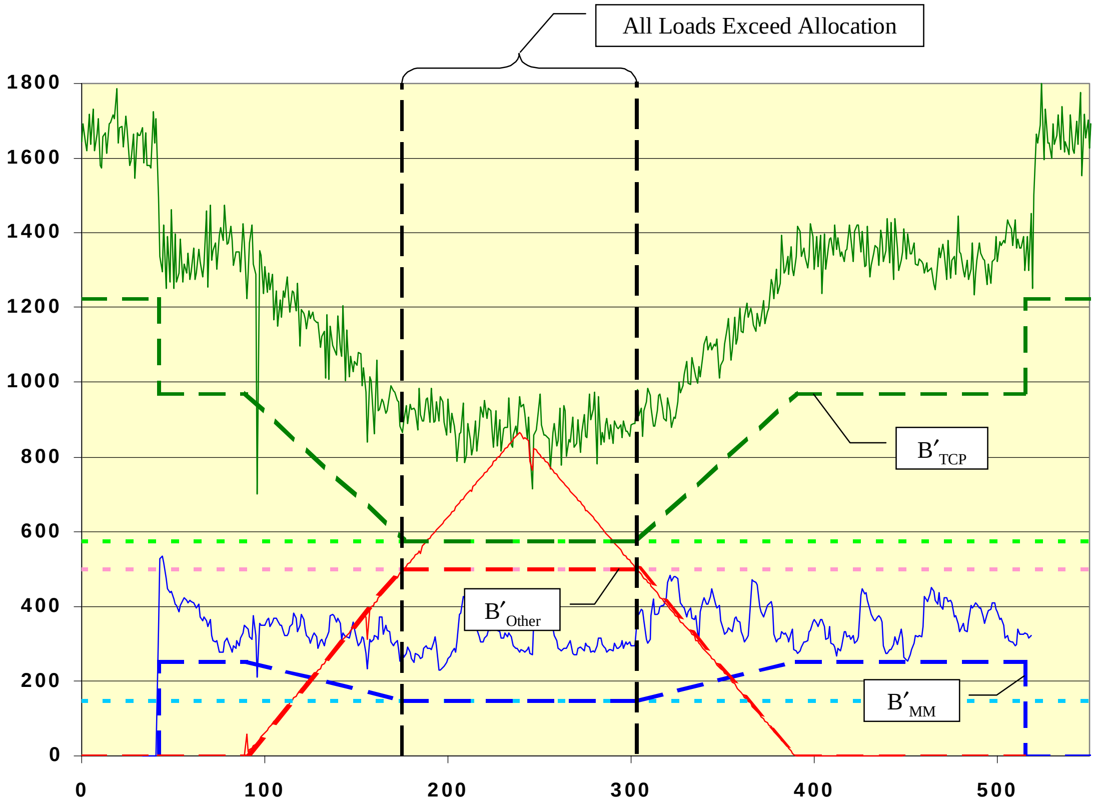


Figure 4.19 Expected Throughput (KB/s) over Time (seconds)

Figure 4.20 shows the actual observed throughput plotted as individual data points compared to the expected throughput (B' plotted as long dashed lines) for each traffic class. Each data point represents the average throughput for a one second interval. Clearly, B' is an accurate predictor of the throughput each class receives as the traffic load changes.

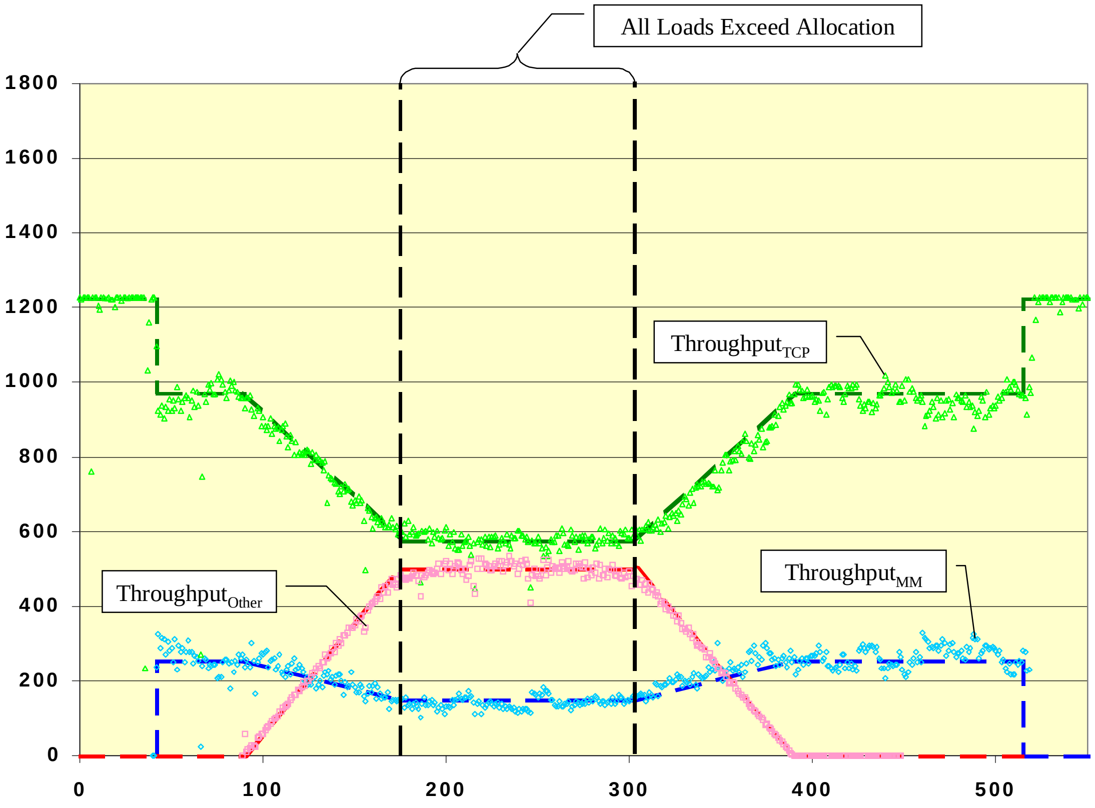


Figure 4.20 Throughput and Expected Throughput (KB/s) over Time (seconds).

Moreover, the ratios of bandwidth allocation are maintained between classes. Consider TCP and multimedia. Figure 4.21 shows the throughput for those two classes of traffic alone. Since multimedia was allocated 150 KB/s and TCP was allocated 575 KB/s initially, the ratio of throughput for these two classes is expected to remain the same, at 150:575, throughout. If the ratio is maintained, then multimedia's throughput should be 21% of the combined TCP and multimedia throughput as long as both classes are generating sufficient load to consume their allocations.

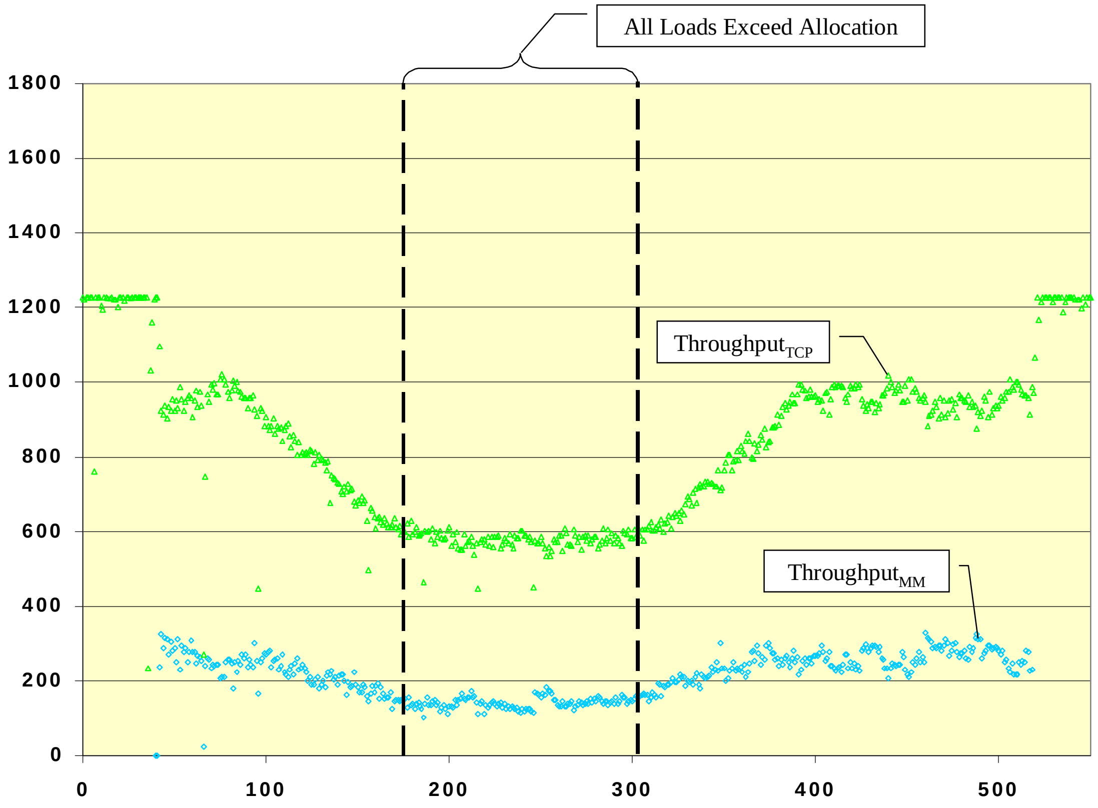


Figure 4.21 Throughput (KB/s) for TCP and Multimedia over Time

Figure 4.22 shows the multimedia throughput as a percentage of the combined multimedia and TCP throughput. The figure clearly shows that multimedia does average 21% for the entire time it is active. Note that this figure covers the period with no *other* traffic, *other* changing linearly, and when all classes have load greater than their allocation. The ratio is maintained throughout.

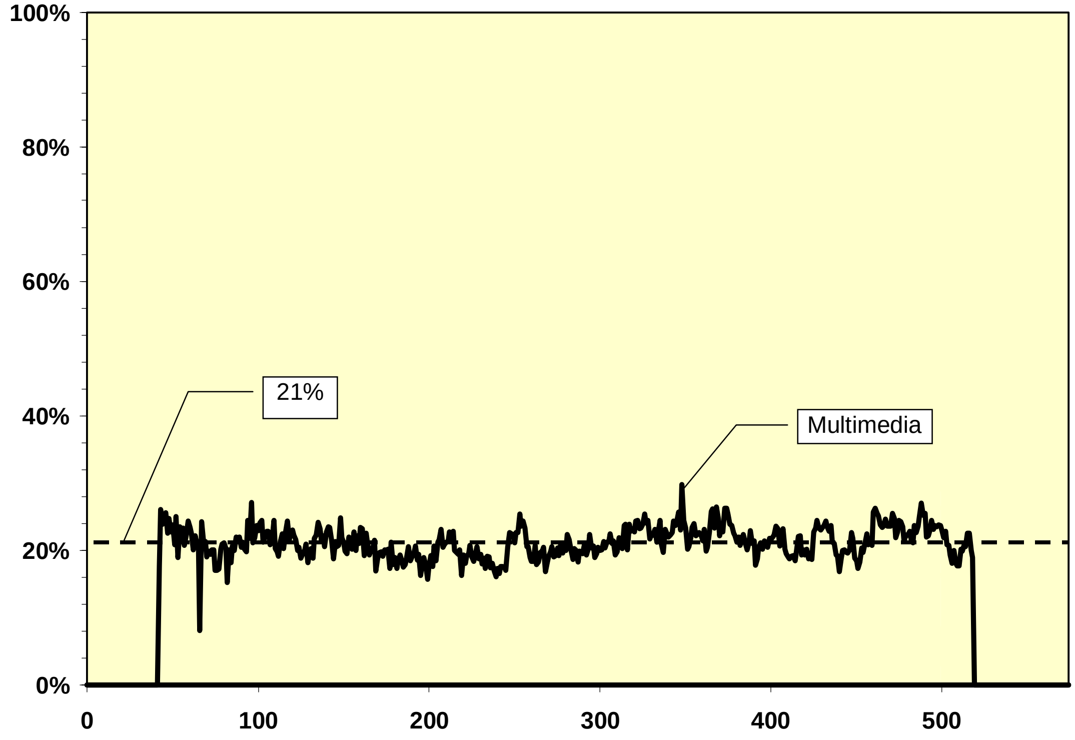


Figure 4.22 Multimedia Throughput as a Percentage of Aggregate Multimedia and TCP Throughput over Time (seconds)

5.2.3. Equations for Predicting Latency

When one or more classes are overprovisioned, the latency caused by queueing delays will decrease. That is because the average queue occupancy for the overprovisioned classes will decrease below Th_{Max} for that class while other classes will obey the limits on their occupancy, resulting in a shorter total queue, and, as a result, less latency resulting from queueing delay. Recall that even though classes can exceed their initial bandwidth allocation by effectively *borrowing* link bandwidth from the classes that are overprovisioned they do not actually borrow queue capacity. Rather, since other classes are not maintaining a queue occupancy equal to their initial threshold settings, those classes that do maintain a queue occupancy equal to their threshold settings represent a larger fraction of the total queue occupancy. As a result, those classes receive a larger fraction of the capacity of the outbound link. Since the average queue occupancy for some classes is lower than the allocated maximum, the aggregate queue occupancy will also be below the allocated maximum, resulting in lower average latency.

To better understand this decrease in latency, recall that thresholds for each class place an upper limit on the queue occupancy, and as a result, the queueing delays induced by that class. This is an upper bound on the average occupancy is shown in Equation 4.19. On average, the queue occupancy for a class is constrained to less than or equal to the threshold for that class.

$$Th_i \geq q\_avg_i \tag{4.19}$$

From that statement it is straightforward to arrive at the relation shown in Equation 4.20. The sum of the thresholds is always greater than or equal to the average queue occupancy.

$$\sum_{i=0}^{N} Th_i \geq \sum_{i=0}^{N} q\_avg_i \tag{4.20}$$

Now, consider that the classes that are overprovisioned will not reach their threshold so their queue occupancy is strictly less than their threshold. This is expressed in Equation 4.21. Note that the load is compared to the reallocated bandwidth, B', not the initial allocation, B. If other classes are overprovisioned a given class may be able to exceed its original bandwidth allocation without reaching its threshold.

$$B_i' > Load_i \Rightarrow Th_i > q\_avg_i \tag{4.21}$$

If this difference between the actual average queue occupancy and the threshold value is expressed with a delta as shown in Equation 4.22 the decrease in latency due to class i can be expressed as $L_{\Delta,\mathrm{i}}$ as shown in Equation 4.23.

$$\Delta_i = Th_i - q\_avg_i \tag{4.22}$$

$$L_{\Delta,i} = \frac{\Delta_i P_i}{C_{outbound}} \tag{4.23}$$

The expected latency, L', given initial threshold values and the actual average queue occupancy for each class can be expressed as shown in Equation 4.24.

$$L' = L - \frac{\sum_{i=0}^{N} \Delta_i P_i}{C_{outbound}} \tag{4.24}$$

While this is a very straight forward way to think about the average queue occupancy and its effect on latency, it is not a useful way to calculate the expected latency because it is not easy to acquire information about average queue occupancy for each class. However the equations from Section 4.1 can be extended to derive an expression of expected latency based on expected bandwidth.

First, recall the equation to express a class's threshold in terms of the desired bandwidth and desired latency. This is shown in Equation 4.25.

$$Th_i = \frac{B_i L}{P_i} \tag{4.25}$$

We can reorganize that equation to express the desired latency in terms of the bandwidth, threshold and packet size as shown in Equation 4.26. This equation is basically an expression of how long it takes a given number of bytes (*Th_iP_i*) to drain with a given drain-rate (*$1/B_i$*).

$$L = \frac{Th_i P_i}{B_i} = \sum_{i=1}^{N} \frac{Th_i P_i}{C} \tag{4.26}$$

In Section 4.1 equations were derived to express the revised bandwidth allocations, B', when some classes are overprovisioned. Since the classes that are underprovisioned, those where $Load_i > B_i'$, still maintain an average queue occupancy of Th_i, a modified form of Equation 4.26, shown in Equation 4.27, can be used to express the expected latency, L'. For each class where the load is greater than the reallocated bandwidth the expected latency can be expressed as the amount of time it takes the class to drain its average queue occupancy at its expected average bandwidth *reallocation*. Also, the expected latency, L', is equal across all classes so once derived for a given class that is underprovisioned, it is derived for all classes, including those that are overprovisioned.

$$\exists i \middle| Load_i > B_i' \ , L_i' = \frac{Th_i P_i}{B_i'}$$

$$L' = L_i' \ , \forall i \tag{4.27}$$

Finally, note that if no class is underprovisioned after the bandwidth reallocations then the link is underutilized and no queue will be expected to form, resulting in an average queueing latency of approximately zero.

5.2.4. Predictability of Latency

Once again, the illustrative experiment is used to demonstrate how accurately the performance of the CBT algorithm can be predicted. Table 4.9 reviews the key events in the experiment and the L' values for each. L' was calculated using equation 4.27 and the value of Th_{MM}, P_{MM}, and B'_{MM} shown.

Th_{MM}=19.04	P_{MM}=807	L=100 ms

Start	Stop	Event	B'_{MM}	L'
42	90	MM and TCP	253	61
90	176	*Other* traffic increasing	253 to 150	61 to 100
176	304	$\mathrm{Load}_{\mathrm{Other}}$ exceeds $\mathrm{B}_{\mathrm{other}}$	150	100
304	390	*Other* traffic decreasing	150 to 253	100 to 61
390	515	MM and TCP	253	61

Table 4.9 Calculating L' Values for the Illustrative Experiment

We measure latency by instrumenting the multimedia traffic generators on the end-systems to record the time each packet is sent and received. (See Appendix A for a discussion of how to insure the clocks are synchronized on sender and receiver.) Since the multimedia traffic is not active during the period (0,42) or (515,607) latency can't be measured during that period so latency is reported and L' calculated only during the period (42,515). The loads and, consequently, the expected latency are constant during three time periods: (42, 90), (176, 304), and (304, 390). Given that the $\mathrm{Load}_{\mathrm{other}}$ is changing linearly during the periods (90,176) and (304,390), B'_{MM} and, consequently, L' must be changing linearly as well, so a line can be plotted between the L' values at either end of the periods with constant latency.

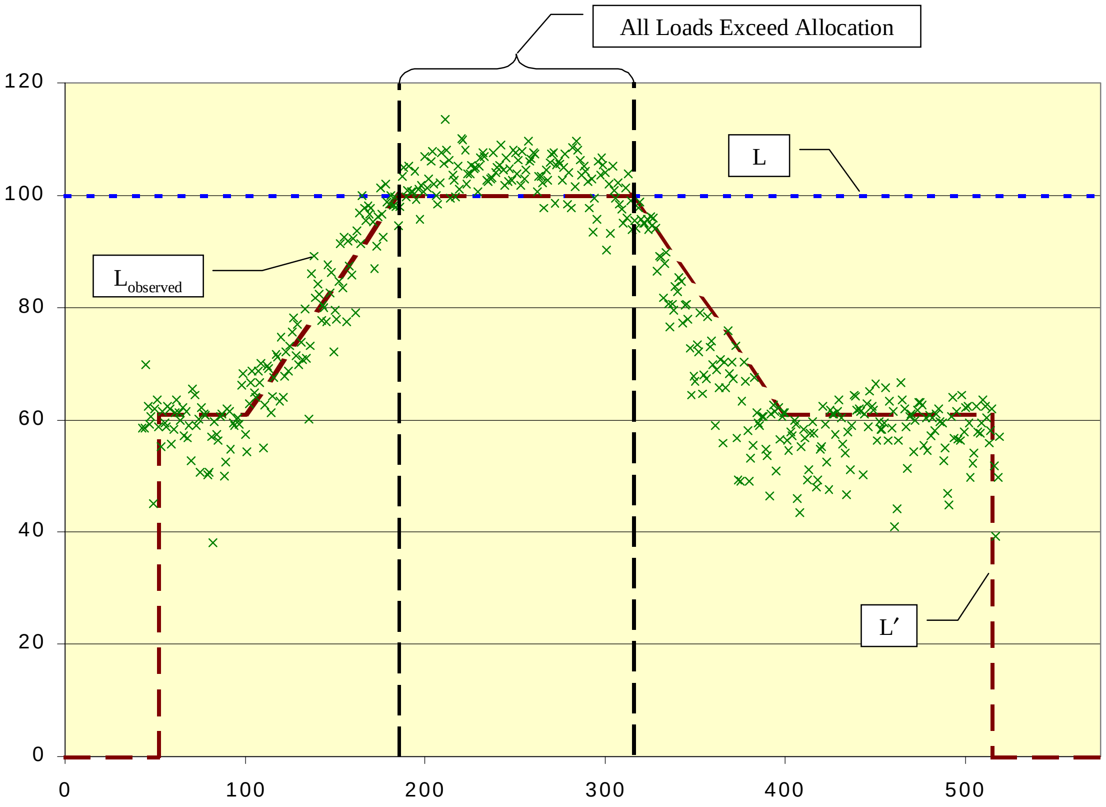


Figure 4.24 Observed Latency Compared to Predicted Latency (ms) over Time (seconds)

We compare *L′* to the observed latency ($L_{observed}$) in Figure 4.24. Each data point represents the average observed latency value for a one-second interval. The long dashed line represents *L′*. The short dashed line represents the initial latency setting, *L*. The predicted latency matches the observed latency quite well. Once again, end-to-end latency is being compared to predicted queueing latency, the observed latency is slightly higher than the predicted values. Forwarding at the other router, propagation, and end-system delays account for this discrepancy. Further, recall that CBT does not promise a strict limit on the latency, but rather a limit on average latency over coarse time scales.

5.3. Summary

CBT effectively divides the link's capacity between the traffic classes in proportion to their initial bandwidth allocations. Moreover, we present equations that can predict the throughput and latency for each class under varying traffic conditions. Empirical evaluation confirms the accuracy of these equations.

6. The Generality of CBT

Although this work focuses on a specific approach to using the general mechanism of CBT to address the heterogeneity presented by the presence of TCP, multimedia, and *other* traffic in today's Internet, CBT is a very general mechanism. Throughout most of this work, CBT is configured to only have three classes. Two of those classes have deterministic drops acting primarily as a constraint mechanism while the TCP class continues to be subject to probabilistic drops in a RED like fashion for feedback. However, the CBT mechanism is general and the number of classes it could accommodate is a factor of memory and the classification mechanism. The algorithm itself is not limited. Further, the drop policy applied to each class is flexible, allowing for anything from an entirely probabilistic policy based on the average queue length to a deterministic drop decision based on instantaneous queue length (essentially FIFO). These changes in policy can be affected simply by changing the input parameters. Although the exploration of this flexibility is left for future work some observations are presented below.

6.1.1. Number of classes

As stated above, the number of classes supported by CBT has no algorithmic limit. One could deploy CBT with only one class, all, and apply a wide choice of drop policies, selected by changing parameter settings as discussed below. One could also allocate many classes and use CBT to provide bandwidth allocation at a finer granularity.

6.1.2. Sensitivity

By setting the weight factor for a given class to a very small value the average can be extremely desensitized to any short-term effects. In contrast, the weight can also be set to one and then drop decisions can be made in response to instantaneous queue size. At that point the maximum threshold also becomes a hard upper limit on the number of packets enqueued for a given class.

6.1.3. Modes

Any of the three RED modes can be eliminated for a class by setting the threshold values. Setting Th_{Min} to zero eliminates the no-drop mode, always giving some feedback. Setting Th_{Min} equal to Th_{Max} eliminates the probabilistic drop mode, using the thresholds only as a source of constraint. And setting Th_{Max} equal to the maximum queue size eliminates forced drops due to the average growing too large. However, forced drops will still occur on queue overflow.

6.1.4. Examples

Using these settings one could implement any of RED, FIFO, or a version of Early Random Detect. To implement any of these, first the number of classes would be set to one. FIFO would simply require a weight factor of one and setting the thresholds equal to the queue length. Early Random Detect simply requires a weight of one, the maximum threshold equal to the queue length, and the minimum equal to the desired threshold. However, in the current implementation, this would not precisely be Early Random Detect as the drop probability would change as a factor of queue size and the number of packets to arrive since the previous drop. RED, of course, simply requires setting the number of classes to one and setting the thresholds, weight and max_p to desired values.

7. Summary

We propose a new active queue management mechanism, class-based thresholds (CBT), for use in Internet routers to both isolate classes of traffic, allocate bandwidth, and manage latency. We have explained the design of the algorithm and present equations for configuring CBT based on desired latency and bandwidth allocations. We have also demonstrated the algorithm’s response to changes in traffic loads is predictable. The accuracy of the equations for predicting performance was confirmed empirically. In the next Chapter, we compare the performance of CBT to other active queue management policies to empirically evaluate how well it meets its design goals.

V. EMPIRICAL EVALUATION

In this chapter, CBT's effectiveness is empirically demonstrated by comparing its performance to other algorithms. Specifically, CBT is compared to the active queue management algorithms: FIFO, RED, and FRED. Additionally, it is also compared to a packet scheduling mechanism, class-based queues (CBQ). Each algorithm was evaluated as it managed the queue servicing a bottleneck link in a laboratory network. Each algorithm was evaluated under a set of varying traffic scenarios featuring different combinations of TCP, multimedia, and *other* traffic. Multimedia performance and network throughput were measured using a combination of end-system instrumentation and network monitoring. The resulting metrics were compared for each algorithm and each traffic scenario.

This chapter is organized as follows. First, the methodology used in conducting these experiments is explained. This includes a description of the network topology, the traffic mixes used, and the measurement periods of interest. Next, the specific metrics gathered are described and their importance explained. The configuration of each algorithm in these experiments is defined and the steps taken to determine the optimal configurations for each algorithm are discussed. Following that, the performance of each algorithm is considered. Each algorithm's performance is evaluated during a set of well-defined measurement periods. Finally, conclusions are drawn about the strengths and weaknesses of each algorithm.

1. Methodology

To compare these queue management algorithms, each was empirically evaluated with a variety of traffic mixes. All experiments were run in a private laboratory network using end-systems for traffic generation and a FreeBSD router running each algorithm. Simulation and production networks were also considered as possible approaches to evaluating these algorithms; however, the use of production networks was quickly eliminated from consideration at this stage of evaluation because of concerns about reproducibility, control of the traffic conditions, and impact on production traffic. Evaluating CBT in a production network is a logical future step but is beyond the scope of this work. In contrast, simulation offered the control and reproducibility that were not available in a production network. Moreover, simulation requires minimal infrastructure, is easily configurable, and has no impact on production traffic. However, simulation may not include all potential factors that effect performance. Because the infrastructure for conducting these experiments in a private network was already available, these experiments were conducted in that framework. The details of the network configuration, as well as more discussion of the choice of experimental frameworks, can be found in Appendix A. A high level description of network configuration and methodology is provided here.

1.1. Network Configuration

The experimental network consists of two 100 Mbps switched LANs connected by a full-duplex, 10 Mbps link which simulates an Internetwork as shown in Figure 5.1. Each LAN is populated with seven host systems used to generate and receive traffic to and from the end-systems on the other LAN. The router is a FreeBSD system with software implementations of each of the algorithms. The implementations are done within the ALTQ framework [Cho98], an extension to the basic FreeBSD IP forwarding mechanism. Artificial delays between each pair of end-systems are used to simulate a wide range of round-trip times between source-receiver pairs and thus ensure synchronization effects are avoided. Each network is monitored to allow collection of packet traces that will be analyzed off-line to assess performance. The end-systems on the left-hand side of Figure 5.1 act as traffic sources, generating traffic in response to requests from the client end-systems on right-hand side. For example, in several experiments we generate simulated web traffic with one end acting as the browsers and the other as servers. While the clients' requests are relatively low bandwidth, the sources on the left-hand side generate sufficient load to overload the 10 Mbps capacity of the simulated Internet, creating overload at the WAN interface on router labeled "bottleneck" in Figure 5.1.

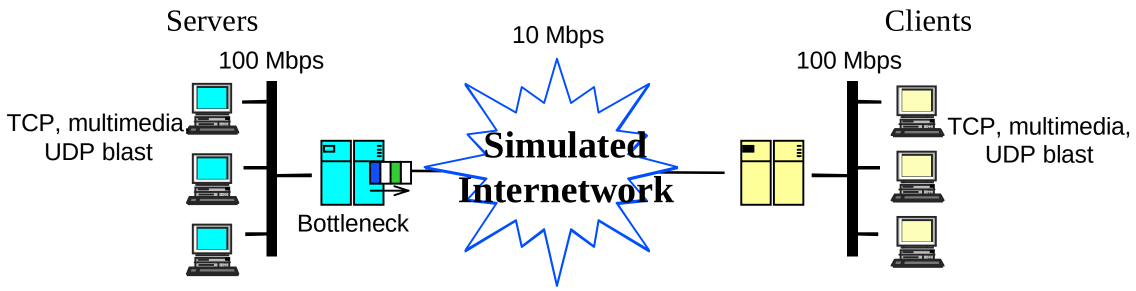


Figure 5.1 Experimental Network Configuration

1.2. Traffic Mixes

The network traffic in these experiments falls into one of three categories: TCP, multimedia, or *other* (an aggressive, unresponsive source referred to as a UDP Blast). In all cases the traffic generators were used to generate hundreds or thousands of flows representing many end-systems. Each experiment includes one type of traffic from each of these three categories. Each category and the types of traffic that category represents is described briefly here. Details regarding each traffic category can be found in Appendix A.

In the experiments, TCP was made the most prevalent traffic category because it represents 95% of the traffic in the Internet. TCP traffic is generated using models of application-level behavior. (The alternative would be to simply replay packet level traces of TCP traffic; however, these packet-level traces would be inadequate because TCP sources adjust the loads they send in response to network performance, hence packet rates could change under different conditions.) TCP traffic is generated using traffic generators that simulate the application level behavior of two of the most common TCP-based applications, world-wide-web traffic, and bulk transfer of data like FTP. In each experiment a subset of the end-systems generate either HTTP or BULK traffic. For HTTP traffic each source simulates multiple HTTP servers responding to requests from the simulated clients systems. The client end-systems modeled, in aggregate, 3,000 HTTP clients (browsers) capable of generating a load of 1.1 MB/s [Christiansen00]. The pattern of requests from the clients and behavior of the servers was based on the HTTP application model developed by Mah [Mah97]. The individual HTTP flows were typically short-lived and transfer few bytes. In contrast, the flows in the BULK traffic model were long-lived and transfer a large number of bytes. For BULK traffic, the simulated FTP clients establish 360 connections distributed evenly among the simulated FTP servers. Each of these connections lasts for the duration of the experiment. In aggregate the BULK sources are capable of exceeding a load of 1.5 MB/s, well in excess of the 1.2 MB/s capacity of the bottleneck link.

As with TCP, two types of multimedia traffic were evaluated in these experiments: MPEG, the common video compression standard, and Proshare, a proprietary video conferencing product from Intel. Each experiment included one of these types of multimedia traffic. The MPEG streams used in these experiments generate 30 frames per second of video using inter-frame encoding with a group of pictures (GOP) of IBBPBB. The streams are playing the movie, "Crocodile Dundee". The size of the different frames is highly variable between 200 byte B-frames that fit in a single packet and 7,000 byte I-frames that span 5 Ethernet-sized packets. In contrast, the Proshare stream generates 15 frames per second of video and 10 frames per second of audio and uses no inter-frame encoding. The variation in frame size for Proshare is much smaller than that of MPEG with each audio frame fitting in one packet and most video frames spanning 2 packets. In both cases, the multimedia streams use UDP as their transport level protocol and are unresponsive at the application-level. Consequently, replaying packet-level or frame-level traces is an effective way to model these traffic types since the packet rate does not change in response to network conditions.

Because the performance of individual multimedia flows will be a factor in comparing the algorithms, it is important to note the characteristics of these flows. The multimedia flows are generated using separate processes for each client-server pair. A single MPEG flow generates a load of 40-50 KB/s. In these experiments four MPEG flows are established to generate an aggregate load of 160-200 KB/s. A single Proshare flow generates a lighter per-flow load of ~27 KB/s. In these experiments six Proshare flows are established to generate an aggregate load of ~160 KB/s. The intention is generate a multimedia load that is a noticeable fraction (13-16%) of the link's capacity.

The third and final category of traffic in these experiments is high-bandwidth, unresponsive traffic. This traffic is referred to as *other* or, in these experiments, as a UDP blast. It is generated as a single, high bandwidth flow using UDP. The UDP blast generates a constant load of 10 Mbps that is capable of consuming all of the capacity of the bottleneck link.

This section sketches out the general traffic patterns and measurements in broad strokes. Details of the traffic mixes and issues involved in generating the traffic can be found in Appendix A. Figure 5.2 shows an example of the traffic mix and loads present on left-hand side of the bottleneck router in Figure 5.1. Each line shows the load generated by each class in isolation. These baseline measurements for each class were gathered independently to demonstrate the capabilities of each class. The timestamps were merely translated to allow for presentation in a single figure. In this example the TCP traffic is BULK and the multimedia traffic is MPEG. The traffic mixes are named based on their TCP and multimedia type. So, this combination is "BULK+MPEG". For all of these experiments, traffic was generated according to the same basic script. First, the TCP traffic generators are started. After allowing some time for the generators to stabilize (before time 0) we begin monitoring the traffic. After approximately 30 seconds, the multimedia traffic generator begins and runs for 180 seconds. At time 90 seconds, the UDP blast traffic starts and runs for 60 seconds. Overlapping these three traffic types in this way gives two key measurement periods to study: the multimedia measurement period and the blast measurement period. The multimedia measurement period covers an interval when only TCP and multimedia traffic are present, presenting performance during a period of moderate overload. The blast measurement period covers an interval when TCP, multimedia, and the UDP blast are present, presenting performance in the presence of a high-bandwidth unresponsive flow.

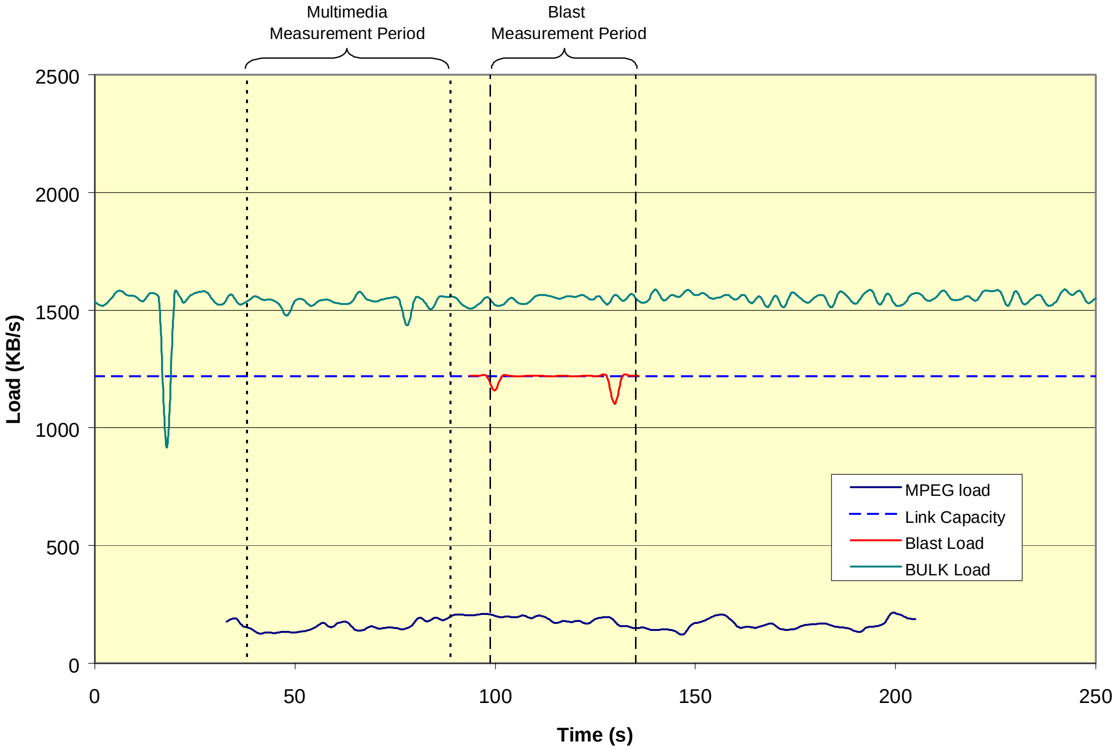


Figure 5.2 Sample Traffic Mix and Measurement Periods.

1.3. Metrics

To compare these algorithms, the network performance is compared during both measurement periods using a variety of metrics across the different traffic mixes. The details of the metrics and our methodology for collecting these metrics are detailed in Appendix A and summarized here.

The throughput on the bottleneck link is considered for all traffic categories. Throughput is measured by collecting a packet trace on the bottleneck link and reporting the average aggregate throughput for each class of traffic at one-second intervals. In the case of TCP, a slightly refined measure of throughput is used. This metric, goodput, does not consider packets that are redundant (i.e. retransmissions). An effective queue management algorithm should allow TCP to have goodput that is a large fraction of the link's capacity, allow multimedia to receive sufficient throughput to match its load, and constrain *other* traffic to a consistent, small share of the link's capacity. Throughput gives a quantitative measure of performance. For data transfers using TCP, goodput is an effective indication of the quality of network service. However, for interactive multimedia it is difficult to evaluate the quality at the end-systems using that metric alone. Latency, loss, and frame-rate are more effective indications of the quality of multimedia interactions so the algorithms are also evaluated based on those metrics. The multimedia data reported is data for one representative flow. (All flows were found to have essentially the same results.) Latency and frame-rate are measured using instrumented traffic generators on the end-systems. A time-stamp in each packet is used to calculate the end-to-end latency of each packet. (The problem of synchronized clocks is avoided by using static routing to allow a single host to act as both the sender and receiver.) For MPEG traffic, the generators report the latency averaged over one second intervals. The Proshare generators report latency packet by packet. This variation in measurements is arbitrary, and a consequence of the traffic generators available. However, although both provide the same general data, packet by packet latency data shows the variability (or lack thereof) for latency while per-frame measurements don't highlight this variability. To maximize interactivity and minimize any end-system buffering needed to smooth jitter, latency should be as small as possible. Particularly, note that the values reported here are latency for packets crossing a 3-hop network with one congested link. While all of the values reported are less than or equal to the 250ms threshold of perceptible latency it is important to minimize latency as most connections cross ~16 hops and queueing delays at routers are cumulative [Paxson96].

For the Proshare traffic generators, loss-rate is a key indicator of the fidelity of the interaction. A single Proshare traffic generator on average generates 36 packets/second with 10 of those packets containing independent audio frames. The video stream generates 15 frames/second of variable size, independent video frames fragmented, on average, into 26 packets. However, because the Proshare traffic is generated based on a packet trace with no media specific information, the generator is not able to report information on frame-rate for Proshare. Instead, the quality of the media stream must be inferred from a secondary metric, loss-rate. The loss-rate is reported as the number of packets that fail to reach the receiver for a representative Proshare stream.

In contrast, the MPEG generator reports frame-rate data. This is a superior measure of multimedia performance. Loss-rate adds little information; however, it is interesting to observe that the relationship between loss-rate and frame-rate as it is not always linear. For example, an AQM algorithm may be biased against large frames that span many packets because the drop mechanism is overly sensitive to the resulting burst of packets associated with that frame. Because I-frames are the largest frames in MPEG and entire GOPs depend on I-frames for decoding, a low loss-rate could lead to a terrible playable frame-rate if one packet per I-frame were dropped. This emphasizes the need to examine both actual and playable frame-rate for media streams that have interframe encodings.

The MPEG traffic generators generate frames at a rate of 30 frames per second and report the actual frame-rate and the playable frame-rate at the receiver. The *actual frame-rate* is based on the number of frames that arrive intact at the receiver, regardless of decoding concerns. The *playable frame-rate* is based on the number of packets that could be successfully decoded given the inter-frame encoding semantics. Playable frame-rate is the more important metric as it indicates the actual quality of the media stream. However, both are reported to point out the relationship between actual and playable frame-rate for different algorithms. Using these metrics, the performance is compared for each queue management algorithm with each traffic mix.

1.4. Configuration of Queue Management Algorithms

To be sure the queue management algorithms are compared fairly, each algorithm was evaluated to determine its optimal parameter settings in the experimental environment. The extensive process used to determine these parameters is detailed in Appendix B. The process and results are summarized below. The optimal parameter settings refer to those settings that offer the best performance based on the goals of the algorithm. For example, for RED the primary criterion is TCP performance as measured both by TCP goodput and the number of retransmissions. A secondary metric is latency as it is an effective indication of the average queue size in these experiments and maintaining a small average queue size is one of the other goals of RED. This is balanced by the third goal of RED, good link utilization. Maintaining good link utilization means buffering some packets so that there are packets to forward during brief intervals when no packets are arriving. In contrast, FRED was evaluated primarily on how fairly it constrains individual flows to 1-*n*th of the link's capacity. In contrast, providing low latency and low loss for multimedia was not a goal of either algorithm so for the purpose of determining parameter settings, neither of those algorithms was evaluated for multimedia performance. However, supporting multimedia effectively is a goal of CBT and CBQ so multimedia performance was a concern in the selection of the optimal parameter settings for those algorithms.

Table 5.1 shows the optimal parameter settings that were selected for each algorithm. In the case of the FIFO, drop-tail algorithm there was only one parameter to consider, the queue length, *maxq*. Different values of maxq were evaluated. Most values offered the same TCP performance but queue lengths of less than 60 resulted in a small decrease in TCP goodput. Since queue-induced latency increased as *maxq* increased, a queue length of 60 packets was selected because it was the smallest value that offered good TCP goodput. That setting is also the default queue length in the FreeBSD implementation.

Algorithm	**maxq**	**w**	**max_p**	**Th_{Min}**	**Th_{Max}**	**minq**
FIFO	60	n/a	n/a	n/a	n/a	n/a
RED	240	1/256	1/10	5	40	n/a
FRED	240	1/256	1/10	5	60	2

Table 5.1 Optimal Parameter Settings for FIFO, RED, and FRED Across all Traffic Mixes

In the case of FIFO, the queue length was chosen independent of the traffic mix. This decision was based on the fact that the buffer in the FIFO algorithm is merely intended to offer some reasonable capacity to accommodate burstiness. In the case of RED and FRED, the optimal parameter settings were selected by evaluating performance with an HTTP-Proshare traffic mix and then confirmed by examining a smaller set of parameter combinations for BULK-Proshare. Different multimedia types were not evaluated because multimedia performance is not a concern in the design of either of those algorithms.

RED and FRED offered a wider range of parameters to consider. To limit the complexity of choosing optimal parameters, the designers' recommended value was used for the weighting factor *w*, the maximum drop probability max_p, and, in the case of FRED, the minimum per-flow queue occupancy *minq*. The maximum queue size, *maxq*, was also fixed at 240 packets as the actual queue size has little effect compared to the threshold settings. The threshold values were the focus of the exploration of the parameter space for RED and FRED. For RED the value of Th_{Max} was the dominant factor for the network performance. Th_{Max} values that were too small resulted in poor efficiency as many of the arriving TCP packets were dropped. Setting Th_{Max} to values of 40 packets or more offered the same efficiency, with 95% of the TCP data reaching the router being forwarded (i.e. a loss rate of 5%). However, Th_{Max} also limited the maximum average queue occupancy. Since RED's design goal was to minimize average queue occupancy this argued for a small value of Th_{Max}. This led to selecting the smallest value of Th_{Max} that offered good TCP efficiency. That value was 40 packets. The value of Th_{Min} had no effect on the metrics we examined so we used the recommended setting for Th_{Min} of 5.

Performance analysis of FRED was more complex. The ratio between Th_{Max} and Th_{Min} was important at it placed a limit on the fraction of the queue that a given flow could occupy. Since one goal is to constrain the UDP blast this argued for a large ratio between Th_{Max} and Th_{Min} to constrain each flow to a small share of the queue during periods of overload. However, queue-induced latency increases as Th_{Max} increases so that argues for minimizing Th_{Max}. Moreover, extremely small values of Th_{Min} over constrain individual flows, including TCP. As a result, the optimal parameters selected were a Th_{Max} of 60 and a Th_{Min} of 5. These values were chosen initially based on TCP performance with HTTP traffic. When the TCP type was changed to BULK the combination of a large number of active flows and BULK's higher load combined to force the queue into a state of consistent overload. Because in FRED each flow is allowed to have two packets enqueued regardless of average queue size, FRED's primary drop mechanisms are circumvented and only the queue size limits queue occupancy. Consequently, changing the threshold settings had no effect on performance. Since the threshold values had no effect, the optimal settings derived from the HTTP traffic are also used for BULK.

The selection of optimal parameters for CBT was a slightly different process since the algorithm can be configured to offer desired bandwidth allocations and a limit on queue-induced latency. Rather than probing a large parameter space, this process simply sought to confirm the calculated optimal parameter settings. First, as with the previous algorithms, fixed values were used for max_p, *maxq*, and the weights associated with each class of traffic. When determining the optimal parameter settings for these experiments the desired latency setting was 100ms. For bandwidth allocations, the class *other* was constrained to 150 KB/s. This was an arbitrary value selected to be a small fraction of the link's capacity. Multimedia was allocated bandwidth based on the average expected load for the multimedia traffic type and TCP was allocated all of the remaining capacity of the bottleneck link. Using the equations from Chapter IV to compute threshold settings based on desired bandwidth allocations and latency values, one should be able to compute the optimal parameter settings. However, to confirm these settings, a range of settings that offered multimedia slightly more and slightly less than the expected multimedia load were also considered. TCP's bandwidth allocations were adjusted correspondingly to match the remaining link capacity. These parameter settings were evaluated based on how well they isolated TCP from non-TCP traffic (TCP goodput), how well *other* was constrained (*other* throughput), and how well multimedia performed (frame-rate, loss, and latency). Upon analysis, the actual optimal allocations were found to be slightly higher than originally thought. Multimedia's loss rate was too high. This was because the load generated by the multimedia traffic types did vary slightly, with brief periods of slightly higher load and the initial allocations, based on the long-term average load for the class, were inadequate to accommodate that variation. To ensure good performance, bandwidth allocations should be based on maximum short-term average generated loads, not long-term averages. The optimal parameter settings for CBT are shown in Table 5.2. Note that threshold settings are non-integer. The average itself is non-integer and allocating fractional shares of the queue allows for more precise control of the bandwidth allocation.

Traffic Mix	**Weight**			**Th$_{Max}$**			**KB/s**		
	other	**Mm**	**TCP**	***other***	**Mm**	**TCP**	**B$_{other}$**	**B$_{mm}$**	**B$_{TCP}$**
HTTP-MPEG	1/4	1/16	1/256	14.28	22.45	85.58	150	195	880
BULK-MPEG	1/4	1/16	1/256	14.29	26.06	59.11	150	205	870
HTTP-Proshare	1/4	1/16	1/256	14.28	22.19	88.35	150	160	915
BULK-Proshare	1/4	1/16	1/256	14.29	25.69	61.74	150	190	885
maxp = 1/10, maxq = 240									

Table 5.2 Optimal Parameter Settings for CBT with 100 ms of Latency

The optimal CBT parameter settings were based on a router configuration that has a queue-induced latency of 100 ms. However, one goal of these resource management strategies is to minimize latency. The bandwidth allocations established previously were used to recompute new thresholds for those allocations with a latency setting of 30 ms. It is possible that changing the latency setting makes these bandwidth allocations sub-optimal for CBT. (For example, the new threshold settings could be small enough to limit the capacity to accommodate bursts.) However, using sub-optimal parameter settings for CBT would only make the other algorithms look better in comparison. Table 5.3 shows the recalculated threshold settings with a latency value of 30 ms.

Traffic Mix	**Weight**			**Th$_{Max}$**			**KB/s**		
	Mm	***other***	**TCP**	***other***	**Mm**	**TCP**	**B$_{other}$**	**B$_{mm}$**	**B$_{TCP}$**
HTTP-MPEG	1/16	1/4	1/256	4.28	6.74	25.67	150	195	880
BULK-MPEG	1/16	1/4	1/256	4.29	7.81	17.73	150	205	870
HTTP-Proshare	1/16	1/4	1/256	4.29	6.66	26.50	150	160	915
BULK-Proshare	1/16	1/4	1/256	4.29	7.71	18.52	150	190	885
maxp = 1/10, maxq = 240									

Table 5.3 Optimal Parameter Settings for CBT with 30 ms of Latency

Finally, AQM techniques must be evaluated in comparison to a packet scheduling mechanism, CBQ. The approach for selecting the optimal parameter settings for CBQ is similar to that used for CBT. CBQ uses parameters that specify the percentage of the outbound link to allocate to each traffic class. These percentages can map directly to bandwidth allocations so the experiments to determine the optimal parameters were simply confirming or making small adjustments to the expected optimal values. As with CBT, CBQ was evaluated by considering throughput for TCP and *other* as well as multimedia performance. As with CBT, the actual optimal bandwidth allocations for multimedia were slightly higher than those calculated because of short term variations in the multimedia load. The optimal percentages appear in Table 5.4. All cases seek to constrain B_{other} to approximately 150 KB/s as was done in CBT. However, the implementation of CBQ only allows integer percentages as allocations so the value of 12% was used to allocate 147 KB/s to *other* traffic and allocate sufficient bandwidth to multimedia to minimize loss and maximize frame-rate.

Traffic Mix	**MM %**	***Other* %**	**TCP %**	**KB/s**		
				B_{mm}	**B_{other}**	**B_{TCP}**
HTTP-MPEG	16%	12%	72%	196	147	882
BULK-MPEG	16%	12%	72%	196	147	882
HTTP-Proshare	13%	12%	75%	159	147	919
BULK-Proshare	14%	12%	74%	172	147	907

Table 5.4 Optimal Parameter Settings for CBQ

Note that optimal parameter settings were calculated for each traffic mix for both CBT and CBQ. Even though the main parameter was the multimedia allocation (B_{mm}), different parameters were selected depending on the TCP traffic type used. This is because HTTP and BULK differ in their aggressiveness and their load. BULK generates a very high-bandwidth load. As a result, there is seldom any excess capacity that multimedia may be able to borrow. As a result capacity must be strictly allocated to meet multimedia's needs. In contrast, HTTP is less aggressive and slightly lower bandwidth, usually operating with a load slightly less than the allocated bandwidth. As a result, the multimedia traffic is able to borrow excess capacity when HTTP is the TCP traffic type. Thus, the bandwidth allocations for multimedia in those scenarios can be smaller.

2. Comparing Algorithms

CBT's performance is compared empirically to that of FIFO, RED, FRED, and CBQ. Each algorithm is first evaluated during the period with a high bandwidth unresponsive flow, the blast measurement period, as that is the scenario that this work seeks to address. The algorithms are next evaluated during the multimedia measurement period, a period of moderate congestion.

Each algorithm was tested with its optimal parameters over five independent runs. The results of each run were examined to be sure the behavior was consistent between runs. Then one representative run for each algorithm was selected for presentation here. The algorithms are compared by examining the performance of individual metrics over time. Although each of the algorithms was tested independently, for ease of comparison they are presented in the same graphs below with time values translated to synchronize the beginning of the measurement periods. For each metric the performance with an HTTP+Proshare traffic mix is closely examined. Then, unless the conclusions vary significantly, the results with the other three traffic mixes are only briefly summarized.

2.1. Blast Measurement Period

During the blast measurement period all three traffic categories are active with the UDP blast generating an unresponsive load sufficient to consume all of the link's capacity. This is the scenario that highlights the tension between responsive and unresponsive flows. CBT should offer TCP and multimedia performance that is better than FIFO and RED. CBT and FRED should offer comparable TCP throughput and effectively constrain *other*. However, CBT should offer better performance than FRED for multimedia since the multimedia flows need slightly more than a fair share of the link's capacity in order to maintain fidelity. Finally, CBT's performance should approach the performance of the packet scheduling algorithm, CBQ.

2.1.1. TCP Goodput

The first metric to consider is TCP goodput. The goodput for each algorithm during the blast measurement period with HTTP+Proshare is shown in Figure 5.3. This plot shows the TCP goodput averaged over 1 second intervals versus time for each algorithm.

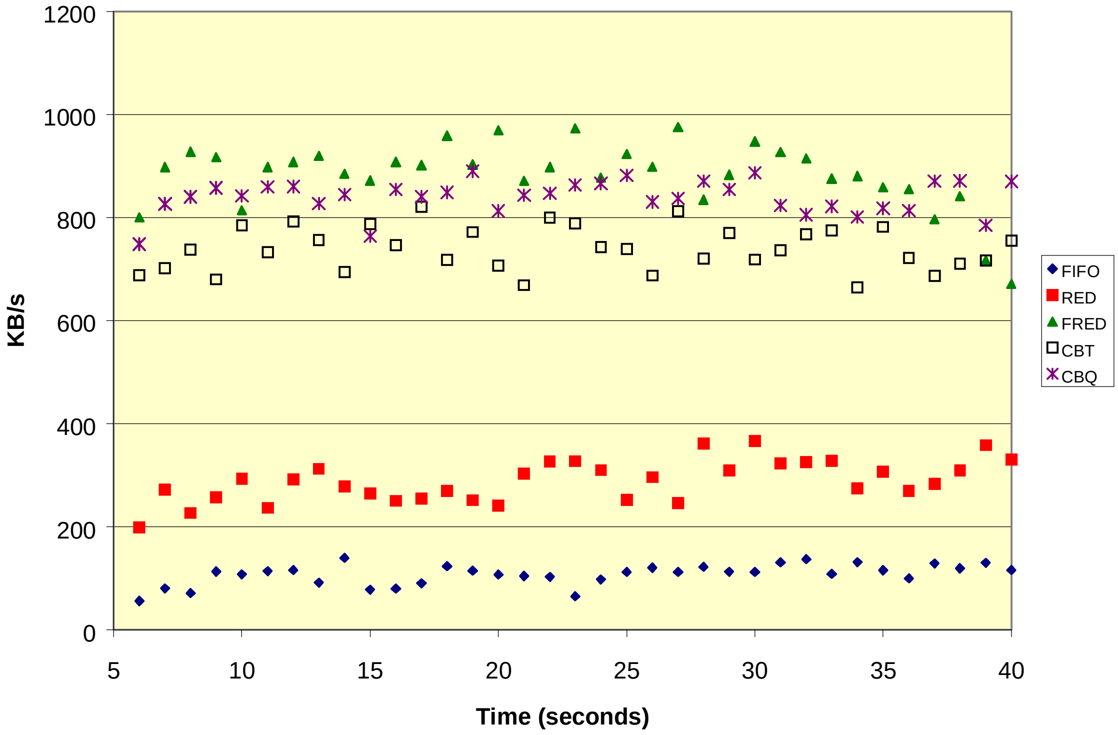


Figure 5.3 TCP Goodput Averaged Over 1 Second Intervals During the Blast Measurement Period for HTTP+Proshare

When both multimedia and the UDP blast are contending for resources along with TCP, FIFO and RED do little to isolate TCP from the effects of unresponsive flows. Goodput with RED is between 200 and 400 KB/s (out of a possible 1,100 KB/s) and FIFO is worse at ~100 KB/s. Under FIFO and RED, packets from all flows are dropped during the overload and the TCP flows respond by reducing their load, allowing the unresponsive traffic to dominate the link. In contrast, with FRED the TCP traffic gets much better goodput (~900 KB/s). Because there are many TCP flows and a small number of multimedia and *other* flows, FRED's per-flow fair sharing constrains the multimedia and *other* flows to a small fraction of the link's capacity leaving TCP with high goodput. Finally, CBT and CBQ's allocations on a class by class basis prove effective. TCP has goodput of ~750 KB/s with CBT and ~850 KB/s with CBQ.

Note that fluctuations in goodput shown in Figure 5.3 are explained by fluctuations in the load offered by the traffic generators. For example, the offered TCP loads with HTTP+Proshare during these periods are shown in Figure 5.4. The varying loads are a result of a combination of factors. TCP's responsiveness to the packet loss accounts for some variation as does the randomness in the sequence of client requests and server responses produced by the traffic generators.

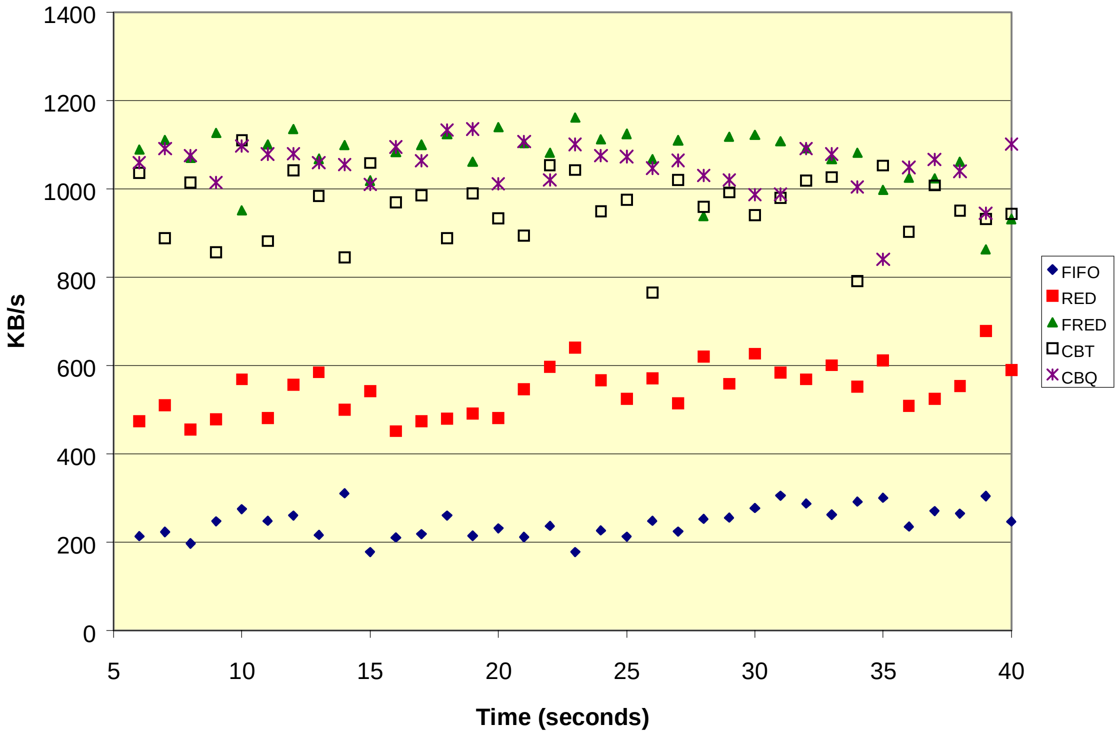
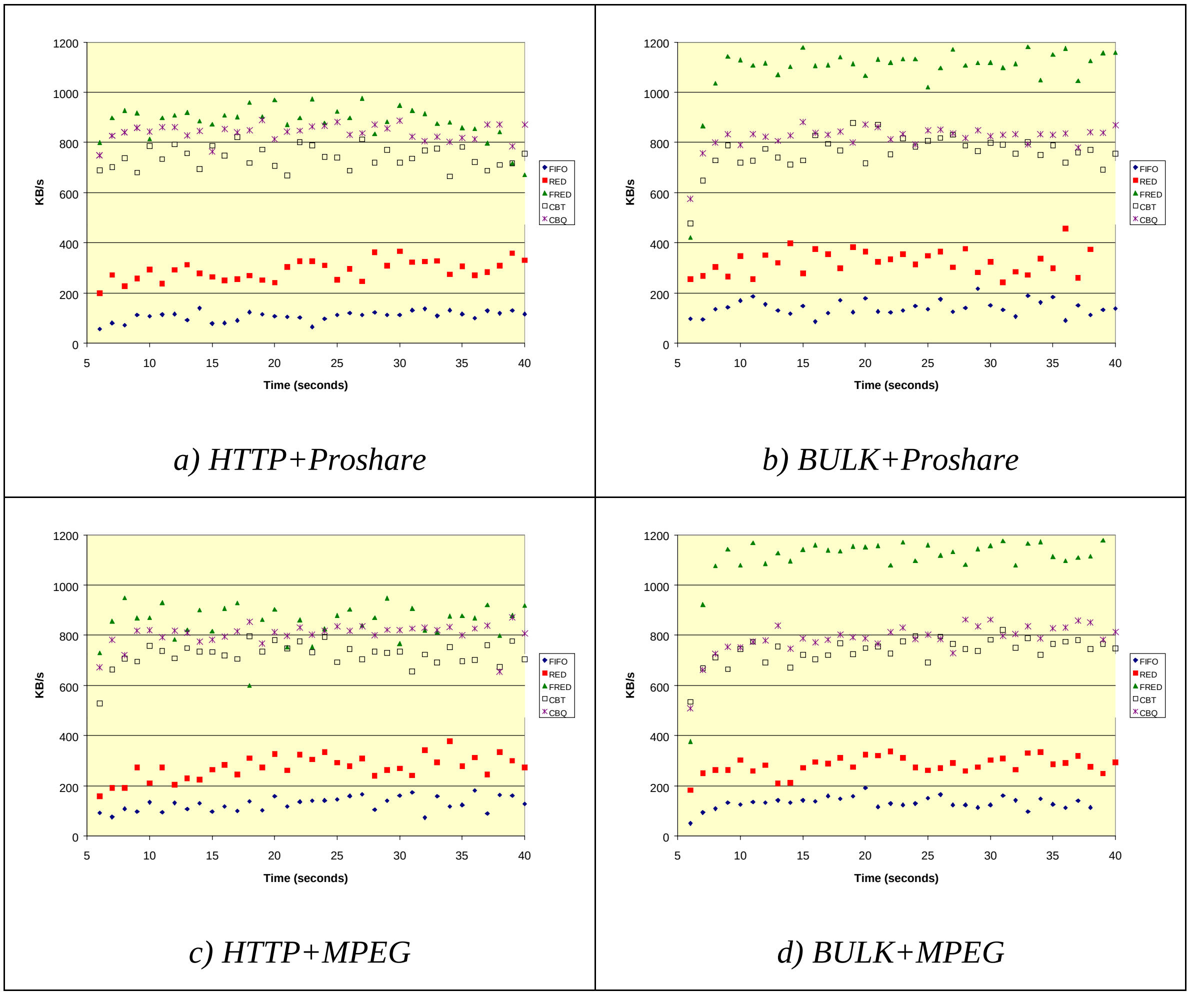


Figure 5.4 TCP Load Averaged over 1 Second Intervals for HTTP+Proshare During the Blast Measurement Period

The performance with other traffic mixes is similar to that with HTTP+Proshare Figure 5.5 shows the TCP goodput for each traffic mix. Note that the goodput of BULK TCP (Figure 5.5*b*, *d*) is slightly higher than that with HTTP (Figure 5.5*a*,*c*). Because the BULK flows are long-lived, higher bandwidth, and more numerous those TCP flows are capable of consuming more bandwidth when it is available. Moreover, goodput for BULK traffic is much higher with FRED because there are typically more active TCP flows with the BULK traffic than with HTTP. As a result, TCP represents a larger fraction of the flows sharing the router's queue and thus the link's capacity. Also note that for CBT and CBQ, TCP goodput with MPEG (Figure 5.5*c*,*d*) is slightly lower than the respective cases with Proshare (Figure 5.5*a*,*b*). This is due to the higher multimedia allocations for MPEG and consequently lower TCP allocations in those experiments. FRED, CBT, and CBQ are much more effective than the other algorithms in isolating TCP from the effects of unresponsive flows.



a) HTTP+Proshare

b) BULK+Proshare

c) HTTP+MPEG

d) BULK+MPEG

Figure 5.5 TCP Goodput for All Traffic Mixes During the Blast Measurement Period

2.1.2. Throughput for *Other* Traffic

The effectiveness of the algorithms in isolating TCP is directly related to how the algorithms constrain high-bandwidth unresponsive flows. If unresponsive flows are constrained TCP will not suffer loss due to queue overflow triggered by unresponsive traffic.

In these experiments, the throughput of the traffic class *other* reflects how effectively unresponsive traffic is constrained. Figure 5.6 shows the throughput for class *other* during the blast measurement period for HTTP+Proshare. The throughput for *other* is essentially the complement of the TCP goodput in Figure 5.5. FRED, CBT, and CBQ effectively constrain *other* to a small fraction of the link's capacity while RED and FIFO allow *other* to dominate the link. Since *other* is represented by a single high-bandwidth flow, FRED constrains the class *other* to approximately $1/n^{th}$ of the link capacity, where n is the number of active flows.

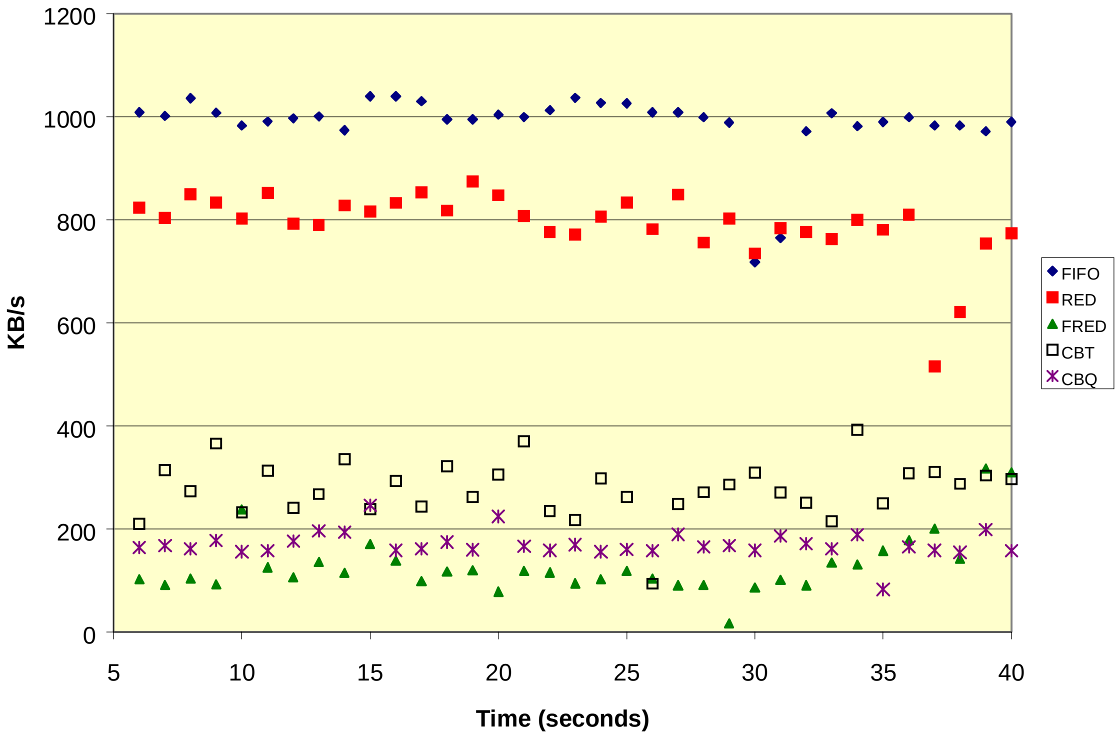
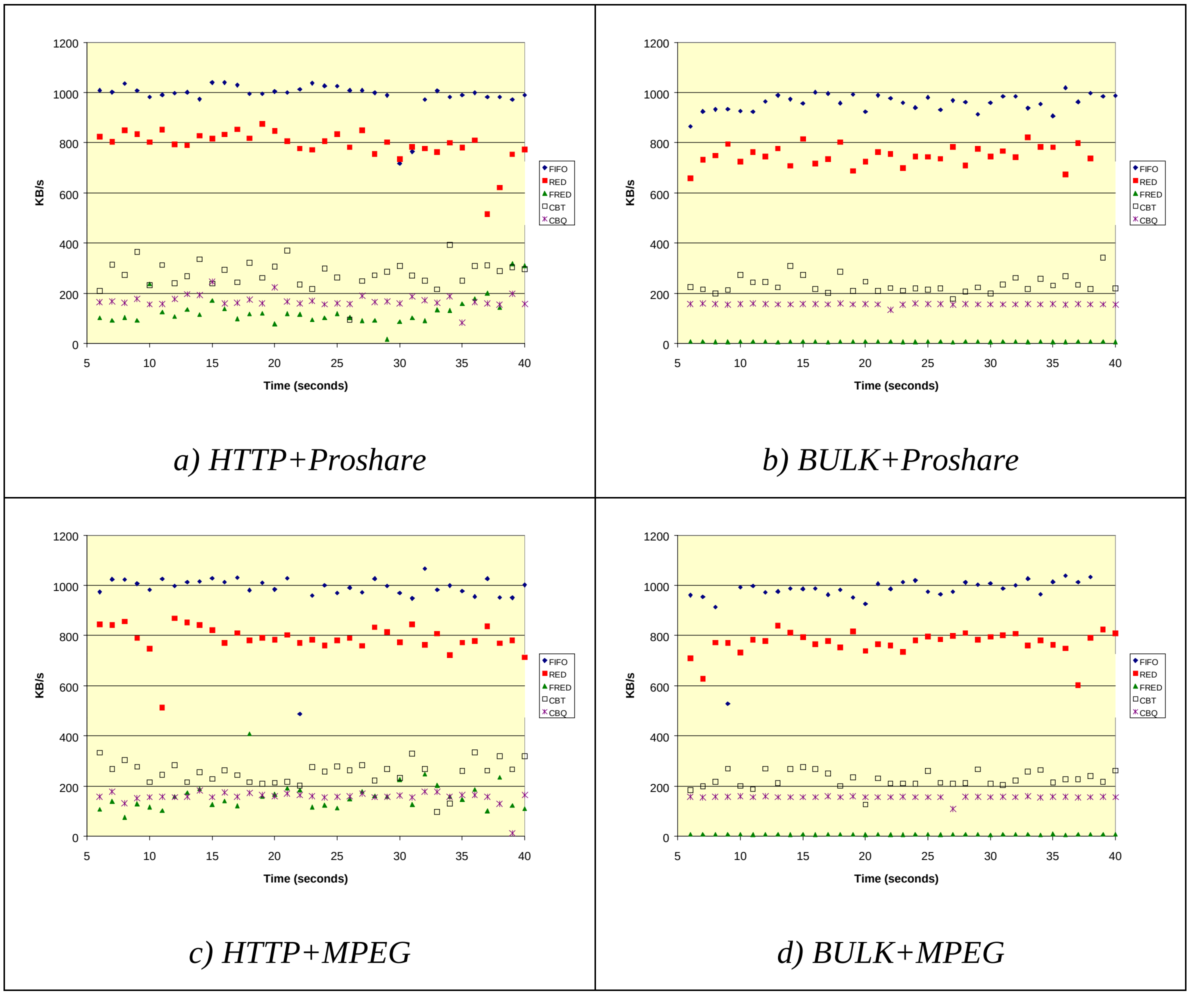


Figure 5.6 Throughput for *Other* Traffic Averaged Over 1 Second Intervals with HTTP+Proshare During the Blast Measurement Period

FRED constrains *other* most severely; however, CBQ constrains *other* more accurately. That is, the bandwidth allocation for *other*, B_{other}, was 150 KB/s and under CBQ its throughput is in that range. CBT allows *other* to exceed this allocation more frequently than CBQ. This can be attributed to several factors: the weighted running average queue occupancy calculation of CBT, CBT's reliance on accurate information on average packet sizes, differences in the mechanics of borrowing for CBT versus CBQ, and the use of early drop mode when dropping packets for TCP in CBT. Consider each. First, CBT's weighted running average is less precise than CBQ's scheduling as a mechanism for allocating bandwidth. Moreover, in CBT the classes may be sampled at different rates. For example, if one TCP packet arrives over an interval while 60 packets drain but 50 *other* packets arrive in the same interval, *other's* average will reflect recent queue occupancy while TCP's average will still be highly influenced by the TCP occupancy at the beginning of the interval. As a result, there may be a subsequent interval when the queue occupancy of TCP or multimedia decreases because that class's average (highly influenced by out of date data) dictates that all arriving packets should be dropped. If *other* maintains its queue occupancy constant during this interval, *other* will have a larger fraction of the packets in the queue and thus a larger fraction of the outbound link. Moreover, because the implementation of CBT used here relies on average packet size values when calculating the threshold values, variation in packet size can result in variation in the resulting queue allocation. Consider an example. If TCP, multimedia, and *other* each have thresholds of 10 packets with expected average packet size of 500 bytes then each class would on average have 5,000 bytes enqueued. Consequently, each would receive one-third of the link's capacity. However, if multimedia actually generated packets of size 1,000 bytes over some interval, then multimedia would have 10,000 bytes enqueued compared to 5,000 for each of the other classes. Consequently, multimedia would receive half of the link's capacity. Note that this reliance on average packet sizes is an implementation limitation. The algorithm can be easily modified to track the number of bytes consumed in the queue instead of packets. Further, CBQ allows for the specification of a hierarchy of borrowing while CBT simply allows all classes to borrow in proportion to their shares. In these experiments CBQ is configured to allow TCP and multimedia to borrow each other's excess first and, only if excess remains can *other* borrow from that excess. As a result, *other* is more tightly constrained.

Finally, CBT uses an early drop policy to manage the queue occupancy by TCP. As a result, TCP packets are dropped before TCP reaches its maximum threshold. The result is that TCP typically maintains average queue occupancy that is less than the maximum threshold. This gives TCP a smaller share of the queue than it was allocated and, thus, a smaller share of the link's capacity. *Other* is able to claim this unused capacity and exceed its initial bandwidth allocation. However, although CBT's constraint of *other* is imprecise, it is still effective in constraining that traffic type.

Further, HTTP is more sensitive to congestion than BULK, allowing more opportunities for the remaining traffic classes to borrow its unused capacity. This sensitivity is due to the fact that most HTTP connections are short-lived. Because the HTTP flows are short-lived they rarely make it out of slow-start mode. Because of this the flows typically operate with a very small congestion window and many of the drops are only detected due to time-out. As a result, the flows sit idle much of the time, leading to a lower average load during periods of congestion than what would typically be seen with longer-lived flows, such as BULK. The data confirms this. Figure 5.7 shows the throughput for class *other* for all of the traffic mixes. Note that *other* is more tightly constrained with CBT when the TCP traffic type is BULK. Moreover, throughput with CBQ is almost exactly 150 KB/s. This is because BULK maintains sufficient load to prevent *other* from borrowing its unused capacity. Similarly, FRED also constrains *other* much more tightly with the BULK traffic mix. Because there are many more active TCP flows with BULK, the fraction of capacity received by each flow is much smaller, thus the single flow of class *other* is more tightly constrained.



a) HTTP+Proshare

b) BULK+Proshare

c) HTTP+MPEG

d) BULK+MPEG

Figure 5.7 Throughput for *Other* Traffic Averaged over 1 Second Intervals with HTTP+Proshare During the Blast Measurement Period

However, in all cases, the relative performance of the algorithms is consistent. FIFO and RED fail to constrain *other*, FRED constrains *other* to a very low throughput, CBT constrains *other*, though not always to its precise allocation, and CBQ constrains *other* very effectively to the 150 KB/s allocated to it.

2.1.3. Multimedia

To effectively support TCP and multimedia an algorithm should isolate not only TCP from the effects of aggressive unresponsive traffic, but should also isolate multimedia from the effects of aggressive unresponsive traffic as well. Multimedia's performance can be evaluated by considering throughput, latency, loss, and frame-rate. First, consider throughput. To evaluate throughput one must consider it relative to the offered load.

Figure 5.8 shows the offered aggregate Proshare load on the router's inbound link during the experiments with HTTP+Proshare. As expected given the scripting and traffic generation the multimedia load is very similar for each experiment, ranging from ~140 KB/s to ~180 KB/s over the duration of each experiment. However, note that the load varies between these two extremes. Consequently, there should be corresponding variation in the throughput.

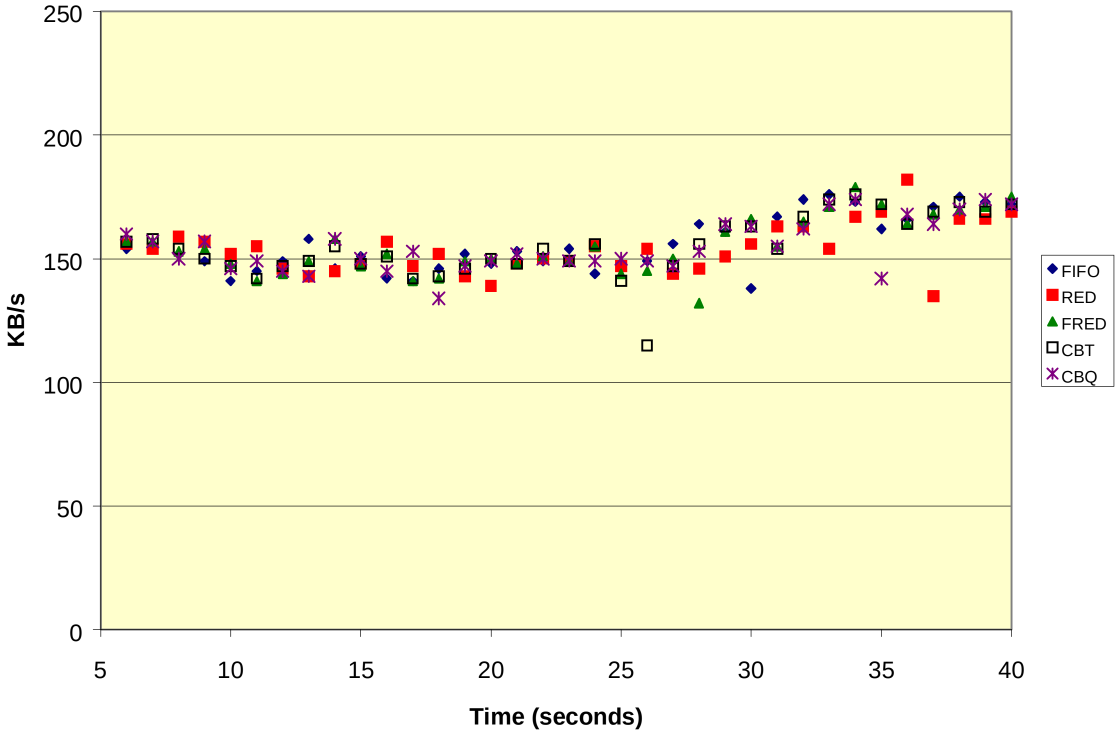


Figure 5.8 Multimedia Offered Load Averaged Over 1 second Intervals for HTTP+Proshare During the Blast Measurement Period

Comparing the offered loads above to the multimedia throughput on the bottleneck link, as shown in Figure 5.9, provides a starting point for evaluating how effectively each algorithm supports multimedia. Precisely how effective the algorithms are will be more apparent when packet loss and frame-rate are considered below. The multimedia throughput is highest for CBT and CBQ (140-180 KB/s). The throughput is nearly equal to the offered load. Classification and bandwidth allocation effectively isolate the multimedia traffic from the effects of the unresponsive flows. Proshare's throughput is nearly equal to the offered load with FRED for the HTTP+Proshare traffic mix. In contrast, multimedia throughput is between 80-120 KB/s for FIFO and RED, well below the offered load. Once again, this is because during periods of overload FIFO and RED drop many packets from all flows.

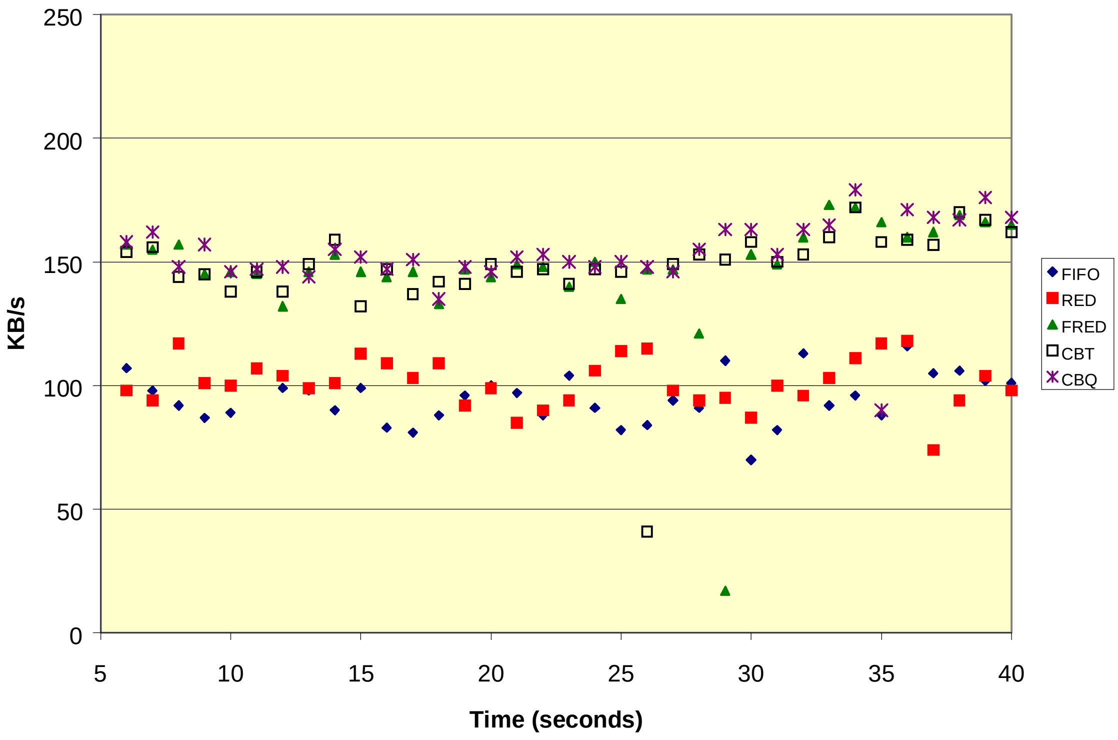
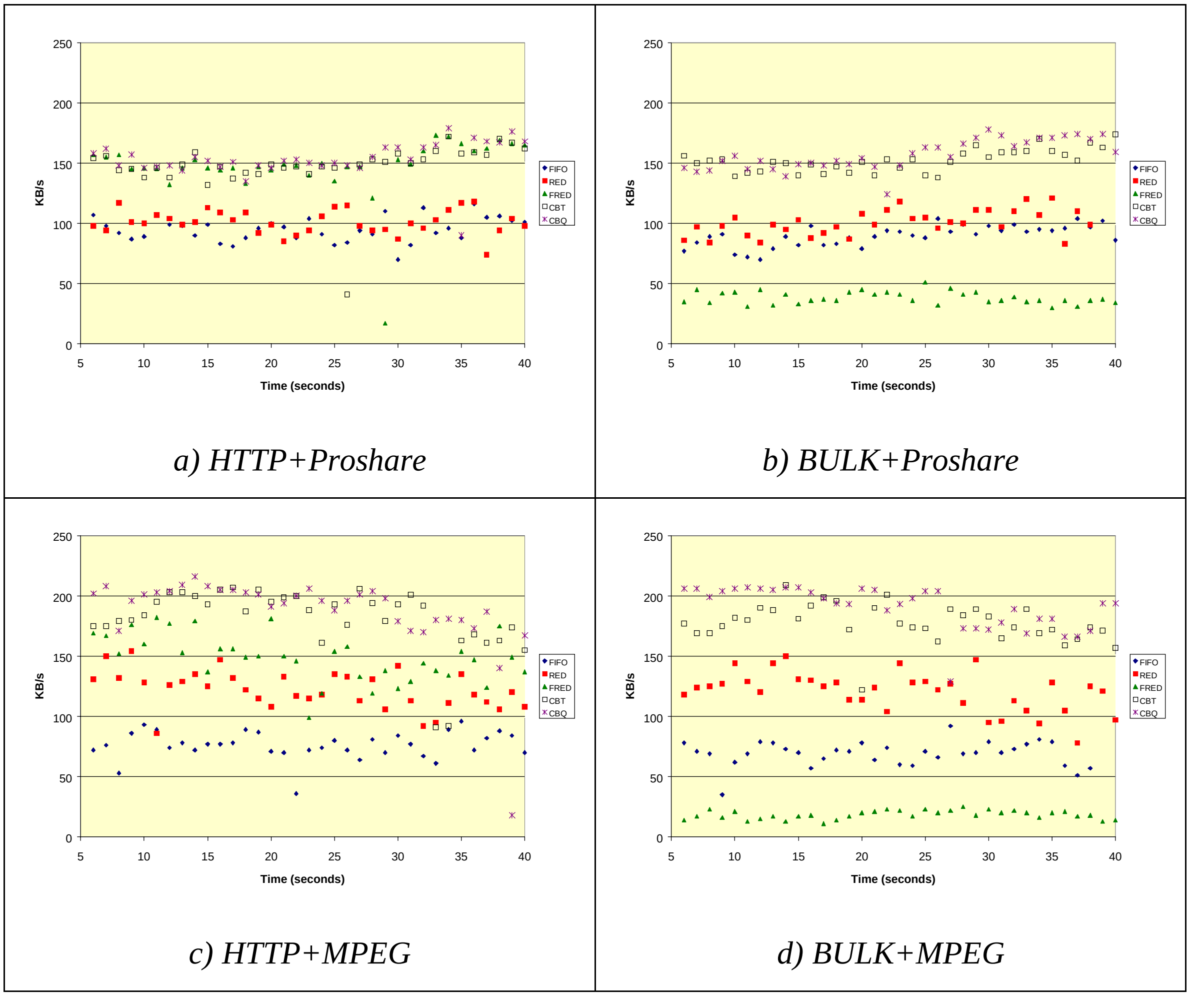


Figure 5.9 Multimedia Throughput Averaged over 1 Second Intervals for HTTP+Proshare During the Blast Measurement Period

Figure 5.10 shows the multimedia throughput for all workload mixes. Multimedia performance with other traffic mixes is consistent with that observed for HTTP+Proshare for all algorithms except FRED. CBT and CBQ offer throughput approximately equal to the offered load while FIFO and RED result in significantly lower throughput.

Multimedia performance with FRED varies as a function of workload mix. Although Proshare's throughput is nearly equal to the offered load for HTTP+Proshare (Figure 5.10*a*), thoughput reduces by two-thirds when the traffic mix is BULK+Proshare (Figure 5.10*b*). The increase in the number of active flows with BULK accounts for this variation because the share of the queue buffers, and thus the link capacity, allocated to each flow decreases as the number of flows increases. Since the number of multimedia flows remains constant the aggregate share of the queue allocated to multimedia decreases. A similar effect occurs for MPEG (Figure 5.10*c*,*d*). However, because MPEG's offered load (both per-flow and aggregate) is higher than Proshare's, the effects of FRED's per-flow constraint are noticeable even with HTTP as the TCP traffic type. Moreover, because FRED's per-flow drop decisions are based on instantaneous queue occupancy, FRED is biased against flows that are bursty, dropping packets from those flows even when the average offered load is less than 1/n^{th} of the link (where *n* is the number of active flows). Since MPEG's I-frames span as many as five packets on average, they are likely to trigger this biased behavior of FRED. The effect of this behavior on playable frame-rate can be severe and is discussed in the section on frame-rate, below.



a) HTTP+Proshare

b) BULK+Proshare

c) HTTP+MPEG

d) BULK+MPEG

Figure 5.10 Multimedia Throughput Averaged Over 1 Second Intervals During the Blast Measurement Period.

In summary, CBT and CBQ offer the best multimedia throughput in all cases. The throughput for the cases with MPEG (Figure 5.10*c*,*d*) is higher because MPEG offers a higher average load than Proshare. FRED offers reasonable throughput for HTTP+Proshare but this is the accidental result of a combination of factors as discussed above. For the other three traffic combinations FRED's multimedia throughput is significantly lower than with CBT and CBQ. Finally, RED and FIFO offer lower throughput in all four cases.

Although multimedia throughput is an effective initial measure of the effectiveness of the algorithms, to fully evaluate the algorithms one must consider latency, loss-rate, and the resulting frame-rate. Those metrics are presented below.

Latency

Latency must be limited to maximize interactivity and limit the amount of end-system buffering necessary for jitter management. Figure 5.11 shows the end-to-end latency for the HTTP+Proshare traffic mix during the blast measurement period. These latency values include the effect of propagation delay and end-system queueing. (However those factors are minimal, on the order of 5 milliseconds.) As a result, the differences in these values directly reflect differences in queue-induced delays at the bottleneck router. Note that in the plots with Proshare as the multimedia traffic type, the latency values are reported on a packet by packet basis, instead of averaging over 1 second intervals. This illustrates the variability between instantaneous latency values which leads to jitter. It also points out the relation between consecutive latency values. During periods of overload consecutive packets will have nearly the same queue-induced latency because the latency is a result of the queue size when the packet arrives and the queue occupancy changes slowly. Queue occupancy changes only when packets arrive and depart.

With FIFO, multimedia has a delay of just under 60 ms. This is a direct result of the maximum queue occupancy used for FIFO, 60 packets. The average size of packets in these experiments was approximately 1000 bytes and each required about 1 ms to be transmitted on the 10 Mb/s bottleneck link. With FIFO's queue constantly full due to the overload, the queue-induced latency approaches 60ms. Similarly, RED's maximum threshold of 40 packets constrains the average queue occupancy, leading to the 35-40 ms of latency observed. For the HTTP+Proshare mix the maximum threshold value of 60 packets also constrains the queue occupancy when using FRED. Note that for both RED and FRED it takes less than 1 ms to process the average sized packet, so the latency is slightly less than 1ms/packet enqueued.

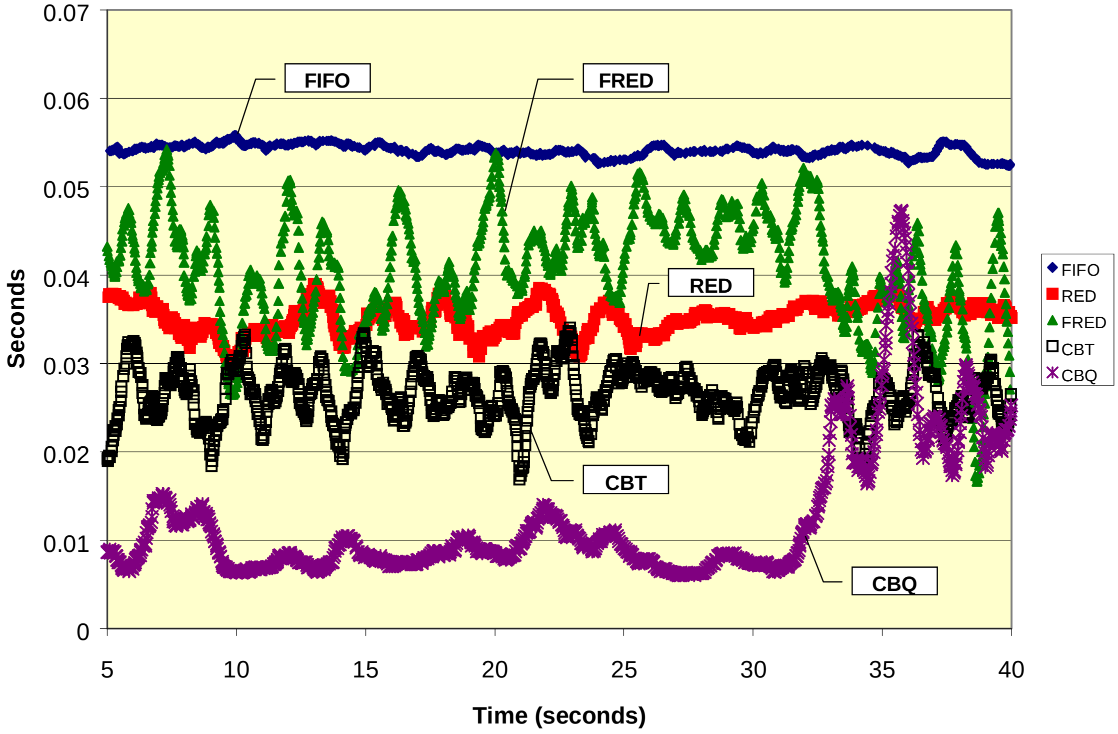
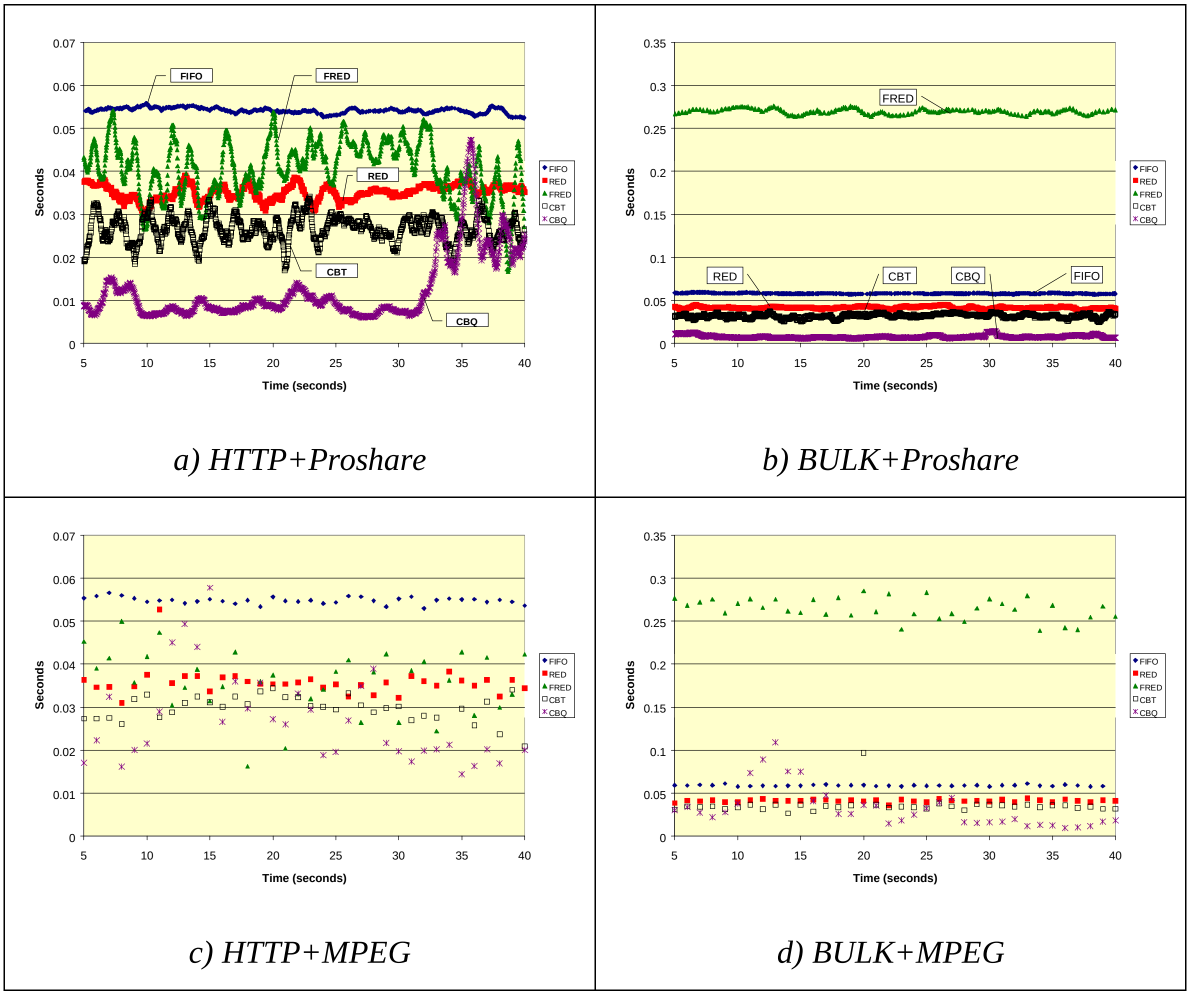


Figure 5.11 End-to-End Multimedia Latency During the Blast Measurement Period for HTTP+Proshare

With CBT, the latency is determined by the aggregate average queue occupancy for all classes which is limited by the aggregate maximum threshold settings for the classes. Moreover, those values are calculated based on the desired maximum average latency. In these experiments that value was 30 ms and thus the latency averages near 30 ms. In contrast, CBQ's latency is usually quite low, around 10ms. This is because each class of traffic has its own queue. If a CBQ class is operating with a load less than its bandwidth allocation, queue occupancy for that class will remain near zero and arriving packets will be forwarded almost immediately. In this experiment, that is precisely what happens for CBQ up until time 32 when latency jumps to 20-50ms. At this point a queue forms because the load generated by multimedia exceeds the bandwidth allocation. Recall from Figure 5.8 that at time 32 the offered load for multimedia increases. In fact, it exceeds the

149 KB/s allocated to multimedia by CBQ, causing a queue to form. Although CBQ has sufficient queue capacity to avoid dropping any of the multimedia packets, this brief variation does result in the increased queueing delay as shown. This phenomenon is also evident for CBQ with BULK+MPEG during the interval (10,15) as shown below.



a) HTTP+Proshare

b) BULK+Proshare

c) HTTP+MPEG

d) BULK+MPEG

Figure 5.12 End-to-End Multimedia Latency Averaged Over 1 Second Intervals During the Blast Measurement Period

Figure 5.12 shows the latency values for all traffic mixes. Note that for the HTTP (Figure 5.12*a*,*c*) and BULK (Figure 5.12*b*,*d*) traffic mixes the latency values are reported on different time scales. Although the results are consistent with those observed for HTTP+Proshare for most of the algorithms, FRED's behavior changes significantly. Although FRED's thresholds are constant at (60,5) across all four traffic mixes, the latency increases dramatically when the TCP traffic type is BULK instead of HTTP. This occurs because there are many more active flows with BULK. Since FRED allows any flow to have two packets enqueued regardless of the average queue length, when there are a large number of active flows the occupancy is constrained only by the size of the queue. In this case the queue size is 240 packets. When BULK is the TCP traffic type, the average packet size is slightly larger than 1000 bytes and requires slightly more than 1 ms to process. This leads to a queue-induced latency of approximately 260 ms. As a result, the latency at this router alone exceeds the perception threshold which is a significant drawback for the FRED algorithm.

Also, notice that CBQ latency briefly grows quite large up to time 15 for BULK+MPEG (Figure 5.12*d*). This occurs because multimedia load in excess of CBQ's bandwidth allocation causes a queue to build for the multimedia class. To understand this behavior, consider the offered load for multimedia in this experiment. Figure 5.13 shows multimedia's offered average load during the blast measurement period as stars. The dashed line indicates the bandwidth allocation for multimedia (B_{mm}) of 196 KB/s. Clearly, the load slightly exceeds the allocation during the period from time 5 to time 16. As a result, a queue builds for the multimedia class and the packets incur significant queue-induced latency as shown in Figure 5.12 (Figure 5.13*d*). This also occurs in the HTTP+MPEG case (Figure 5.13*c*) but the effect is not as severe because HTTP does not use all of it allocated capacity so multimedia can borrow from TCP to address this slight overload.

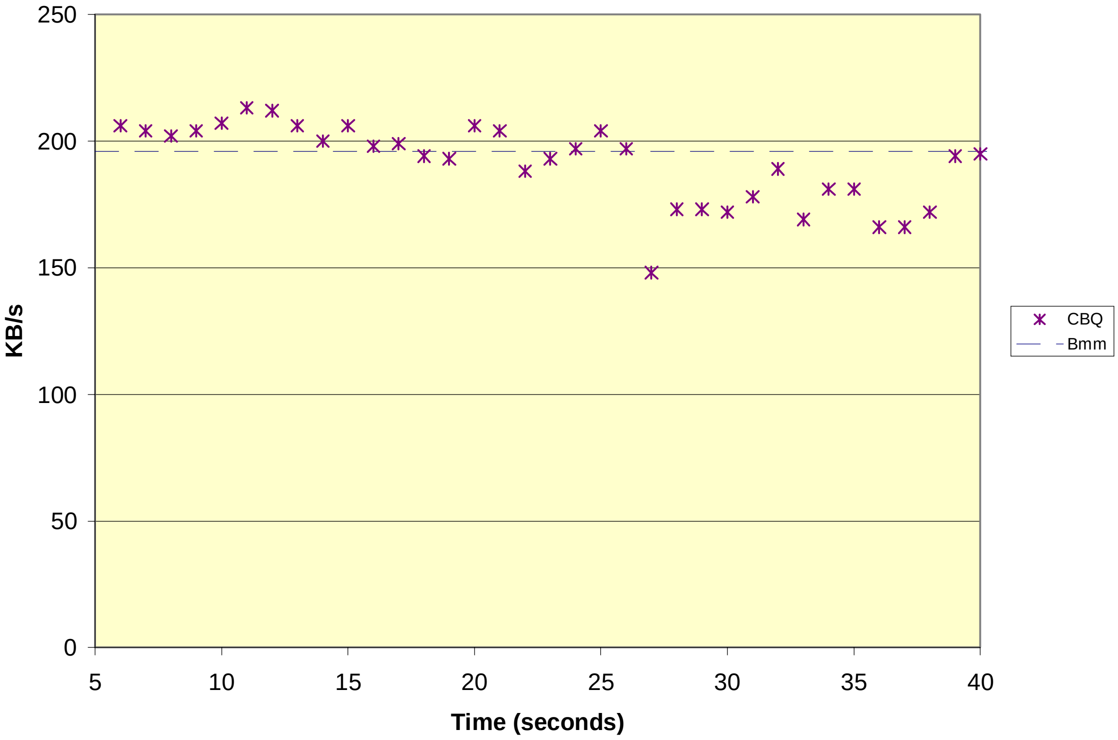


Figure 5.13 Multimedia Load Averaged Over 1 Second Intervals for BULK-MPEG for CBQ During the Blast Measurement Period

In summary, CBQ offers the lowest latency, though it is subject to variation if the offered load for multimedia exceeds the allocation. Latency with FRED can be unpredictable. Although the maximum threshold limits the queue occupancy when there are moderate numbers of active flows, when the number of active flows approaches the queue size the queue-induced latency is limited only by the queue size. Moreover, even in the best case, FRED's optimal parameter settings offer latency on order of 50ms, second highest of all of the algorithms. Similarly, RED and FIFO offer poor latency performance relative to the other algorithms. The latency value can be managed for these algorithms by changing the queue size or Th_{Max}, respectively, but when optimal parameters were selected it was noted that decreasing these values limited the capacity to accommodate bursty arrivals and decreased TCP efficiency. In contrast, CBT is able to consistently constrain queue occupancy and limit queue-induced latency to a desired value, in this case 30ms.

Loss and Frame-rate

The final metrics considered for multimedia were loss-rate and frame-rate. The loss-rate for multimedia affects the playable frame-rate but the relationship is not necessarily linear. Issues such as interframe encoding and marshalling of frames into many packets mean that loss of one packet may translate to the loss of many frames. Further, some queue management algorithms may be biased against bursty arrivals, causing packet losses to be concentrated in large frames spanning multiple packets because those packets are generated simultaneously and hence may arrive at the router nearly simultaneously. First we consider the loss rate under all traffic mixes and then the corresponding frame-rates (for MPEG).

Traffic Mix	**Multimedia Loss-rate (Packets/Second)**				
	FIFO	**RED**	**FRED**	**CBT**	**CBQ**
HTTP+Proshare	14.8	11.9	2.2	1.2	0
HTTP+MPEG	19.0	13.7	9.2	2.6	0
BULK+Proshare	15.7	13.0	27.7	1.1	0
BULK+MPEG	24.1	14.7	33.1	4.5	0

Table 5.5 Loss Rates for Multimedia During the Blast Measurement Period

The loss-rates in packets/second are shown in Table 5.5. This metric is useful when comparing algorithms for a single media type. Clearly CBQ's loss-rate of zero across all traffic mixes is superior. Likewise, CBT offers the second lowest loss-rate for all mixes, five to ten times better than the other algorithms in all but one case. The only exception is FRED's loss-rate for HTTP+Proshare which is only slightly higher than CBT's. However, FRED's loss-rate for HTTP+MPEG is higher and for the BULK traffic mixes FRED offers the highest loss-rate of all. The reasons for this behavior were addressed in the section on multimedia throughput above. Finally, the loss-rates for FIFO and RED are uniformly high with FIFO's consistently slightly higher than RED.

To better understand the effect of these loss-rates, it is helpful to consider the resulting frame-rates. Consider frame rate with HTTP+MPEG. (Only the MPEG traffic generators offer frame-rate information so using HTTP+Proshare here isn't possible.) The actual frame-rates (i.e., the number of frames successfully received independent of decoding concerns) for MPEG are shown in Figure 5.14. The MPEG sender is transmitting 30 frames per second. CBQ is able to deliver at that rate while CBT maintains a frame-rate near 28 frames per second with some variation. As expected from the drop-rates, FRED offers a degraded frame-rate of 20-25 frames per second while FIFO and RED offer poorer frame-rate performance. RED drops the actual frame-rate below 20 frames per second while FIFO averages around 15 with a good deal of variability.

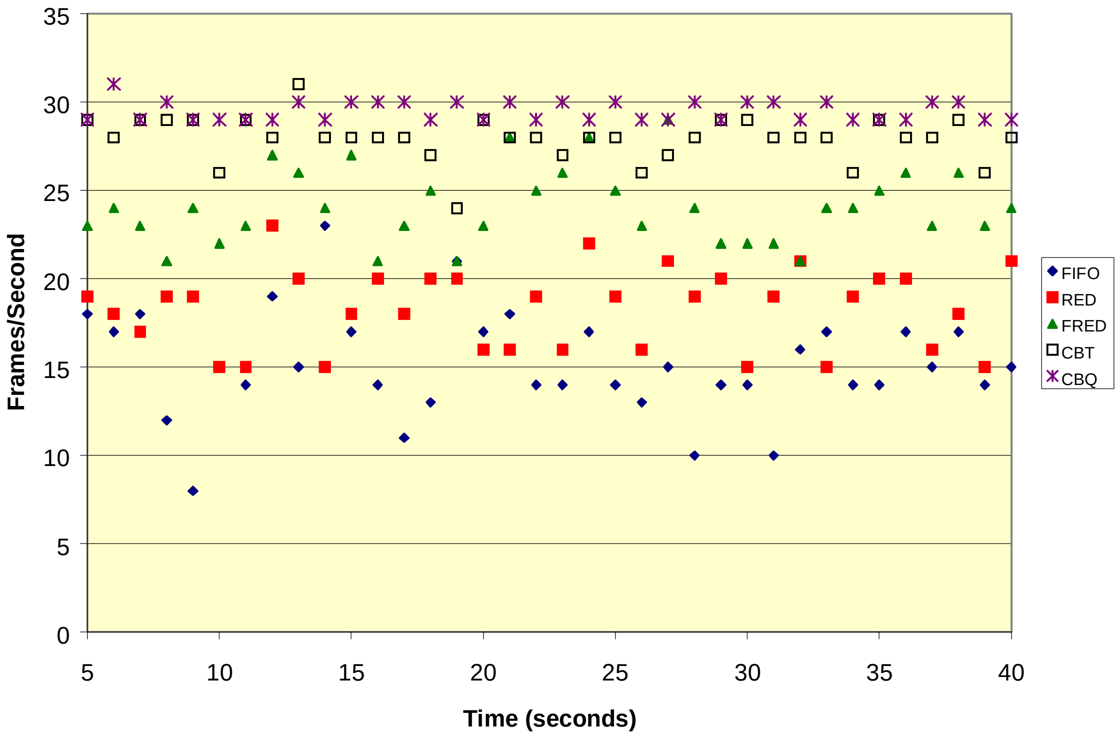
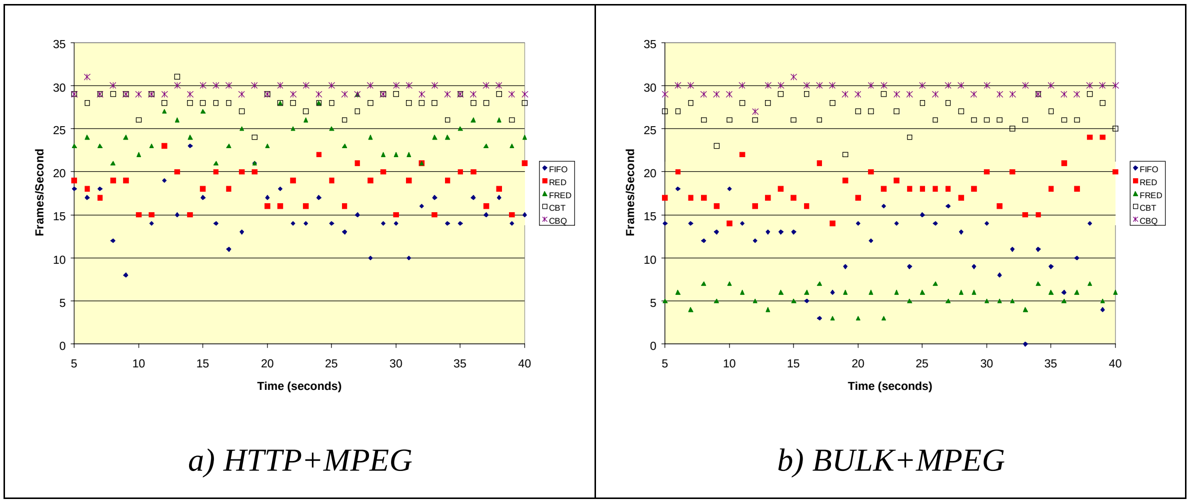


Figure 5.14 Actual Frame-Rate Averaged Over 1 Second Intervals for HTTP+MPEG During the Blast Measurement Period

Figure 5.15 also shows the actual frame-rate information for BULK+MPEG (Figure 5.15*b*) as well as HTTP+MPEG (Figure 5.15*a*). The MPEG frame-rate with BULK traffic is slightly poorer in all cases. This is to be expected as we have previously seen that BULK flows consume more of the bandwidth than HTTP. In the case of RED and FIFO this means that the drop-rate must be even higher and so the drop-rate for MPEG increases and the actual frame-rate decreases. As expected, the effect of the BULK traffic is most severe with FRED as the frame-rate degrades to approximately 5 frames per second. In the case of CBT, the fact that BULK is more consistent in its consumption of allocated bandwidth means there is little excess capacity for multimedia to borrow when it briefly exceeds its threshold so MPEG's frame rate is slightly lower in those instances. In contrast CBQ has sufficient buffer space to queue excess packets when multimedia does exceed its threshold so no packets are dropped. Although the actual frame-rate is an indication of network performance it does not give a strong indication of the quality of the media stream. To do that, one must consider the playable frame-rate.



a) HTTP+MPEG

b) BULK+MPEG

Figure 5.15 Actual Frame-Rate Averaged Over 1 Second Intervals for MPEG During the Blast Measurement Period

Recall that the *playable frame rate* is the number of packets that can be successfully decoded for playback. For a media encoding that has interframe dependencies, such as MPEG, the playable frame-rate can vary significantly from the actual frame-rate. For example, if an I frame is lost the entire GOP cannot be decoded. In these examples the GOP is IBBPBB so loss of an I frame results in the effective loss of 6 frames because they cannot be decoded. Figure 5.16 shows the playable frame-rate for HTTP+MPEG. Only CBQ has a playable frame rate approaching the actual frame rate. However, the relative performance of the algorithms remains consistent with the exception of FRED. With CBT the stream maintains a frame rate of 20 frames per second or better 80% of the time.

However, RED degrades to approximately 5 frames per second and FIFO's performance is even worse, as almost no frames can be decoded for playback. Although FRED had an actual frame-rate between 20-25 frames per second the playable frame-rate varies between 0 and 25 frames per second. As previously discussed, FRED's use of instantaneous queue occupancy for per-flow drop decisions introduces a bias against bursty arrivals. In this case the larger I-frames span as many as 7 packets which arrive nearly simultaneously at the router, triggering this bias. Consequently, the subsequent frames of the GOP cannot be decoded. The results for BULK+MPEG (Figure 5.17b) are consistent with those seen for HTTP+MPEG.

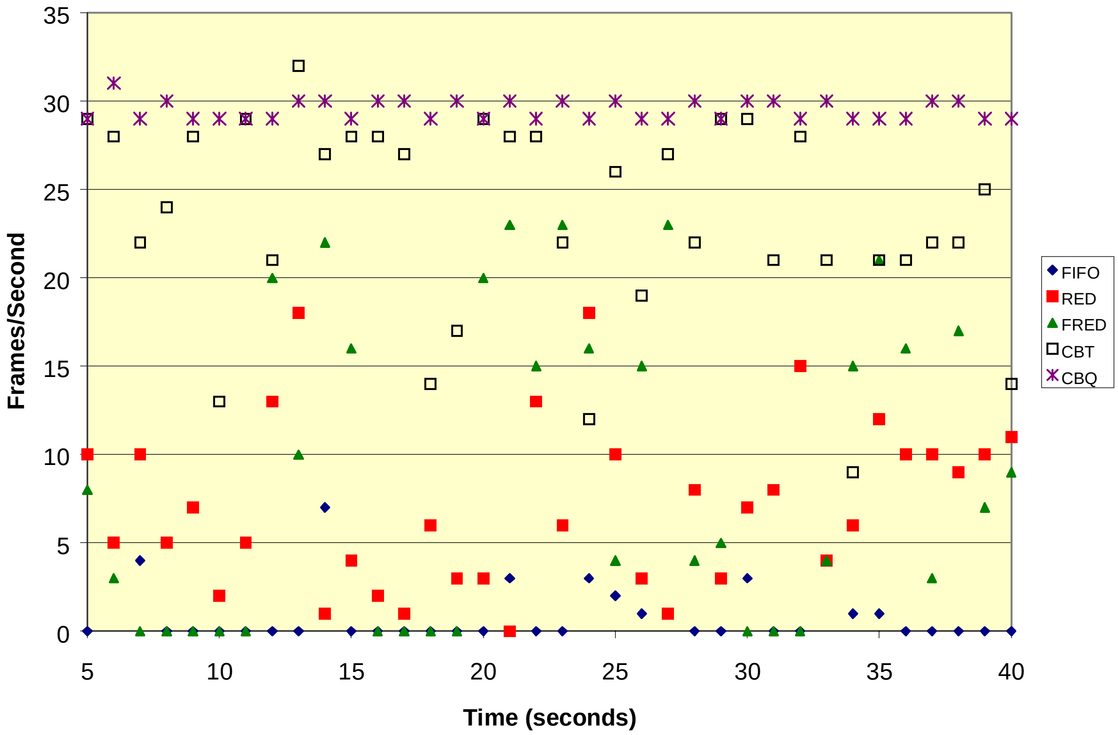
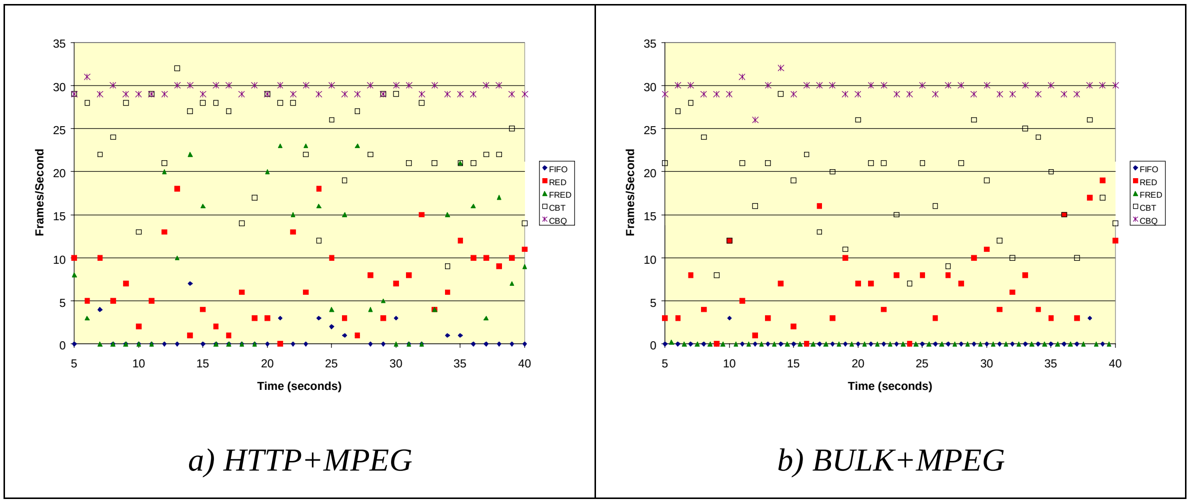


Figure 5.16 Playable Frame-Rate Averaged Over 1 Second Intervals for HTTP+MPEG During the Blast Measurement Period



a) HTTP+MPEG

b) BULK+MPEG

Figure 5.17 Playable Frame-Rate Averaged Over 1 Second Intervals During the Blast Measurement Period

Examining frame rate and loss-rate emphasizes the fact that quality of a multimedia stream does not degrade continuously. In a file transfer, increased drops increase the time necessary to transfer an object from sender to receiver but the object is still useful when it arrives. In contrast a drop rate of 20% can lead to a playable frame-rate near zero making the media stream useless.

2.1.4. Summary of Blast Measurement Period

These experiments confirmed the vulnerability of FIFO and RED to aggressive, unresponsive flows. TCP goodput was low, *other* dominated the link's capacity, and multimedia throughput and thus loss-rate and frame-rate were poor. Although queue-induced latency was tolerable at 60 and 40 ms, it mattered little because the high loss-rate severely degraded the multimedia streams.

In contrast, FRED offers good TCP performance and effectively constrains *other* but the mechanism used to constrain *other* also over constrains multimedia. In three out of the four traffic mixes examined this results in high loss for multimedia. Moreover, FRED's use of per-flow instantaneous queue occupancy in the drop decision leads to a bias against bursty packet arrivals. Consequently, MPEG's large I-frames are very likely to be subject to loss and, consequently, has a poor playable frame-rate even when the loss-rate is low. Further, when the number of active flows approaches the size of the queue, FRED's performance equates to that of FIFO, leading to a high drop-rate and high queue-induced latency.

CBQ offers the best performance. The bandwidth allocations accurately indicate the throughput each class of traffic actually receives. When properly configured TCP goodput is high, *other* is effectively constrained, and there is no multimedia loss and a very high frame-rate. However, during periods of brief overload the queue-induced latency can vary significantly.

CBT also offers good performance though the bandwidth allocations less precisely control the resulting throughput. When properly configured, TCP goodput is high, *other* is effectively constrained, and multimedia loss is low, resulting in a high frame-rate. Moreover, queue-induced latency is predictable and configurable.

The measurements during the blast measurement period demonstrate the CBT meets its design goals, isolating TCP and multimedia from each other and from the effects of aggressive, unresponsive flows. Moreover, CBT performs better than the other AQM algorithms examined and the performance is comparable to that of the packet scheduling mechanism, CBQ.

2.2. Multimedia Measurement Period

The second measurement period considered is the multimedia measurement period. The multimedia measurement period is a period when only TCP and multimedia traffic are present. There is no *other* traffic. The goal in this section is to simply demonstrate that CBT has no ill effects on TCP or multimedia and that CBT offers performance that is comparable or superior to the other AQM algorithms. Most of the queue management algorithms handle this scenario reasonably although there are some slight variations in performance. However, FRED does demonstrate the same pathological behaviors with BULK during the multimedia measurement period as it did during the blast measurement period. To evaluate the algorithms during the multimedia measurement period, the analysis parallels that from the blast measurement period.

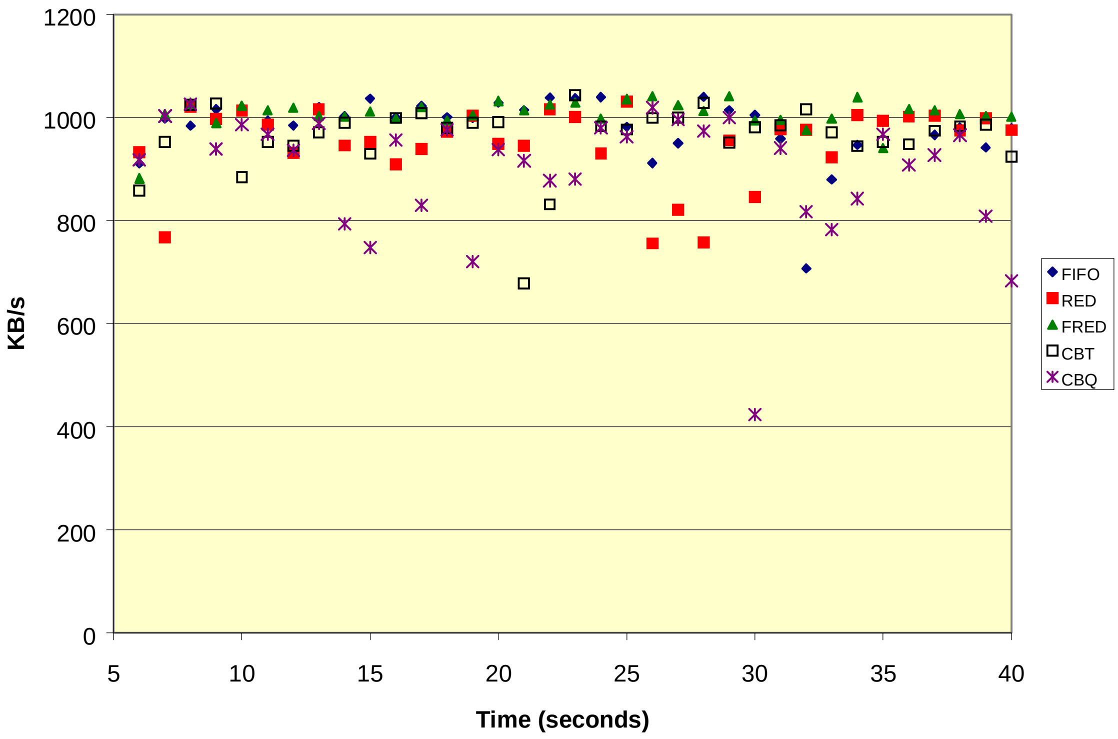
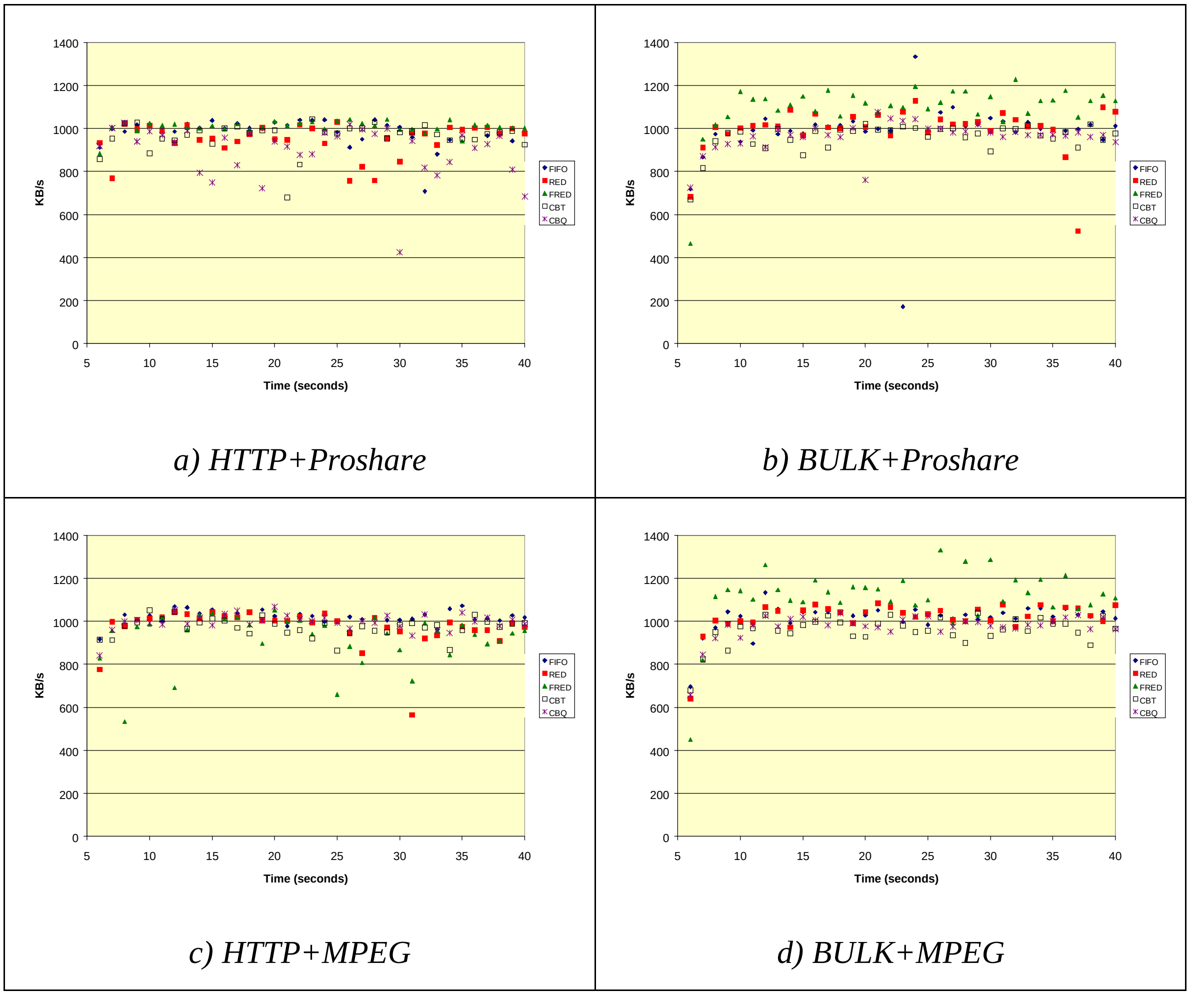


Figure 5.18 TCP Goodput Averaged Over 1 Second Intervals During the Multimedia Measurement Period for HTTP+Proshare

2.2.1. TCP Goodput

Without aggressive, unresponsive traffic present, TCP and multimedia should be able to share the link with TCP getting most of the link's capacity. The TCP goodput measurements shown in Figure 5.18 confirm this. (The low goodput reading for CBQ at time 30 seconds is an artifact due to the measurement technique.) For HTTP+Proshare, the TCP goodput is approximately 1,000 KB/s for all of the algorithms with each algorithm showing occasional fluctuations due to variation in load.



a) HTTP+Proshare

b) BULK+Proshare

c) HTTP+MPEG

d) BULK+MPEG

Figure 5.19 TCP Goodput Averaged Over 1 Second Intervals During the Multimedia Measurement Period for All Algorithms

The goodput values for the other traffic mixes (Figure 5.19) are comparable. However, the goodput is slightly higher for the traffic mixes featuring BULK than the mixes featuring HTTP because BULK maintains a consistently higher load. The goodput is especially high for BULK with FRED because the large number of BULK flows relative to multimedia flows leads to TCP receiving a larger fraction of the link capacity. Otherwise, the performance for a given traffic mix is nearly identical across all of the algorithms.

2.2.2. Multimedia

For multimedia performance, first consider the throughput under each queue management algorithm as a coarse measure of performance. Next, metrics more specific to multimedia, such as latency, and frame-rate are considered.

Throughput

Figure 5.20 shows the multimedia throughput during the multimedia measurement period for HTTP+Proshare. As expected, during periods of light congestion without aggressive traffic, the choice of algorithm has little impact on multimedia throughput. The dips between time 20 and 30 are due to unavoidable flaws in the measurement technique that lead to some samples being lost. The throughput is consistent across all algorithms.

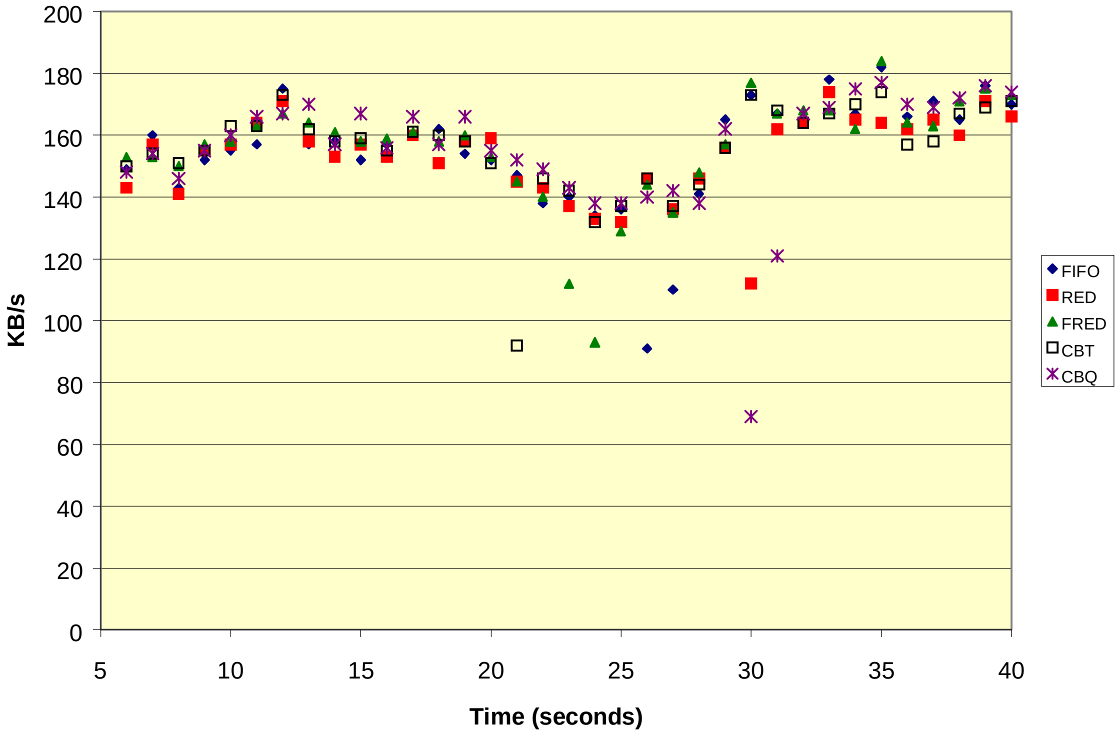
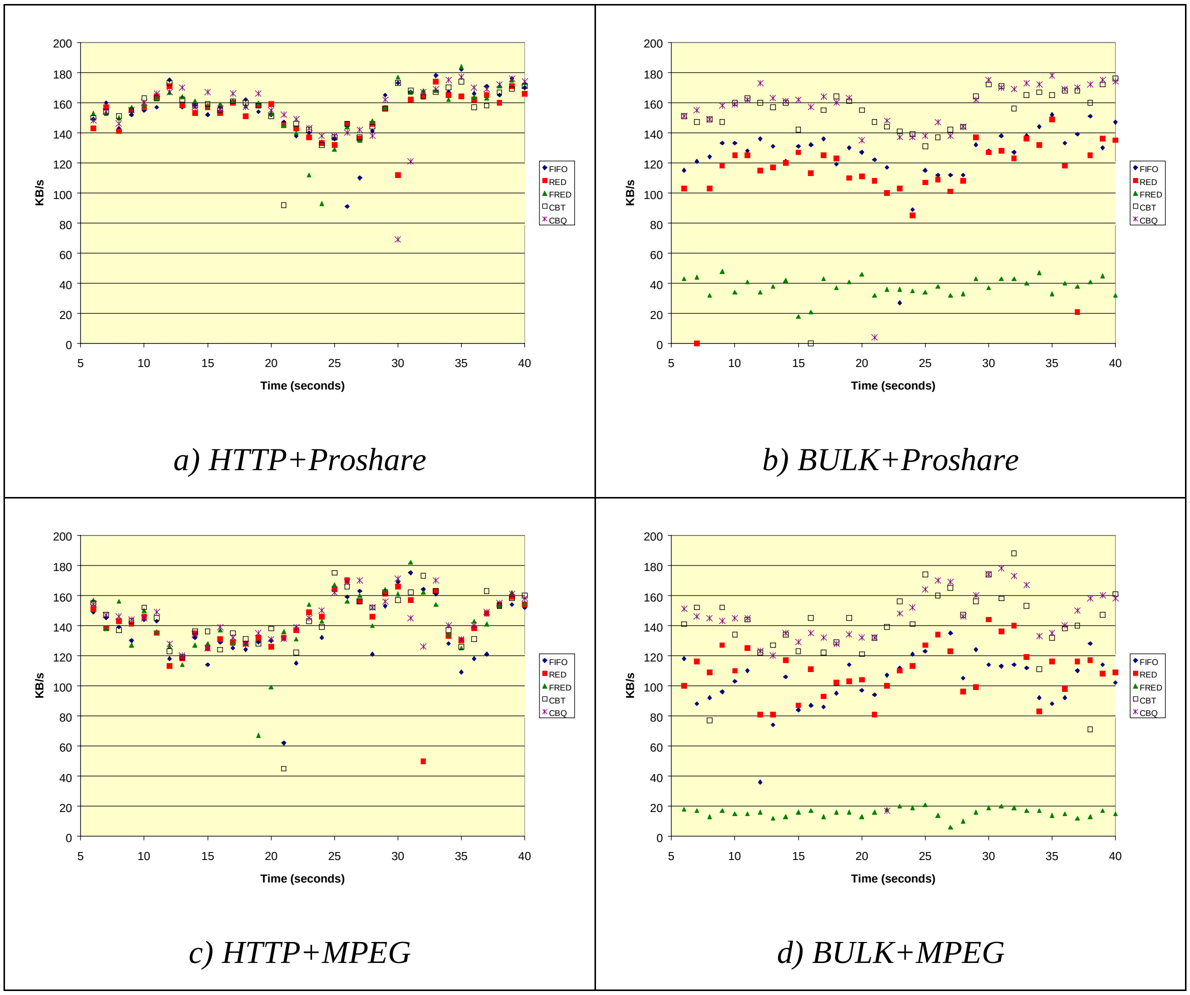


Figure 5.20 Multimedia Throughput Averaged Over 1 Second Intervals During the Multimedia Measurement Period for HTTP+Proshare



a) HTTP+Proshare

b) BULK+Proshare

c) HTTP+MPEG

d) BULK+MPEG

Figure 5.21 Multimedia Throughput Averaged Over 1 Second Intervals During the Multimedia Measurement Period

However, it is interesting to observe how the performance changes when the TCP traffic is BULK instead of HTTP. Recall that the BULK traffic generators offer a more consistent and higher load than the HTTP traffic generators. As a result, the network is in a more persistent state of overload with the BULK traffic mixes even without the UDP blast. When the network is overloaded, the differences in the multimedia performance under the queue management algorithms are more apparent. Figure 5.21 shows the multimedia throughput for all traffic mixes. While the HTTP traffic mixes (Figure 5.21*a*,*c*) offer comparable performance across all of the algorithms, the BULK mixes (Figure 5.21*b*, *d*) highlight the value of the bandwidth allocation offered by CBQ and CBT. For example, the multimedia throughput for CBT and CBQ with BULK+Proshare (Figure 5.21*b*) is consistent with the throughput observed for HTTP+Proshare (Figure 5.21*a*).

However, the throughput drops slightly for RED and FIFO. Moreover, as observed during the blast measurement period in Section 2.1.3, FRED's fair sharing severely constrains multimedia due to the large number of active BULK flows.

Latency

As noted for the blast measurement period, the end-to-end latency reported is primarily a function of the queue-induced latency for each algorithm. For the HTTP+Proshare mix (Figure 5.22), the latency is relatively low for all of the algorithms because the congestion is not persistent enough to test the limits of queue occupancy for each algorithm. FIFO does have high latency (40 ms) up until time 25 but then a brief decrease in the TCP load allows the queue to drain slightly so when the slight overload resumes the latency stays around 20-30 ms. CBT, RED, and FRED operate below their maximum occupancy throughout, resulting in latency ranging from 10 to 40 ms. As always, CBQ's separate queues for multimedia packets allows them to pass through the router quickly, incurring little latency. These latency values are all tolerable.

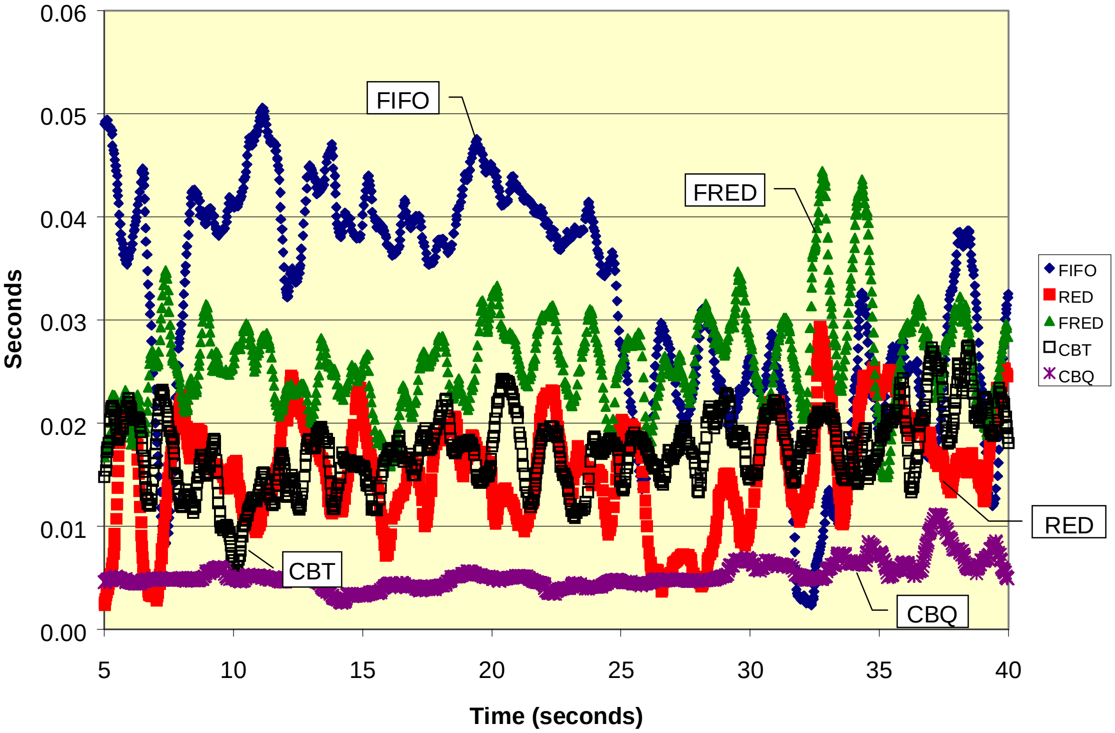
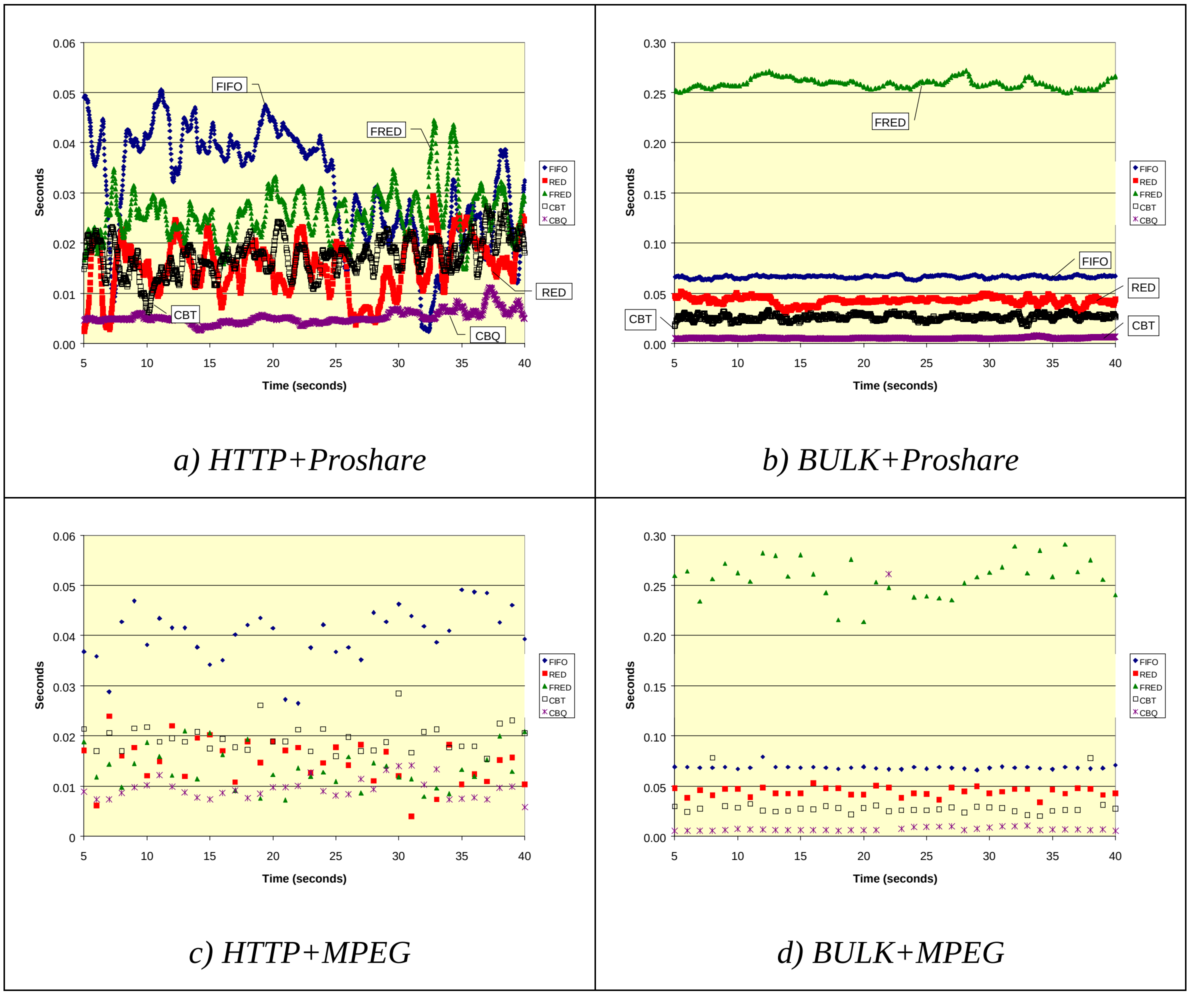


Figure 5.22 End-to-End Multimedia Latency During the Multimedia Measurement Period for HTTP+Proshare

However, the latency values are more interesting when the TCP traffic component is BULK. Figure 5.23 shows the latency for all of the traffic mixes. Note that because the results vary the BULK and HTTP mixes are presented on different scales. The latency for HTTP+MPEG (Figure 5.23*c*) is comparable to that observed for HTTP+Proshare (Figure 5.23*a*), but the values for BULK+Proshare (Figure 5.23*b*) and BULK+MPEG (Figure 5.23*d*) are quite different. These differences are due to the fact that the BULK traffic causes a more persistent state of overload. As a result, when BULK traffic is present queue occupancy is limited only by queue size or threshold settings. FRED offers the worst performance as the large number of flows present in the BULK mixes lead to queue occupancy constrained only by the maximum queue size. The queue size of 240 combines with BULK's large packet size to produce a queue-induced latency of 260 ms. In contrast, the second worst case is FIFO. FIFO's queue size of 60 packets leads to a latency of only approximately 65 ms. RED's average queue-induced latency is constrained only by the limit on average queue occupancy, Th_{Max}, set at 40 packets. In contrast, CBT's latency is slightly lower than the configured 30ms of latency. This is because the class *other* is not using its allocated queue capacity. As a result, the average aggregate queue occupancy is lower than intended, leading to lower queue-induced latency. In this case, the effect is small as the threshold for *other* is only 4.3 packets. This corresponds to approximately 4 ms of latency. In contrast, CBQ's behavior does not change as multimedia is able to traverse the router quickly in all cases as a result of separate queues for each class. Once again, CBT offers better performance than any of the other AQM algorithms.



a) HTTP+Proshare

b) BULK+Proshare

c) HTTP+MPEG

d) BULK+MPEG

Figure 5.23 Multimedia Latency During the Multimedia Measurement Period

(Note: BULK and HTTP are on different time scales.)

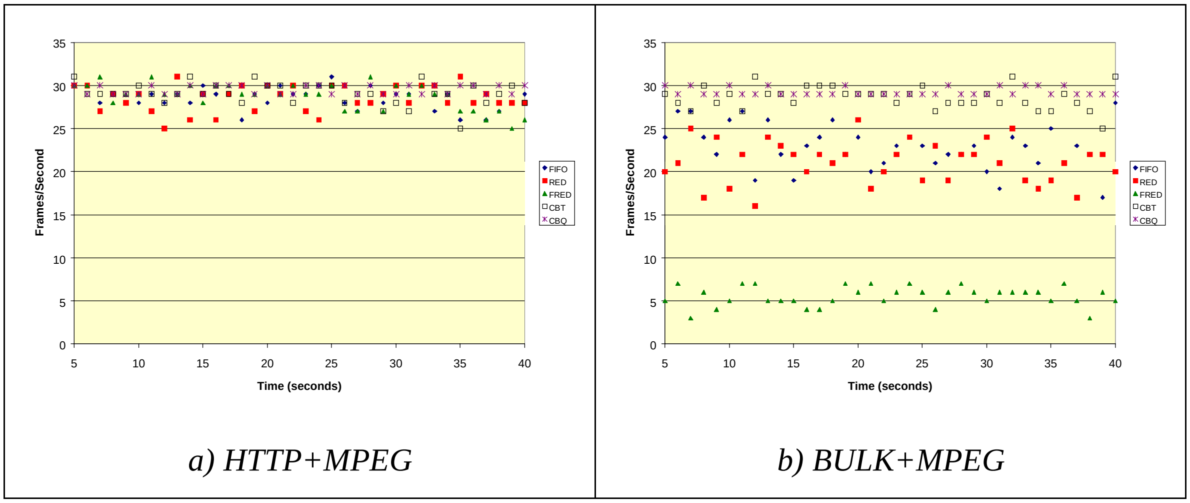
Loss and Frame-Rate

Finally, we consider the quality of the multimedia streams, first by considering the loss-rate and then examining the resulting frame-rates. Although there is an inverse relationship between loss-rate and frame-rate, as noted during the blast measurement period, that relationship is not strictly linear due to interframe encodings and biases in some algorithms. First, consider the loss-rate. Table 5.7 shows the loss-rates for each algorithm across all traffic mixes. The loss-rates are lower in all cases than the corresponding values during the blast measurement period. This is expected since TCP and multimedia are not competing for bandwidth with *other* traffic. However, for the AQM algorithms the loss-rate for traffic mixes using BULK is significantly higher than the loss-rate for traffic mixes that use HTTP. This is because the BULK traffic generates sufficient load to maintain a state of persistent congestion. Since RED and FIFO drop packets from all flows equally during periods of congestion, queues build and packets are discarded. And FRED again offers a higher drop-rate for multimedia as it constrains the multimedia flows to a fair share. With many flows FRED's fair share is much less than multimedia's offered load. In contrast, CBQ's and CBT's allocations result in a low loss-rate for multimedia.

Traffic Mix	**Multimedia Loss-rate (Packets/second)**				
	FIFO	**RED**	**FRED**	**CBT**	**CBQ**
HTTP+Proshare	.5	.9	.96	.7	0
HTTP+MPEG	1.4	.9	1.11	.62	0
BULK+Proshare	6.9	8.8	27.6	.91	0
BULK+MPEG	8.2	10.2	32.8	1.1	0

Table 5.7 Loss-Rates for All Algorithms during the Multimedia Measurement Period

These loss-rates have a clear impact on frame-rate. Given the observed loss-rates one expects to realize a high actual frame-rate for all queue management algorithms for the traffic mixes featuring HTTP. However, one expects more variation in frame-rate for the mixes featuring BULK since the loss-rate was high in those cases. The data supports this conclusion. Figure 5.25 shows the frame-rates for MPEG with HTTP (Figure 5.25*a*) and with BULK (Figure 5.25*b*). (Recall that the Proshare traffic generator was not instrumented to provide frame-rate data so only MPEG is considered.) When HTTP is the TCP type, all queue management algorithms lead to an actual frame-rate on the order of 30 frames per second. However, in the BULK case, CBT and CBQ's bandwidth allocation leads to frame-rates of 30 frames per second but the losses associated with RED and FIFO lead to frame-rates of 20-25 frames per second. Moreover, FRED's fair shares severely constrain the frame-rate, limiting MPEG to approximately 5 frames per second.



a) HTTP+MPEG

b) BULK+MPEG

Figure 5.25 Actual Frame-Rate Averaged Over 1 Second Intervals During the Multimedia Measurement Period

However, the actual frame-rate is merely a first approximation of the quality of the media stream. Since MPEG uses interframe encoding, a relatively high actual frame-rate can still lead to a low playable frame rate if the lost frames are the I-frames. Figure 5.26 illustrates this point. CBQ still maintains a frame-rate of approximately 30 frames per second and CBT is nearly that, with occasional dips (probably due to lost I-frames.) However, RED and FIFO degrade the frame-rate closer to 25 frames per second. FRED leads to more variation in the frame-rate due to its bias against the large I-frames.

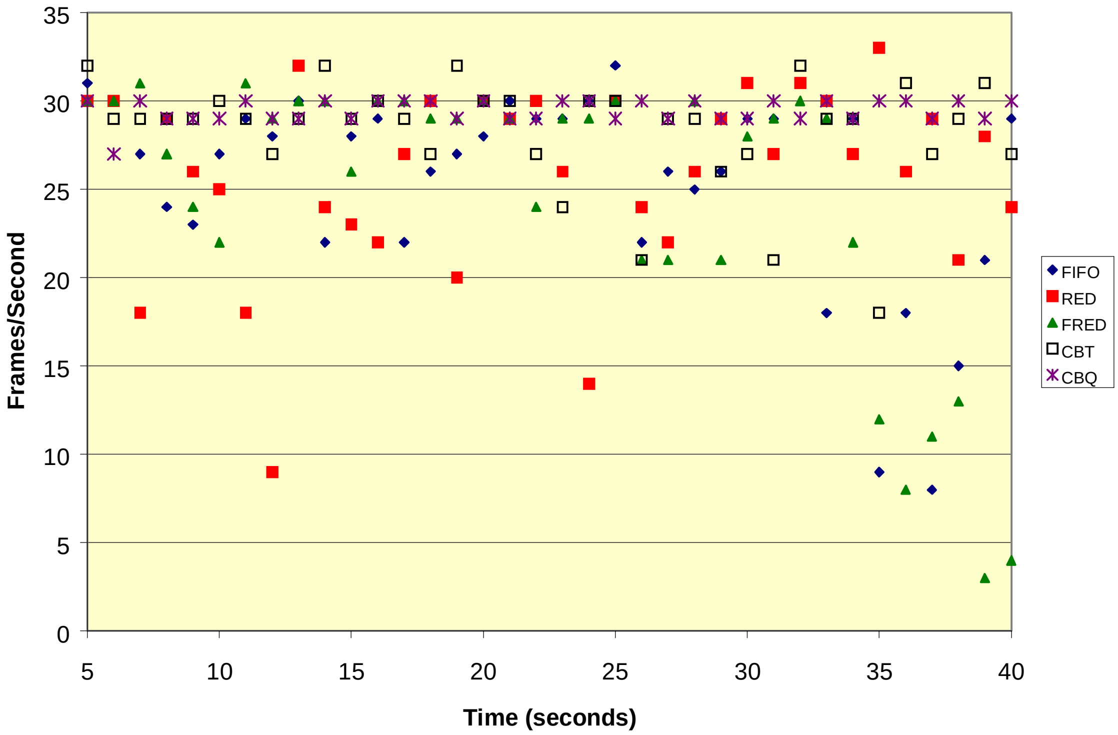
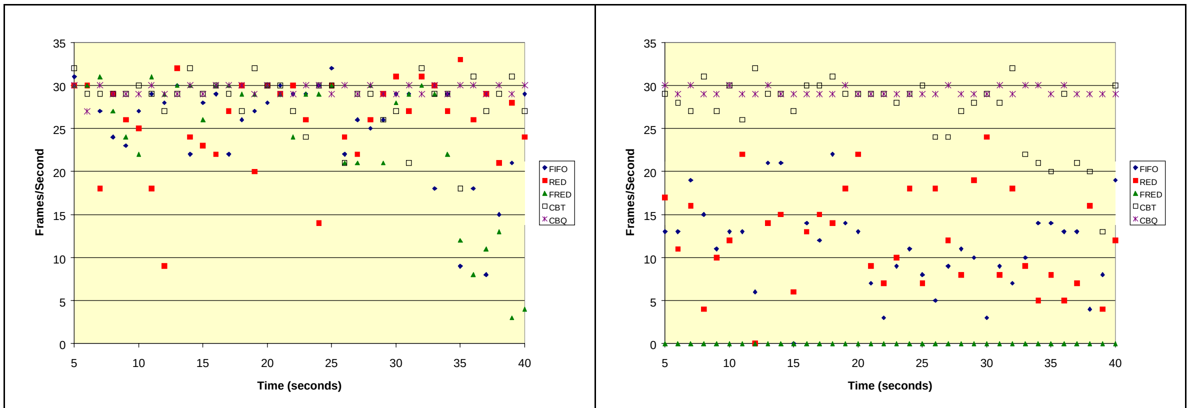


Figure 5.26 Playable Frame-Rate for HTTP+MPEG

The effect is even more extreme when BULK is present. Figure 5.27 shows MPEG's playable frame-rate for both HTTP (Figure 5.27*a*) and BULK (Figure 5.27*b*). Consider the BULK case. Once again, the bandwidth allocations offered by CBQ and CBT are effective and allow MPEG to maintain a higher frame-rate than with the other algorithms. CBQ' s frame-rate is consistently on the order of 30 frames per second while CBT averages 25 frames per second. However, the playable frame-rate with RED and FIFO degrades to on the order of 10 frames per second. Again, FRED offers the worst performance as the frame-rate drops to zero.



a) HTTP+MPEG

b) BULK+MPEG

Figure 5.27 Playable Frame-Rate During the Multimedia Measurement Period

2.2.3. Summary of Results for Multimedia Measurement Period

The measurements during the multimedia measurement period present two different scenarios. First, the measurements using HTTP as the TCP traffic type point out that all of the queue management algorithms are generally comparable during periods of low or transient congestion. TCP throughput, multimedia throughput, and loss-rates differ little between algorithms. However, since there is some congestion a queue does form and the latency does vary depending on the algorithm used. In that scenario, CBQ offers the minimal latency while CBT also offers predictable queue-induced latency. FIFO, RED, and FRED do constrain the latency as a function of their parameter settings as well but these parameter settings are selected to maximize metrics like TCP efficiency and that leads to higher queue-induced latency in these experiments.

The second scenario to consider is when BULK is the TCP traffic type. BULK flows are long lived and the oscillations of TCP allow the BULK traffic to maintain a higher level of congestion. Consequently, those experiments demonstrate the interaction between multimedia and TCP during periods of persistent congestion that are not caused by high bandwidth unresponsive flows. In this scenario, CBQ and CBT continued to perform well, with both mechanisms isolating multimedia from the effects of the high bandwidth TCP flows. All of the algorithms offer high TCP goodput with BULK. FRED's goodput is particularly high since the BULK flows represent almost all of the active flows. However, this leads to poor performance for multimedia throughput with FRED as each of those flows is constrained to a fair share that is less than the intended load for those multimedia flows. In contrast, the other algorithms are more comparable for multimedia throughput, although CBT or CBQ bandwidth allocation offers superior performance. Further, the latency measures underscore FRED's poor performance when the number of flows is large. FRED's multimedia latency is nearly four times that observed for any other algorithm. Loss-rate and frame-rates also make it very apparent that RED and FIFO fail to protect these relatively low bandwidth multimedia flows from the effects of high bandwidth TCP while FRED's per-flow fairness constraints and its bias against bursty arrivals directly limit the multimedia quality. However, CBT, like CBQ, offers low loss, low latency, and high frame-rates while maintaining high TCP goodput. CBT meets its design goal of providing better performance for multimedia without having negative impact on TCP.

3. Summary

To evaluate the queue management algorithms, the algorithms were empirically evaluated in a laboratory network using multiple mixes of TCP, multimedia, and *other* traffic. Each traffic type was instrumented and the algorithms were evaluated based on a variety of metrics including throughput for each traffic type as well as multimedia latency, loss, and frame-rate. Class-based thresholds (CBT) addresses the tension between responsive and unresponsive flows in the Internet by isolating TCP from the effects of aggressive unresponsive traffic while also offering improved performance for multimedia. CBT limits queue-induced latency and allocates sufficient bandwidth to multimedia traffic to insure that the applications maintain acceptable frame-rates. Moreover, CBT offers performance superior to that of the FIFO, RED, and FRED queue management mechanisms.   Specifically, FIFO and RED are unable to constrain unresponsive flows effectively allowing unresponsive flows to dominate the link as responsive flows reduce their load in response to packet drops. FRED constrains aggressive unresponsive flows but presents other problems for multimedia. FRED's most effective parameter settings for TCP performance can result in high queue-induced latency. Further, when each flow receives a fair share of the queue, the resulting per-flow throughput can be insufficient to maintain the desired frame-rate for multimedia streams.   Finally, CBT compares favorably to a representative packet scheduling mechanism, Class-Based Queueing (CBQ) and demonstrates that bandwidth can be allocated effectively using queue management.

VI. SUMMARY AND CONCLUSION

A fundamental tension exists between responsive and unresponsive flows in the Internet today. Responsive flows respond to congestion by reducing the load they generate. The cooperative behavior of many responsive flows alleviates congestion by reducing the aggregate load to less than the capacity of the congested link. In contrast, unresponsive flows ignore congestion and maintain their load. Consequently, when both types of flows share a congested link, responsive flows suffer because of their responsive behavior. Unresponsive flows maintain their load and are able to use a disproportionately large fraction of the link's capacity. Most of the traffic in the current Internet is responsive because it uses a responsive transport protocol, TCP. Most applications use TCP because it offers a reliable ordered byte stream. However, many classes of applications, such as interactive multimedia, have no compelling reason to use TCP as their transport protocol because reliability isn't a primary concern. Moreover, the mechanisms used to provide reliable delivery have a negative impact on performance, making TCP undesirable for those applications. Responsiveness should be encouraged but applications that are not well suited to TCP should also be supported. This dissertation addresses the problem of supporting and encouraging responsive behavior while still providing reasonable performance for applications like multimedia that aren't well suited to TCP.

Current approaches to this problem span many layers of the network stack. At the application layer, work has focused on allowing the application to respond to congestion by using temporal and spatial scaling adaptations to adjust the bit-rate or packet-rate [Delgrossi93],[Hof93],[Talley97]. At the transport layer, new transport protocols have been proposed. These protocols are responsive but unreliable and, thus, do not carry the overhead associated with providing reliability [Cen98], [Sisalem98]. However, all of these approaches require changing or rewriting existing applications. They do nothing to address the applications that are already deployed and in use. Supporting these existing applications requires a network-centric approach. Moreover, making applications responsive would leave them vulnerable to the effects of other unresponsive flows. Limiting the effects of other unresponsive traffic can only be accomplished with a network-centric approach. Most of the current work at this level has focused on packet scheduling. Another approach is *active queue management* (AQM) techniques that adjust the way in which a router manages the queue of packets that builds during periods of congestion. Most AQM approaches focus on providing better support for responsive flows. Specifically, they identify and aggressively constrain unresponsive flows through packet drops with the intention of encouraging application and protocol designers to use responsive techniques [Floyd98], [Braden98]. This leads to poor performance for unresponsive flows, including interactive multimedia.

The goal of this dissertation is to demonstrate that AQM techniques can be used to isolate TCP and multimedia traffic both from one another and from the effects of other unresponsive traffic. Specifically, classifying traffic by type (TCP, multimedia, *other*) and setting proper per-class thresholds on the packet drop decision can result in limiting the queue induced latency and provisioning of link bandwidth similar to the use of weights with a scheduling mechanism, but without the complexity of implementing a scheduler. This work makes four contributions toward this goal:

1. We designed and implemented a novel AQM algorithm called Class-Based Threshold (CBT) to address the tension between responsive and unresponsive flows by offering bandwidth allocation via buffer allocation.

2. We have shown that CBT can be tuned to offer prescribed levels of performance for each traffic class. We presented analysis that predicts network performance when using CBT based on initial configuration parameters, explained how this analysis can be used to derive optimal parameter settings given desired network performance, and empirically demonstrated the accuracy of this analysis.

3. We have presented data that contributed to our understanding of how existing AQM schemes work. We examined the empirical behavior of many current AQM

algorithms as well as one packet scheduling algorithm across a wide range of parameter settings and traffic mixes, articulated relationships between parameters and performance metrics, and evaluated the results to determine optimal parameter settings for each algorithm and traffic mix.

4. We empirically demonstrated that CBT effectively isolates TCP while providing better-than-best-effort service for multimedia. We compared CBT's performance to the optimal performance for each of the other algorithms. We showed CBT provided better protection for TCP than RED and FIFO and better multimedia performance than RED, FIFO, and FRED.

The rest of this chapter discusses each of these contributions and then presents some possibilities for future work in this area.

1. Class-Based Thresholds

Class-based thresholds is a novel algorithm for active queue management which effectively isolates classes of flows and limits queue-induced latency. Limiting the average queue occupancy controls queue induced latency. Classes of traffic are isolated from one another by classifying packets as they arrive and managing each class's queue occupancy independently. Since all classes share a single FIFO queue, each class's share of the queue equates to its share of the capacity of the outbound link.

Although the support for classes of traffic is general, the implementation demonstrated here is configured for three classes of traffic, TCP, multimedia, and *other*. The average queue occupancy for each class is managed by applying the RED algorithm to each class [Floyd93]. The maximum threshold value for each class equates to a loose upper limit on that class's average queue occupancy. Consequently, the ratio between the maximum thresholds for each class corresponds to the ratio between maximum average queue occupancy for each class. As a result, when all classes are active and maintaining their maximum queue occupancy the ratio between these thresholds is also the ratio between the bandwidth each class is able to consume on the congested link.

Using queue management to allocate bandwidth to each class of traffic isolates the classes from one another. For example, if *other* exceeds its queue allocation then packets from class *other* will be discarded. However, as long as TCP and multimedia are within their allocations, they will suffer no drops and receive their allocated share of the outbound link. Moreover, the aggregate of the classes' thresholds divided by the capacity of the outbound link equates to the queue-induced latency so the threshold settings determine an upper limit on the average latency.

2. Analyzing CBT

CBT offers predictable performance. The performance of different classes of traffic interacting with CBT can be computed using equations derived here. Performance depends on each class's threshold on queue occupancy, link capacity, average packet sizes, and load generated by each class. The most straightforward case is the worst case analysis. Assume that each class is generating load sufficient to maintain average queue occupancy equal to that class's threshold (i.e., load exceeds bandwidth allocation during a period of congestion). In that case the expected queue-induced latency is the sum of the products of each class's average packet size and threshold divided by the link capacity as shown in equation 6.1.

$$L = \frac{\sum_{i}^{n} Th_i P_i}{C} \tag{6.1}$$

Similarly, each class's throughput is a fraction of the link capacity equal to the product of that class's threshold and average packet size divided by the product of each class's threshold and average packet size summed across all classes as shown in equation 6.2.

$$B_j = \frac{Th_j P_j}{\sum_{i}^{n} Th_i P_i} C \tag{6.2}$$

These equations may be inverted to derive optimal threshold settings based on desired bandwidth allocations and limits on latency.

Moreover, CBT's performance can also be predicted when some classes are using less than their allocated bandwidth. Queue induced latency decreases when some class is not using its full queue allocation. Using the analysis presented in this dissertation, one can determine the expected latency when classes of traffic are operating with loads below their bandwidth allocation. Moreover, this analysis explains the limits on how much each class can increase their throughput when there is excess capacity available. Essentially, classes borrow the unused capacity of other classes. This borrowing is shown to be implicit in the design of the algorithm. The limits on average queue occupancy do not change. Rather, borrowing is the result of other classes having an increased share of the reduced queue occupancy when some classes do not use their full allocation. Classes with loads greater than their allocation essentially can borrow unused capacity in proportion to their initial allocations.

Using the analysis developed here, one may compute the latency and the throughput for each class as a function of each class's load and the initial parameter settings for CBT. The accuracy of this analysis was confirmed empirically by ranging the parameter settings and traffic mixes and comparing the resulting performance to that predicted using the derived functions.

3. Empirical Analysis of Algorithms

Four queue management algorithms, FIFO, RED, FRED, and CBT were empirically evaluated to determine optimal parameter settings across different traffic mixes. A packet scheduling algorithm, class based queueing (CBQ), was also evaluated as a baseline to compare with the AQM techniques. Each algorithm was evaluated based on the initial design goals of that algorithm. A range of queue sizes was considered for FIFO with TCP goodput, link utilization, and queue-induced latency as key metrics. The primary result found was a linear relationship between queue size and queue-induced latency. However, goodput and efficiency did decline for very small queue sizes. Consequently a moderate queue size (60 packets) was selected for FIFO.

RED was evaluated using many of the same metrics as FIFO as the values of the maximum and minimum threshold values were varied. Many relationships were confirmed. First, queue-induced latency was linearly related to the maximum threshold size because the maximum threshold places an upper limit on average queue occupancy. TCP goodput and efficiency were also related to the maximum threshold value, with particularly poor performance for threshold settings less than 40 packets. Since efficiency and goodput should be maximized while latency should be minimized, a maximum threshold of 40 packets was indicated. In contrast, there was no apparent relationship between the minimum threshold setting and any of the metrics. Consequently, the original recommendation of the RED designers, a minimum threshold of 5, was used.

Like RED, FRED was evaluated as the values of the maximum and minimum threshold values were varied. FRED was evaluated using most of the same metrics as RED and FIFO but a greater emphasis was placed on FRED's ability to constrain aggressive, unresponsive flows (i.e., *other*). The relationships between the threshold values and their resulting effects were complex. Although the maximum threshold limited the average queue-induced latency during periods with moderate numbers of active flows, it had no effect when the number of active flows approached the maximum threshold. Since every flow was allowed to enqueue two packets, the probabilistic and forced-drop mechanisms were short circuited and only the actual queue size limited occupancy. Although the queue induced latency was not limited with many flows, performance with a smaller number of flows argued for the use of a small maximum threshold to limit average queue induced latency in that case. Additionally, the ratio between the maximum and minimum thresholds was found to be important. This ratio determined the fraction of the queue each flow might receive. To minimize the throughput of a misbehaving flow the ratio should be large. However, this conflicts with the desire to minimize latency by using a small value for the maximum threshold. Moreover, extremely small values of the minimum threshold over-constrain individual flows, including TCP, leading to poor efficiency. Consequently, threshold values were selected to balance these concerns, using a maximum threshold of 60 packets to limit queue induced latency and a minimum threshold of 5 to allow mildly bursty flows to operate without drops during periods of moderate congestion while still limiting each flow to a small share of the queue.

The evaluation of CBT and CBQ was more straightforward because the relationship between the parameters for those algorithms and the resulting performance is more clearly defined. Consequently, the expected optimal parameter settings were calculated with the goal of constraining *other* traffic to a small fraction of the link, assuring multimedia had sufficient allocation for high fidelity media, and allocating the remaining (substantial) fraction of the bandwidth to TCP. The resulting settings for each algorithm were evaluated relative to other settings that varied the allocations for multimedia and TCP. This analysis led to the observation that using long term average throughput as an estimate of the necessary bandwidth allocation was insufficient. There were periods of loss for multimedia for both algorithms when using the expected optimal parameters. Because of this, the parameters had to be tuned to base multimedia's allocation on the maximum short-term average load generated by multimedia. With that adjustment, both algorithms were configured to offer low latency, low loss and high fidelity for multimedia, and good throughput and efficiency for TCP.

4. Empirical Comparison of Algorithms

Finally, CBT was compared to FIFO, RED, and FRED as well as a packet scheduling policy (CBQ) to empirically demonstrate CBT's effectiveness addressing the tension between responsive and unresponsive flows. Using a private laboratory network and traffic generators the algorithms were compared with a variety of traffic mixes and metrics. Two key measurement periods were identified. During the blast measurement period the algorithms operated in the presence of high bandwidth TCP and multimedia traffic combined with aggressive, unresponsive traffic. During the multimedia measurement period the algorithms operated in the presence of only the TCP and multimedia traffic. The hypothesis was that when compared to other AQM algorithms during the blast measurement period, CBT would offer clearly superior performance as determined by a combination of metrics for the TCP, multimedia, and *other* traffic. CBT should also offer performance at least as good as the other AQM techniques during the multimedia measurement period. Further, CBT was expected to compare favorably with the performance of the packet scheduling approach, CBQ. All these hypotheses were confirmed.

4.1. Blast Measurement Period

Results from the blast measurement period confirmed the vulnerability of FIFO and RED to aggressive, unresponsive flows. TCP goodput was low, *other* dominated the link's capacity, and multimedia throughput and thus loss-rate and playable frame-rate were poor. Although queue-induced latency was tolerable at 60 and 40 ms, it mattered little because the high loss-rate severely degraded the multimedia streams.

In contrast, FRED offers good TCP performance and effectively constrained *other* but the mechanism used to constrain *other* also overconstrained multimedia. In three of the four traffic mixes examined, the overconstraining resulted in high loss for multimedia. Moreover, FRED's use of per-flow instantaneous queue occupancy in the drop decision leads to a bias against bursty packet arrivals. Consequently, MPEG's large I-frames are very likely to be subject to loss and, consequently, result in a poor playable frame-rate even when the loss-rate is low. Further, when the number of active flows approaches the size of the queue, FRED's performance equates to that of FIFO, leading to a high drop-rate and high queue-induced latency.

In contrast, and as expected, CBQ offers the best performance. The bandwidth allocations accurately indicate the throughput each class of traffic actually receives. When properly configured, TCP goodput is high, *other* is effectively constrained, and there is no multimedia loss which results in a very playable high frame-rate. However, during periods of brief overload the queue-induced latency can vary significantly.

Finally, Class-based Thresholds also offers good performance although the bandwidth allocations less precisely control the resulting throughput. When properly configured, TCP goodput is high, *other* is effectively constrained, and multimedia loss is low, resulting in a high playable frame-rate. Moreover, queue-induced latency is predictable and configurable.

Performance during the blast measurement period demonstrates that CBT meets its design goals, isolating TCP and multimedia from each other and from the effects of aggressive, unresponsive flows. Moreover, CBT performs better than the other AQM algorithms examined and the performance is comparable to that of CBQ, the packet scheduling mechanism.

4.2. Multimedia Measurement Period

The experiments during the multimedia measurement period present two different scenarios. First, the measurements using HTTP as the TCP traffic-type point out that all of the algorithms are generally comparable during periods of low or transient congestion. TCP throughput, multimedia throughput, and loss-rates differ little due to algorithm choice. However, since there is some congestion, a queue does form and the latency varies depending on the algorithm used. In that scenario, CBQ offers the minimal latency while CBT also offers predictable queue-induced latency. FIFO, RED, and FRED do constrain the latency as a function of their parameter settings as well but their parameter settings are selected to maximize metrics like TCP efficiency and that leads to higher queue-induced latency in these experiments.

The second scenario to consider is when BULK is the TCP traffic type. BULK flows are long lived and numerous so the oscillations of TCP allow the BULK traffic to maintain a higher level of congestion. Consequently, those experiments demonstrate the interaction between multimedia and TCP during periods of persistent congestion which are not due to high bandwidth unresponsive flows (i.e., *other*). In this scenario, CBQ and CBT continued to perform well, with both mechanisms isolating multimedia from the effects of the high bandwidth TCP flows. All of the algorithms offer high TCP goodput for BULK traffic. FRED's goodput is particularly high since the BULK flows represent almost all of the active flows. However, this leads to poor performance for multimedia throughput with FRED as each of those flows is constrained to a fair share that is less than the intended load for those multimedia flows. In contrast, the other algorithms are more comparable for multimedia throughput, although CBT and CBQ's bandwidth allocation offers superior performance. Further, the latency measures underscore FRED's poor performance when the number of flows is large. FRED's multimedia latency is nearly four times that observed for any other algorithm. Loss-rate and frame-rates also make it very apparent that RED and FIFO fail to protect these relatively low bandwidth multimedia flows from the effects of high bandwidth TCP flows while FRED's per flow fairness constraints and its bias against bursty arrivals directly limit the multimedia quality. However, CBT, like CBQ, offers low loss, low latency, and high frame-rates while maintaining high TCP goodput. CBT meets its design goal of providing better performance for multimedia without having a negative impact on TCP.

4.3. Summary

These experiments empirically demonstrated that CBT effectively isolates TCP while providing better-than-best-effort service for multimedia by comparing CBT's performance to the optimal performance for each of the other algorithms.

5. Future Work

Although this work confirmed the thesis and made important contributions, there are several opportunities for future work. This section covers a few of the possible directions to pursue for future work.

5.1. Limitations and Suggested Refinements

The analysis conducted as part of this dissertation identified some limitations of the CBT algorithm. In some cases there are straightforward refinements to the algorithm that should address these issues. For others, further study would be required to determine the best approach to the problem. Both types of limitations are presented below, along with proposed solutions or approaches.

5.1.1. Imprecision in CBT

Although, CBT's bandwidth allocations, latency limits, and resulting performance can be predicted at a coarse level, the predictions (and allocations) are imprecise. This imprecision arises from several sources:

1. Reliance on accurate predictions of average packet sizes.

2. Disparities in the sampling rates for different classes.

3. Lack of guidelines for assigning weights for each class.

4. Assuming that all classes, particularly TCP, will be able to maintain average queue occupancy equal to the maximum threshold.

Each of these sources of imprecision are addressed below.

Accurate Predictions of Average Packet Size

If actual average packet sizes vary from those used when calculating thresholds, the actual performance may vary from the intended allocation. Recall that threshold settings are calculated as the product of the desired bandwidth allocation and latency divided by the average packet size. The product of the desired bandwidth and latency determines the threshold on the number of *bytes* that the class should have in the queue. Dividing by the average packet size converts this limit from units of bytes to packets. If the average packet size for a given class is inaccurate, this can change the share of the queue and, thus, the outbound link, allocated to that class. Consider an example. Suppose three classes each have thresholds of 10 packets and an average packet size of 500 bytes is assumed for all classes. Further, assume all classes are capable of consuming the full capacity of the outbound link. If the average packet size predictions are accurate, each class can have 5,000 bytes enqueued on average, each occupying one third of the queue, and thus receiving throughput equal to one third of the capacity of the outbound link. However, if the average packet size of one class was actually 1,000 bytes that class could have 10,000 bytes enqueued to the 5,000 bytes of the other classes, allowing that class to use half of the link's capacity. Similarly, a class would receive less than its bandwidth allocation if the average packet size were actually smaller than the original estimate. The solution to this problem would be simply to set thresholds in terms of the average bytes enqueued rather than packets.

Prior AQM algorithms typically set thresholds in units of packets instead of bytes because it simplified management of the average. Each packet arrival or departure resulted in a simple increment or decrement. Byte measurements require examining the packet length. In prior AQM algorithms, both approaches yield essentially the same results as long as the average packet size has little variation [Floyd97b]. However, for CBT, where different classes have different thresholds, errors in the estimated average packet size effectively change the thresholds on queue occupancy and, thus, the bandwidth allocations. Fortunately, modifying the algorithm to use bytes instead of packets is straightforward. Tracking the average number of bytes simply requires examining at an additional field (i.e., length) of the packet at enqueue and dequeue and, respectively, adding or subtracting that packet's length to a byte count instead of simply incrementing or decrementing the packet count. With this change, the threshold value can be computed as the simple product of the bandwidth allocation and the desired latency. This change eliminates the need to estimate the average packet size and removes one source of imprecision.

Disparities in Sampling Rates

Disparities in sampling rates for each class may make the classes' averages incomparable. In CBT multiple averages are maintained. The ratio between these averages determines the ratios between throughput for each class. Since these averages are used to make drop decisions and are intended to determine the ratio of queue occupancy between the classes, it is important that they be comparable to one another. That is, they should represent the average behavior over the same interval of time, or at least an interval of the same magnitude. However, because CBT extended the averaging mechanism from RED, this is not the case in the current implementation. In CBT, like RED, each class's average occupancy is sampled and updated whenever a packet of that class arrives. Consequently the sampling rates for each class differ as the arrival rates for each class differ. Recall that the averages are weighted moving averages. Consider two classes, one with an arrival rate of 5 packets per second and another with an arrival rate of 40 packets per second. This means the class with the low rate of packet arrivals may still be strongly influenced by the queue occupancy from 1 second ago. In contrast, the class with the high packet arrival rate will give a sample from 1 second ago 35 orders of magnitude less weight. (The sample from 1 second ago will have been multiplied by (1-*w*) 35 times.) This problem was also identified by others and named "unsynchronized weighted-average updates" [Chung00]. This leads to concerns that considering ratios between these averages has little meaning with regard to the ratio of throughput for each class. The fact that the empirical analysis yields results that correspond to computed values seems to indicate that, in the long term, the effect is negligible. However, it would be worthwhile to further consider this issue and alternatives to more effectively synchronize these averages.

One alternative approach would be to maintain the multiple averages and update the average for each class whenever a packet arrives for any class. The strength of this approach is that the averages all represent the same period. However, taking multiple samples of a queue size that isn't changing (because another class is having rapid packet arrivals) may allow recent behavior to unfairly influence a class's average. For example, consider a situation where the multimedia class was idle for a long time, then a burst of 3 multimedia packets arrived to a large queue. Assume those packets were successfully enqueued (due to a small average queue occupancy) and were followed by a burst of 100 *other* packets that were all dropped during the time it takes the 3 multimedia packets to traverse the queue. If we sampled all classes on every packet arrival, the average for multimedia with typical weights would probably reach 3 just because of that one burst, possibly leading to drops of subsequent multimedia packets that arrive. This behavior is also undesirable. Resolving this issue will require further study.

Guidelines for Assigning Weights

Guidelines for determining the weighting factors for each class are needed. The weighting factor used to factor each new sample into the average helps determine each class's sensitivity to short term behaviors. In the current implementation, different weighting factors are used for each class. *Other's* weighting factor is high, with each new sampling contributing 25% to the new average. In contrast, the weighting factor for TCP is low, 0.4%. The intent was to strictly constrain *other* while allowing TCP to have more variable behavior. However, this may also make the averages incomparable. Further analysis of the effects of these weighting factors is in order.

Assuming Classes' Averages Match the Maximum Threshold

When classes don't maintain average queue occupancy equal to the maximum threshold they don't receive their fair share of link bandwidth. This is a particularly a problem for responsive flows as they often find a stable operating point that maintains average queue occupancy less than the maximum threshold. In the empirical analysis, TCP traffic, especially HTTP, frequently failed to achieve throughput equal to the intended bandwidth allocation. This was because RED's probabilistic drop mode interacts with the responsiveness of TCP. These two mechanisms lead to the aggregate TCP load converging to an operating point that maintains an average queue occupancy in the range of the probabilistic drop mode but less than the maximum threshold. Since TCP doesn't use its full share of the queue (i.e., the maximum threshold), it doesn't receive its full share of the link capacity either.

There are several approaches to address this problem. One approach would be to accurately articulate the relationship between the threshold settings and the resulting average queue occupancy for TCP. However, experience indicates the average queue occupancy is also determined by the traffic type. While average occupancy with HTTP was below the maximum threshold, maximum occupancy was maintained with BULK. Consequently, a better approach might be to dynamically adjust the thresholds for *other* and multimedia relative to the average TCP occupancy. Of course, such a mechanism would have to be able to recognize situations where TCP is simply using less than the intended bandwidth allocation and not adjust the other thresholds in that case. However, this approach may lead to more accurate bandwidth allocations for TCP.

5.1.2. Accurately Predicting Bandwidth Needs

Bandwidth allocations with CBT are only effective if the needs of the traffic classes can be predicted accurately. This problem is not unique to CBT. It is a problem for all resource allocation schemes (e.g., CBT and CBQ). In this work, when the optimal parameter settings were being selected, it became apparent that using the expected average load to predict the necessary bandwidth allocation was insufficient. Brief loads in excess of the average resulted in poor latency or increased drops for multimedia. The simplest solution in the context of the work presented here was to determine the peak load for each class and allocate accordingly. In a more complete framework, policing and shaping agents may be responsible for managing classes of flows to assure they conform to the negotiated bandwidth allocations. Another alternative may be to dynamically update the threshold settings based on current traffic patterns [Chung00]. However, this approach needs further refinement if minimum levels of service are to be assured.

5.2. Further Analysis

In addition to considering changes to the algorithm there are also opportunities for further analysis of both the CBT algorithm (current or modified) and the other algorithms considered in this dissertation. Although the complexity and overhead of the algorithms are considered briefly in this work, it would be worthwhile to more thoroughly analyze the complexity and overhead associated with each algorithm and the impact of that overhead on traffic performance. Additionally, it would be interesting to consider the behavior of each algorithm under more realistic network conditions, both more complex test networks, actually deploying algorithms in a production environment, and considering other traffic patterns that may point out strengths and weaknesses of the algorithms. In this section we briefly consider each of these areas of analysis.

5.2.1. Complexity and Overhead

Minimizing packet processing overhead is a serious concern for router manufacturers. If packet-processing overhead becomes the dominant factor in the time required to forward packets it is possible that the router will not be able to forward packets fast enough to maintain full link utilization. Computing the average(s) for RED, classifying flows for FRED, examining additional packet fields for CBT, or maintaining a schedule for CBQ all increase the overhead associated with processing each packet. In addition to analyzing the theoretical complexity of each algorithm, it would be enlightening to instrument the algorithms to record the number of cycles required for different subroutines of each algorithm to quantify the effect each subroutine has on packet processing and end-system performance. Once instrumented, the algorithms could be tested with both representative loads and loads designed specifically to test the limits of the algorithms. For example, generating high-bandwidth traffic with minimum size data payloads would emphasize the effect of the packet-processing overhead. Similarly, generating a very large number of simultaneous flows would test the limits of flow classification for FRED.

Additionally, it is important to limit the amount of state that must be maintained because cache memory is expensive. Moreover, as memory requirements grow, the access time may grow as cache sizes are exceeded, effecting the packet-processing overhead.

Therefore, it would be worthwhile to analyze the amount of state and buffer space required for each algorithm under different conditions.

5.2.2. More Realistic Conditions

In this dissertation, time and space considerations limited the variety of conditions used to evaluate each algorithm. The traffic mixes used here were intentionally simplified to limit the number of independent variables in each experiment. Moreover, the traffic patterns tended to be extreme in terms of load or having relatively constant bit-rate. Further, the network topology was the simple "dumbbell model", using two networks joined by one link of limited capacity. All of these limitations were necessary in the initial analysis in order to determine relationships between specific factors and performance. However, considering other combinations of traffic and network topologies may reveal additional issues with the behavior of the algorithms. Future analysis may consider networks with more complex topologies. For example, the effects of having multiple routers in sequence running the same or different queue management algorithms may be considered to determine the results of flows experiencing drops at multiple consecutive routers.

Another possibility may be to use traffic loads that are much more variable in terms of load, packet size, arrival rates, and combinations of traffic types. All of these factors may effect how the algorithm behaves. For example, we have already shown that for CBT and CBQ it is important that the bandwidth allocations represent the largest short-term average load, not the long-term average load. Additionally, variability in packet arrival rate may effect the behavior of the algorithm, as with FRED's biased behavior towards bursty flows. Other sensitivities may be discovered with more experimentation. Another possible variation may distribute the load for class *other* across a large number of flows. The hypothesis would be that FRED would fail to constrain *other* in this scenario because each flow would be individually allowed to maintain its fair share, leading to a large aggregate load for other. Finally, after extensive testing in the laboratory environments, the logical next step would be to deploy CBT in a production environment, perhaps the router connecting a campus LAN to an ISP.

5.3. Deployment Issues

Another avenue for future work involves addressing the issues necessary to deploy CBT in production networks. In this work, many simplifying assumptions were made as the focus was on analyzing CBT's effectiveness for bandwidth allocation and latency management. Robustness and flexibility were secondary concerns. For example, packets were classified based on a combination of protocol identifier and destination port. All TCP packets were classified as TCP. All UDP packets with destination port addresses in the range 6000-6010 were classified as multimedia. And, all other packets were in the class *other*. In a true production environment, the classification process must be much more flexible. If the bandwidth allocations are intended to be adjustable based on newly negotiated service profiles, mechanisms for receiving these adjustments must also be considered. All of these issues are considered below.

5.3.1. Packet Classification

The primary requirements for packet classification are accuracy and speed. Additionally, flexibility in defining how to classify packets is a highly desirable feature. The options for packet classification vary extensively. Classification may be done with a filter based on addressing information or by using a heuristic to determine the packet's class based on the recent behavior of the flow [Floyd98]. Alternatively, classification might be done based on tag bits in the packet header that are set at the end-systems and/or ingress and egress routers for a given ISP. Packet classification is relevant to many approaches to quality of service and is the subject of active research. The strengths and weaknesses of several approaches are highlighted below.

Classification based on address filtering is a common approach. Recent results show such classification can be done in $O(\log_2(\text{address bits}))$ time [Waldvogel97]. However, during periods of congestion this time does not effect end-to-end latency as long as the packet processing time is less than the time required for transmitting a minimum size packet on the outbound link. Since transmission of prior packets happens in parallel with the classification of arriving packets, if the time to transmit the previous packet is higher, then that determines the latency incurred. Although $O(\log_2 n)$ is very efficient, it is still an additional overhead for each packet during periods without congestion. Moreover, during these periods, the contribution of classification to end to end latency is O(m $\times$ $\log_2$(address bits)) where m is the number of routers that classify packets. This address classification approach also requires a significant amount of state as each address to be filtered against must be stored in each router that the packets might pass through. The benefit of this approach is that packets are classified on a per flow basis allowing more precise control and the classification is controlled locally.

In contrast, some approaches seek to push the task of packet classification towards the edges of the network, classifying and then tagging packets at the end-systems or at ingress routers. This approaches offers two major benefits. First, the number of times a packet must be classified is reduced. The number of classifications may be reduced to one, if the first classification is trusted, or possibly to once per autonomous system (e.g., an ISP) traversed. Second, because ingress routers support a smaller number of flows than core backbone routers the amount of state that must be maintained is also reduced. However, this approach requires a level of trust, with down stream routers trusting the classification of packets from upstream sources. This can be addressed in part by verifying the classification of packets at the ingress point for each autonomous system. Another issue is that tagging packets uses bits in the packet header. The number of bits available to encode the class tag limits the number of classes. As a result, it is not reasonable to have a large number of classes with this approach. However, it is reasonable to encode a small number of classes, such as the three used by basic CBT. These three classes could easily be specified in the type-of-service field in the IP header. The differentiated services architecture specifies that six bits of that field are available to specify different service levels, offering potentially sixty-four service levels and three of those could be the services outlined her for CBT [Nichols98].

Another approach that may have value for CBT, is classification based on responsiveness as determined by examining a flow's recent drop history [Floyd98]. This approach keeps track of the recent drop history for each active flow (i.e., flows with packets enqueued) and classifies packets as TCP-friendly (i.e., responsive in a TCP like manner), unresponsive, or high-bandwidth. This classification determines a flow's characteristics by examining that flow's recent drop history. The flow's actual arrival rate can then be compared to the expected arrival rate for a TCP flow under the same drop conditions to determine if the flow is responsive or not. Moreover, the flow's arrival rate can be compared to the aggregate arrival rate at the queue to determine if the flow is "high-bandwidth". To integrate this approach with CBT, all flows could be initially classified as TCP and after an interval in which they are subject to the RED drop policy, be reclassified based on their drop history if they are unresponsive. This approach has the benefit of actually classifying flows based on their behavior and not on other information, such as protocol identifiers, that may be spoofed. However, it requires state be maintained for the flows in each router. Moreover, short-lived flows may terminate before reclassification occurs.

Packet classification remains an open area of research, with several promising approaches. Before CBT can be widely deployed one approach must be selected for use. Moreover, this section addressed the mechanics of how to classify packets within the router. The administrative issue of how to decide which flows belong to which classes (in an address-based classification) is also an open question.

5.3.2. Negotiating Allocations

For CBT to be deployed and widely used there will also have to be some mechanism to negotiate bandwidth allocations. These topics, reservation protocols and admission control, are areas of active research. Approaches range from dynamic, short-term, per flow reservations, to static, long-term, allocations for each of a small number of classes. For CBT, a differentiated services type of approach may be best [Nichols97]. Rather than negotiate short-term per flow reservations at each router, administrators of autonomous systems (AS) may negotiate long-term allocations with neighboring AS administrators. These administrators may then allow their clients to request allocations either in the short or long term, based on current allocations of the bandwidth under contract with the next AS to be traversed. Each AS may be responsible for shaping traffic at egress points and policing traffic (using CBT) at ingress points. This work still must address issues such as dynamic routing and balancing resource utilization against insuring allocations meet peak demands.

6. Summary

This dissertation demonstrates that an active queue management algorithm, class-based thresholds (CBT), can effectively isolate responsive traffic from the effects of unresponsive traffic. Moreover, CBT also isolates multimedia from the effects of other unresponsive traffic. Further, analysis shows that CBT can be configured to allocate bandwidth and manage latency and that the performance under varying loads is predictable. Additionally, optimal parameter settings are determined for a collection of other algorithms and in accomplishing this goal, relationships between parameters and performance metrics are identified and articulated. Using these optimal settings, the algorithms are empirically evaluated and CBT's superiority confirmed. Finally, potential areas for future work are identified and discussed.

APPENDIX A. METHODOLOGY

This dissertation empirically compares the behavior of CBT to other algorithms under the same network conditions. Different parameter settings are also compared for a given algorithm. This appendix covers the details of the experimental methodology used in conducting these experiments. First is an explanation of the decision to use a private network for the measurements instead of simulation or a production network. Then the network configuration is discussed. Section 3 describes the traffic types used and how they are generated. Section 4 details the specific ways traffic is mixed for different experiments. Section 5 describes the data collection techniques. Section 6 addresses statistical issues: reproducibility, representativeness, and comparability. Section 7 concludes with a discussion of the specific metrics derived, what they mean, and how they are presented.

1. Experimental Model

Experiments described herein were performed on a private laboratory network. The network is configured to offer a single congested link. This simple model is acceptable because the experiments are not intended to be representative of the day to day behavior of the Internet. Instead, they are intended to test the effects of different active queue management (AQM) algorithms and parameter settings under controlled network conditions on different types of TCP and multimedia traffic. Examining the results of these controlled experiments gives better understanding of the algorithms and their effects of their parameters under different network conditions.

1.1. Alternatives for Experimentation

In preparing for the empirical evaluation of the AQM algorithms and parameter settings several alternatives were considered for the evaluation methodology. These included simulation, production networks, and a private experimental network configuration. Each alternative has strengths and weaknesses. Below, the alternatives for experimental environment are discussed and the reasoning for choosing to use a private laboratory network explained.

Since production networks are the location one would ultimately deploy these algorithms, experiments in production networks offer the most realistic conditions for evaluation. However, experimenting in production networks presents several problems. First among these is reproducibility. In production networks the traffic loads vary widely and unpredictably. As a result, it is difficult to compare different algorithms against the same network conditions in a production network. Moreover, it is impossible to conduct controlled experiments if conditions can not be reproduced. This work seeks to observe the behavior of the algorithms under specific pathological states. It is difficult and undesirable to establish such conditions in a production network. Finally, it is impractical to deploy a variety of experimental algorithms and parameter settings in a production network. Because these queueing algorithms affect all traffic, not just traffic generated for the experiment, the use of production networks must be limited to validation of experimental results. For initial analysis a controlled environment is preferable.

A controlled environment allows one to control the number of variables in a given experiment to focus on the effects of specific factors. For example, to have reproducible experiments, the network load is controlled using traffic models and a script to control the changes in load and traffic mixes. This controlled environment also simplifies the analysis and focuses on the effects of specific events, like the introduction of a particular traffic pattern or changes in parameters, instead of accounting for a wide number of variables.

There are two alternatives for establishing such controlled experimental environments: simulation and private experimental networks. Simulation offers numerous benefits. Simulations can be conducted using a single computer, requiring much less physical infrastructure than experiments in a real network. Given sufficient computing and storage resources, one can conduct many experiments simultaneously by running each simulated experiment as a separate process. Moreover, simulations are easily reproducible as the network topology and traffic conditions are all factors of input parameters or configuration files. Further, it requires minimal effort to change experimental conditions. For example, changing network topology simply requires editing the input file that specifies the topology. Simulation offers precise control over all factors, even controlling the rate at which time passes allowing instrumentation to appear to have no impact on performance. However this detailed control can also introduce complications. This level of control is based on accurately modeling all of the factors that contribute to network conditions. These factors are very complex. They range from the obvious factors such as network topology, traffic types, and average load from different traffic streams to less obvious factors such as the behavior of different transport protocols, the behavior of the network media itself (e.g. ethernet's collision detection and back-off), or timing and scheduling effects in the operating system. As a result, it is possible that a simulation will not account for all of the possible factors and their effects. Only by comparing simulation results to actual physical experiments can the simulation model be validated.

In contrast to simulation, using a private physical network carries a great deal of overhead. First, this approach is much more expensive than simulation. One must have end-systems and network infrastructure sufficient to support the desired topology. Second, the complexity of managing the network can be significant. A change to the network topology requires reconnecting cables and changing network routes, possibly even installing different network interfaces. One must also take care to be sure the network is properly configured for each experiment as a physical network may be shared by many researchers. However, these complications are balanced by the benefits of using a physical network for experiments. The factors involved are real and can affect network conditions and consequently, affect performance of the algorithms. For example, in a simulation one may model the router queues using a steady drain-rate. However, in practice some network interfaces drain the router queue in a bursty manner, using a threshold scheme with high and low watermarks associated with on-board buffers, resulting in seemingly erratic queue drain-rates. This type of behavior has a significant affect on algorithms that rely on average queue behavior to infer network conditions. Such factors must be considered. Experiments using a physical network uncover such factors; simulation may not. Moreover, factors like the behavior of the transport-level protocol stacks do not have to be modeled with a physical network since it uses the real transport protocol stack on each end-system. Similarly, using real network components (especially operating systems) provides the opportunity to assess issues of overhead which simulation fundamentally cannot do.

However, there is still a need for some simulation in this approach. The behaviors of the application-level protocols are modeled in order to allow for reproducibility. Application models must be used because some transport-level protocols, like TCP, change their behavior in response to network conditions. Moreover, application level behaviors also depend on network performance. For example, in a web-browser, one cannot follow a link until the page containing that link is displayed. Consequently, if network performance is poor the time between consecutive page requests would increase. Consequently, simply recording and replaying a packet-level trace is insufficient. (This is issue is discussed in more detail in Section 3.4.) The applications are also simulated with models because of issues of scale. For example, in a small private network it is impractical to set up a collection of web-servers with hundreds or thousands of users. Instead the behavior of users and web-servers are modeled with application models. The details of such models are described below in section 3. Both simulation and private networks require this type of modeling. Finally, experimentation in a physical network offers a more direct path to deployment and final testing in production networks as the implementation is not a model built on a simulator but an actual implementation that could be deployed once the parameterization and testing have been completed.

In summary, both simulation and empirical experiments are worthwhile. However, a physical network is used, not a simulation. The combination of control and reproducible conditions made both alternatives attractive. However, a laboratory infrastructure is well suited for empirical experiments in a controlled network. The infrastructure was already in place so the cost of acquiring and configuring the experimental infrastructure is not a concern. Also, these experiments repeatedly use the same simple network topology (see section 2) so reconfiguration is not a concern. As such, the fact that a private network more accurately reflects real world factors was the deciding factor in the decision to use the private network instead of simulation. Although a private laboratory network is used, simulation is a viable alternative, particularly because many experiments can be conducted simultaneously on a small set of computing resources.

2. Network Configuration

A very simple network topology is used for the experiments in this dissertation. This topology is not intended to be representative of the Internet as a whole and leaves out many factors, such as the effect of feedback from multiple routers using AQM techniques under various levels of overload. Moreover, it is difficult to define what configuration would be representative of the Internet. Instead of attempting to model the entire Internet in a laboratory network, this simple configuration limits the number of variables, focusing on the effects of very specific factors on the traffic traversing a single bottleneck link. This limits the scope of this work to focus on a basic understanding of the relationships between queue management algorithms, parameters, and performance.

2.1. Network Configuration

The logical structure of the network is shown in Figure A.1. This simple network topology includes a set of traffic sources on the left side of the figure and a series of traffic sinks on the right side. This arrangement of the end-systems assures that all overloads occur in the desired network queue, eliminating concerns about the effects of overload on other links or router queues. This network configuration is intended to model the behavior of a router serving as an egress point for a campus or other autonomous system.

The router shown in crosshatched red is studied. At this router, the capacity of the link on the left exceeds the capacity of the one on the right, creating a bottleneck. Directing most traffic from the left to the right of the figure creates this bottleneck. The two routers on the bottleneck link are connected by a full-duplex connection so MAC-level contention for the link is not an issue. Beyond the bottleneck link, the other end-systems are varying distances from the router (in terms of propagation delay) but there are no other bottleneck points or opportunities for congestion. This is a conscious design decision to simplify the analysis. As this work focuses on the behavior and effects of a single router, consideration of the interaction of multiple routers is left for future work.

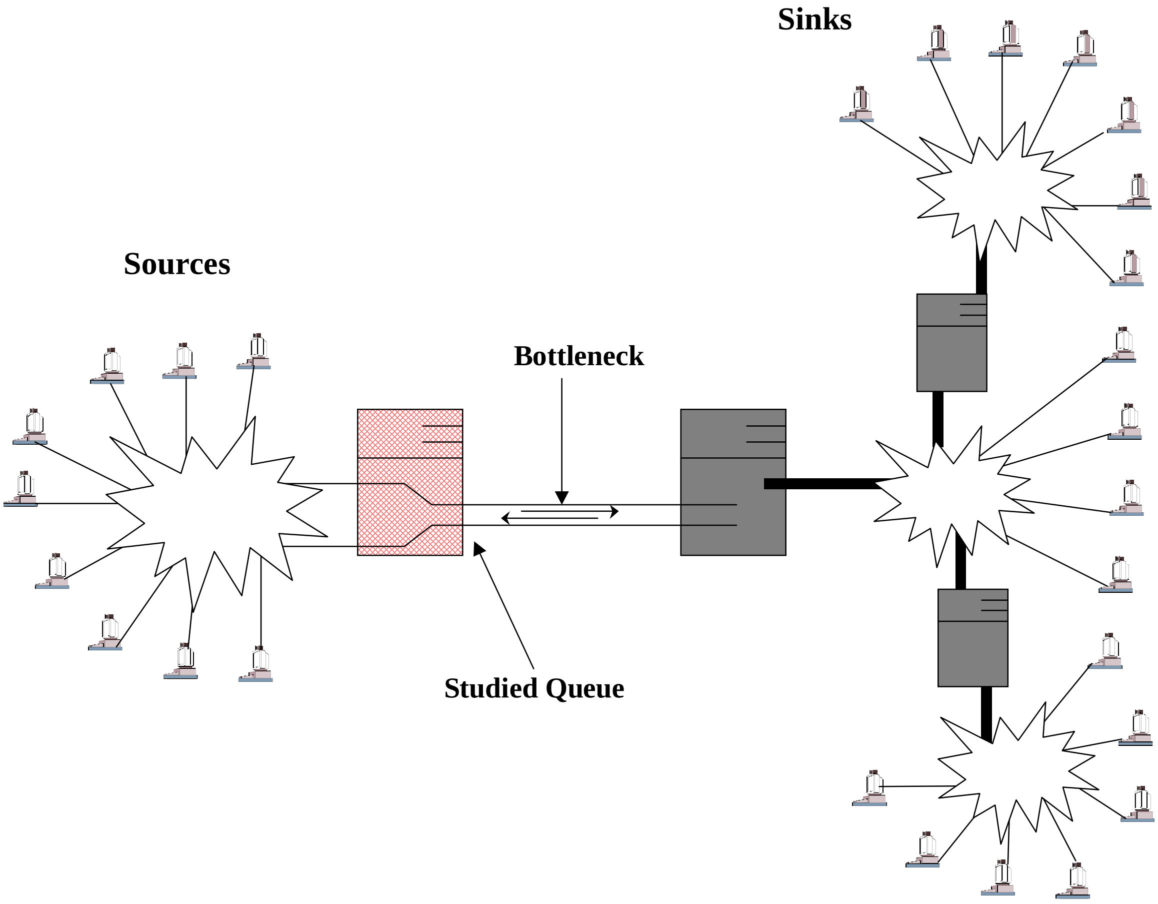


Figure A.1 Logical Network Configuration

2.1.1. Physical Network Configuration

In reality, the logical network configuration must be implemented within the laboratory environment. Figure A.2 shows the physical network configuration. It consists of a combination of 12 end-systems, 2 systems (daffy, bollella) serving as network routers, and an additional system (yosemite) as a network monitor. All of these machines run FreeBSD version 2.2.8. The end-systems all have 10Mb/s connections to one of two Catalyst switches. There are 7 end-systems connected to either switch. The bottleneck link represents the connection to an ISP while the switch labeled “138” represents the campus network and the switch labeled “134” represents the various networks beyond the bottleneck point. Artificial delays on the inbound interfaces of the sources (Section 2.1.4) give the appearance that the "138" end-systems are at varying distances from the campus. Additionally, each of these catalyst switches is connected to a 100 Mb/s hub. The hubs are used to provide a monitoring point as described below in Section 5.

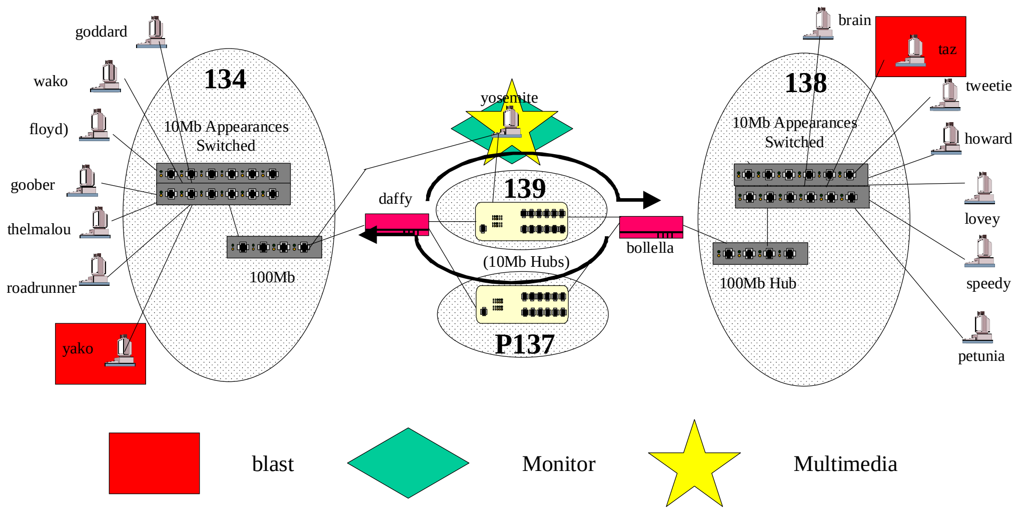


Figure A.2 Physical Network Configuration

The router labeled daffy is the focal point of the work. It is the bottleneck router connecting the 100Mbs, campus network (134) to the 10Mb/s link. Daffy runs the different queueing algorithms analyzed (see section 2.1.3). Bollella is the router on the other end of the bottleneck link. Contention at Bollella is not an issue since most of the generated traffic flows from the 134 network to the 138 network with only small requests and acknowledgements travelling from the 138 network to the 134 network. (The networks are referred to by the third octet of their IP addresses. The 'P' refers to a private, 192.168.xxx.xxx, network.) To simulate a full-duplex link between bollella and daffy, the routers are connected to two hubs, 139 and P137. Static routes allow bollella to forward all packets to daffy across the P137 hub and, likewise, for daffy to forward all packets to bollella across the 139 hub. The effect is that of a single 10Mb/s full-duplex link. Hubs were used instead of a true full-duplex connection to allow network monitoring by an independent system.

The traffic on each of these logical networks was monitored from yosemite. That machine used 3 network interfaces to connect to the hubs. The actual monitoring is discussed in Section 5.1, below. Each component of the network configuration is discussed at more length below.

2.1.2. End-systems

Physical constraints limited the number of end-systems available for use in the experiments. However, multiprocessing and multiplexing at the application protocol level allowed the 7 end-systems on either side of the bottleneck router to simulate the traffic associated with thousands of simultaneous client-server interactions. These end-systems served as sources and sinks for each traffic model. The details of traffic generation are discussed at length in section 3. Most of the end-systems were sources or sinks for TCP traffic. However, one end-system on each network (yako, taz) was a source or sink for aggressive UDP traffic. Finally, a single machine (yosemite), with interfaces on both the "134" and "138" networks served as the multimedia source and sink. This allowed precise measurement of network latency because the sender and receiver time-stamps used the same clock.

2.1.3. Router Configuration

While the end-systems are critical for generating traffic to evaluate the algorithms, it is the routers in their roles as forwarding devices that are the focus of this work. The actual routing decisions are of little concern. As such, static routes were used to establish the desired network topology. There were no routing protocols or dynamic routing issues in this network configuration. Different queueing policies were evaluated by activating them on the bottleneck router using the ALTQ [Cho98] framework. ALTQ provides a framework for implementing and activating alternative queueing policies using FreeBSD systems as network routers. The standard distribution of ALTQ includes implementations of the FIFO, RED, and CBQ algorithms. For these experiments, the FRED and CBT algorithms were also implemented within the ALTQ framework.

2.1.4. Induced Delay

To simulate realistic conditions, there must be variable round-trip times between end-systems. This avoids the synchronization effects that can occur when TCP flows have the same round trip time. To accomplish this, delays were artificially introduced using dummynet [Rizzo97] on all of the source machines. This tool introduced delays on the arriving packets. So the delay is introduced for acknowledgements just before they reach the original sender. The effect is to increase the apparent round trip time as measured on the sender. Since none of the protocols or applications used one way delay in any fashion this technique was acceptable. If systems were concerned with one-way delay (or delay on specific links) delay could have been introduced at other points in the network. Table A.1 describes the delays between end-system pairs:

Sources	**Sinks**						
	brain	**howard**	**lovey**	**speedy**	**petunia**	**tweetie**	**Avg.**
floyd	15	0	14	60	98	30	36.16667
goober	30	15	28	112	42	50	46.16667
thelmalou	10	45	80	84	154	60	72.16667
roadrunner	73	126	30	70	56	70	70.83333
goddard	40	5	96	15	49	110	52.5
Avg.	33.6	38.2	49.6	68.2	79.8	64	55.56667

Table A.1 Delays

The machines down the left-hand side are the senders. The machines across the top row are receiver machines. All times are in milliseconds. This combination of round trip times is based on round trip times to various internet sites from the laboratory at the University of North Carolina, measured using ping. They represent typical delays associated with visiting various representative web-sites. With the network topology established, we now consider the types of traffic modeled and evaluated.

3. Traffic

In almost all of the experiments there are three main types of traffic: TCP, multimedia, and *other*. (The main exception is in the baseline experiments where baselines are established for each traffic type in isolation.) Within the main classes of TCP and multimedia, two different application types were considered for each class. These applications represent different extremes in the behavior of each class. Each traffic type is briefly summarized below and followed by more detailed descriptions of each traffic class and its significance.

Class	**Application**	**Load (Mb/s)**		**Packet Size (Bytes)**	
		Average	**Variability**	**Average**	**Variability**
TCP	**HTTP**	9-10	Moderate	1062	Moderate
	BULK	>10	Very Low	1440	Very Low
Other	**Blast**	10	None	1024	None
Multimedia	**Proshare**	1.3	Low	700	Moderate
	MPEG	1.3-1.5	High	811	High

Table A.2 Summary of Traffic Types

Table A.2 summarizes some of the characteristics of the traffic generators. The table has a row for each traffic type and shows the average load each type can generate on the inbound link of the router, assuming no other traffic types are present. It also shows the average packet sizes for each type and indicates the variability in both load and packet size associated with each class. These ranges of variability will offer a more complete understanding of how the different router mechanisms behave under different conditions.

For these experiments, TCP traffic can be either a set of HTTP flows or a set of BULK data transfers. The HTTP traffic offers a large number of short-lived connections transferring mostly small (~1KB) objects. In contrast, the BULK traffic offers a small number of long-lived flows transferring data at the maximum possible data rate. The BULK transfers generate greater than 10Mb/s of load in aggregate across 240 flows. These traffic types were chosen to examine the effects of extremely different types of responsive traffic.

The multimedia traffic can either be Proshare or MPEG. The Proshare flows are near constant bit-rate but with variable packet sizes due to fragmentation and the mixing of audio and video frames. In contrast, MPEG has more variability. MPEG uses three different frame types: Intraframe (I), Bi-directional (B), and Predictive (P). The MPEG I-frames can be significantly larger than the P-frames and B-frames, producing more variability in the packet sizes and bit-rate over small time scales. Moreover, the MPEG stream is also more variable over larger time scales as the encoding process responds to scene changes or variations in the amount of motion in a given scene.

The *other* traffic, called a UDP blast, is constant bit-rate. The total load generated by the *other* traffic always sums to 10Mb/s. However, the number of flows can be varied producing this load, with each flow generating a load of 1/nth of 10 Mb/s where N represents the number of flows.

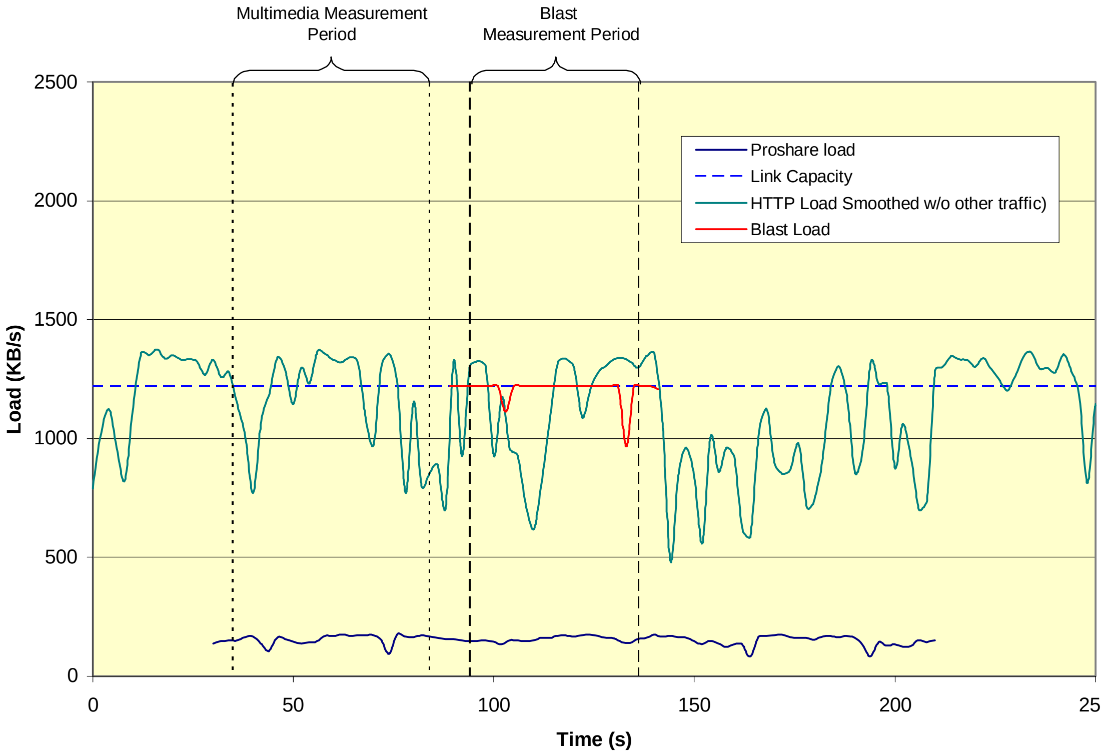


Figure A.3 Example of a Traffic Mix

3.1. Understanding the Traffic Loads

Figure A.3 shows an example of the type of plot used to describe the traffic mixes. As each traffic pattern and the various mixes are introduced below, each part of this figure (or one like it) will be considered in greater detail. This figure is presented here to explain the general nature of these plots. The figure shows representative load generated on the bottleneck router's in-bound link by each of the traffic types in an experiment. *Load* is defined as the bandwidth used on the router's inbound link and *throughput* is defined as the bandwidth used on the router's outbound link. Throughput is a result of many factors, especially the behavior of the router's congestion control mechanisms. Because the aggregate load for a given traffic type never exceeds the capacity of the inbound link the load is generally invariant for a traffic type. The exception to this statement occurs in the case of a responsive protocol, like TCP. However, these figures focus on the potential load each type could generate assuming that traffic type is the only one present in the network, not the load generated when other types of traffic are present.

Consider Figure A.3. The y-axis is the average load in kilobytes per second. The load measurements are averaged over 2 second intervals. Other plots may also show the average over 100ms intervals to demonstrate short-term variability. The x-axis is the translated time in seconds. Most of the plots of individual traffic classes below are on the same scale as the one above in order to facilitate comparison. For those cases that zoom in to illustrate details, the change in scale is clearly indicated.

The horizontal dashed line indicates the capacity of the bottleneck (the router's outbound) link. The HTTP traffic load is able to exceed the bottleneck link's capacity because this is load measured on the in-bound link that has capacity of 100Mb. The vertical dashed lines delimit monitoring intervals that include different traffic mixes. The data series themselves are the load generated by each traffic class. The details of the different traffic mixes and these monitoring intervals are discussed below. Each traffic type is considered in turn. The discussion begins with the simplest, *other,* proceeds to multimedia, and then to the most complex: application models using TCP.

3.2. Other Traffic

The simplest traffic type modeled is the *other* traffic. *Other* traffic is any traffic that is neither TCP nor multimedia. *Other* traffic is aggressive: high-bandwidth, unresponsive, and constant bit-rate. The *other* traffic type is made up of UDP flows referred to as a UDP blast. The UDP blast is a flow or collection of flows offering a load of approximately 10Mb/s.

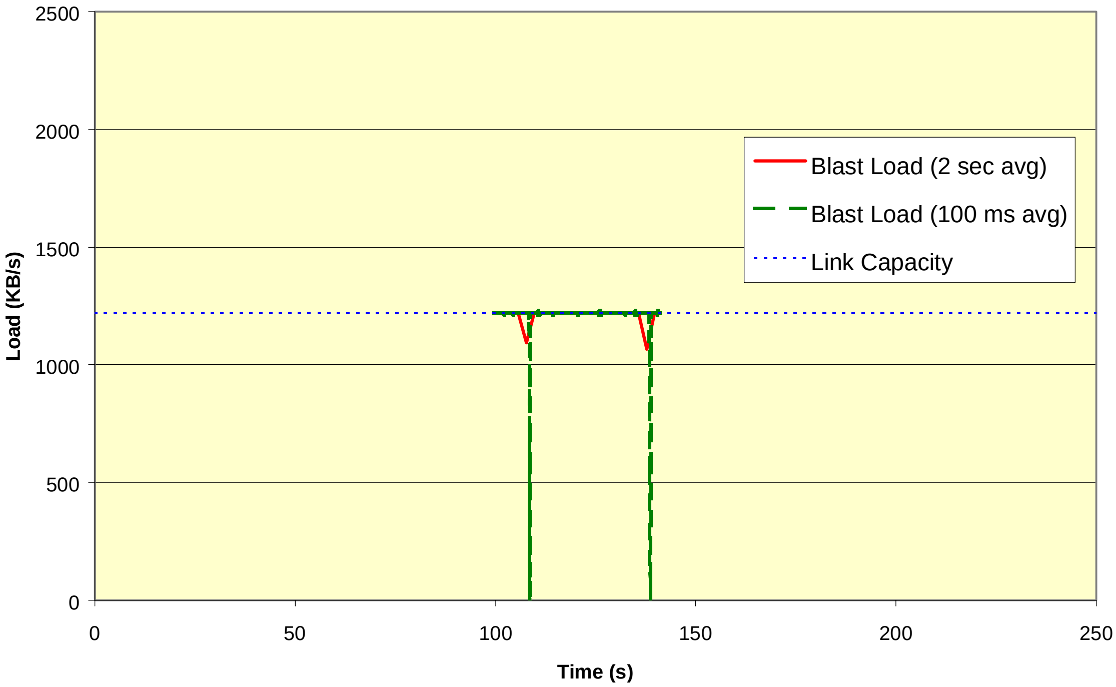


Figure A.4 Load Generated by a UDP Blast

Figure A.4 shows the load generated by the blast during one experiment. The figure shows the load on the inbound link of the router averaged over both 100ms and 2 sec intervals. There is very little variability in the load. Two spikes are visible during the 100ms average due to flaws in the measurement technique. The network-monitoring tool sometimes fails to record any data during a small interval, resulting in intervals with no recorded load. This absence of data is interpreted as zero load, leading to the spikes shown. Averaging over a larger interval decreases the effect of these lost samples. Note that the load for *other* is bounded above at 10Mb. This limit is a result of the 10Mb link from the UDP blast source to the 100Mb first-hop network. This 10Mb load is sufficient for these experiments, however, as the concern is with insuring that the UDP blast maintains a load capable of consuming the entire capacity of the 10Mb bottleneck link. This traffic pattern is generated by periodic transmission of uniform sized packets. The experimental default was one flow transmitting 1 KB packets at a rate of 1,270 packets per second resulting in a target load of 1.3 MB/s. Since this load exceeds the capacity of the 10Mb uplink, it results in a 10Mb throughput on the network. The UDP blast traffic is a constant-bit-rate traffic stream consisting of one flow. The packet sizes are 1K and the load is 10Mb/s.

3.3. Multimedia

The second class of traffic is multimedia. One of the goals of this work is to provide better support for multimedia while still isolating TCP from the effects of both multimedia and *other*. Two different types of multimedia traffic are modeled: Proshare™ and MPEG. Proshare, a video-conferencing application from Intel™, presents low-bandwidth, low-variability video and audio streams, without inter-frame dependency. Alternatively, MPEG presents a moderate-bandwidth, moderate-variability video stream with inter-frame dependency. Both traffic types are unresponsive to congestion. Since the traffic does not react to network conditions the load generated is independent of network conditions. As a result, traces of each type of traffic can be replayed to generate the traffic streams. Each traffic type is explained in more detail below.

3.3.1. Proshare

Proshare presents a low-variability multimedia traffic stream. Six Proshare streams are generated using six traffic generators, each modeling one side of a single Proshare video-conference. Figure A.5 shows the load generated by the six Proshare traffic-generators. The load is shown on a scale of 0-2,500 KB/s to make it easy to compare this plot directly to later plots that show different traffic types which generate higher loads.

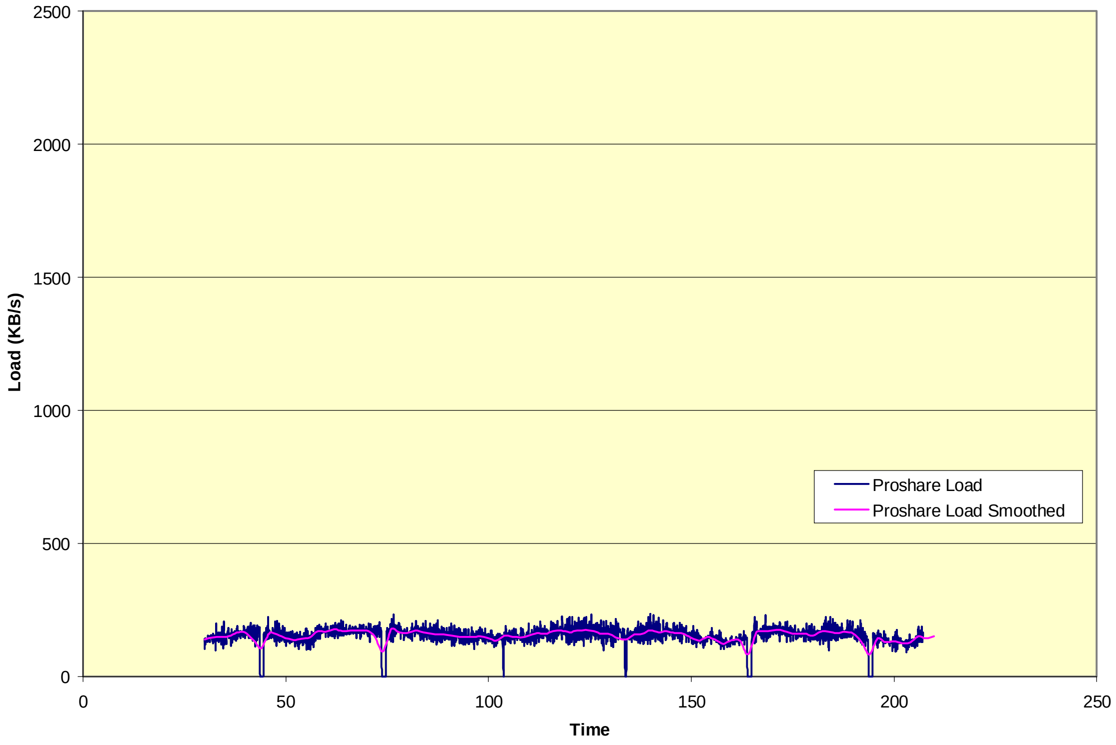


Figure A.5 Proshare Load

Six conferences are modeled because their aggregate load (approximately 160 KB /s, 1.3 Mb/s) is substantial enough to consume a noticeable fraction (13%) of the link's capacity without being excessive.

The Proshare traffic stream is generated by direct playback of a packet-level trace of a Proshare media stream. The original trace recorded an unresponsive Proshare Audio/Video Conference [Nee97] for 294 seconds. The original application generated both audio and video. The audio was a 16 Kb/s audio stream consisting of ten 200 byte "audio-frames" per second. There was no silence suppression so the frame size and rate were uniform. The video was the high-quality, "smoother" option video generating a 200 Kb/s video stream. The video stream consisted of fifteen 13.3 Kb (1.66 KB) frames per second. Fragmentation to the MTU size (1,500 bytes) meant that most video frames generated two packets. There was some variation in the encoding resulting in occasional frames smaller than the MTU size, leading to an average packet rate of 26 pkts/second for video. Combined with the 10 pkts/second for audio, this resulted in a packet rate of 36 pkts/second.

This packet trace approach is viable because the Proshare video streams were unresponsive so the traffic pattern was independent of the associated network conditions. Using the packet trace makes the load associated with a single Proshare flow highly reproducible. Because the same trace is used for each of the Proshare generators, care is taken to avoid synchronization of the six Proshare streams. Synchronization would result in amplified extremes in the aggregate load. To avoid this possibility, each traffic generator begins at a random offset into the trace. The generator simply transmits packets of the appropriate size at the inter-packet intervals specified in the trace. When the end of the trace is reached, the generators starts at the beginning again.

It is important to note that the frames in a Proshare stream are independent. Decoding any given frame does not depend on the content of any other frames. As a result, a lost packet only affects the frame that packet belongs to. This will be important when presenting results on loss-rate and their effect on the application. The Proshare traffic generator presents 6 flows uniformly generating in aggregate a load of approximately 160 KB/s with an essentially constant bit-rate. The bit-rate does vary slightly due to variable packet sizes resulting from fragmentation effects and the variation between audio and video frame sizes. However, the six Proshare flows generate an average aggregate load of 160 KB/s with low variability.

3.3.2. MPEG

The second multimedia data type is MPEG-1 [Le Gall91]. MPEG differs from Proshare in several important ways. First it presents a much more variable traffic load, over both very small intervals (per frame) and large ones (tens of seconds). Second, MPEG uses interframe encoding to achieve its high compression rate. As a result, a single lost reference frame may result in many frames that cannot be decoded. MPEG and other interframe encodings are popular in the Internet today so it is important to consider the effect of different router policies on multimedia flows of this type as well.

Traffic Generation Details

MPEG traffic is modeled using an MPEG traffic generation tool. The tool is trace-driven. The input trace specifies the size and type of each frame of the movie as well as the frequency of frame generation, but it contains no actual content. The MPEG tool generates packets of the appropriate size and transmits them at the appropriate time. The input trace is based on an MPEG encoding of the movie "Crocodile Dundee". This movie was 97 minutes long and the MPEG encoding uses a *Group of Pictures* (GOP) having a sequence of frame types IBBPBB with a frame-rate of 30 frames per second. As with Proshare, multiple media streams are generated. In the case of MPEG there are 4 streams, each flow starting at a different offset into the trace. The starting points are offset in increments of 5,000 GOPs (1,000 seconds/16 minutes) in each instance in order to avoid amplified extremes due to synchronization effects. This MPEG tool is instrumented for data gathering. The details are discussed in the section on measurement below (Section 5.2).

Traffic Characterization

The aggregate load generated by the 4 flows ranges between 140-190 KB/s, but is variable at both small and moderate time scales. MPEG’s variability at different scales stems from different sources. At the very small scale, the MPEG traffic may be variable for two reasons. First, individual flows have high variability because of the different frame types used in the encoding process. Second, because this variability is periodic, as it is associated with the frame order, it may be amplified if the flows happen to synchronize on a GOP boundary. Both concepts are explained below.

First, consider the behavior of a single MPEG flow. Figure A.6 shows the MPEG frame sizes for each frame type. Figure A.7 shows the number of packets the frames would span when fragmented. Both plots reflect a one-minute interval of the movie. During this interval the intra-pictures (I-Frames) can range from between 5 KB- 9 KB, spanning 4 to 8 packets. In contrast, the interpolated pictures (B-Frames) are always less than 1 KB easily fitting in exactly 1 packet. The P-Frames lie somewhere in between.

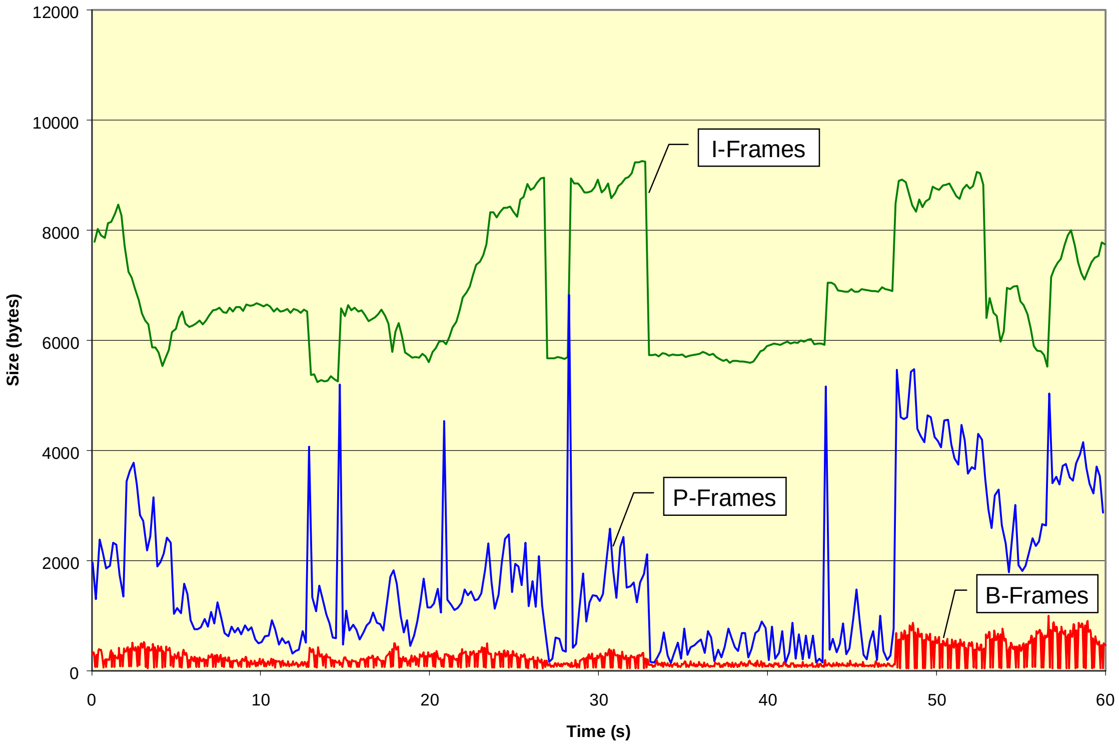


Figure A.6 MPEG Frame sizes by Type

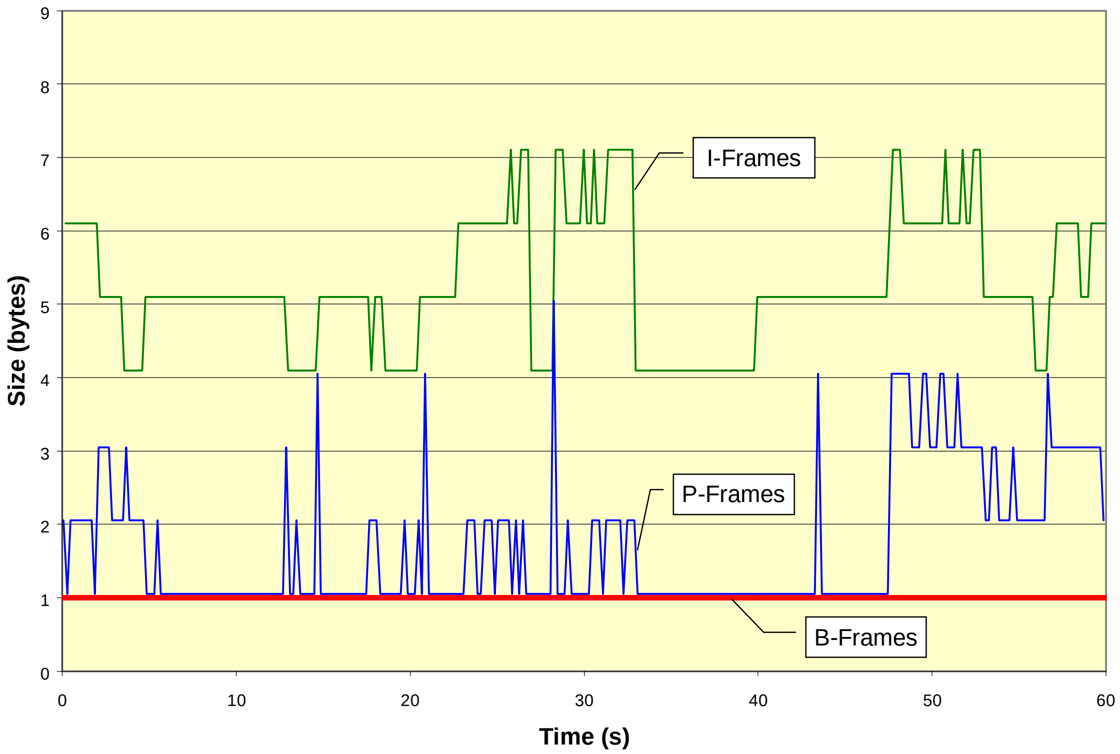


Figure A.7 MPEG Packets per Frame (Ethernet Packets)

These frames are from a GOP with organization IBBPBB. As these different frame types are transmitted the instantaneous load can vary by an order of magnitude. To appreciate this, consider a 50-second segment of the movie. Figure A.8 and Figure A.9 respectively show the average aggregate load and packet-rate generated by all the frames of an MPEG stream with a 66 ms network sampling interval. This 66 ms sampling interval is based on the fact that in many experiments the typical drain-time of a router's queue averaged 66 ms. In a sense this interval offers a snapshot of the potential occupancy of the router's queue during congestion. Clearly the variability is quite high, ranging from almost 8000 bytes during an interval to nearly 0 bytes almost immediately. The packet transmissions over the interval range from 2 to 8 packets.

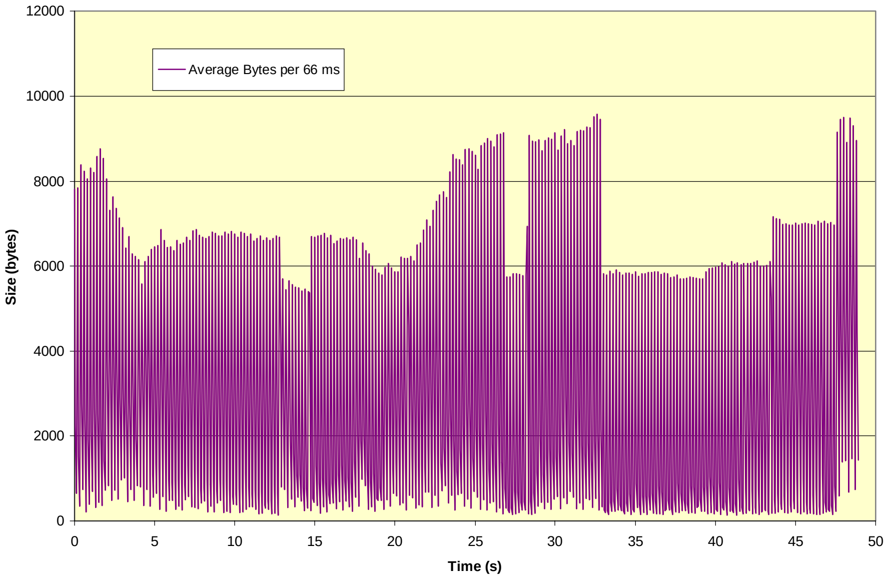


Figure A.8 MPEG Average Bytes per 66ms

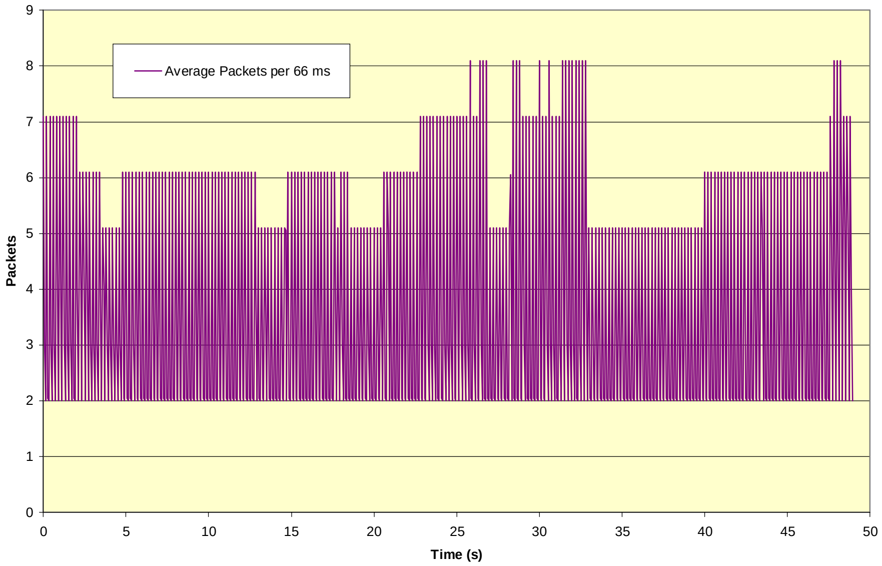


Figure A.9 MPEG Average Packets per 66ms

The aggregate MPEG behavior can display short-term variability as well. Figure A.10 shows the average (over 2 seconds) MPEG load aggregated for all 4 of the MPEG flows. This figure actually shows the average load from two different runs of the MPEG tool although the two are nearly indistinguishable because the performance is nearly identical. With this trace driven approach, one would expect the average loads to be identical and they almost are. This variability is accounted for by the fact that the exact start time of each MPEG traffic generator may vary on the order of a second.

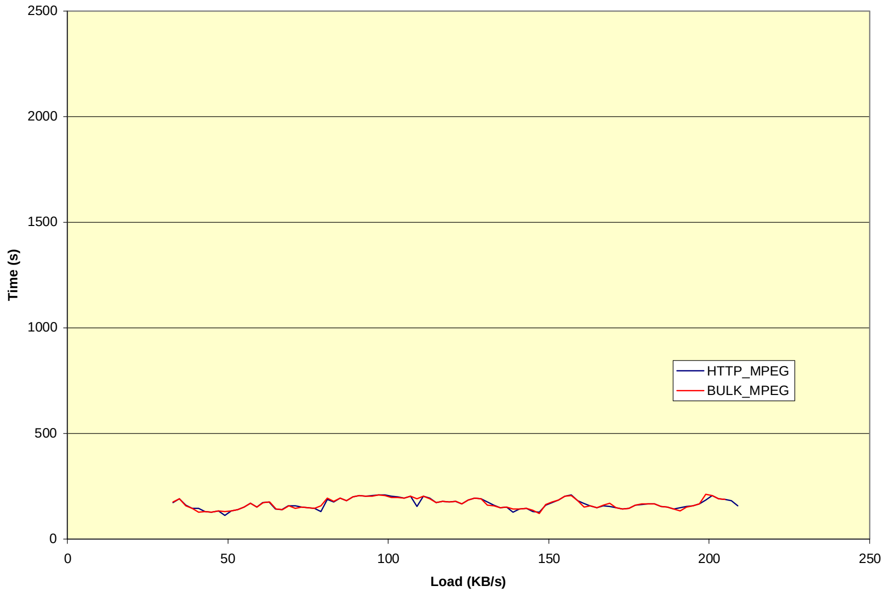


Figure A.10 Average MPEG Loads from Different Traffic Mixes

However, this small variation in start times can result in significant variations in load on small time-scales. Figure A.11 and Figure A.12 show two runs at 100ms (highly variable) and 2 second (smooth) averaging intervals. The same MPEG traffic pattern was used for each run. Notice that although they maintain the same average on the 2-second scale, on the 100ms scale the run in Figure A.12 shows much more variation than the run in Figure A.11.

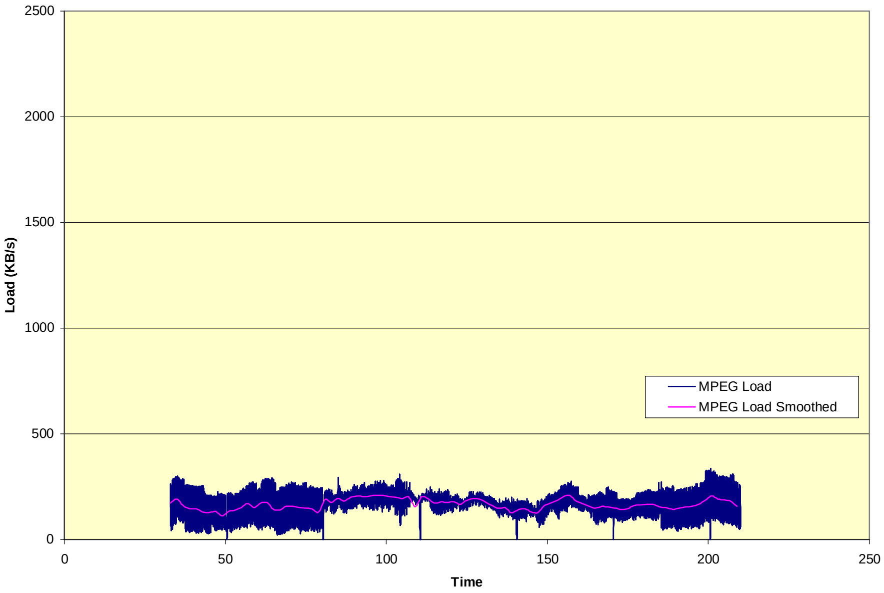


Figure A.11 MPEG Load During Run #1

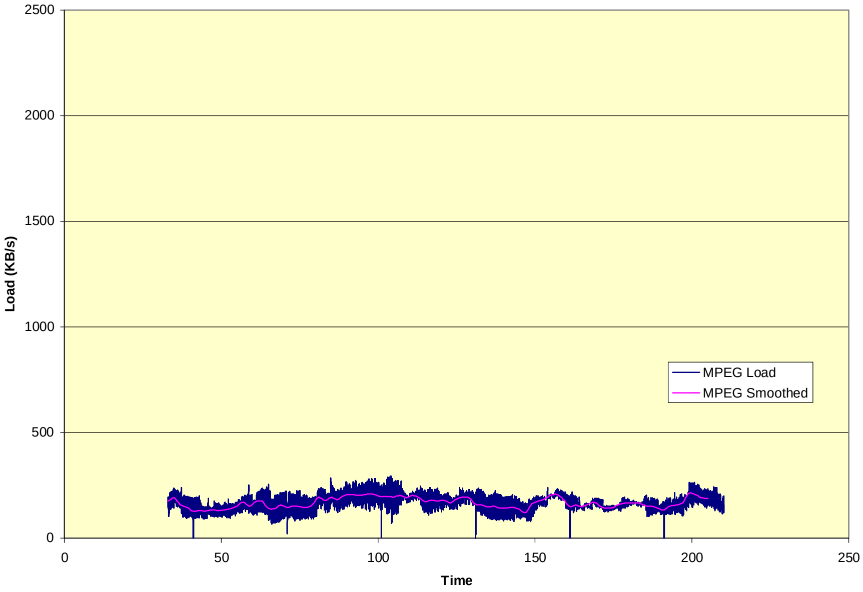


Figure A.12 MPEG Load During Run #2

Note that the 100ms sampling (blue) of the load appears very different between the two runs with the run #1 experiment appearing much more variable. This is actually simply a small time-scale synchronization effect. The source of the difference can be accounted for by considering amplification effects resulting in a form of synchronization between MPEG streams. Consider two extreme examples. Recall that all four MPEG streams use a GOP of IBBPBB with each frame sent 33 ms apart. If all four of the MPEG traffic generators happen to synchronize on GOP boundaries, they may all send the large I frames nearly simultaneously followed by a series of small B-Frames, then moderately sized P-Frames. At a granularity of 100 ms, which will include 3 frames from each stream, having all of the I-frames in one interval leads to no I-frames in the next interval. The resulting disparity results, on average, in samples with 4 I-frames and 8 B-frames totaling approximately 29 KB compared to subsequent samples with 4 P-frames and 8 B-frames totaling approximately 8 KB. Obviously this will result in high variability in the sampling. In the other extreme, if the I and P frames are equally distributed between the samples, all samples will include 2 I-frames, 2 P-frames, and 8 B-frames, totaling approximately 19 KB in all samples.

This variability in bandwidth is relevant because some queue management algorithms (e.g. FRED) do use the short-term occupancy of the queue as an indication of network conditions. The queues used in these experiments can fully drain in a time on the order of 100ms. As a result, the apparent state of the network as inferred by these algorithms can appear to change radically over that interval if the sources are synchronized.

In addition to the short-term variability, MPEG streams also show more long-term variability both in packet sizes and in average load. Figure A.13 illustrates this variability both between frame types and even between frames of the same type over the entire movie. The figure shows the frame sizes in bytes vs. each frame of the movie. While the B-frames are consistently fairly small (1-2 KB) the I-frames clearly range from almost 14,000 bytes to 2000 bytes and P-frames range from 10,000 to less than 1 KB.

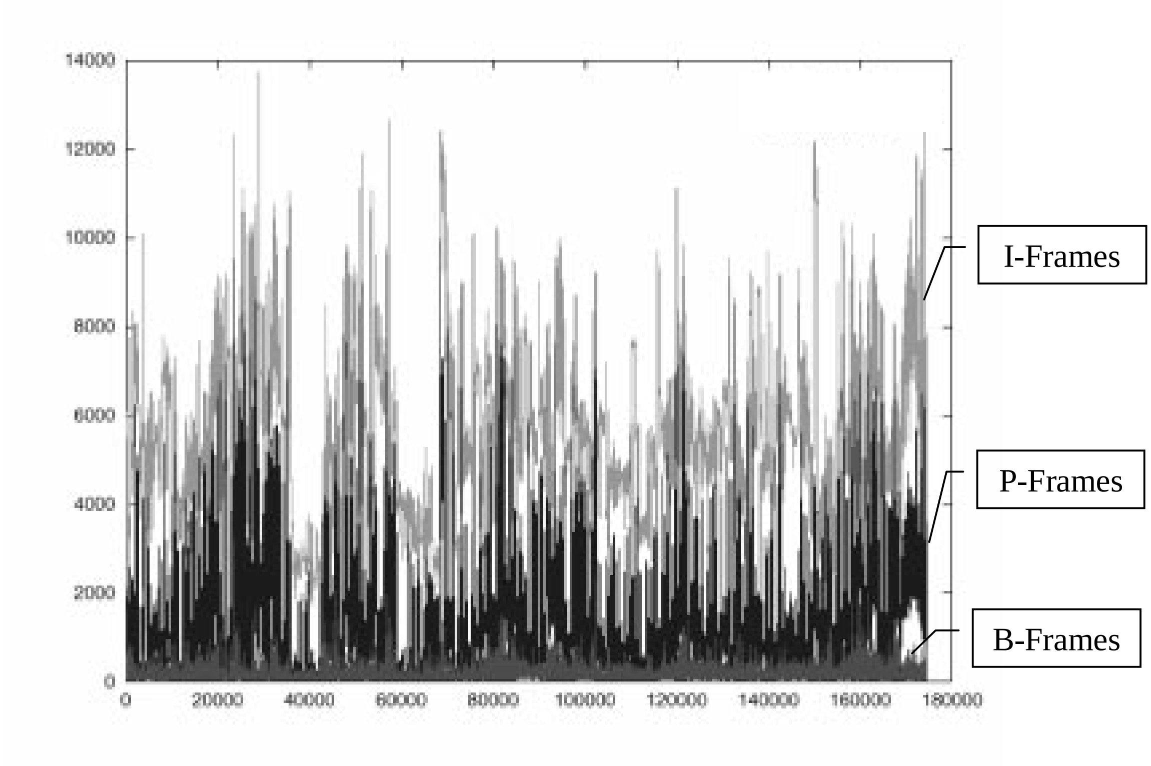


Figure A.13 MPEG Frame Sizes over Time (Crocodile Dundee Movie)

Over this longer duration, variation in the scene and motion within the scene being encoded results in variation in the frame sizes because the redundancy between frames varies. Figure A.14 shows a closer look at the average load generated by the aggregate MPEG traffic. The average load climbs to over 200 KB/s, presumably during scenes with a great deal of motion or rapid scene changes. Likewise, the load averages as low 130 KB/s, presumably due to a single still shot or close-up. Moreover, these variations may cover significant intervals. For example, during the period marked *measurement period 1* the average load is 145 KB/s, while during *measurement period 2* it is 197 KB/s.

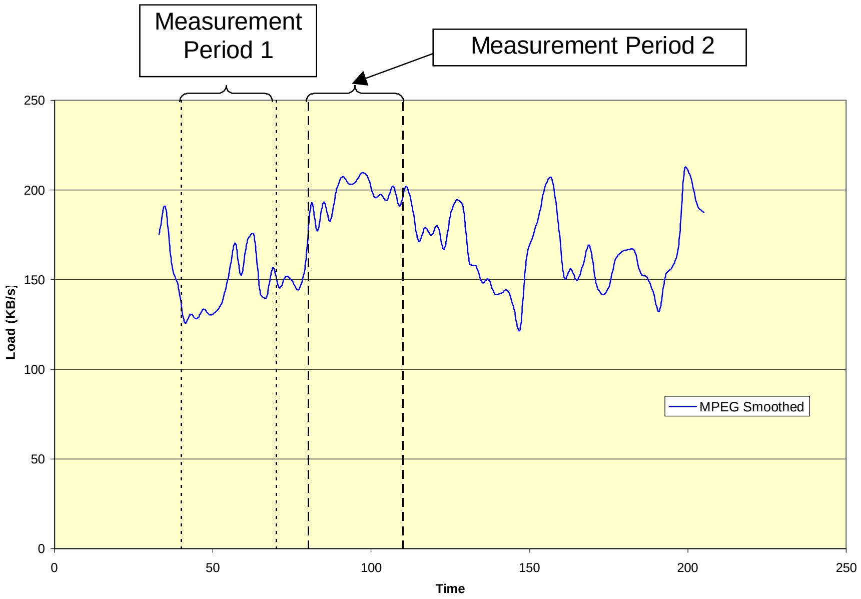


Figure A.14 MPEG Load

This long-term variation is a potential source of concern if two different intervals are monitored and compared to one another. The aggregate load in a variety of experiments and measurement periods is shown in Table A.4. The details of these measurement periods are explained in section 4.1. The concern here is simply that the average loads are different in different periods.

Exp. Set	**Measurement Period**	**KB/s**	
		Average	**Range**
A	**1**	164	138-191
	2	160	143-170
B	**1**	183	162-191
	2	154	145-168
C	**1**	186	174-192
	2	151	144-168

Table A.4 MPEG Loads in Different Experiments and Measurement Periods

However, these problems are tolerable. The measurements periods used in this dissertation are coarse-grained. Moreover, as explained in Section 4, the same intervals are measured across all comparable experiments and results from measurement period 1 are never compared to measurement period 2. As a result, the comparisons are valid. Only CBT and CBQ have any concern with expected load and in both instances the correct approach is to provision for the worst case behavior. In the metrics, loss-rate can be examined to get a sense of the effectiveness of any algorithm whether or not the inbound load is known. Moreover, even the largest variations are only 50 KB/s. The significance of this variation depends on perspective. 50 KB/s is 4% of the link capacity so the effect on the majority of the link's load, TCP, is negligible. However, 50 KB/s is 20-25% variation in the MPEG load so the MPEG performance could vary significantly and must be considered when provisioning for algorithms like CBT and CBQ.

Sensitivity to Loss

Another important issue related to throughput is sensitivity to loss. A low loss rate may have little value if the few frames that are lost are needed to decode the other frames of the media stream. In that case, some of the frames that do arrive are also useless. MPEG traffic is particularly sensitive to loss. There are two major reasons that MPEG is so sensitive. First, because some frames span many packets, a single lost packet renders the other packets of that frame useless. Second, because of inter-frame encoding, the loss of a single frame may make other frames of that GOP useless. Consider each issue.

First, as Table A.5 shows, MPEG’s I-frames are quite large, averaging almost 7 KB (and ranging 2-14 KB). In this MPEG stream the I-frames span 6 packets on average while B-frames never span more than 1 packet. (Note that the number of packets per frame is not computed by dividing the average bytes per frame by the MTU. Rather, it is computed by actually counting the number of packets for each frame, which is always an integer value, and dividing by the number of frames.) If drops are uniformly distributed (a goal of RED) a given packet-loss rate will on average result in 1.5 times as many lost I-frames as B-frames because the four B-frames in a given GOP represent 4 packets while the singe I-frame spans 6 packets. Moreover, the frame loss-rate for I-frames would be 6 times the packet loss-rate because when one packet in an I-frame is lost, all of the other packets in that frame have to be discarded. This does not consider any relation that might exist between burstiness and drop-rate (as in algorithms like FRED) which may make larger frames more susceptible to loss. Those issues will be considered for those algorithms where it is a factor.

Frame Type	Bytes Per Frame	Packets per Frame
I	6916	5.26
B	241	1
P	1646	1.69

Table A.5 MPEG Average Packet Statistics by Type

This increased probability that drops will occur in I-frames leads to the second reason MPEG is sensitive to loss. While loss of a single packet will always cause the effective loss of the frame that packet belongs to, loss of a packet in a reference frame also results not only in loss of that frame but also in loss of all of the frames that refer to that frame due to interframe encoding. This leads to effective frame loss-rates that exceed the packet loss-rate. *Effective loss-rate* is defined to include the frame that is successfully delivered but useless because of the loss of other frames required to correctly decode the successfully delivered frames.

Frame Type	Packets per Frame	Frames per GOP	Effective Lost Frames
I	6	1	8
B	1	4	1
P	2	1	5

Table A.6 MPEG Frame Losses Assuming a Single Frame Loss

Table A.6 illustrates the effects of packet loss on frame loss. Recall that in this MPEG stream each GOP contains one I-Frame, 4 B-frames, and 1 P-Frame. Those frames span on average 6, 1, and 2 packets respectively. Loss of a single I-frame results in the effective loss of more than an entire GOP. It results in the effective loss of the preceding B-frames that were interpolated using the I-frame as a reference and on the subsequent B-frames and P-frames up to the next I-frame. Loss of one packet in an I-frame results in the effective loss of 8 frames of video. Loss of a single P-frame results in the loss of the B-frames that depend on it. In this encoding, 5 frames (1 P-frame and 4 B-frames) are effectively lost whenever a P-frame is lost. Lost B-frames do not affect other frames and results in the loss of only that one frame.

On average the GOP spans 12 packets. If the probability of a packet loss is 1 in 12, then on average one packet, and thus one frame, will be dropped in every GOP. Table A.7 shows the effective frame loss rate with this packet loss rate. Since half of the packets in the GOP are I-frames the probability that the dropped packet will be part of an I-frame is 50%. 33% of the lost frames will be B-frames and 16% will be P-frames. Each of these losses results in effective losses of 8, 1, or 5 frames, respectively. Multiplying the probability that a given frame type will be lost by the number of effective lost frames associated with that frame type and summing gives us the average number of frames that may be lost per GOP with this loss rate. Effectively, nearly 43% of the frames are lost when the packet loss rate is only 8% (1 in 12).

Frame Type	**Average Packets per GOP**	**Probability of Packet Loss in Frame Type**	**Effective Frame Loss Rate (frames/GOP)**	**Loss Rate**
I	6	50%	4.00	33%
B	4	33.3%	.333	2.8%
P	2	16.6%	.833	6.9%
Total	12	100%	5.166	43%

Table A.7 MPEG Frame Drops with 1 in 12 Packet Loss

Figure A.15 shows the relationship between the packet loss rate and the frame loss rate for this MPEG stream. The frame loss rate is almost 5 times the packet loss rate in this GOP format. This analysis is somewhat over simplified but it is accurate for this drop-rate. When the packet drop rate exceeds 1 packet per GOP the effects of multiple drops in one GOP must be accounted for. Moreover, the fact that B-frames may be affected by different I-frames and/or P-frames is ignored. However, this simplified analysis makes the point. MPEG's interframe encoding and large reference frames combine to make MPEG very sensitive to packet loss.

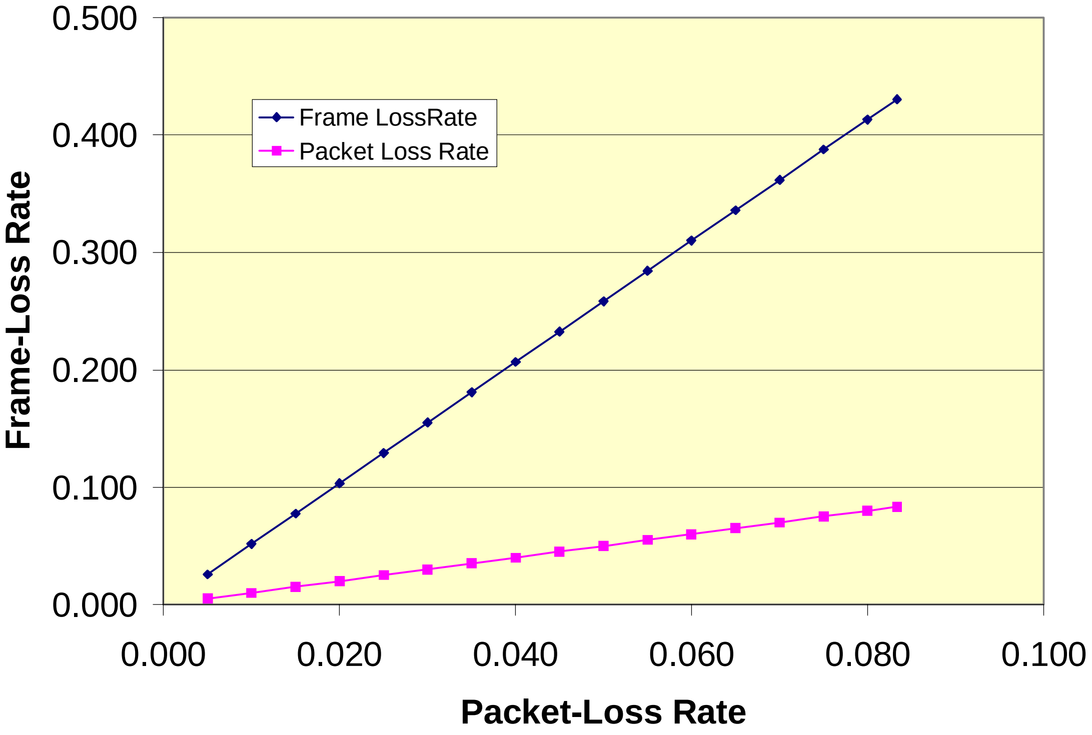


Figure A.15 MPEG Packet Loss vs. Frame Loss

3.4. TCP

As the predominant transport protocol in the Internet today, TCP is a key traffic type. Because TCP is a responsive protocol the trace-driven approach used for the multimedia traffic types is not applicable. At the packet level, the TCP implementation adjusts the rate at which it generates and transmits packets in response to network conditions. Moreover, user or application behavior also varies depending on network performance. Unlike unresponsive multimedia where the application simply generates and transmits frames periodically, TCP applications often have application level exchanges of requests and replies. In the case of HTTP, the generation of application level messages depends on the performance of the network. For example, as a user follows links through a sequence of pages he must receive the current page before following links on that page. As a result, if the time required to receive a page increases, the time between page requests will also increase. Because of these factors, these traffic types must be modeled at the application level.

The specific modeling approaches taken for each TCP traffic type are discussed below. Two different types of application level protocols that depend on TCP were studied. The first is the most popular application protocol in the Internet, HTTP. The modeled HTTP traffic is referred to simply as HTTP throughout this dissertation. The second type was an alternative model which contrasted some of HTTP's extremes (e.g., short-lived, light-weight connections) with long-lived, high-bandwidth bulk data transfers. This model is referred to as BULK throughout this dissertation. The HTTP and BULK models are described in more detail below. First, consider the general details of the TCP implementation and configuration for these experiments.

For these experiments all of the end-systems are configured to have TCP sender and receiver windows of 16K. The round-trip times between the end-systems vary between approximately 0 and 126 milliseconds as discussed in Section 2.1.4. This combination of window size and round-trip times should allow any single TCP flow to generate at least 1 Mb/s of traffic based on the delay-bandwidth product. Of course, other factors, such as application behaviors, TCP’s response to congestion, contention for shared media, or network-based delay may limit the actual per-flow throughput. The curious reader may find the relevant details of the TCP implementation used in Table A.8. The items listed and explanations of their significance can be found in [Allman99].

FreeBSD 2.2.8 TCP Implementation (4.4 BSD)	
Basic Congestion Control:	OK
Extensions for High Perf:	OK (RFC 1323)
Selective Acknowledgements:	NO
Delayed Acknowledgements:	OK (RFC 1122)
Nagle Algorithm:	OK
Larger Initial Windows:	NO (RFC 2414)
ECN:	Optional (RFC 2481) on
Bugs (See RFC 2525)	
Bugs in 4.2BSD:	NO
Window Size Bug:	YES
Other Issues **(See draft-ietf-ippm-btc-framework-01.txt)**	
What happens when (cwnd == ssthresh)?	Acts like cwnd < ssthresh (slow-start)
Default ssthresh:	1 MB

Table A.8 TCP Specifics

3.4.1. HTTP

HTTP flows are often referred to as “web mice” as they can be thought of as large numbers of small, short-lived entities scurrying about the network. These mice contrast the elephants of the Internet, large file transfers, which are represented by the BULK transfers. HTTP flows spend most of their time in slow-start and some flows never see the effects of congestion control. This traffic is also burstier than that generated by an application that is constantly trying to transmit data. Instead HTTP flows may consist of a series of request-reply exchanges as the client requests a page, waits for the page to arrive, requests the embedded objects within the page, waits for those object to arrive, and then requests another page. HTTP traffic generation used the model developed by Mah, et al [Mah97] and implemented by [Christiansen00].

Although HTTP flows are individually low bandwidth flows, the large volume of HTTP clients active at any given time represents the majority of the network load as discussed in I.1.1.1. In order to compare experiments, the number of independent variables that change from experiment to experiment must be limited. One such variable is the average load generated by the HTTP traffic model. Christiansen et al., conducted an analysis of the load generated in our experimental network as the number of browsers was varied [Christiansen00] and found a linear relationship.

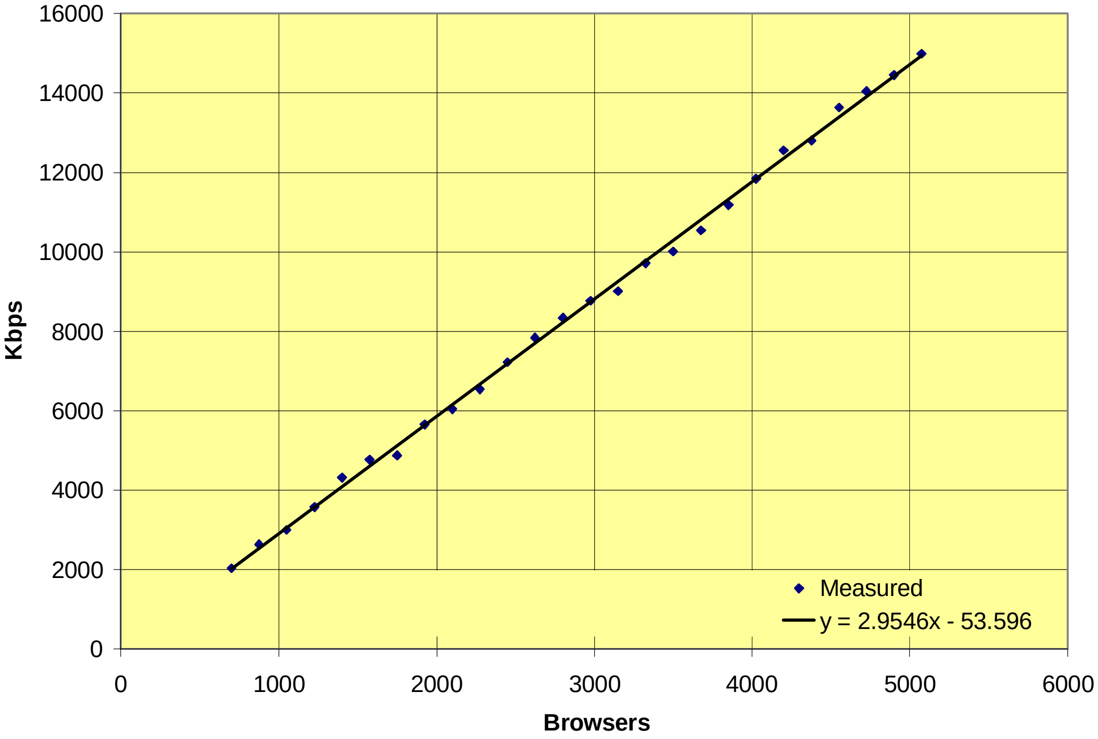


Figure A.18 Generated Load vs. Simulated Browsers

Figure A.18 shows this relationship between the number of simulated browsers and the generated load. They also confirmed that other concerns such as socket limits and client/server processor speeds were not factors in this network configuration. On a 100Mb/s network, they measured the load generated as they varied the number of clients. They then applied a linear regression to find the relationship between the number of browsers and the resulting load. That relation is expressed by the equation $y = 2.9546x – 53.596$

where *x* is the number of browsers and *y* is the resulting load. Using this equation and the observed measurements yields the result that 3,000 browsers generate a load of 1.1 MB/s or 8.8 Mb/s, or approximately 90% of the capacity of the link. This is the standard HTTP configuration used in the experiments. The load is divided between 5 servers being contacted by 6 client machines. Each client runs a model simulating 500 browsers distributing requests across the 5 servers. The router algorithms affect the replies traveling from the servers to the clients (left to right in Figure A.1). The replies represent the majority of the network load for HTTP. This load combined with the 140-190 KB/s of MPEG traffic (160 KB/s for Proshare) should keep the bottleneck link shown in Figure A.1 fully utilized.

The relationship between browsers and load was based on averages over hour long runs. Actually, those runs lasted for more than an hour but the first 25 minutes were ignored because start up effects in the traffic model were very apparent. Consequently, the model used in these experiments also runs for 25 minutes before collecting any measurements.

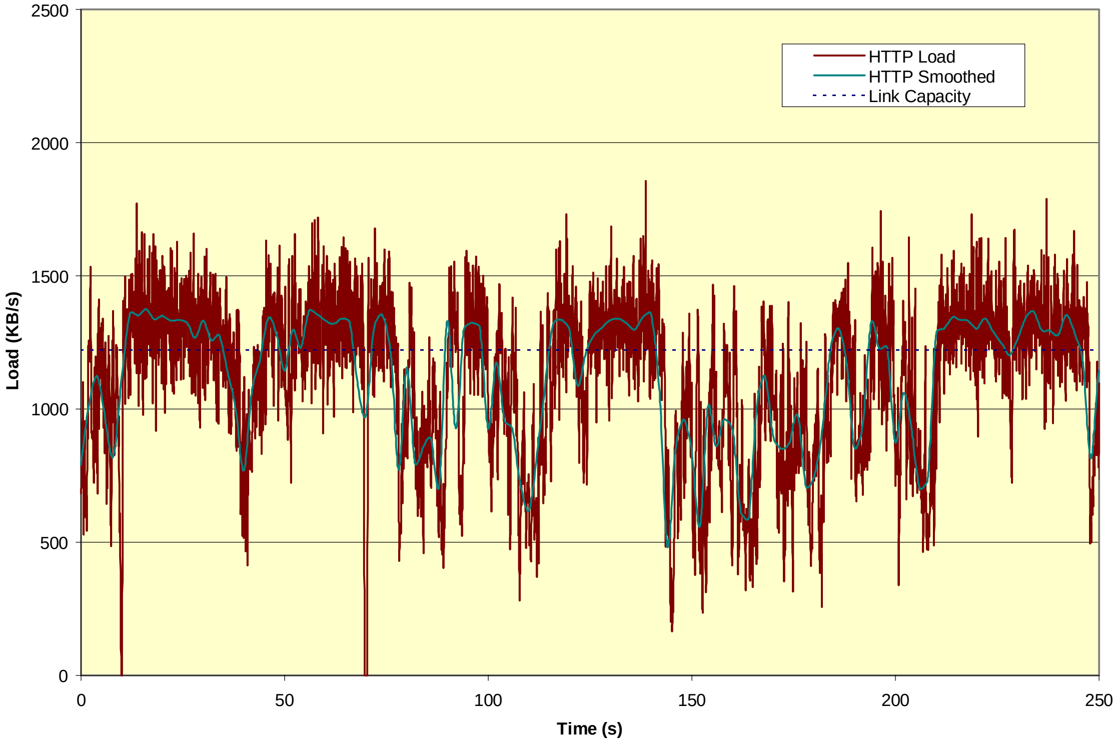


Figure A.19 HTTP Load Alone (but with bottleneck link of 10Mb/s)

Figure A.19 shows the load generated by the HTTP model when no other traffic types are present. The load line is based on averages over 100ms intervals while the smoothed line is based on averages over 2 second intervals. The load is quite variable on both time scales. The behavior of the clients is very bursty, even when aggregated over 3,000 active clients. Note that the load often exceeds 1.2 MB/s, the capacity of the bottleneck link. This is because new connections are always starting and trying to use more capacity. When this effect is aggregated over 3,000 clients it results in the TCP load often exceeding the bottleneck link's capacity. However, as discussed above, the behavior without other traffic types present only gives a general sense of the traffic's behavior. The traffic's responsiveness to changing network conditions must also be considered.

Recall both TCP, the simulated web browser, and users, are responsive to changing network conditions. To illustrate the change in load due to these responsive elements, the load in the presence of other traffic types is also examined. Figure A.20 and Figure A.21 show the load for HTTP with and without different traffic types present on the bottleneck link. Figure A.20 adds the scenario with other traffic to the results shown in Figure A.19. Both show the respective loads averaged over both 100 ms and 2 second intervals. Figure A.21 shows only the averages over 2 second intervals for easier viewing. Figure A.20 shows that the load is bursty in both cases.

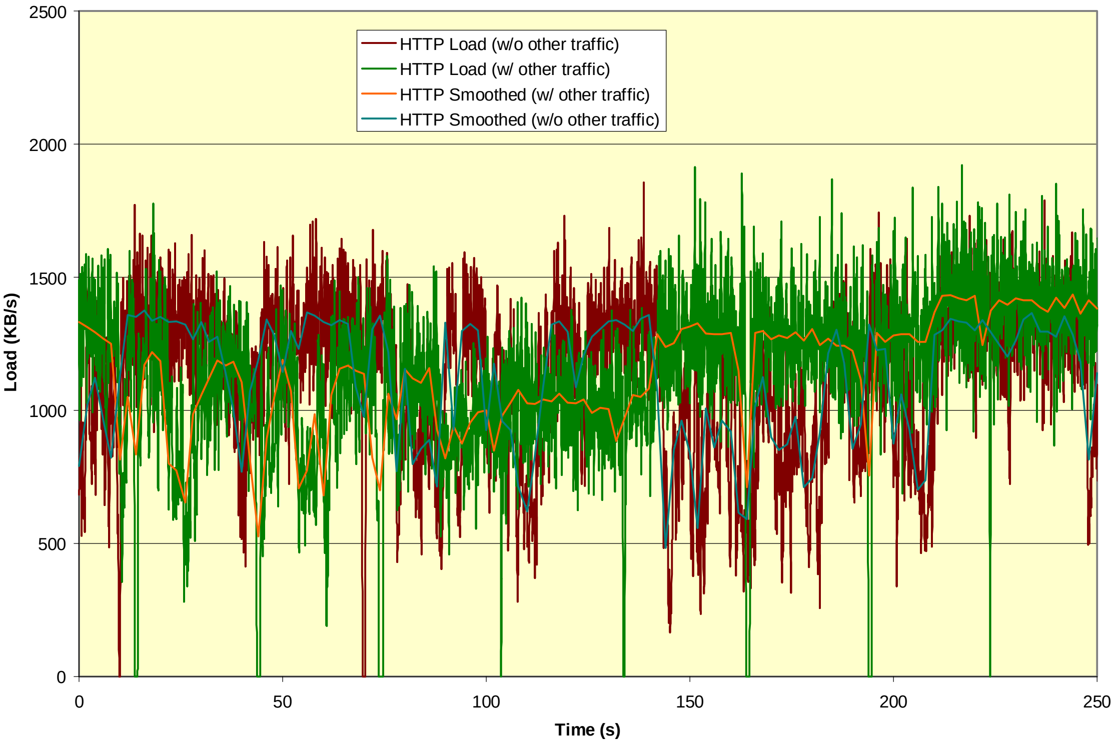


Figure A.20 HTTP Load with and without other traffic types present.

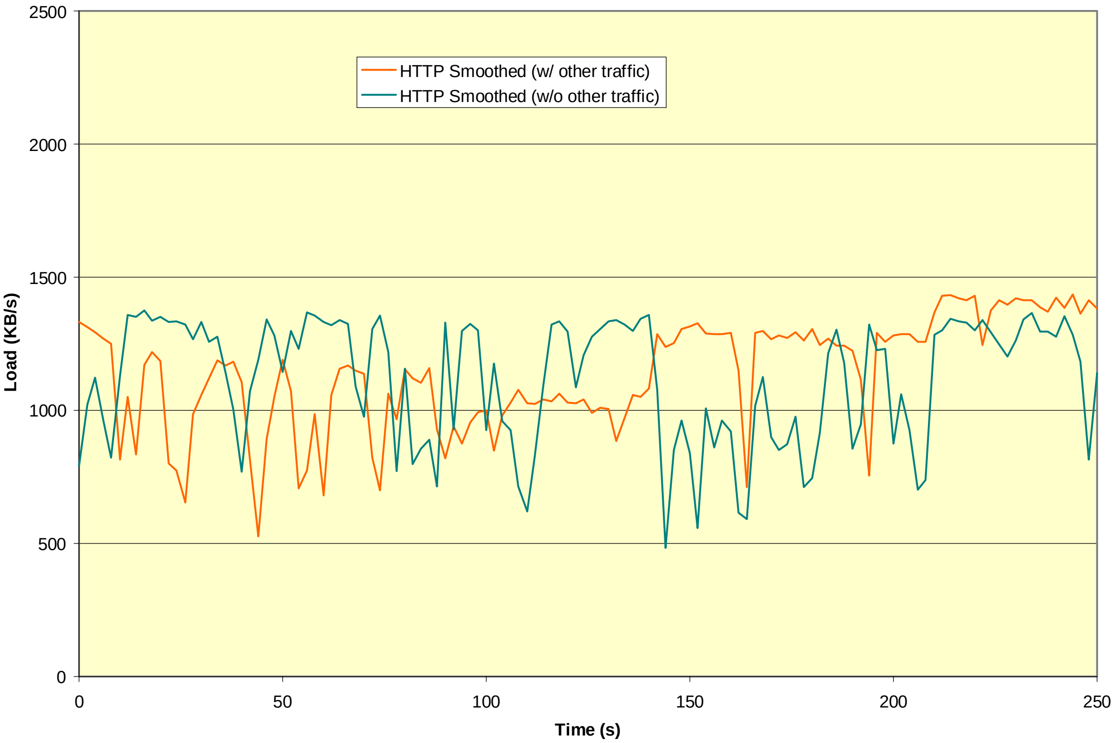


Figure A.21 HTTP Average Load with and without Other Traffic Types

Note that in Figure A.21 and Figure A.20 the HTTP behavior (i.e. generated load) in the presence of other traffic types can be better than the HTTP behavior without other traffic types. This happens in the HTTP model because individual browsers continue to start independent of the behavior of other browsers. The duration of browsing sessions may increase because of increased response times due to congestion while sessions continue to start at the same rate. If this is the case, the number of overlapping sessions will increase. More simultaneous browsing sessions increase the application level demand on the network during a given interval. Recall that the duration of the browsing sessions increases because of the need for retransmissions due to congestion. Because the HTTP flows are numerous as well as short-lived and low bandwidth the TCP congestion window may have limited effect reducing the average load in response to drops. However, the resulting retransmissions increase the load on the network.

Moreover, HTTP flows are particularly sensitive to loss because of their short lives. Because of their short-lives the synchronization handshake represents a significant fraction of the connection's packets. Since HTTP flows represent 75% of the flows in the Internet this means a significant fraction of all packets are associated with TCP initialization. TCP is particularly sensitive to loss during initialization and losses result in time-outs on the order of seconds. Many HTTP flows have lifetimes less than one second. As a result, drops during initialization can result in order of magnitude increases in the response time for an HTTP request.

Summary

The HTTP traffic generators model the application level behavior of web-browsers and generate network traffic based on this model. The model considers user-think time, average page sizes and the number of embedded objects associated with a given page. The model also reacts to network conditions. At the transport level, TCP responds to network congestion by adjusting its congestion window to control the load placed on the network. At the application level, the browser only initiates requests for new objects or pages when it has received the necessary information from previous requests. The network performance determines how long the model must wait for the objects to arrive.

The HTTP model offers a traffic pattern that is variable for both single flows and in aggregate over small and large time scales. It generates load capable of consuming 90% of the link's capacity.

3.4.2. BULK

In contrast to the “web mice” presented by HTTP, BULK transfers are the "elephants" of the Internet. Although small in number, these large, long-lived flows represent a significant proportion of the Internet traffic (after the world-wide web traffic). While WWW traffic represents 75% of the bytes flowing across the internet and 70% of the flows, FTP represents 5% of the bytes (25% of what remains after WWW) but only 1% of the flows. These flows are typically both long-lived and high-bandwidth.

These flows represent the opposite extreme from the TCP traffic presented by HTTP. They last long enough to experience the effects of congestion windows and are able to consume the available bandwidth of the network. The number of flows is 360 total, 60 for each of the 6 clients. This is 12% of the number of maximum clients possible with the HTTP model. However, because these flows are always active, the number of flows is likely much greater than the average number of active TCP connections in HTTP. Many flows were used so that time-outs would be less common. (In early experiments with a small number of TCP flows, the TCP load was very low in all the configurations because many of the flows were in retransmission time-out. With the larger number of flows, if any flow, or small group of flows experience retransmission time-out other flows are able to maintain the load.) Even with the maximum RTT (120ms + 100ms) and 16K windows, only 16 TCP flows are required in order to consume the capacity of a 10Mb/s link so these flows are capable of fully-utilizing the network.

This model focuses only on modeling the transport level behavior associated with BULK flows, focusing purely on TCP's reaction to network conditions. Starting and stopping different flows is not considered. Instead all flows transfer data as fast as conditions allow and run for the duration of the experiment. Each of the 6 clients establishes 60 connections, 12 to each of 5 servers, for a total of 360 flows (60/client, 72/server). These servers establish flows that all start nearly simultaneously and simply send as fast as possible (at the application level) for the duration of the experiment. There is some delay associated with starting 60-70 processes on a given machine so the flows do not actually start at the same instant. To address that issue the BULK traffic model runs for 12 minutes before any other traffic type is introduced or measurements taken. At the application level, each server is always transmitting so TCP's congestion avoidance mechanism must regulate the actual transmission rate to avoid congestion. Even so, the aggregate effect of 360 flows constantly oscillating between reducing their load in response to loss and increasing their load to probe for available capacity are able to exceed the capacity of the bottleneck link. Figure A.22 shows the load generated by this BULK traffic on the 100 Mb link before the bottleneck router when no other traffic types are present.

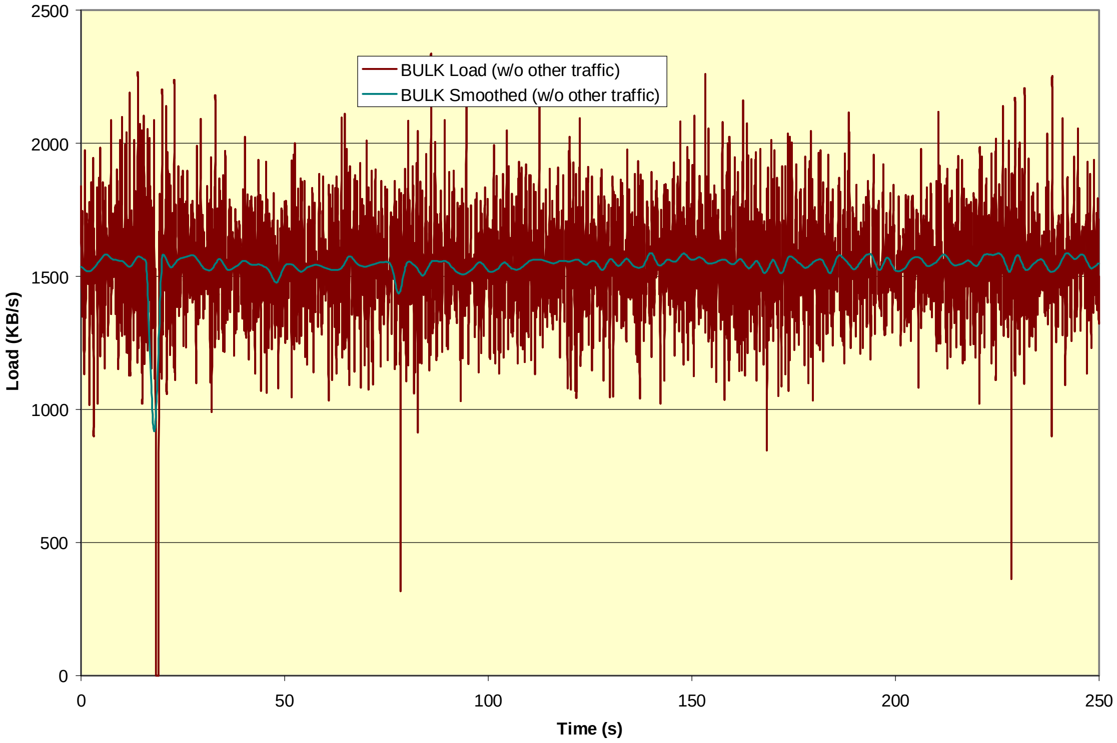


Figure A.22 BULK Load with no other traffic types (but with 10Mb/s bottleneck link)

Clearly, the BULK traffic is capable of exceeding the bottleneck link's 1.2 MB/s capacity. However, one should not assume this constant load of 1.5 MB/s reflects unresponsiveness on the part of the BULK traffic load. Rather, it is the result of a large number of TCP flows testing the network's available capacity by probing. As some flows reduce their congestion windows in response to loss, others are increasing their congestion window and taking advantage of the capacity released by those flows that are backing off. This probing behavior also explains the short-term variability seen in the average over 100 ms intervals. BULK's responsiveness is demonstrated when other traffic types are added to the mix. Figure A.23 illustrates this phenomenon.

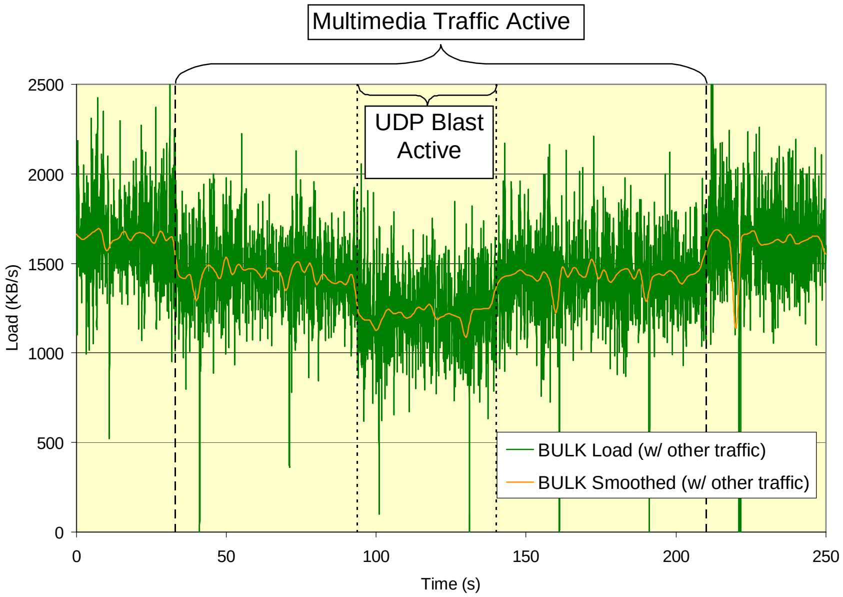


Figure A.23 BULK Load in the Presence of Other Traffic Types

Recall this plot shows the load on the *inbound* link to the router, before contention for the bottleneck link occurs. The changes in the load are the result of the traffic generators changing the load they are generating (by adjusting the congestion window in the sender), not the direct result of drops in the router's queue. The BULK traffic load clearly decreases as each new traffic type, first multimedia, then the blast, is introduced and increases as they are removed. This data was actually gathered while the CBQ scheduling algorithm that offered TCP a significant minimum allocation of the bottleneck link's capacity was running. Otherwise, as shown in Chapter III, TCP's responsiveness would be evident as a complete collapse of the TCP load when the UDP blast is present.

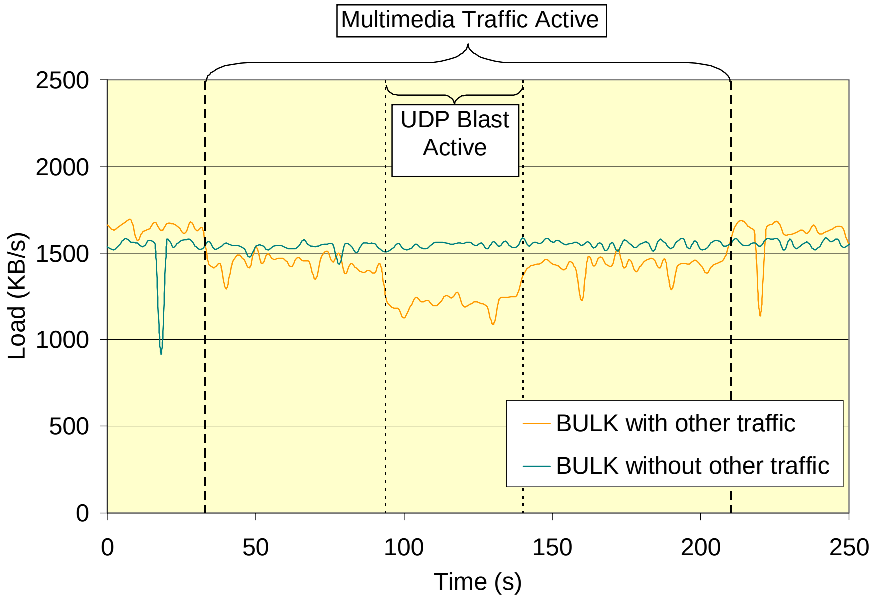


Figure A.24 Average Bulk Load with and without Traffic

Figure A.24 compares the load generated by the BULK traffic generators with and without the other traffic types. This simply allows for direct comparison of the offered BULK load in both scenarios.

4. Traffic Mixes

Combinations of the traffic types established above are mixed and then the effects of the different router-based queue management techniques on each mix are analyzed. Most of the experiments are composed of a set of measurement intervals. First measurements are collected with TCP and multimedia, and then with a UDP blast added. This allows one to consider separately the interaction between TCP and multimedia and the effects of aggressive, unresponsive flows on each.

All possible combinations of the traffic types discussed above are not explored. First, only one of each type of TCP, multimedia, and *other* is used in each traffic mix. Second, specific settings are used for each traffic type (e.g., 3,000 HTTP clients, 360 BULK flows, six Proshare flows, 4 MPEG flows, and 1 UDP blast flow of 10Mb/s) as the standard settings for each traffic mix. These settings change only as necessary to illustrate specific conjectures. The focus in this work is not exploring wide ranges of traffic conditions but on studying the behavior of a few specific traffic mixes in detail to better understand the effects of each algorithm. Additional concerns include understanding the effects of different parameter settings for each queue management algorithm and, for comparability, reproducibility.

4.1. Timing of when each type of traffic is introduced

All of the traffic mixes follow the same basic pattern. TCP traffic generators start and are allowed to stabilize. Then the multimedia traffic begins. After a brief interval, the blast traffic begins. Then the blast traffic stops, followed by the multimedia, and finally the TCP, with a delay between each action. During each experiment two periods are considered, one including TCP and multimedia, and the other including all three traffic types.

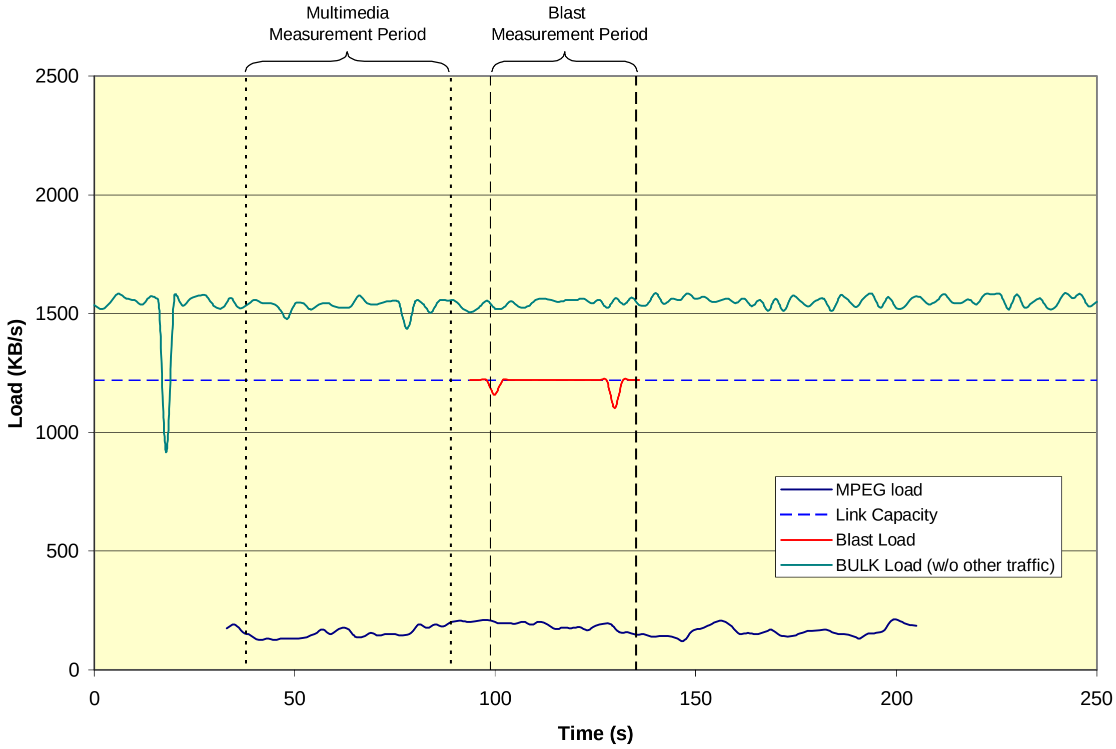


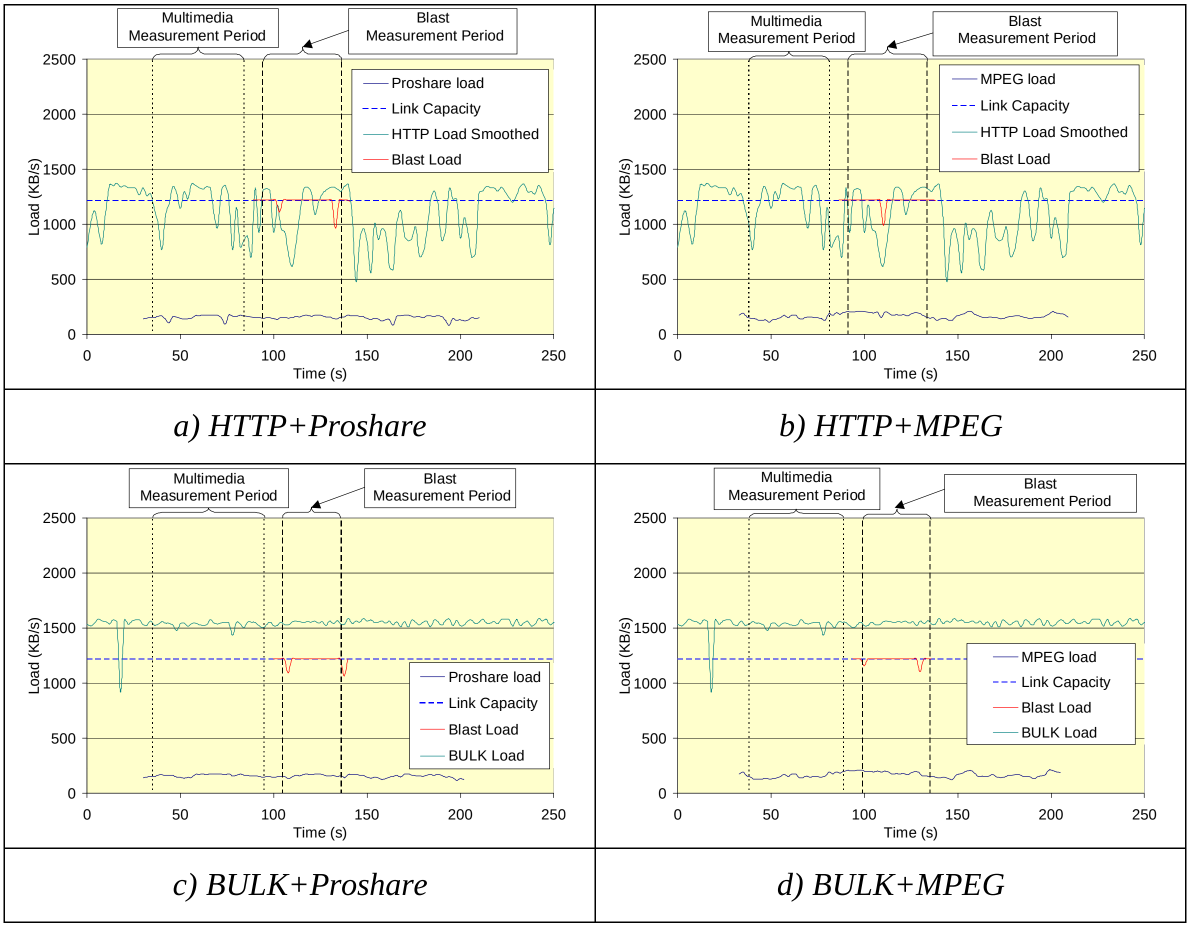
Figure A.25 Traffic Mix for BULK+MPEG

Figure A.25 illustrates the timing of the introduction of the different traffic mixes and the measurement periods. A BULK+MPEG experiment serves as an example. (The type of TCP and multimedia traffic used determines the name of the traffic mix. In this case the TCP type is BULK and the multimedia type is MPEG so it is BULK+MPEG.) The figure shows the typical potential load on the inbound link during such an experiment. All loads are averaged over 2 second intervals. The plot also shows the capacity of the bottleneck link, to allow easy assessment of the overload created by the traffic mix. In the case of MPEG and Blast, since those traffic types are unresponsive the load is the same across all experiments. Recall that because TCP is responsive its actual load does change based on traffic conditions. The plot above shows the potential BULK load that can be generated when no other traffic types are present, not the actual load during an experiment.

Figure A.25 shows the timing of the BULK+MPEG traffic mix and the loads presented by the different classes of traffic. This figure is translated to the time that the network monitoring begins. At that time (0 on the x-axis), the BULK traffic has been running for 12.5 minutes before the period shown. After the network monitoring begins, the MPEG traffic is introduced around 32 seconds later. The blast is introduced around time 93 seconds. The blast stops at time 140 seconds and the multimedia at time 210 seconds. Finally, the TCP stops at time 314 seconds.

This analysis focuses on two measurement periods. The first is the period during which multimedia and TCP are running, called the *multimedia measurement period*. The second, the *blast measurement period*, is the period during which multimedia, TCP, and the blast are running. The multimedia measurement period begins 5 seconds after the multimedia is introduced and ends 5 seconds before the UDP blast begins. The blast measurement period begins 5 seconds after the blast begins and ends 5 seconds before it stops. Measurements are gathered during each of these intervals. The metrics themselves are discussed in section 5. After the blast terminates, the multimedia stream and then the TCP streams also stop at intervals of several minutes. The termination of the generators is of no consequence for the analysis and they are deactivated in this way simply for the convenience of the experimenter.

These measurement periods and general timing patterns apply independent of the specific traffic mixes as shown in Figure A.26. Note that the loads generated, particularly those generated by HTTP, do vary significantly between measurement periods. This is not a concern however, as the focus is on comparing a single traffic mix and measurement period across multiple algorithms, not on comparing different traffic mixes or measurement intervals to one another. In other experiments a single traffic mix and measurement period are compared across multiple parameter settings for a given algorithm. But, once again, variation between measurement periods or traffic mixes is not a concern.



a) HTTP+Proshare

b) HTTP+MPEG

c) BULK+Proshare

d) BULK+MPEG

Figure A.26 All Traffic mixes

There is one variation in this method that is not shown on these plots. The variation lies in the increased time TCP flows are allowed to stabilize in the case of HTTP. This stabilization is necessary to reach a point where the HTTP flows are starting and stopping in a manner comparable to the Internet. Consider that there is no time at which a network link suddenly goes from an idle state to thousands of active HTTP flows. This sudden start is quiet chaotic and requires time to stabilize. Because the HTTP model requires 20-25 minutes to stabilize, allow the HTTP traffic generators run for 25 minutes before the interval shown. Although this stabilization period is important to obtain consistent results, the difference in time required for HTTP and BULK to stabilize has no effect on the analysis. As a result, all results are translated to have this startup period before time 0.

5. Data Collection

Evaluating a queue management algorithm consists of three steps: creating conditions to evaluate the algorithm, collecting data about the performance of the algorithm, and analyzing that data to extract useful metrics. This section focuses on the data collection techniques used to collect data on the application-level performance, network performance, and the behavior of the router. At the application level, instrumented traffic generators record information such as frame-rate, packet-rate, loss rate, and latency. At the network level, packet-level tracing is used to record the traffic traversing specific links in the network. The infrastructure for data collection and the general information collected are described below. The analysis and specific metrics are considered in more detail in Section 7.

5.1. Network Monitoring

Packet level traces are captured at selected points in the network using **tcpdump**. All packets are recorded to a trace file and filtered for selected flows during post-processing. Figure A.28 shows the monitoring points. One machine monitors both networks. This uses the same clock to generate the time stamp for both traces, allowing the traces to be synchronized and compared during analysis. This machine has the capacity to monitor the highest loads used in these experiments and capture 99.5% of all packets. Traffic on each logical network passes through an Ethernet hub to provide a shared medium that can be used as an access point for monitoring. Traces are recorded during the multimedia and blast measurement periods indicated in Section 4.1.

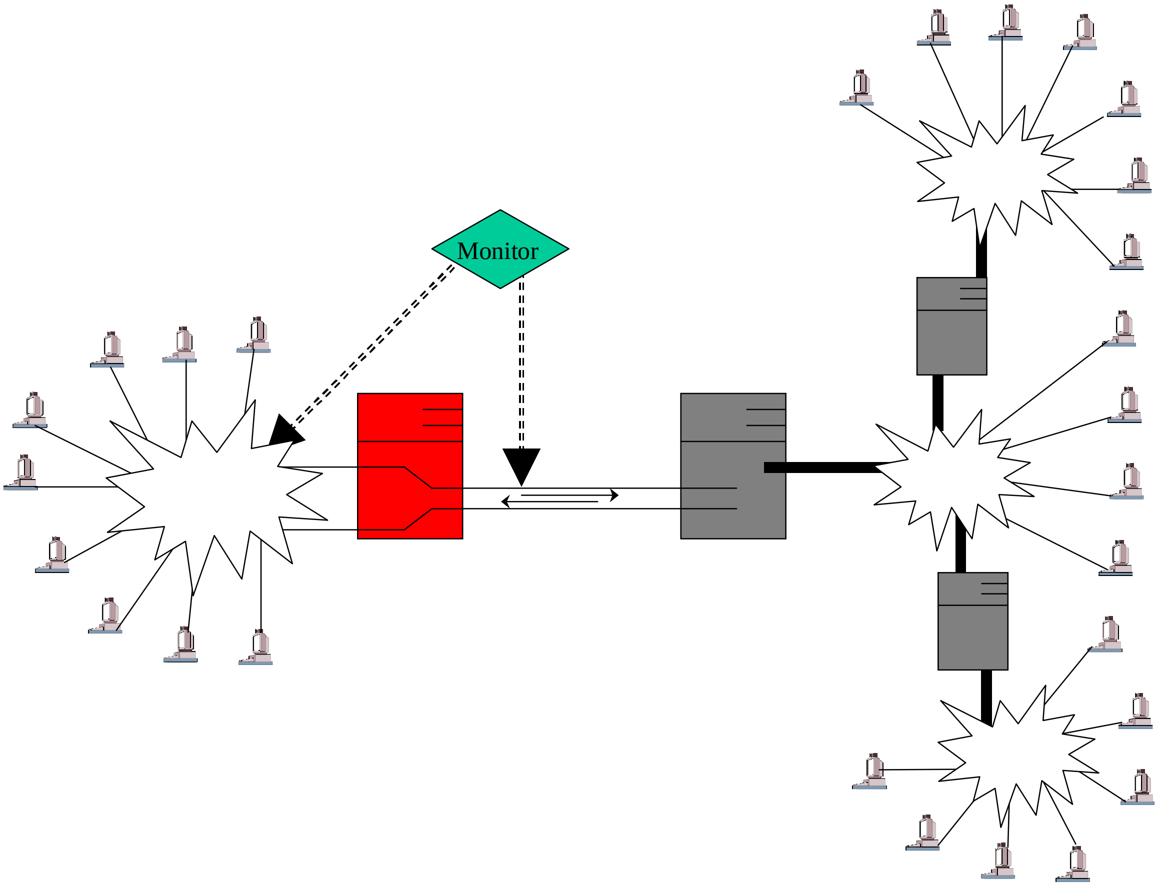


Figure A.28 Location of Network Monitoring

5.2. Application Level Monitoring

All of the applications in these experiments are actually traffic generators. As such, it was easy to instrument and modify the applications to acquire application-level measurements. Specifically, packet payloads included sequence numbers and timestamps to allow detection of loss, out of order delivery, and latency at the receiver for protocols that could not otherwise report such information. Network monitoring alone was used to collect the necessary data for TCP and *other* traffic as well as for some multimedia data. However, most multimedia measurement data were gathered by application level instrumentation.

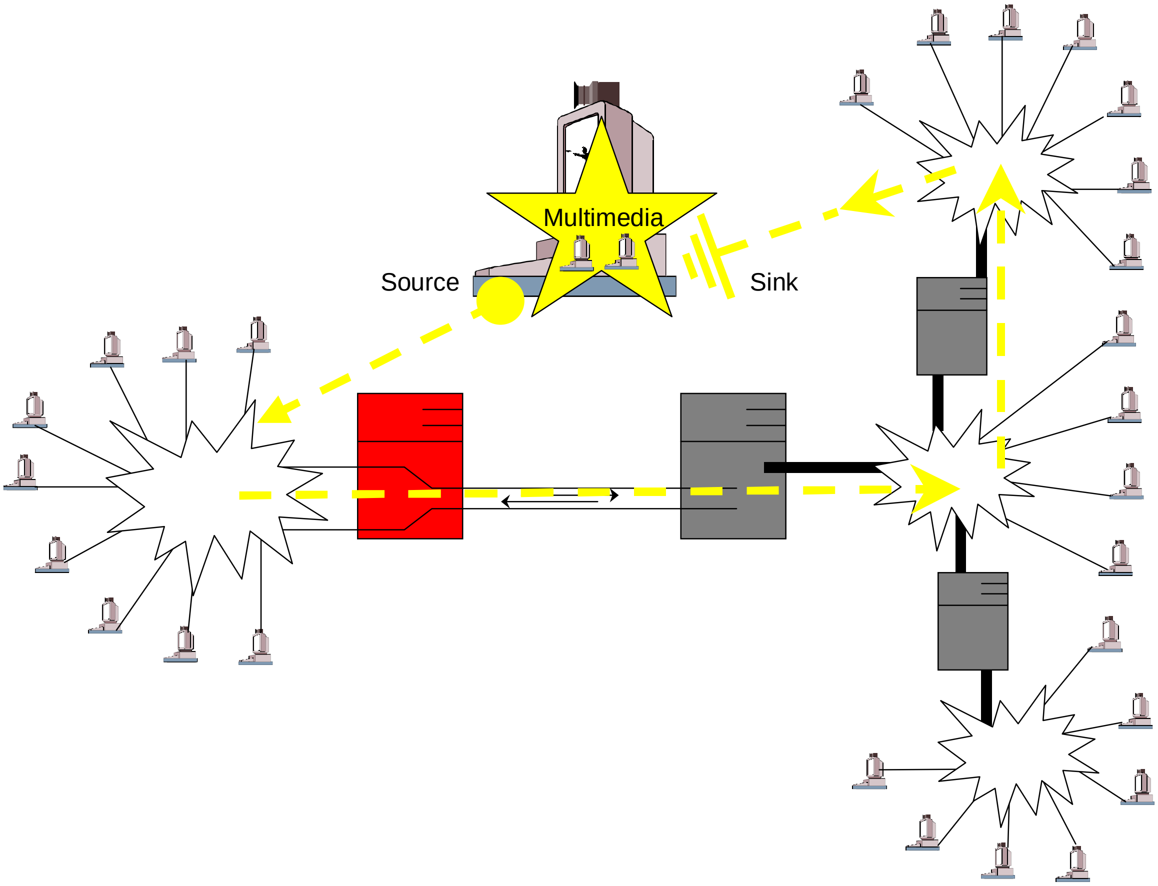


Figure A.29 Location of Multimedia Traffic Generator

As shown in Figure A.29, one multi-homed end-system acted as the source and sink for multimedia traffic. Multimedia traffic was forced to traverse the network using static routes. This configuration used one clock to record the time packets were transmitted and the time they were received, offering accurate one-way latency measurements. The specific data collected varied between the two media types and is described below.

5.2.1. Measuring the Performance of Proshare

The Proshare traffic generator is instrumented to report packet-level statistics at one second intervals. The Proshare simulator is based on a raw packet trace so it does not distinguish between frames of audio and video. The generator is also aware of the minimum MTU size along the path and does fragmentation at the application level so that packet drops can be determined accurately at the application level. Over each interval, the generator reports the current time, the bits received, the packets received, the bit rate, the packet rate, and the average latency of the packets delivered. It also reports the cumulative number of dropped packets and the cumulative number of out-of-order packets through the interval. While the generator records this data for the life of the application, the data associated with the two measurement periods is sliced out and examined independently. The data from these “slices” is used to plot behaviors of both individual and aggregate flows. To compare these runs with other experiments, the samples are averaged over the measurement period and the class's average performance reported over the entire period. Specific metrics considered are discussed in section 7.

5.2.2. Measuring the Performance of MPEG

The receiver for the MPEG traffic generator is specifically designed to report performance statistics. It is aware of the expected GOP structure and uses that information to determine which arriving frames could be decoded based on the other frames that have arrived. Like the Proshare generator, the MPEG generator also is aware of the minimum MTU size along the path and does fragmentation at the application level so that packet drops can be determined accurately. Each packet includes a time-stamp and a sequence number to allow for calculating latency and detecting out of order delivery. The generator reports only the network latency, the time between when the packet is transmitted and when it is received. It does not consider the latency involved in decoding the frames, particularly time spent waiting for frames that are the target of forward references in bidirectional encodings.

The MPEG traffic generator reports statistics at the end of every sampling interval. For these results, the sampling interval was one second. The statistics reported include a timestamp, and some statistics averaged over the previous interval and others averaged over the duration of the run. The statistics reported for the current interval include the playable frame rate, the actual frame rate, throughput, average latency, packet loss rate, and packets received per second. The generator also reports a running average of the frame rate over the life of the stream. The details of these metrics and the ways in which this data is analyzed are described in Section 7.

5.3. Router Queue Management Overhead

As the queue management algorithm increases in complexity one must consider the effect of the additional complexity on the per-packet processing time. As the enqueue and dequeue operations grow increasingly complex, the number of operations required to process each packet can lead to the packet processing overhead dominating the time it takes a packet to traverse the router. If packet processing time grows too high, it becomes the limiting factor for throughput on the outbound link instead of the physical transmission speed of the link. Evaluating this overhead is beyond the scope of this work but we report some limited observations.

5.4. Summary

Network behavior is monitored with packet traces, and application performance with application level instrumentation. With the infrastructure in place to collect this information, next consider the statistical issues involved with analyzing this data.

6. Statistical Issues

In this work five different router queue-management algorithms are compared to one another over a large set of parameter settings and traffic patterns. Because the results are compared between experiments, reproducibility is an issue. To address the need for reproducibility, controlled traffic mixes described in Section 4 are used in the closed network environment described in Section 2. Additionally, because the use of random sampling in the traffic generators creates variations between experiments, care must be taken to insure the results are representative and not single instances of extreme behavior. Addressing this concern involved conducting multiple experiments for each set of parameter settings. In most cases each experiment was repeated five times. In a few instances there were more runs. Multiple instances of each experiment were also used because the experimental setup is complex and components may sometimes fail in ways that are most easily detected by changes in the experimental results. (Once detected, one can confirm that the change was due to a failure, not a pathological state of the algorithm.)

Because the volume of data makes comparing the second-by-second behavior of individual runs impractical, average behavior over a given measurement period is considered and compared, both over time and across multiple runs. However, select instances are also compared for illustrative purposes. For the same reasons, in some instances the analysis focuses on the behavior of either an entire traffic class in aggregate or the behavior of a single representative flow, but never the individual behaviors of all the flows.

For a given metric and experiment, the samples for that metric are averaged over the selected measurement period. That value is then averaged with the same values for other runs of the same experiment (i.e., with the same algorithm, traffic mix, and parameter settings). One value that represents the average performance of that metric across all runs with the given configuration of the queue management algorithm, traffic mix, and algorithm remains. For example, consider comparing TCP throughput during the blast measurement period. First, the average TCP throughput over small (100 ms) intervals is recorded during the experiment. After the experiment is complete, the values during the blast measurement period are extracted and averaged to determine the average TCP throughput during the blast measurement period for that instance of the experiment. The average TCP throughput values for all runs with the same algorithm and experimental conditions (queue management parameters, measurement period, and traffic mix) are then averaged together to produce a single value. Comparing that value to the same value for other algorithms and/or experimental conditions highlights the differences. Throughout this work the average performance of multiple executions of the same experiment are presented. Usually the analysis considers only the aggregate performance of a particular traffic class or, occasionally, of a single, representative flow. Limiting the evaluation to long-term average behavior neglects some aspects of performance. Averaging over intervals on the order of 30 seconds removes the opportunity to examine the short-term or instantaneous behavior of the system. Further, examining aggregate performance prevented examination of the fairness between flows in a given class. However, this type of analysis is beyond the scope of this work.

7. Performance metrics

Throughout this dissertation the focus is on comparing the effects of different queue management algorithms on a wide range of network and application metrics. They range from throughput and latency at the network level to playable frame-rate in the application. In this section the metrics are defined, techniques for deriving them described, and their significance explained.

7.1. Bandwidth Usage Measures

Bandwidth usage refers to the amount of data crossing a network segment over a given time interval. Bandwidth usage is considered from many perspectives and is usually referred to in bytes per second, though it is also referred to in bits per second. At times packets per second may also be considered; however, packets per second is a less precise measure unless the average packet size is known.

Bandwidth usage on a particular link or series of links is a somewhat ambiguous measure of end-to-end performance. To clarify the basic terms used to refer to different kinds of bandwidth are defined here:

- *Capacity* is the bandwidth available on a given link.
- *Load* is the bandwidth demand placed on a network element or the bandwidth demand generated by a given flow or class.
- *Throughput* is the amount of data traversing a network element over a given interval. The load at a router is determined by the throughput on the inbound links. However, the throughput of the router is the limited by the capacity of the router and its outbound link.
- *Utilization* is the lesser of the ratio of load to capacity or 100%. Since the utilization of a given network element may not exceed one hundred percent, some data may be lost if the load is greater than the capacity. These drops limit the usefulness of throughput as a metric in some cases.
- *Loss rate* is an expression of the fraction of traffic that is discarded at a given network element. Loss rate is most commonly used regarding unresponsive and, usually, unreliable protocols. (Loss rate also effects responsive protocols but its effect on the actual goodput is the key measurement in those cases.) Also loss-rate commonly refers to packets per second, not bits per second.

In addition to these basic terms, there are a few others that are more complex. While throughput is a straight-forward measure, it fails to consider the overall usefulness of the packets transiting the link. That is, packets that are ultimately dropped further along the network path or that are duplicates of data that has already traversed the link, are not a good use of the link’s capacity. *Goodput* is used as a measure of the traffic that is useful and is strictly defined as the rate at which data is ultimately acknowledged by the TCP receiver. Considering goodput leads to a metric for considering the effectiveness of a protocol's responsiveness. Responsive protocols are supposed to be able to adjust their load to match the available capacity of the network (i.e., the available capacity at the bottleneck link). If they succeed in this, the responsive traffic's load at the source should equal its throughput and end-to-end goodput. The effectiveness of the response is a measure called *efficiency.* Efficiency is the ratio of goodput to load.

Finally, note that although *capacity, load, throughput, utilization, goodput, efficiency,* and *loss rate* were used to refer to all of the traffic on a single network element above, those terms also apply relative to subsets of flows and across multiple network elements. For example, comparing the end-to-end goodput for all TCP flows to their load on the source network may be used determine TCP's efficiency. Alternatively, the loss rate for a representative multimedia flow may be considered. Below the derivation of each measure is explained.

Most of the bandwidth metrics are gathered from packet traces (Section 5.1) collected using **tcpdump.** Another tool, **ifmon**, reads the tcpdump file format, filters packets according to specified rules, and generates reports of the bandwidth consumed by those packets. Filter rules use protocol type, source and destination IP addresses and port numbers to group packets into classes of TCP, multimedia, or other. The tool reports average bandwidth. It reports this bandwidth in three forms. At configurable intervals, it reports the average over the interval as well as the running average up to that point. It also reports the average over the entire period. Unless otherwise stated, the tool samples over intervals of 100ms. This interval results in averages that reflect some amount of the burstiness of the traffic while still capturing the average behavior. However, most of the results presented compare the average bandwidth over the entire period to the same value in other experiments.

Because a closed and controlled network is used, simplifying assumptions can be made to derive these metrics. First, capacity is a known constant for each link in this network configuration. Similarly, utilization can be measured simply by dividing the load at a given network element by the capacity of that element. Since all the traffic generator sources are connected to the router's inbound network and that network is over provisioned the bandwidth on that link is the load being generated by the traffic generators. As a result, generated load is simply the bandwidth used on the router's inbound link. Further, knowledge that there is no source of loss beyond the bottleneck router simplifies the consideration of loss, goodput, and efficiency as well. Since end-to-end load can be derived from the inbound link and end-to-end goodput from the outbound link, the ratio between the two loads is the efficiency. Finally, the difference between the throughput of the outbound link and to the load on the inbound link divided by the load is the loss rate. Although we can calculate loss rate by comparing load to throughput this is not the way it is calculated here. Instead, the analysis relies on application level instrumentation. The multimedia applications are instrumented to report the number of packets received and dropped over every one-second interval. This information is used to calculate the multimedia loss-rate over a given interval.

7.2. Latency

With interactive multimedia, end-to-end latency has a significant impact on the quality of the communication. If latency is too high, interaction becomes difficult or impossible. End-to-end latency is measured in the multimedia applications by recording a time-stamp at transmission and receipt of each packet. The transmission time-stamp is carried in the payload of the packet so it can be compared to the time the packet is received. Latency can be calculated by taking the difference between these two values. Recall that using the same end-system as sender and receiver allows the same clock to be used for both time-stamps, insuring an accurate measure of the end-to-end latency for the system. Further, the only bottleneck point in the system is the router. As a result, the only factor contributing to any significant change in the end-to-end latency is the queue-induced latency on the bottleneck router. As such, end-to-end latency measurements can be compared to one another in order to compare the queue-induced latency associated with different algorithms or parameter settings. As with bandwidth, the average multimedia latency is reported across a given measurement period in multiple runs of the same configuration.

7.3. Frame-rate

Delivered frame-rate is a key performance measure for multimedia data. Two measures come into play here: actual frame-rate and playable frame-rate. The actual frame-rate is based on the number of frames that arrive successfully at the receiver. This measure ignores any inter-frame encoding concerns such as lost reference frames. The playable frame-rate does consider inter-frame encoding and only counts frames that can be decoded and played successfully. It also does not include frames that arrive out of order if the preceding frames have already played.

In this work, Proshare traffic is generated and measured only at the packet level so no frame-rate information is available. Because of this, the resulting frame-rate for Proshare cannot be computed since the frame a lost packet is associated with is unknown. As a result one can not tell if the loss affects the audio or video frame-rate. Moreover, two lost packets may be part of different frames or a single video frame. Thus, only packet rates are reported for Proshare.

In contrast, the application level instrumentation of the MPEG traffic generator does report the true frame-rate and playable frame-rate. Since the encoding scheme of MPEG is based on dependencies between frames, the loss of one frame, or part of one frame, also makes the frames that reference that frame useless. As a result, the distribution of drops can have a significant effect on the playable frame-rate, even as the drop-rate remains constant. Given the standard GOP of IBBPBB used by this trace, a single lost fragment in an I-frame results in the effective loss of all of that GOP's frames in the playback stream. Because some AQM algorithms may have a bias against large packets (or sequences of fragments) like I-frames, this is a particularly relevant measure.

8. Summary

This dissertation empirically compares the behavior of CBT to other algorithms under the same network conditions. In addition, different parameter settings are also compared for a given algorithm. This appendix explained the details of the experimental methodology used in conducting these experiments including network topology, traffic mixes, data collection, and metrics considered.

APPENDIX B. CHOOSING OPTIMAL PARAMETERS

Before experiments can be conducted to compare algorithms to one another, fairness of the comparison must be assured. Towards that end, experiments were conducted to determine the parameter settings which give optimal performance for each algorithm. Optimal performance is defined for each algorithm in terms of the goals of that algorithm. For example, RED's goal is improved feedback to responsive flows and shorter queues. On the other hand, FRED's primary goal is to provide fair sharing of network resources. When there is no discernible difference in the primary metrics secondary metrics are considered. For example, with RED the focus is first on TCP goodput as a measure of the efficiency of TCP. (Goodput is explained in Appendix A.) If RED provides better feedback to the senders they should be able to more accurately adjust the loads they generate, resulting in higher TCP goodput. If the goodput is equivalent for multiple parameter settings, then another metric, network latency, is used to distinguish between those settings. Network latency is a reflection of average queue length, and minimizing average queue length is another of RED's goals. In contrast, when selecting the optimal parameters, metrics related to multimedia performance under RED are not emphasized because improving multimedia performance was not a goal of RED's design.

Although selecting optimal parameters for each algorithm is important, it was also necessary to limit the number of parameters considered both for practical concerns associated with finite experimental resources and because some of the parameters are known to have more effect on the performance than others. For RED, FRED, and CBT the experiments focused on the threshold settings while holding the weighting factor, the maximum drop probability, and queue length constant. For other algorithms, parameters that served the same general purpose as the thresholds were explored. For CBQ, a wide range of bandwidth allocations was considered. For FIFO the only parameter, the queue size, was varied. Details of the analysis of each algorithm are discussed separately below.

First, the general experimental methodology is discussed. After establishing the approach, the analysis of each algorithm is presented in turn and finally, the optimal parameter settings are summarized.

1. Methodology

To determine the optimal parameter settings, these experiments used a standard network configuration in a private laboratory network as described in Appendix A. In that network configuration each queue management algorithm was tested on a bottleneck router connecting a 100 Mb/s and a 10 Mb/s network. The queue management algorithm manages the queue servicing the 10 Mb/s link. The traffic mixes used to evaluate each algorithm and the reasons for selecting those mixes are detailed in the discussion of each algorithm. Detailed explanations of traffic mixes as well as the metrics measured can be found in Appendix A.

In these experiments each algorithm was evaluated under different traffic conditions and varied combinations of parameter settings. Initially a range of parameter combinations was selected based on knowledge of the algorithm and previously published results. Then one experiment was conducted for each of the selected parameter settings and the results evaluated. Some of those parameter settings were selected simply to confirm the hypothesis that those settings would result in sub-optimal performance. After evaluating these initial runs, the experiments with parameters in a range that seemed likely to yield optimal results were repeated and the focus shifted to selecting the optimal parameter combination from this set of repeated experiments. Each experiment was repeated five or six times to lessen the effect of any anomalous performance from a single run. Table B.1 shows an example of the number of executions of each experiment in the evaluation of the RED algorithm. For RED, only the minimum and maximum threshold parameters were varied. Each column is a setting of the maximum threshold and each row represents a setting for the minimum threshold. The values in the table represent the number of runs for each parameter combination. For example, there were 6 runs of the (Th_{Min}, Th_{Max}) combination

(5,60) and 1 run of the combination (5,30). Note that although the matrix shows all parameter combinations, only a subset of these was considered. The empty cells (e.g. (0,10)) indicate that no experiments were conducted for that parameter combination. Many of the empty combinations are invalid because the maximum threshold would be less than the minimum threshold. These are darkly shaded in the table. In other cases combinations were skipped because they would not add any information and practical concerns motivated limiting the number of experiments conducted where possible. Those cells are simply blank. Such a grid is presented for each set of experiments.

Min	**Max**									
	10	**15**	**20**	**30**	**40**	**60**	**80**	**100**	**120**	**180**
0				1	5	5			5	5
2				1	5	5			5	5
4				1	5	5			5	5
5	1	2	1	2	5	6	6	6	6	5
10	▨	2		1	5	5			5	6
15	▨	▨		2	5	5			5	5
20	▨	▨	▨	1	5	5			5	5
25	▨	▨	▨	1	5	5			5	5
30	▨	▨	▨	▨	5	5			5	5
60	▨	▨	▨	▨	▨	▨	5	5	6	4
120	▨	▨	▨	▨	▨	▨	▨	▨	▨	5

Table B.1 Example Showing the Number of Runs for Each Parameter Combination.

Basing the decision to eliminate a parameter combination on a single run of an experiment may seem unsound. However, no parameter combination was ever discarded on the basis of one run alone but, rather, on the pattern indicated by runs of a group of similar parameter settings. For example in Table B.1, after conducting one run of each parameter setting it was evident that all experiments with a maximum threshold less than 30 had poorer TCP performance than those with a maximum threshold greater than or equal to 30. Because this result was consistent across all such runs and because it was expected based on knowledge of the algorithm (explained in section III.3.2) those parameters were eliminated. Additional experiments were conducted only for maximum threshold settings of 40 or greater. Although the approach was iterative, in this document only the final results that include all runs are shown. This presentation does not affect the resulting analysis or conclusions.

Having established the experimental methodology, now proceed to consider the experiments themselves. First, FIFO's performance relative to queue size is considered. Next RED and FRED and their performance relative to the maximum and minimum threshold settings are studied. Finally, the process for choosing the settings for CBT and CBQ is explained and those parameters are demonstrated to be optimal.

2. FIFO

The FIFO, or drop-tail algorithm, has only one parameter: maximum queue size (maxq). (The FIFO algorithm is explained in chapter III.) To choose the optimal value, the queue size is varied at the bottleneck link and the resulting performance is examined. Consider queue sizes of 30, 60, 120, 240, and 360 packets. Experiments were conducted for all four of the traffic mixes (Appendix A). The results were comparable for all mixes. For brevity a subset of those experiments is presented here. For each setting of maxq and traffic mix the experiment was repeated 5 times. The results for the selected metric were averaged over the measurement period. Those results were then averaged across all 5 runs. Plots illustrating the performance of different metrics as queue size was varied are presented below. All of the plots will show the maximum queue size, in packets, along the horizontal axis and the values of a particular metric along the vertical axis.

Latency was the first metric considered. The maximum queue size should determine an upper limit on the maximum average queue-induced latency. This limit can be calculated using equation B.1. Latency is a function of the average packet size, the maximum queue size (maxq), and the link capacity, *C*.

$$Latency = \frac{\overline{packet\_size} * \mathrm{maxq}}{C} \tag{B.1}$$

In this experimental setup the link capacity was 10 Mb. The average packet size varied as a function of the traffic mix and as a function of the traffic mix enqueued. However, the maximum packet size was 1,500 bytes, the size of the ethernet MTU. So the maximum average latency in milliseconds is 1.2 * maxq. In these experiments the average packet size is less than the maximum MTU so smaller latency values will often occur.

Figure B.1 shows the end-to-end latency during the blast measurement period (when all three traffic types are present). As the maximum queue size increases so does the average latency. The correlation is clearly linear. This occurs because the traffic load, particularly the *other* traffic, exceeds the capacity of the bottleneck link. As a result, the overload keeps the queue full almost all of the time during the blast measurement period. Consequently, the maximum queue size is limiting the average queue occupancy and, thus, the average queue-induced latency.

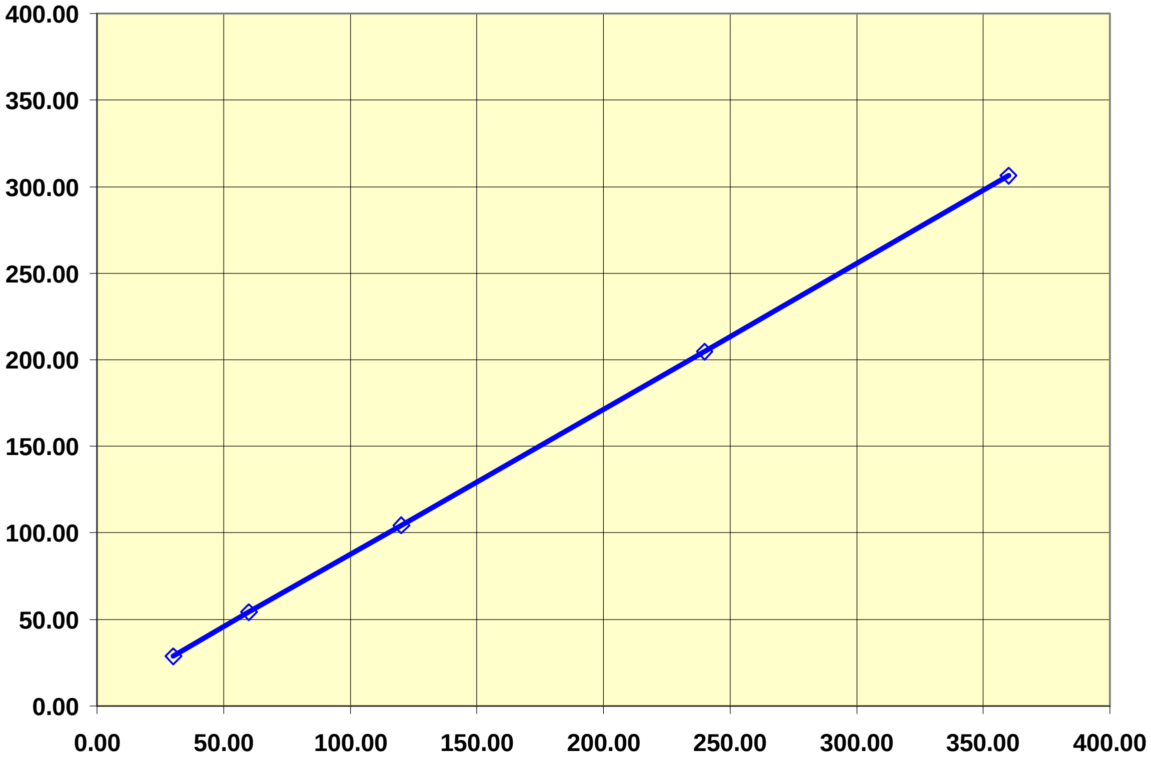
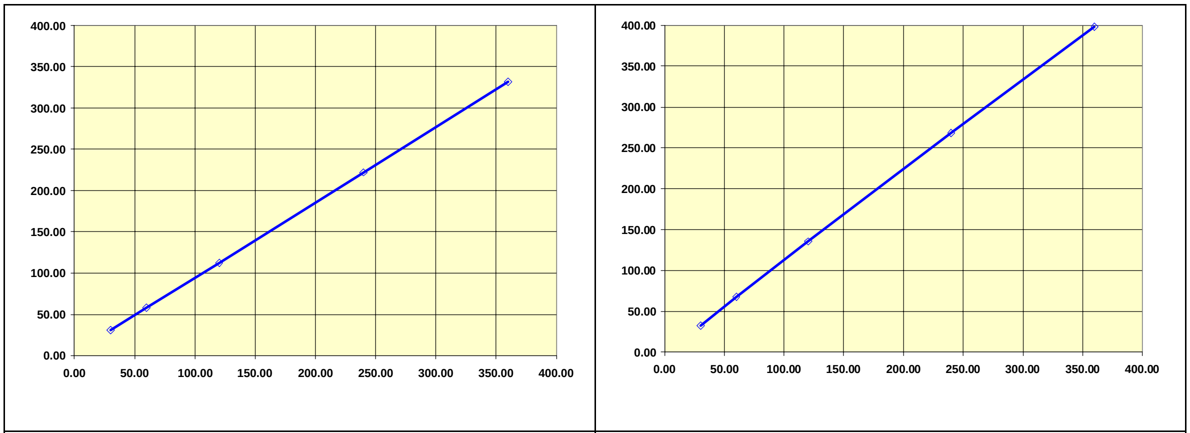


Figure B.1 Latency (ms) vs. Maximum Queue Size (packets) with FIFO during the Blast Measurement Period (HTTP-Proshare)

It is interesting to note that in the case of the BULK-Proshare traffic mix, the latency is higher during the multimedia measurement period than it is during the blast measurement period. Figure B.2 *(a)* below shows the latency during the blast measurement period while Figure B.2 *(b)* shows the latency during the multimedia period. The higher latency during the multimedia measurement period seems counter-intuitive. Since the blast period includes BULK, Proshare, and other traffic one might expect the queue occupancy, and thus the latency, to be higher in that case. However, the load is sufficient to keep the queue fully occupied most of the time in both cases because BULK offers a high load. Moreover, in the blast measurement period most of the BULK traffic has decreased its load in response to congestion so most of the packets in the queue are of type *other*. The average packet size of *other* is 1,080 bytes. However, in the case of the multimedia measurement period, most of the packets in the queue are of type BULK. The average packet size of BULK is 1,500 bytes. This difference in the average packet size also results in a difference in the average queue occupancy in bytes and is the major reason for the latency results observed.



a) Blast Measurement Period

a) Multimedia Measurement Period

Figure B.2 Latency (ms) vs. Maximum Queue Size (packets) with FIFO (BULK-Proshare)

The central observation for latency is that the maximum queue size and queue-induced latency are linearly related during periods of overload. As a result, increasing the maximum queue size increases the queue-induced latency. This argues for using a small maximum queue size.

However, the primary motivation to having a queue in a router is to allow for bursty arrivals to be buffered and serviced during idle periods in order to maintain good link utilization. This argues for a large maximum queue size. Consider the effect of the maximum queue size on the link utilization during the multimedia measurement period. The multimedia measurement period is considered because the packet arrivals may be bursty enough to illustrate the effects of queue drain on link utilization. This contrasts with the blast measurement period which has a higher traffic load leading to full queues almost all of the time. Figure B.3 shows the average throughput on the bottleneck link during the multimedia measurement period. Each data series represents one of the traffic mixes. If the queue size limits the router's ability to buffer bursty arrivals and avoid an empty queue, this should be reflected in a higher average throughput on the bottleneck link. Another factor effecting the throughput on the bottleneck link is the actual load arriving at the router and this varies between traffic mixes. The loads for traffic mixes using HTTP are typically a little lower than those using BULK and this is reflected in the results. As Figure B.3 shows, there is little change in the throughput on the bottleneck link for the BULK-mpeg and BULK-Proshare traffic. In fact, for all of the traffic mixes and maximum queue sizes the generated load is sufficient to saturate the link and hence the new is nearly fully utilized. Because of the, the experiment does not give a strong indication of the actual benefit of increased queue size for bursty traffic with conditions of moderate load. However, since these are the measurement periods and traffic combinations that will be used to evaluate the queue management algorithms, this analysis is appropriate. To choose the optimal setting, note that in these experiments although the HTTP mixes have a bit more variation, the queue size of 60 gives performance almost as good as any other value for all of the traffic mixes. Given that a smaller maximum queue size results in significantly better latency these results indicate a queue size of 60 is the current likely optimal setting.

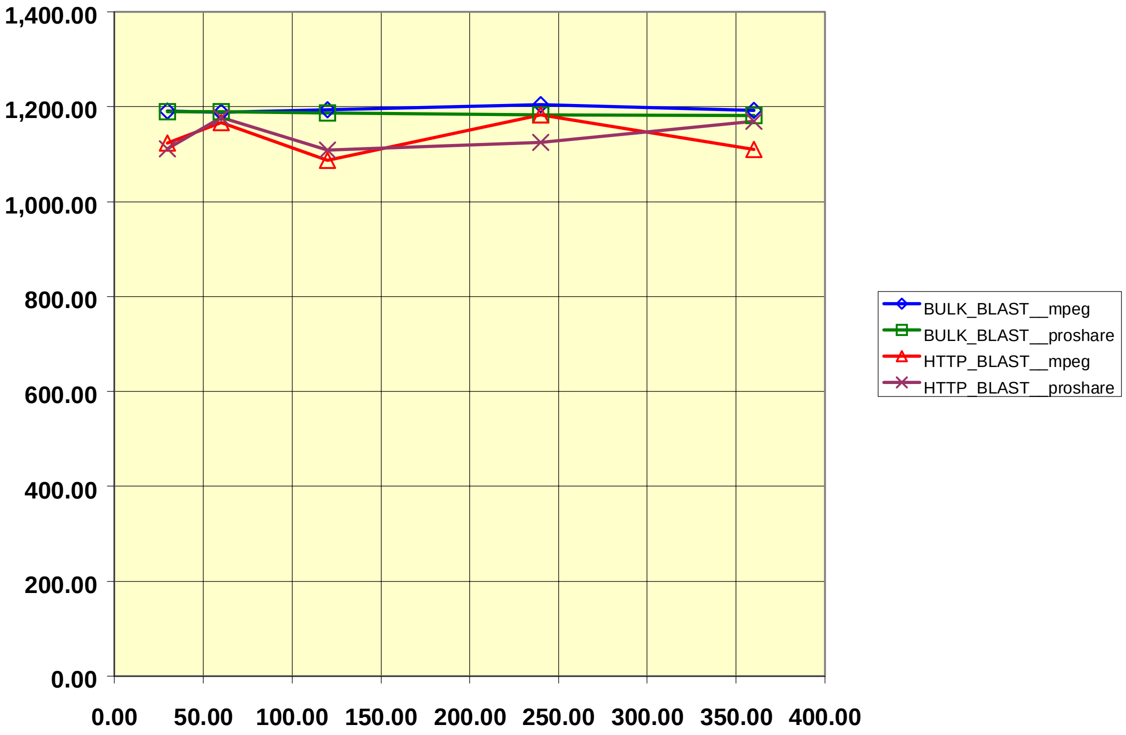


Figure B.3 Maximum Queue Size (packets) vs. Throughput (KB/s) with FIFO for All Traffic Mixes during Multimedia Measurement Period

Generally, as expected, FIFO gives poor performance for the other metrics during the blast measurement period. Multimedia packet loss is high and frame-rate is low during the blast measurement period because the *other* traffic is able to dominate the queue. Figure B.4 shows the packet loss rate for Proshare during the blast measurement period and Figure B.5 shows the frame-rates for MPEG. The key observation for choosing the optimal parameter setting for FIFO is that the loss-rate and frame-rate are relatively invariant and unacceptable with respect to the maximum queue size.

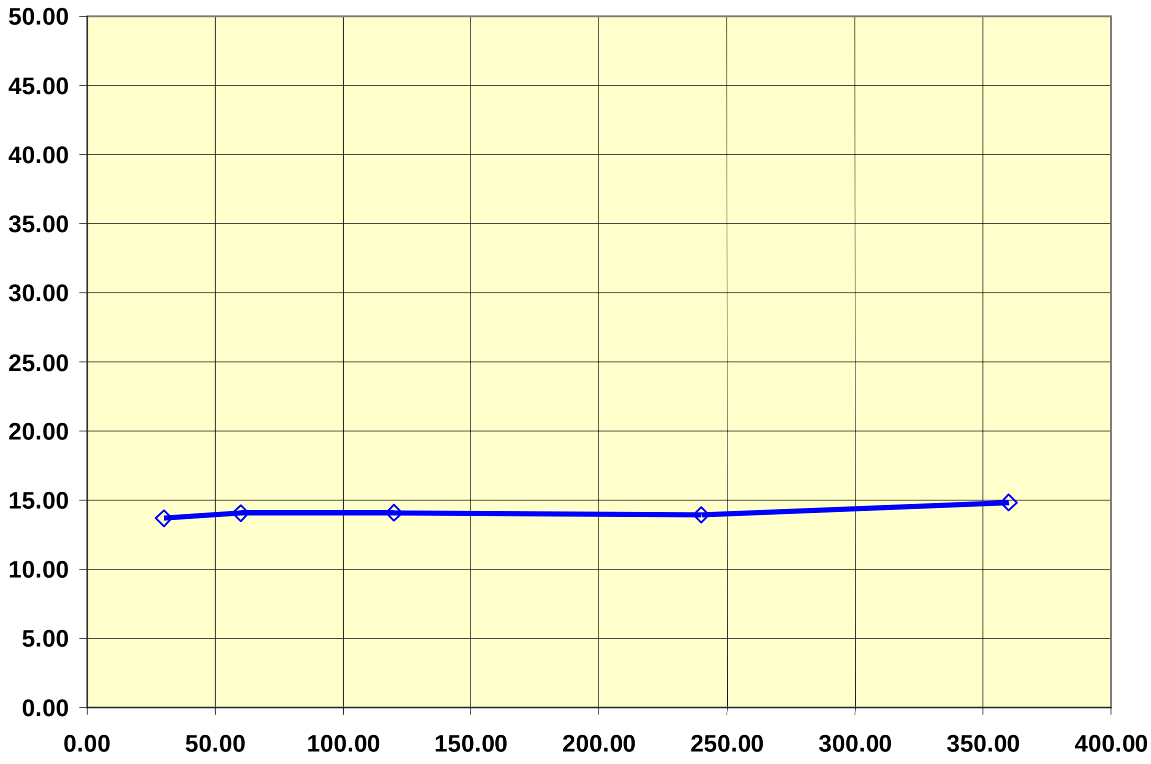
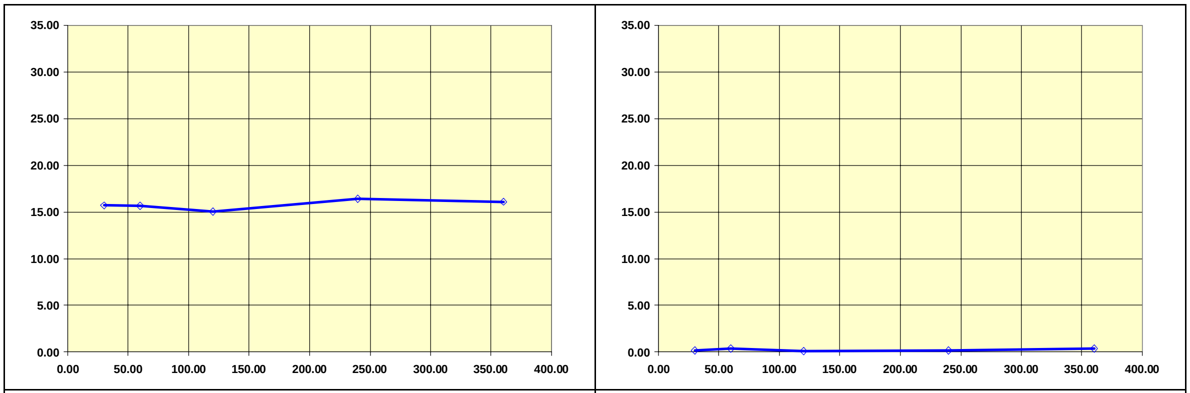


Figure B.4 Maximum Queue Size (packets) vs. Packets Lost (packets/second) for FIFO with HTTP-Proshare during Blast Measurement Period

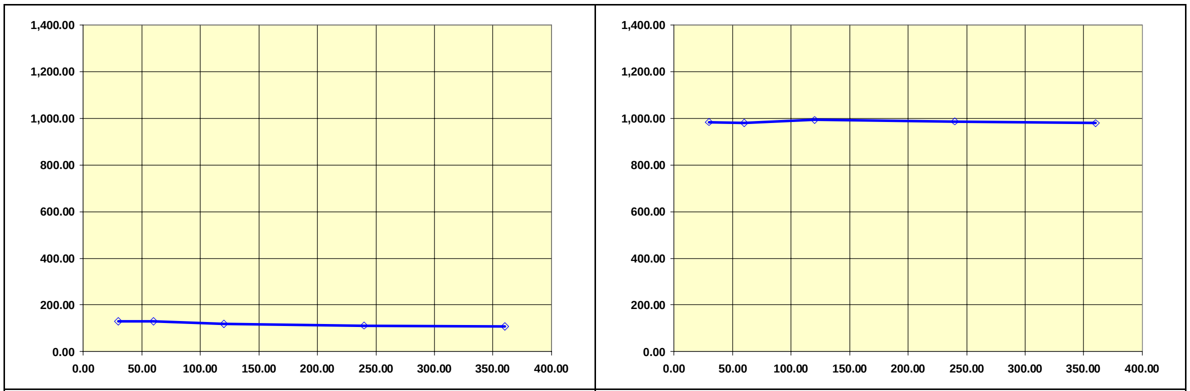


a) MPEG True Frame Rate

b) MPEG Playable Frame Rate

Figure B.5 Maximum Queue Size (packets) vs. Frame-rate (frames/second) for FIFO with HTTP-MPEG during the Blast Measurement Period

One of the reasons for the poor multimedia performance is that the UDP blast is able to dominate the queue managed by the drop-tail (FIFO) algorithm. Because the UDP blast is high bandwidth and unresponsive the drop-rate is very high for all packets. However, even with a large percentage of packets dropped, the UDP blast still gets high throughput while lower bandwidth flows like multimedia have their throughput decreased significantly. This is evident in Figure B.6. During the blast measurement period the TCP throughput on the bottleneck link is approximately 150 KB/s while the *other* throughput (the UDP blast) has throughput of 1,000 KB/s. Once again, in choosing the optimal queue length for FIFO, the key observation is that the performance is relatively invariant as the queue size changes. If anything the TCP throughput decreases slightly as the queue size increases. This argues slightly against increasing the maximum queue size.



a) TCP Throughput

b) Other Throughput

Figure B.6 Maximum Queue Size (packets) vs. Throughput (KB/s) with FIFO during the Blast Measurement Period

The optimal maximum queue size for FIFO in these experiments is 60 packets. This conclusion is based on the following observations:

- Latency is directly correlated with the queue size, increasing as the queue size increases.
- Although the link utilization is relatively uniform for all traffic mixes and queue sizes, a queue size of 60 offers performance equal to or better than the other settings, particularly the queue size of 30 packets for the HTTP mixes.
- Multimedia loss rate and frame-rate do not change relative to queue size.
- Throughput for TCP and other traffic classes does not vary relative to queue size.

Since latency is the only metric strongly effected by the queue size, a small queue size should be used to minimize latency. The link utilization metric encourages the selection of a maximum queue size of 60 instead of 30.

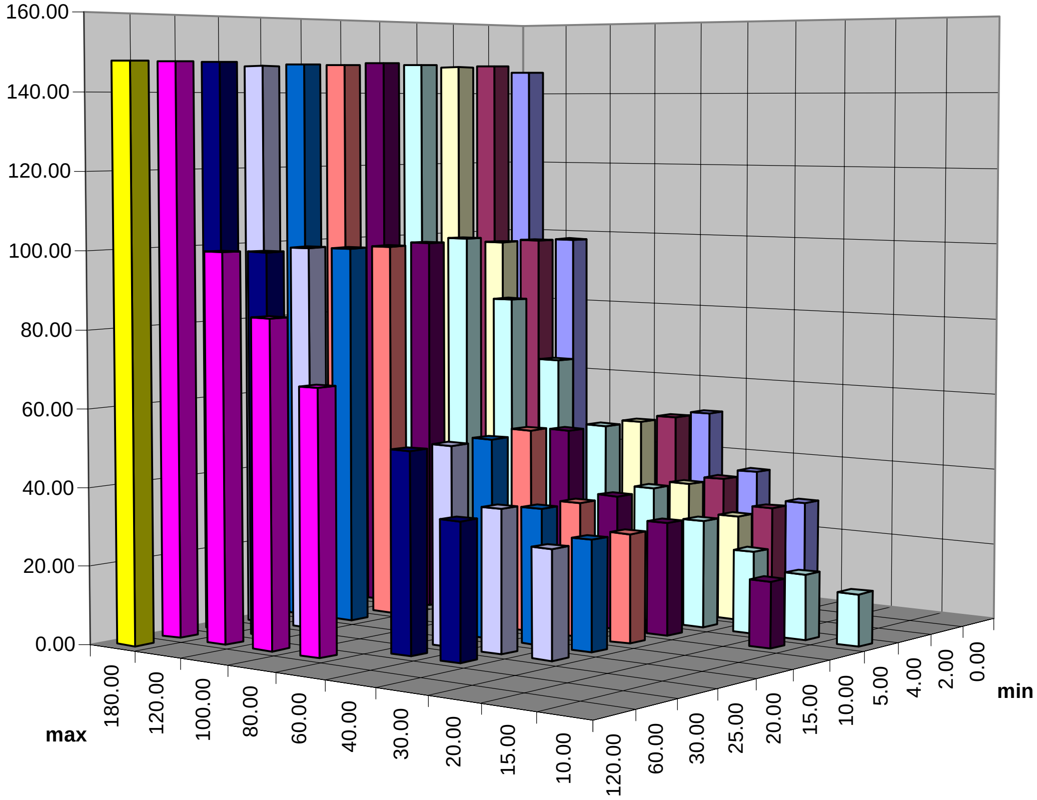
3. RED and FRED Analysis

Next, consider the selection of optimal parameters for RED and FRED. Because most of the parameters and much of the analysis for RED and FRED are similar, it is convenient to address some issues that apply to both. The focus here is on the values of the minimum and maximum threshold parameters for each algorithm, while holding the other parameters constant. The reason for this decision will become apparent as the experiments are discussed.

To begin, to determine the optimal parameter settings each algorithm must be evaluated with a range of threshold settings. Charts representing the individual performance metrics for the combination of parameter settings are considered. Those parameter combinations that offer poor performance are eliminated through the analysis of different metrics until only parameter settings that offer good performance across all key metrics remain. Before beginning the actual analysis it is helpful to explain the charts and tables used in making this analysis.

3.1. Understanding the Charts and Figures

Figure B.7 shows an example table and chart for analyzing one of the performance metrics (latency) under RED. The caption for the figure indicates the metric (latency), traffic mix (HTTP and Proshare), measurement period (Blast), and the algorithm (RED) used for this set of experiments. The chart is a three dimensional bar graph with each bar representing the results of an experiment. The maximum threshold setting (max is shorthand for Th_{Max}) runs along one horizontal axis and the minimum threshold setting (min is shorthand for Th_{Min}) runs along the other horizontal axis. These are the RED thresholds on the average queue occupancy in packets. Note that although the threshold values are numeric labels on these axes, the numeric values are indices to identify the parameter combination and not a scale. For example, the increment from a minimum threshold of 0 to 2 is the same size in the chart as the increment from 60 to 120. Also note that the colors group values by the minimum threshold setting. That is, all runs with the same minimum threshold setting have the same color. For example, all experiments with a minimum threshold value of 0 are represented by light blue bars. (For careful study, obtaining a color copy of this document is recommended.) The vertical axis is the metric being considering. In this case the metric is network latency as measured with the instrumented multimedia application, Proshare. The value represented by a given bar is the value of the metric averaged over the measurement period and across all runs with that parameter combination. More specifically, in the case of latency, the instrumented Proshare application reports the mean latency over one second intervals. Tthe values reported during the measurement period for a given run are then arithmetically averaged to obtain that run's mean latency. Finally, the arithmetic average of the mean latency values reported for all of the runs is computed and that value reported. The chart offers a graphical representation of the data that allows one to discern trends and relationships visually. For example, in this plot one can see that the latency value is correlated with the maximum threshold value but not the minimum thresholds.



Min	Max									
	10	15	20	30	40	60	80	100	120	180
0				27	35	50			98	146
2				28	34	50			98	147
4				27	35	50			98	147
5	13	16	21	28	35	51	67	83	99	147
10		16		29	34	51			99	148
15				28	34	52			98	147
20				28	34	51			98	147
25				27	36	51			99	146
30					35	51			99	147
60							67	83	99	147
120										147

Figure B.7 Latency (ms) during Blast Measurement Period for RED and HTTP and Proshare

For more precise analysis, the data is also presented in tabular form. The table beneath the chart is another representation of the same data. Each column represents a setting of the maximum threshold and each row represents a setting of the minimum threshold. The contents of each cell represent the average value of the metric with the combination of parameters for that row and column. For example, the experiment with a minimum threshold of 10 and a maximum threshold of 40 had an average latency of 34 ms.

As the analysis of parameter settings proceeds some parameter combinations are eliminated from consideration. Shading in the tables indicates parameter combinations that are eliminated. Dark gray shading indicates combinations previously eliminated by analysis of other metrics. For example, as we will later see, the values in Figure B.7 with minimum threshold values of 5 or less were eliminated after analysis of the same experiments with respect to another metric which is not shown for this example. In this case, while analyzing the effects of the parameters on throughput it was evident that minimum threshold values of 5 or less gave unacceptable throughput. Light gray shading indicates combinations eliminated by the current metric. For example, if maintaining average network latency of less than 50 ms were a concern, these results would indicate all combinations with a maximum threshold value of 60 or more are unacceptable. Those values are shaded with light gray. When combinations have poor performance with respect to the current metric and a previous metrics (e.g., (5,60)) those cells remain shaded with dark gray. Finally note that shading of the row or column labels indicates all runs using that threshold setting have been eliminated. For example all rows representing minimum threshold settings of 5 or less are darkly shaded, indicating those values were eliminated by a prior metric. Those cells that are not shaded remain in contention as optimal parameter settings.

3.2. Fixed Parameters

While the threshold values were varied, RED's and FRED's other parameters remained constant. Thresholds have the most direct effect on the performance of the network with the RED and FRED algorithms they were examined and the other parameters of the algorithms were held constant. For both algorithms, the queue length was 240, the weight factor was 1/256, and, the maximum drop probability was 10%. Additionally, FRED’s minq value was set to 2. The drop probability and minq values were selected based on the recommendation of the algorithm designers. The RED designers recommend this value of maxp based on the fact it is a slightly higher drop probability than that seen in the steady state in a router [Floyd97a]. The FRED designers recommend using a minq of 2 in their work [Lin97]. This value of minq is intended to insure that all flows can have a few packets enqueued even when severely constrained.

The choice of the weighting factor and the queue size are intertwined. The weighting factor can be thought of as a "time constant" that determines how closely the instantaneous queue occupancy and the weighted average are bound together [Floyd97a]. If the weight is too high, the average will be too sensitive to bursty behavior. If the weight is too low, the average will not indicate congestion until the queue is overflowing. Even more, the weighting factor determines how large a packet burst the queue can accommodate without triggering the dropping mode of the RED algorithm. Equation B.2 shows the weighted average that results when a burst of *L* packets arrives at an idle queue. This equation is explained in [Floyd93].

$$avg = L + 1 + \frac{(1 - w_q)^{L+1} - 1}{w_q} \qquad \text{(B.2)}$$

In this work the queue size was set large enough to insure it would not be a factor in he dropping scheme. Maintaining an average queue size small enough to insure reasonable queue-induced latency, on the order of 40 packets, was also a concern. With a weighting factor of 1/256 a burst of 158 packets would result in an average queue size of 40.39 packets, more if the average was greater than zero initially. As such, the weighting factor of 1/256 and queue size of 240 are sufficient.

4. RED

Thresholds have the most direct effect on the performance of the network with the RED algorithm. As a result, choosing the optimal threshold values is the focus here. The settings for RED's other parameters, the weighting factor, maximum drop probability, and queue size, are held constant throughout so that the effect of varying threshold settings can be isolated. The choice of the fixed parameter values was discussed in Section 3.2. Here, the threshold settings and their effects are examined.

4.1. RED Threshold Settings

In selecting optimal parameters for RED, first the following metrics were considered in the order listed:

1. Goodput for TCP.
2. Aggregate Throughput.
3. TCP Efficiency.
4. Network Latency.

RED's design goals include elements that concern each of these metrics. First, RED wishes to offer better feedback to responsive flows. If feedback is effective, the flows should be able to stabilize at a load that minimizes retransmissions and losses while still obtaining high throughput. This is best measured by examining the goodput across all TCP flows. While the RED algorithm seeks to provide effective feedback by discarding packets when congestion is imminent, it also seeks to avoid unnecessary drops in the face of bursty traffic which may lead to an empty queue and idle outbound link. Aggregate throughput on the outbound link is an effective metric for assessing the parameter settings in this regard. If the parameter settings lead to unnecessarily aggressive packet discards, the queue will empty and, consequently, the outbound link will have idle periods, resulting in lower average aggregate throughput.

After considering aggregate throughput, TCP efficiency is the next metric considered. It is another, more precise measure of the effectiveness of the feedback. This efficiency is expressed as the ratio of TCP throughput on the outbound link to the TCP load on the inbound link. If the feedback mechanism is effective, the senders should be able to adjust their load to match the available capacity resulting in throughput equal to the generated load and efficiency approaching 1.0. Finally, one of RED's goal is to maintain shorter average queues, both to leave room to accommodate bursts and to minimize queue-induced latency. Because the bottleneck router's queueing delay is the only source of latency in this network configuration, measuring the network latency between the sender and receiver can estimate the average queue length. This latency is measured using the instrumented multimedia applications.

Using these metrics eliminates many of the possible parameter combinations but parameter combinations that offer no discernible difference in performance remain. The choice between those alternatives relied on the recommendations of the designers of the RED algorithm and the studies they conducted in their original work.

With this elimination process established, consider the experiments themselves. While selecting the optimal parameters for RED, the HTTP and Proshare traffic mix was used for the base set of experiments. The same type of experiments were then conducted with a more limited parameter space for the BULK and Proshare traffic mix. These traffic mixes were considered because RED's focus is on improving feedback for responsive flows. As a result, the performance of the TCP traffic is the emphasis. Hence, performance with MPEG is not examined for these experiments. Moreover, RED has no mechanisms for distinguishing between either individual flows or classes of flows. As such, the MPEG and Proshare traffic types are both simply unresponsive flows that generate approximately 150-200 KB/s of load on the network. Although it is possible that the RED parameter settings will have different effects on those two multimedia types that issue is deferred until the multimedia performance is considered when the algorithms are compared to one another. This choice is made because RED does not intend to improve multimedia performance. And hence, the multimedia performance is not a concern as a metric for selecting the optimal parameters of RED. However, RED does intend to give better feedback to responsive flows. Consequently, TCP performance is a concern so the TCP traffic types are varied. Because HTTP and BULK flows represent two extremes both in lifetime, bandwidth, and number they may respond differently to the different drop distributions resulting from the different parameter settings. This leads to considering the traffic mixes first of HTTP and Proshare and then of BULK and Proshare while selecting the optimal parameters.

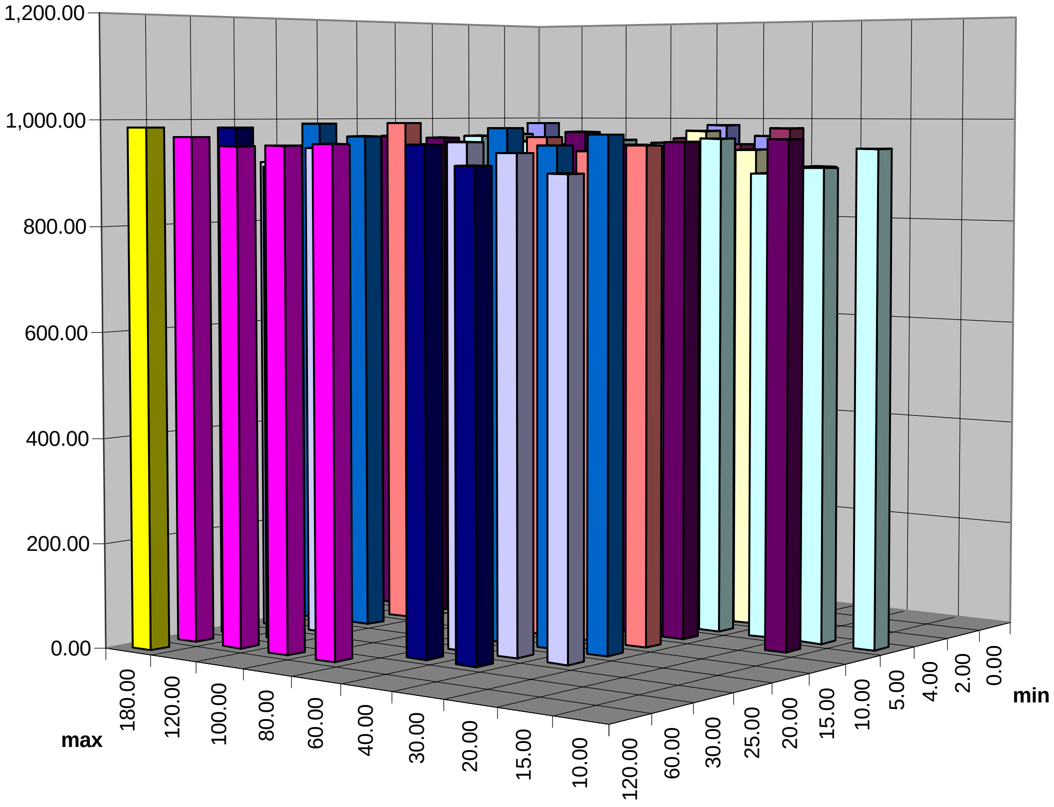
4.1.1. HTTP and Proshare

While evaluating RED, the minimum thresholds on average queue occupancy ranged from 0 to 120 and the maximum thresholds ranged from 10 to 180. Table B.2 shows the number of experiments conducted for each parameter combination. The effect of these parameter combinations on each of the key metrics is considered below.

MinTh	MaxTh									
	10	15	20	30	40	60	80	100	120	180
0				1	5	5			5	5
2				1	5	5			5	5
4				1	5	5			5	5
5	1	2	1	2	5	6	6	6	6	5
10		2		1	5	5			5	6
15				2	5	5			5	5
20				1	5	5			5	5
25				1	5	5			5	5
30					5	5			5	5
60							5	5	6	4
120										5

Table B.2 Count of RED Experiments for Each Parameter Combination with HTTP-Proshare

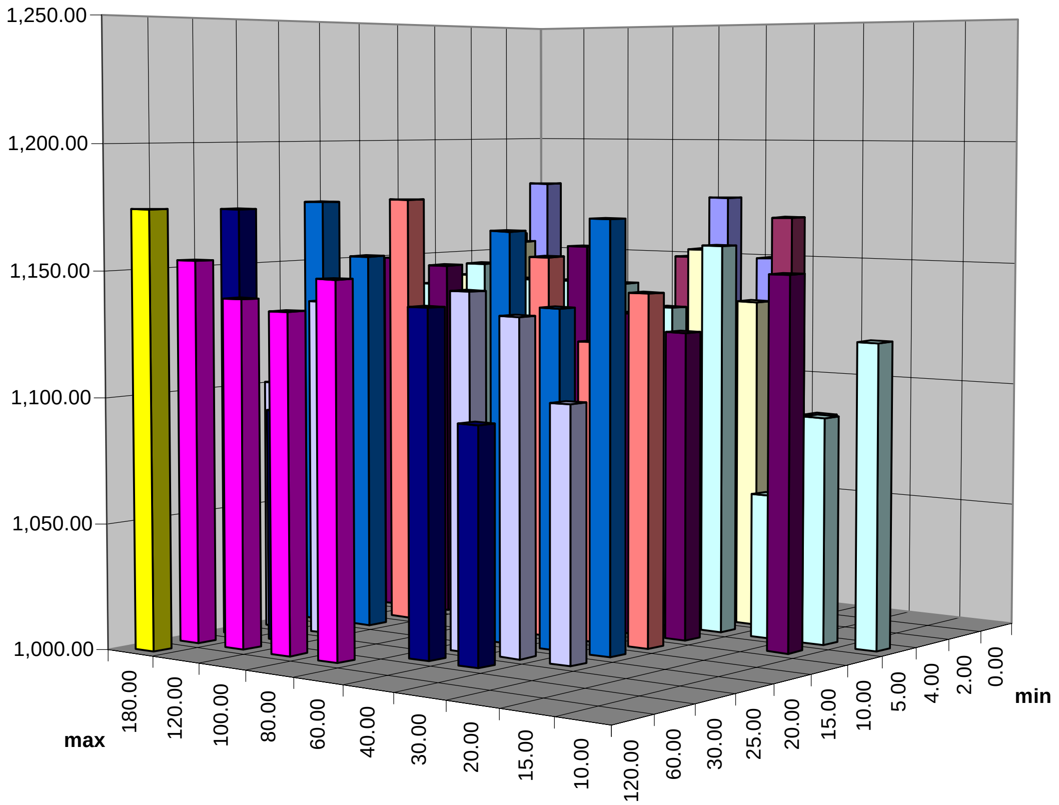
The analysis begins by considering the performance of a traffic mix of HTTP and Proshare across the different RED threshold combinations. The initial criterion for evaluating the parameters is the resulting TCP goodput. RED should insure that TCP throughput remains high and should provide effective feedback to the senders since packets dropped by RED are distributed evenly across flows. Although RED's signaling mechanism, packet drops, necessitates retransmissions, these retransmissions should be minimized if the drops are well distributed. To evaluate RED's effectiveness in this regard, consider the goodput metric. TCP goodput is the amount of unique TCP data that successfully reaches the receiver. Unnecessary retransmissions or dropped packets may contribute to the throughput on a given link, but the capacity they consume is wasted if they do not contribute to the end-to-end TCP goodput. RED seeks to insure both that TCP throughput remains high and that TCP flows' use of the network is efficient. Goodput is an effective measure of both of these criteria. However, ranging the parameter settings had little effect on goodput. Figure B.8 shows the TCP goodput for RED during the multimedia measurement period with Proshare and TCP traffic. The goodput is uniformly in the 900-1000 KB/s range. Multimedia consumes ~150-200 KB/s of capacity so the TCP traffic appears to have stabilized at a load that is near the available capacity of the bottleneck link. Based on this metric no parameter combinations can be eliminated.



MinTh	MaxTh									
	10	15	20	30	40	60	80	100	120	180
0				903	964	986			946	989
2				981	948	960			924	968
4				940	976	936			967	951
5	945	909	896	962	954	958	949	954	965	946
10		964		957	951	975			962	964
15				952	939	965			992	939
20				972	952	983			966	991
25				902	938	958			945	916
30					917	954			910	984
60							956	952	950	966
120										986

Figure B.8 TCP Goodput (KB/s) During Multimedia Measurement Period with RED for HTTP and Proshare.

Next, consider the aggregate throughput on the outbound link. The performance as measured by this metric is shown in Figure B.9. This figure shows the aggregate throughput during the multimedia measurement period. Note that the Z axis (link utilization) starts at 1000 KB/s and ranges up to 1250 KB/s. The results are magnified in this way to discern any trends that might not be apparent with a larger scale. This data indicates the link is operating at 85-95% of capacity in these experiments. However, again there are no obvious trends to indicate a relationship between the parameters and the performance of this metric so no parameter combinations are eliminated.

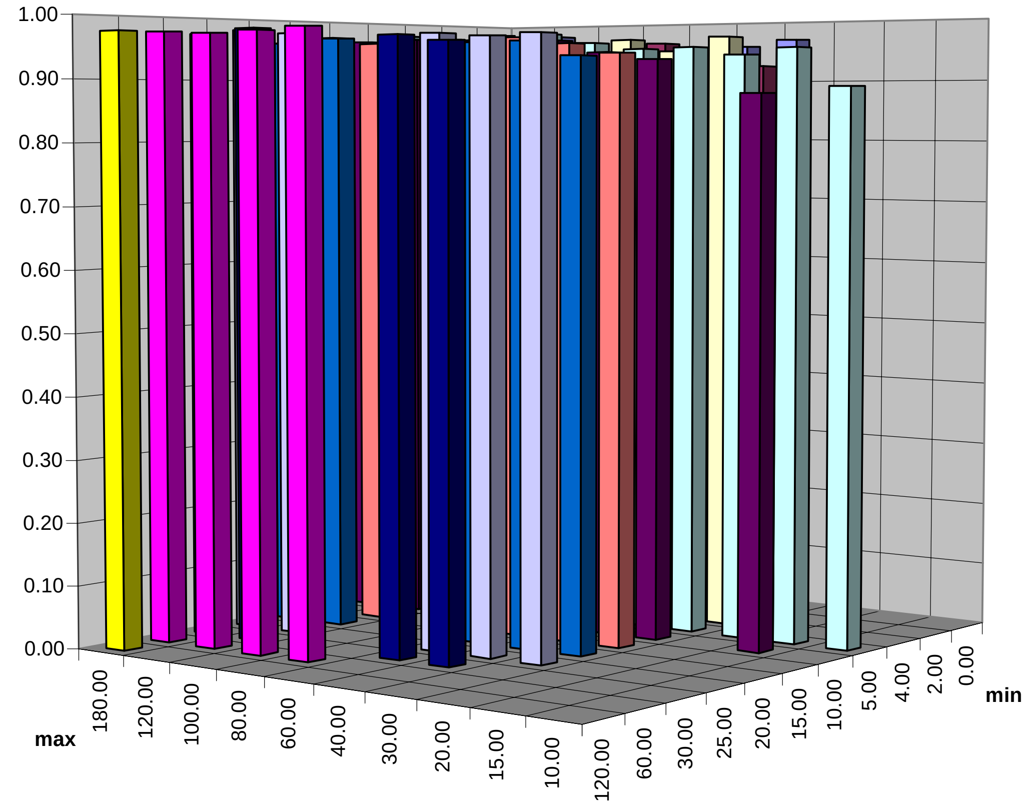


Min	Max									
	10	15	20	30	40	60	80	100	120	180
0				1,083	1,149	1,175			1,141	1,179
2				1,168	1,130	1,150			1,115	1,157
4				1,134	1,155	1,127			1,155	1,140
5	1,122	1,091	1,058	1,158	1,132	1,141	1,141	1,141	1,147	1,137
10		1,149		1,124	1,130	1,157			1,147	1,149
15				1,141	1,120	1,153			1,175	1,123
20				1,171	1,135	1,165			1,152	1,174
25				1,101	1,133	1,142			1,135	1,101
30					1,093	1,137			1,092	1,173
60							1,148	1,134	1,138	1,153
120										1,174

Figure B.9 Aggregate Throughput (KB/s) during Multimedia Measurement Period with RED for HTTP and Proshare

Since goodput and aggregate throughput did not indicate optimal parameters, next consider efficiency. The key to maintaining high efficiency is effectively notifying responsive senders when congestion is imminent without giving false signals because of bursty packet arrivals. Achieving this effect requires carefully balancing the minimum and maximum threshold values. Setting the maximum threshold too small can result in false signals because a small burst may increase the average queue occupancy enough to reach the maximum threshold and trigger forced drop mode. Clearly, dropping all arriving packets due to a single burst of packets would result in very low efficiency. Moreover, even with a large maximum threshold, setting the minimum threshold too close to the maximum threshold limits the time flows have to respond to actual congestion. Recall that it takes one round-trip time for the senders to detect a dropped packet and adjust their transmission window. During that interval the average queue size continues to grow as the senders maintain their load. As a result, a smaller range between the thresholds means the droprate grows very quickly (as the average approaches the maximum threshold), resulting in lower efficiency. Therefore, it is important to maintain a reasonable range between the minimum and maximum thresholds. Moreover, the average may reach the maximum threshold and trigger the forced drop mode when the range is small. Whenever RED is in the forced drop mode efficiency will suffer greatly as all arriving packets are dropped. However, efficiency should improve as the maximum threshold is increased and a reasonable range between the thresholds is maintained.

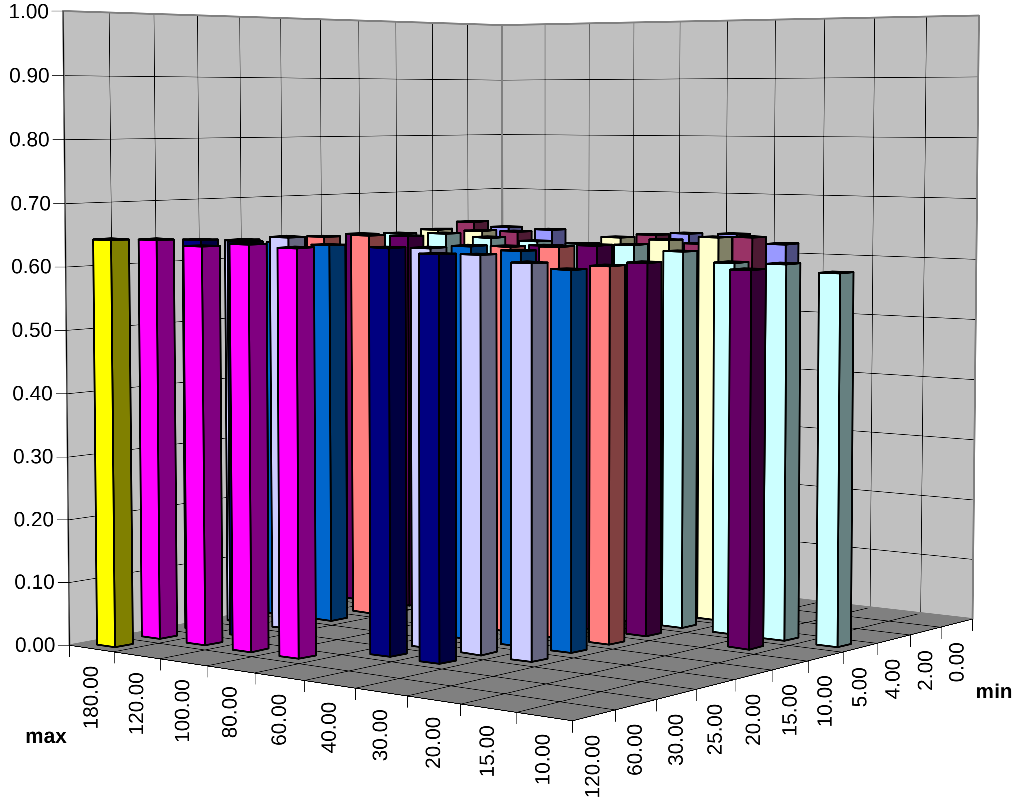
Figure B.10 shows the efficiency metric during the multimedia measurement period and Figure B.11 shows the efficiency metric during the blast measurement period. For RED, during the multimedia measurement period the outbound TCP traffic is roughly 95-98% of the inbound and the ratio is fairly uniform across all of the parameter settings. However, the settings with a maximum threshold greater than an average queue occupancy of 30 packets have better efficiency than those with a smaller maximum threshold setting. Therefore, the parameter combinations with a maximum threshold of 30 packets or less are eliminated.



Min	Max									
	10	15	20	30	40	60	80	100	120	180
0				0.97	0.96	0.95			0.98	0.97
2				0.92	0.96	0.96			0.97	0.98
4				0.97	0.95	0.97			0.97	0.97
5	0.89	0.95	0.94	0.95	0.95	0.96	0.98	0.98	0.97	0.98
10		0.88		0.93	0.94	0.96			0.97	0.97
15				0.94	0.96	0.97			0.96	0.97
20				0.94	0.96	0.96			0.97	0.96
25				0.97	0.97	0.97			0.98	0.98
30					0.96	0.97			0.98	0.97
60							0.98	0.98	0.97	0.98
120										0.97

Figure B.10 TCP Efficiency during the Multimedia Measurement Period with RED for HTTP and Proshare

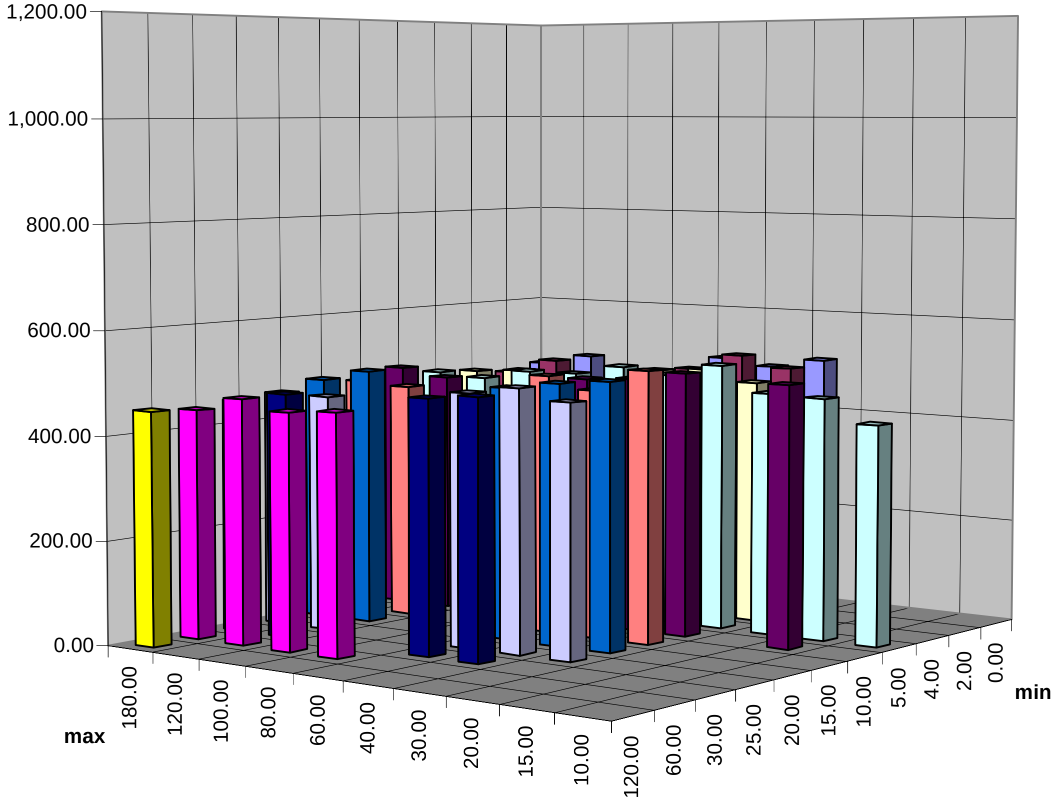
During the blast measurement period (Figure B.11) the efficiency ratio is much worse, with only about 60-64% of the inbound TCP traffic making it to the outbound link. Clearly, the TCP sources have trouble adjusting their load to match the available capacity. However, once again, the efficiency is nearly uniform across all of the available parameter combinations. This indicates the overload resulting from the presence of the aggressive, unresponsive UDPblast is the major factor in this performance. However, once again the maximum threshold settings of 30 or less have slightly worse performance, reinforcing the decision to eliminate them.



Min	Max									
	10	15	20	30	40	60	80	100	120	180
0				0.62	0.63	0.63			0.63	0.63
2				0.63	0.62	0.63			0.63	0.65
4				0.64	0.63	0.63			0.63	0.63
5	0.59	0.60	0.60	0.62	0.62	0.62	0.62	0.63	0.63	0.63
10		0.60		0.60	0.63	0.62			0.63	0.63
15				0.60	0.63	0.63			0.64	0.63
20				0.60	0.63	0.63			0.62	0.63
25				0.61	0.62	0.63			0.64	0.63
30					0.63	0.63			0.64	0.64
60							0.63	0.64	0.63	0.64
120										0.64

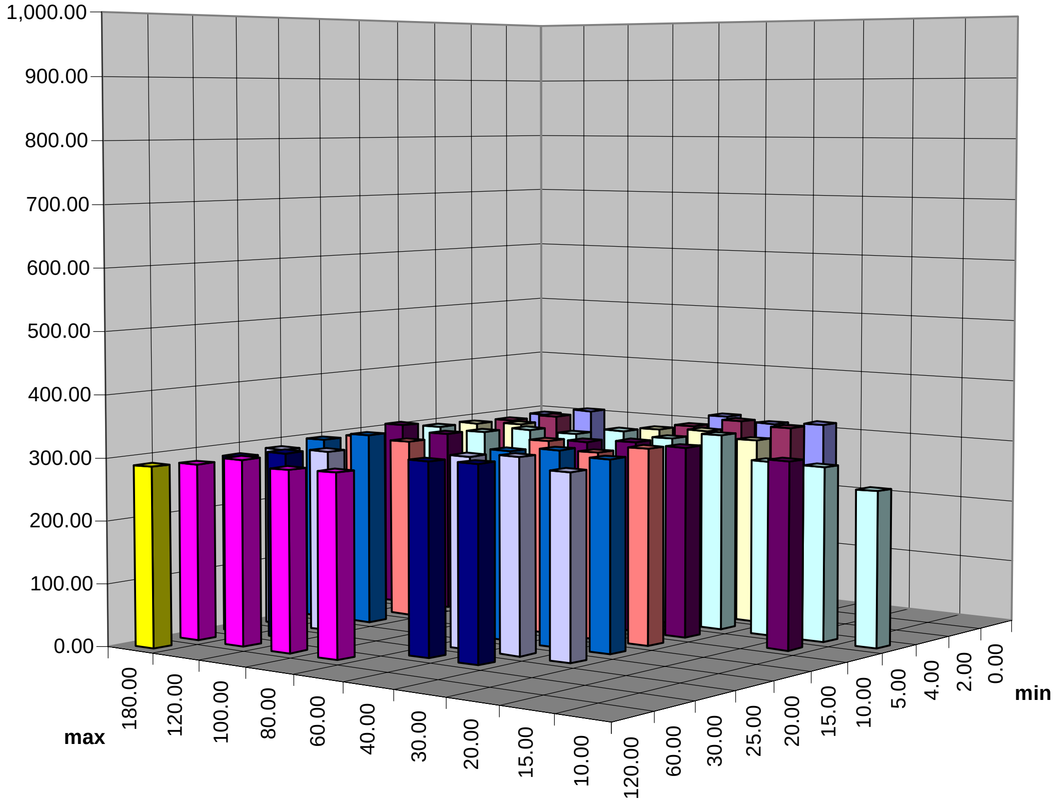
Figure B.11 TCP Efficiency during Blast Measurement Period with RED for HTTP and Proshare

Note that the very low efficiency rating during the blast period should not be interpreted to indicate TCP does not reduce its load in response to the congestion caused by the UDP blast. As shown in Figure B.12, TCP's load reduces to less than ~500 KB/s. However, the UDP blast is capable of consuming the entire capacity of the bottleneck link. Even with well distributed drops, TCP's throughput on the outbound link, shown in Figure B.13, is only ~300 KB/s. The problem is that while the load does decrease, it only decreases in half, while the throughput on the outbound link decreases to 30% of the performance during the period without the blast. TCP's inability to reduce its load in this scenario is attributed to the number of flows involved. With the large number of active TCP flows, there are always enough flows probing for available link capacity (or, in the case of HTTP starting a connection) to maintain a load of 500 KB/s.



Min	Max									
	10	15	20	30	40	60	80	100	120	180
0				505	485	497			484	463
2				496	516	482			481	451
4				475	495	487			468	461
5	423	466	469	517	492	502	483	481	462	467
10		501		510	493	486			470	483
15				523	478	500			460	465
20				511	500	485			499	475
25				481	500	484			457	457
30					493	483			470	450
60							459	452	470	442
120										448

Figure B.12 TCP Load (KB/s) during Blast Measurement Period with RED for HTTP and Proshare



Min	Max									
	10	15	20	30	40	60	80	100	120	180
0				312	307	314			305	293
2				314	319	305			303	290
4				302	311	307			297	293
5	250	280	282	319	307	312	301	302	292	294
10		299		307	309	303			297	305
15				314	300	313			292	293
20				306	313	305			311	297
25				295	311	305			292	289
30					309	306			298	286
60							291	288	297	282
120										288

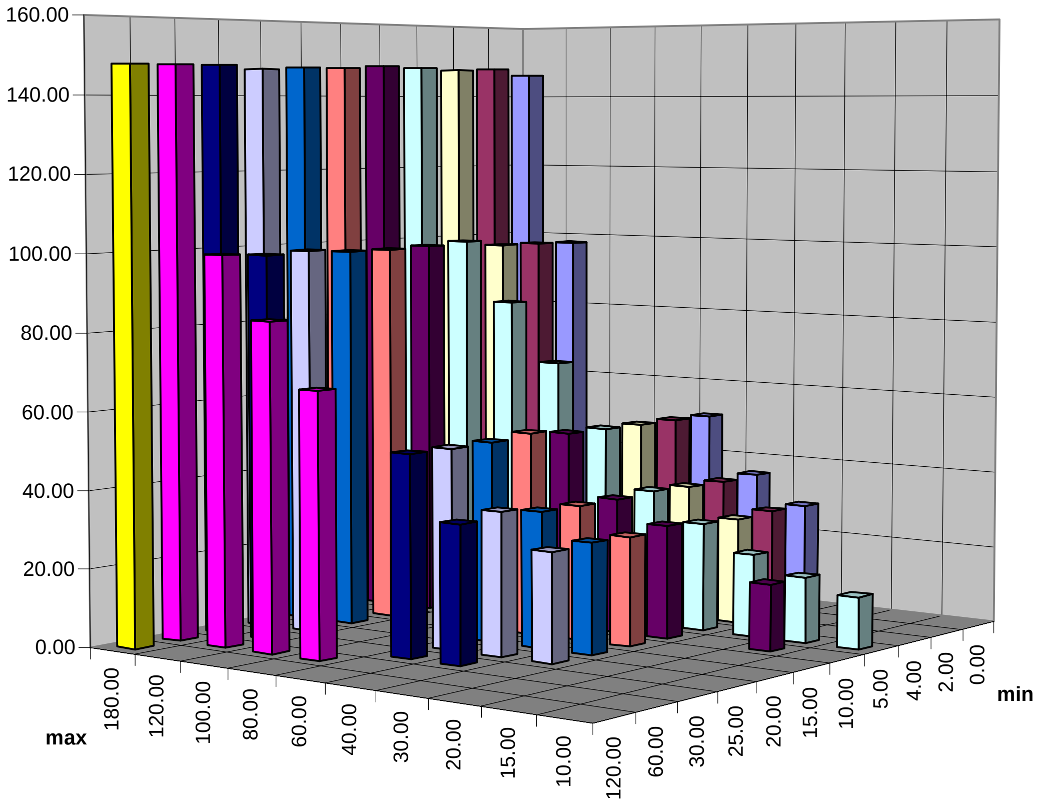
Figure B.13 TCP Throughput (KB/s) during Blast Measurement Period with RED for HTTP and Proshare

To select a suitable set of parameters, consider network latency as a measure of the effectiveness of RED. The latency should be directly related to the maximum threshold setting. The maximum threshold places an upper limit on the average queue occupancy and the queue occupancy determines the queue-induced latency. Recall that in these experiments an aggregate load is maintained that usually exceeds the capacity of the bottleneck link during the multimedia measurement period and always exceeds the capacity during the blast measurement period. As a result the queue occupancy and queue-induced latency is limited by the maximum threshold's limit on the average queue occupancy.

The queue-induced latency can be predicted as a function of the capacity of the outbound link, the average packet size, and the maximum threshold. If the link is continuously congested the queue will build up and have an average occupancy equal to the maximum threshold. Any arriving packet will be delayed for the time necessary to service the packets ahead of it in the queue. Since the queue is serviced at the rate of the outbound link and the link's capacity is expressed in bytes per second, to determine the length of the queue-induced delay the queue occupancy must be expressed in bytes. The occupancy of the queue in bytes is the product of the occupancy in packets and the average packet size. The average latency can then be calculated by dividing the average queue occupancy in bytes by the capacity of the outbound link as shown in B.3.

$$Latency = \frac{pkt\_size * Th_{Max}}{C_{outbound}} \tag{B.3}$$

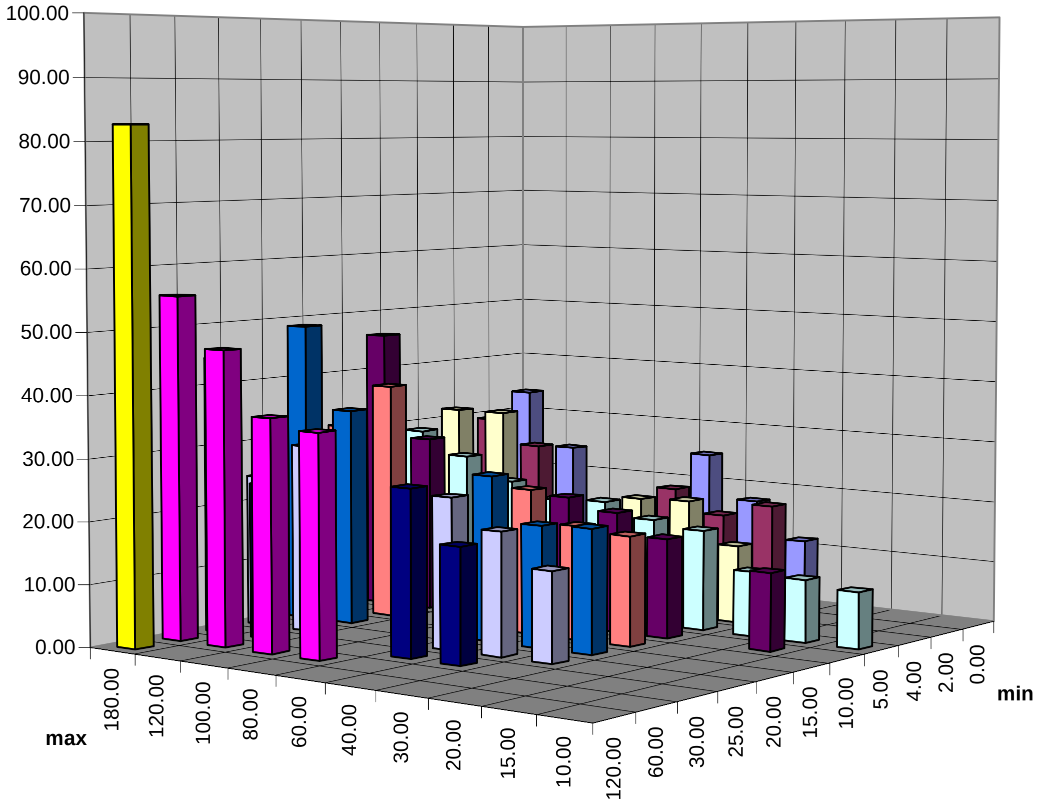
This equation can be used to predict the results of the experiments. Consider an example. Figure B.14 shows the network latency measure during the blast measurement period. During the blast measurement period the observed latency with a maximum threshold of 120 ranges from 98-99 ms. Compare this to the expected value using Equation B.3. In this configuration the queue services a link with a capacity of 10Mb/s and an average packet size of 1,000 bytes. The calculated latency for 1,000 byte packets and an average depth of 120 packets is 96 milliseconds. The difference of 2-4 milliseconds between this value and the observed value is because 2-4 milliseconds is the actual time required for the packet to traverse the network and sender and receiver protocol stacks when the network is not congested. The latency measurements, particularly those during the blast measurement period, clearly confirm that maximum threshold and network latency are directly related. Moreover, the results show no relation between the minimum threshold and the latency.



Min	Max									
	10	15	20	30	40	60	80	100	120	180
0				27	35	50			98	146
2				28	34	50			98	147
4				27	35	50			98	147
5	13	16	21	28	35	51	67	83	99	147
10		16		29	34	51			99	148
15				28	34	52			98	147
20				28	34	51			98	147
25				27	36	51			99	146
30					35	51			99	147
60							67	83	99	147
120										147

Figure B.14 Network Latency (ms) during Blast Measurement Period for RED with HTTP and Proshare

Figure B.15 shows that the maximum threshold is also related to the latency for RED during the multimedia measurement period. Although the relationship is not as strong, it is apparent. The relationship is not as strong because the link is not severely overloaded during the multimedia measurement period. As a result, the RED algorithm is effective in managing the queue size so an average queue occupancy equal to the maximum threshold is not maintained.



Min	Max									
	10	15	20	30	40	60	80	100	120	180
0				11	17	24			24	33
2				18	16	19			25	29
4				12	19	19			31	31
5	9	10	10	16	17	19	19	21	25	28
10		12		16	19	21			28	46
15				17	18	23			38	31
20				19	19	26			35	48
25				14	19	24			30	24
30					18	26			24	44
60							35	37	47	55
120										82

Figure B.15 Network Latency (ms) during Multimedia Measurement Period with RED for HTTP and Proshare

Because link utilization seemed best for maximum threshold values greater than 40, only consider the latency values that result from setting Th_{Max} greater than 40. Since latency decreases with the maximum threshold, examining the results during the blast measurement period (Figure B.14) reveals that the latency values for a maximum threshold of 40 and 60 are generally the lowest among those settings still under consideration. The latency values for a maximum of 60 is generally ~50ms and the latency values for a maximum threshold of 40 are generally ~35ms. As a result, a maximum threshold of 40 is chosen. Although this Th_{Max} is rather small, note that because this threshold setting is compared to the average queue size, not the instantaneous, it does still allow the queue to accommodate larger bursts during periods when the network is not congested. As a result, end-to-end latencies as high as 158 ms are possible if the queue size is large enough, though rare, due to a burst (see section 3.2) but the average should be around 35ms. Based on the queue-induced latency metric all values with a maximum threshold of 60 or more are eliminated.

Parameter settings including a wide range of minimum threshold settings combined with a maximum threshold of 40 remain. With no other key metrics to consider, consider the recommendations of the original designers or RED [Floyd97a]. They point out the minimum threshold should be kept small to insure feedback whenever a queue begins to build. However, they also note setting the minimum threshold as small as one or two packets does not allow for much burstiness in the arrival process without triggering the random drop mechanism. As a result, they recommend a minimum threshold setting of 5. Although the expected difference in performance is not evident in these experiments, this recommendation is accepted. Thus, for RED, threshold values of (5,40) should be used as the optimal setting.

4.2. BULK and Proshare

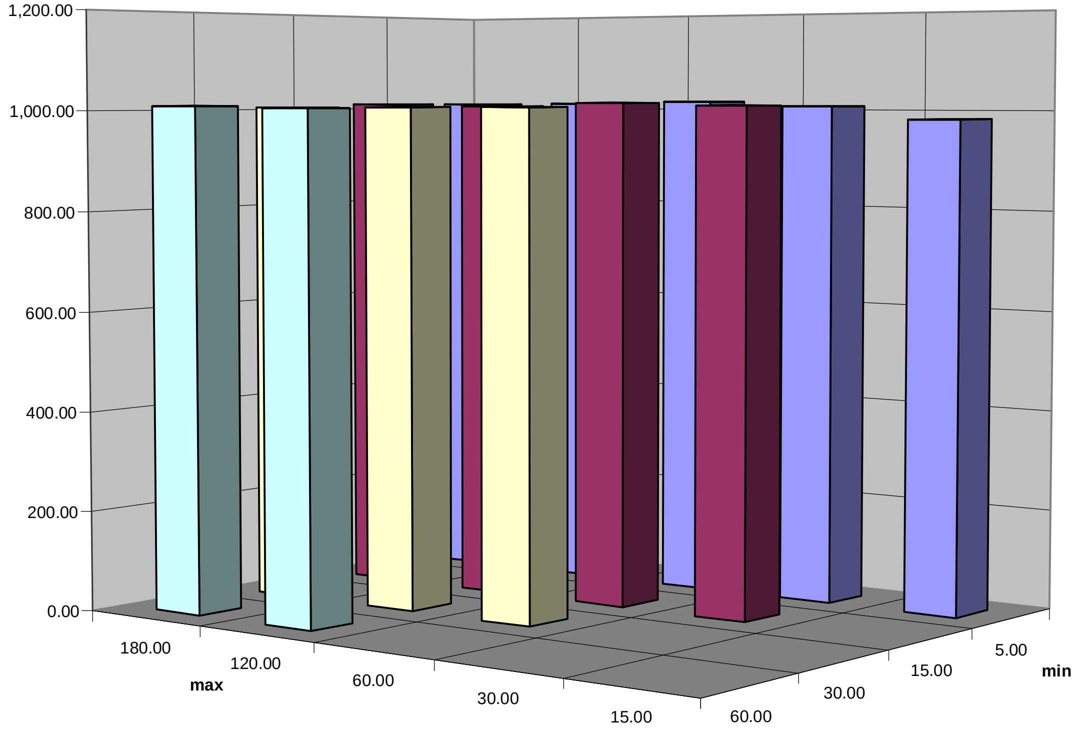
While the previous section considered the performance of RED with HTTP and Proshare traffic, this section repeats the evaluation using BULK as the TCP load. Table B.3 shows the parameter combinations considered for this traffic mix. A smaller set of experiments were used with this traffic mix because the results of these experiments were expected to confirm the findings with HTTP and Proshare. If discrepancies had appeared, the parameter space would have been explored in more detail as necessary.

MinTh	MaxTh				
	15	30	60	120	180
5	6	5	2	5	4
15		4	2	5	5
30			2	5	4
60				5	4

Table B.3 Count of RED Experiments for Each Parameter Combination with BULK and Proshare.

In fact, the analysis of these experiments did confirm the parameter selections from HTTP and Proshare. Analysis of the goodput, aggregate throughput on the bottleneck link, and efficiency metrics did not indicate any optimal parameters (though efficiency and goodput did once again increase slightly as maximum threshold increased). The latency results were consistent as well. Average latency increased as a function of the maximum threshold just as with the HTTP and Proshare traffic mix. These findings reaffirm the selection of (5,40) as the optimal threshold settings for RED.

There was one interesting finding in these experiments, however. Although the efficiency during the multimedia measurement period was nearly uniform, it was significantly lower with BULK and Proshare (74-77%) than it was with HTTP and Proshare (95-99%). This result is discussed when the efficiency data is presented below.

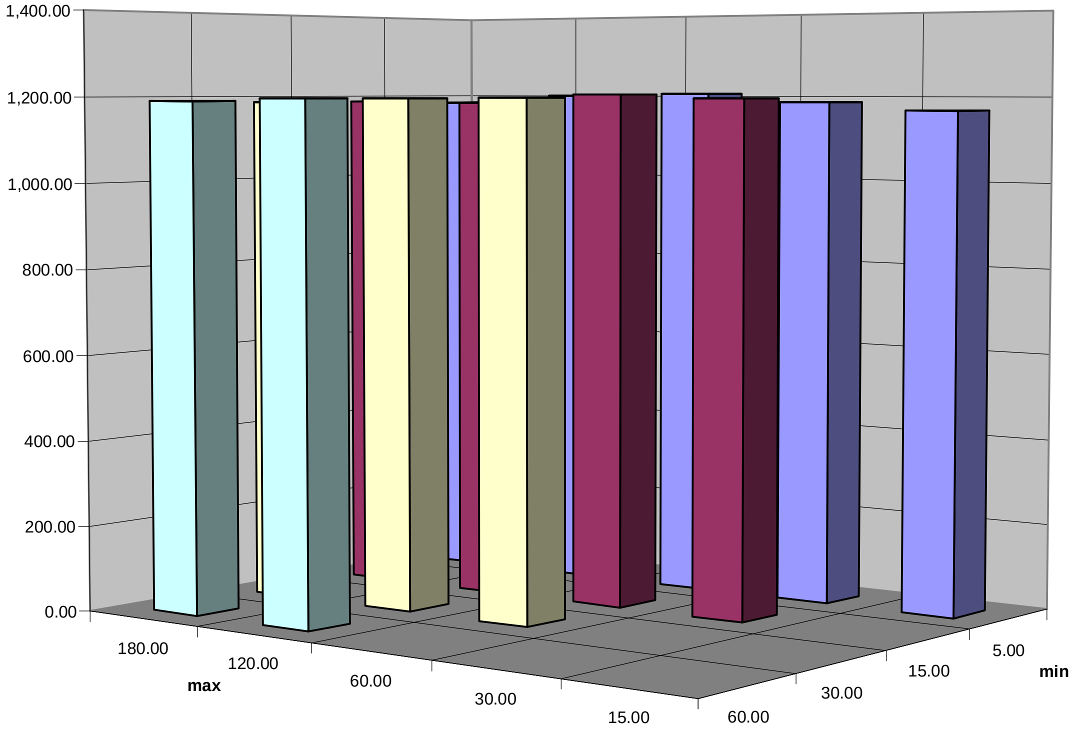


MinTh	MaxTh				
	15	30	60	120	180
5	983	1,008	1,016	1,011	1,008
15		1,011	1,015	1,007	1,010
30			1,008	1,006	1,005
60				1,007	1,009

Figure B.16 TCP Goodput (KB/s) During Multimedia Measurement Period with RED for BULK and Proshare.

Just as with HTTP, this analysis begins by examining the TCP goodput. The TCP goodput during the multimedia measurement period, shown in Figure B.16, is similar to what was observed with HTTP and Proshare. The goodput is relatively uniform, although the goodput is a little worse for the maximum threshold value of 15. However, no parameter settings are eliminated based on this performance metric.

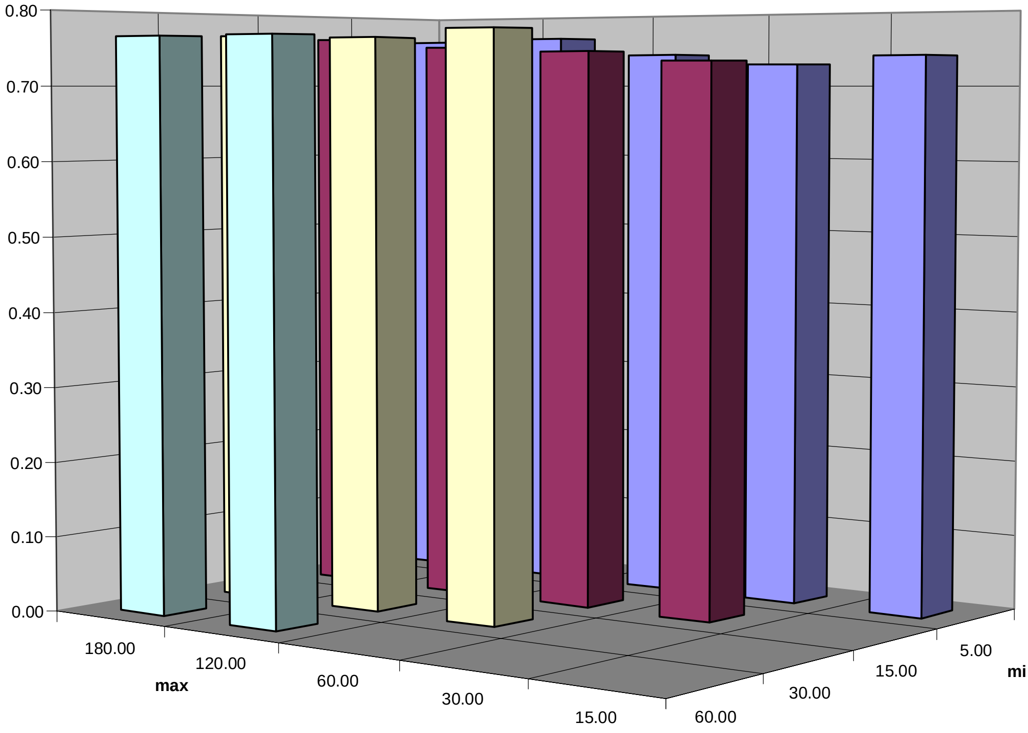
Next consider aggregate throughput on the bottleneck link during the same period. Figure B.17 shows the aggregate throughput on the outbound link. Once again the results are uniform so parameter settings are eliminated.



MinTh	MaxTh				
	15	**30**	**60**	**120**	**180**
5	1,168	1,187	1,206	1,201	1,183
15		1,197	1,206	1,185	1,187
30			1,199	1,196	1,186
60				1,197	1,190

Figure B.17 Aggregate Throughput (KB/s) during Multimedia Measurement Period with RED for BULK and Proshare

Next, RED’s efficiency without the blast is shown in Figure B.18. About 75% of the inbound TCP traffic reaches the outbound link. Once again, changing the thresholds does not affect the efficiency so no parameters can be eliminated based on this metric.

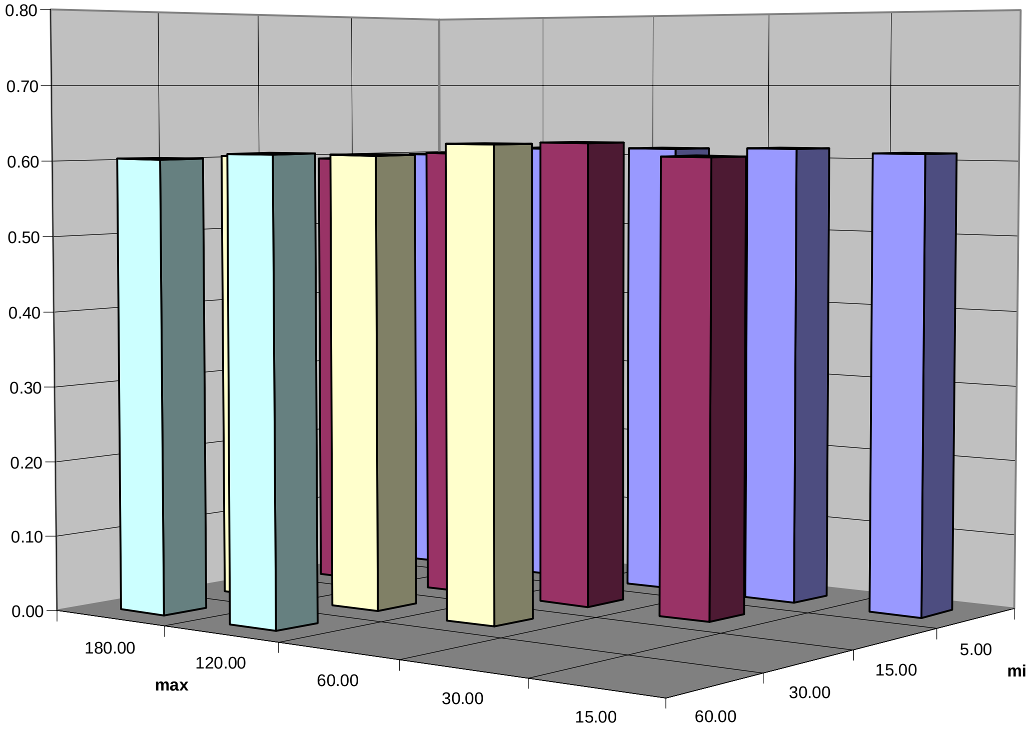


MinTh	MaxTh				
	15	30	60	120	180
5	0.74	0.73	0.74	0.77	0.76
15		0.73	0.75	0.75	0.77
30			0.78	0.76	0.77
60				0.77	0.77

Figure B.18 TCP Efficiency during Multimedia Measurement Period with RED for BULK and Proshare

Recall that using the HTTP and Proshare traffic mix efficiency values between 94% and 98% were observed during the multimedia measurement period. This is because HTTP and Proshare generated a load near the capacity of the bottleneck link. However, BULK alone is able to exceed the capacity of the bottleneck link. As a result, BULK's behavior during the multimedia measurement period is somewhere between that of HTTP during the measurement and blast periods. The network is more overloaded than the multimedia measurement period for HTTP and Proshare, but less overloaded than the blast measurement period. As a result, the BULK flows are constantly probing for excess capacity and then backing off again, leading to the overload.

Next, consider TCP efficiency during the blast measurement period, as shown in Figure B.19. RED consistently gives 61-63% efficiency during the blast. This measure does nothing to indicate a preferred threshold setting.

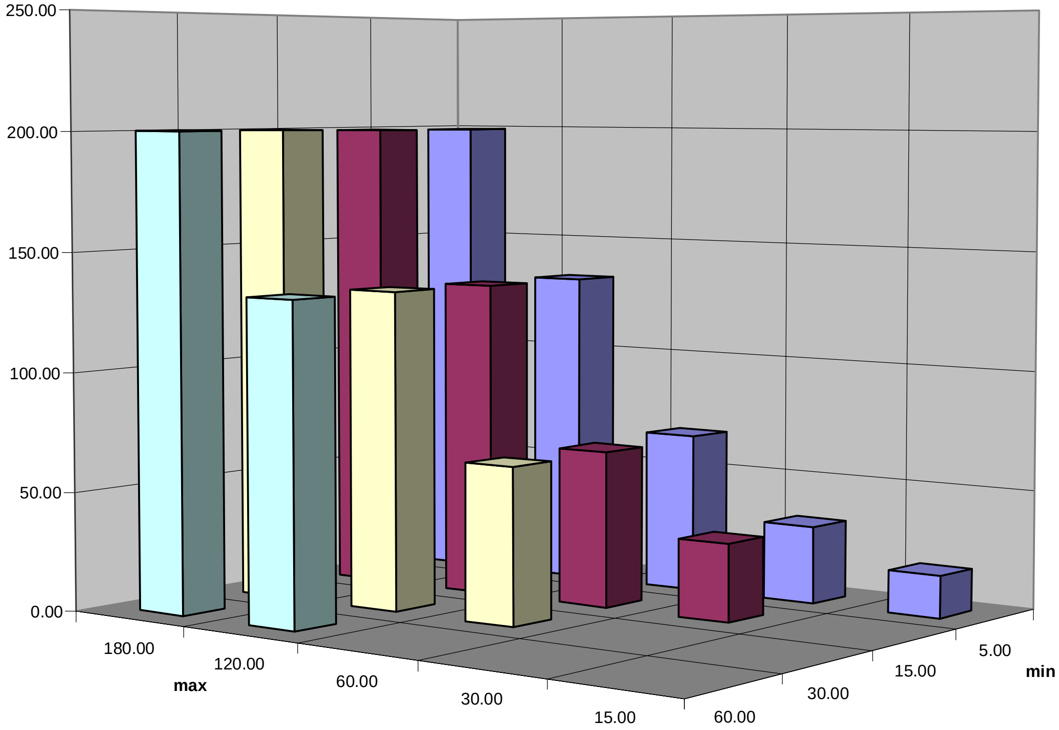


MinTh	MaxTh				
	15	30	60	120	180
5	0.61	0.62	0.61	0.61	0.60
15		0.61	0.62	0.61	0.60
30			0.62	0.61	0.60
60				0.61	0.60

Figure B.19 TCP Efficiency during Multimedia Measurement Period with RED for BULK and Proshare

Finally, consider network latency in order to try to identify the optimal threshold settings. Figure B.20 shows the network latency measured using the instrumented Proshare traffic generator. The latency increases as the maximum threshold increases but is independent of the minimum threshold. The queue-induced latency latency equation (B.3) holds again. BULK packets are bigger on average than the HTTP packets, resulting in average packet size of ~1,400 bytes. For this average packet size, a link capacity of 10 Mb/s, and a threshold of 60, latency of 67 ms is expected and a latency of 66 ms is observed for thresholds (5,60).

Note that the HTTP and Proshare mix only reached the expected latency limit during the blast measurement period while BULK and Proshare reached the limit during the multimedia measurement period. This happens because the BULK traffic alone can overload the bottleneck link, resulting in near constant overload and a queue occupancy near the maximum threshold. With HTTP and Proshare, this network state only occurred when the blast was active.



MinTh	MaxTh				
	15	30	60	120	180
5	17	32	66	133	198
15		32	65	133	199
30			65	133	199
60				133	200

Figure B.20 Network Latency (ms) during Multimedia Measurement Period for RED with BULK and Proshare

These latency results reiterate the observations from the HTTP experiments, keeping the maximum threshold small minimizes latency. Those values that result in latency greater than 50 milliseconds, those with maximum threshold of 60 or greater, are eliminated. Since none of the results from the BULK experiments conflict with the observations with HTTP traffic, and HTTP traffic accounts for 70% of the flows in the Internet, the previously selected threshold values of (5,40) are maintained.

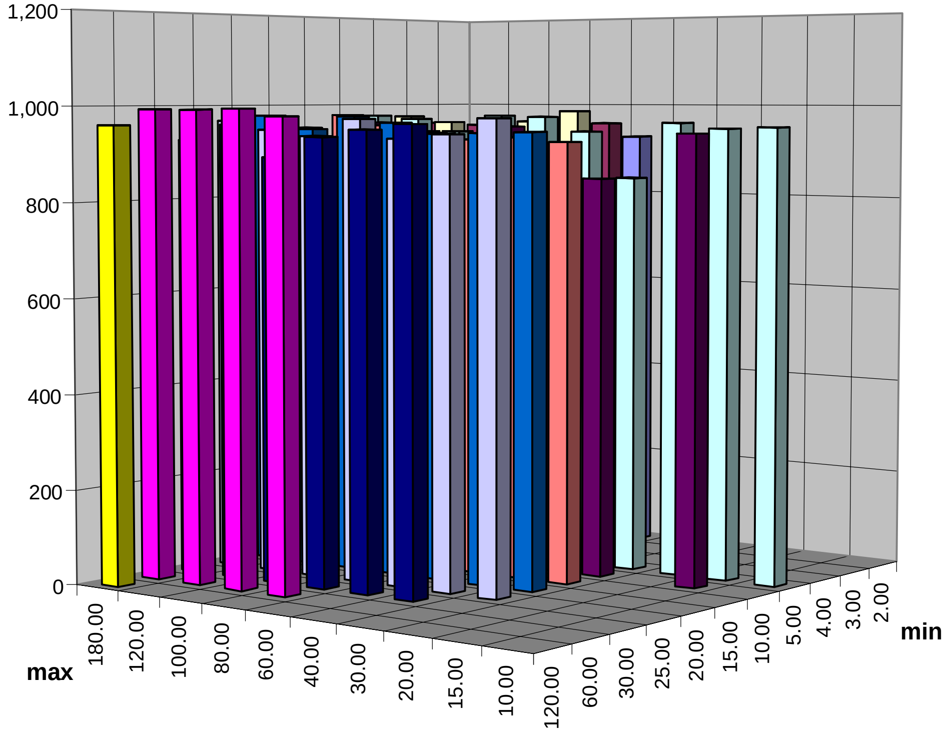
5. FRED

Next, consider the selection of optimal parameters for FRED. As with RED, practical considerations demand that this work focus on a subset of the parameters. As with RED, the two parameters expected to have the greatest impact are the minimum and maximum thresholds. The other parameters are held constant (Section 3.2) and the effects of ranging the minimum and maximum thresholds are examined. After determining the values for the constant parameters, the next task was determining the range of threshold combinations to consider. As with RED, every possible combination of threshold settings was not explored. Further, once trends were established from examining single runs for all the selected parameter combinations, the focus turned to the optimal range of combinations and additional experiments were conducted with those settings to validate those results. Table B.4 shows the parameter combinations considered and the number of runs for each. The criteria for evaluating these parameters is explained below.

MinTh	MaxTh									
	10	15	20	30	40	60	80	100	120	180
2						5	5	5	5	5
3						5	5	5	5	5
4						5	5	5	5	5
5	1	1	1	1	5	5	5	5	5	5
10		1		1	5	5	5	5	5	5
15				1	5	5	5	5	5	5
20				1	5	5	5	5	5	5
25				1	5	5	5	5	5	5
30					5	5	5	5	5	5
60							5	5	5	5
120										5

Table B.4 Count of FRED Experiments for Each Parameter Combination with HTTP-Proshare

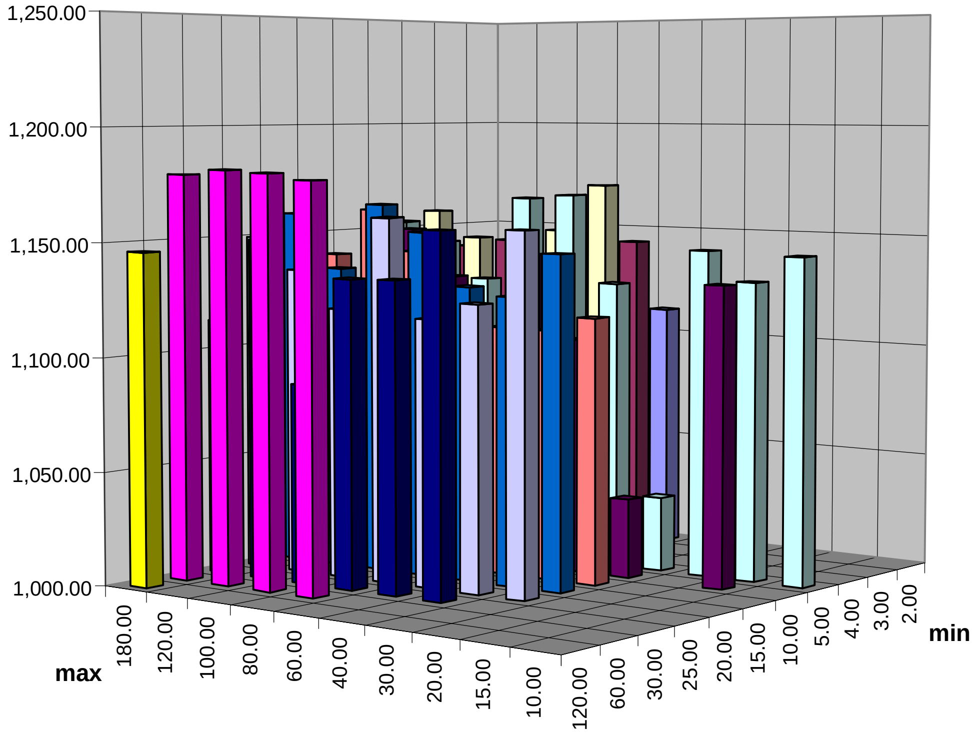
Choosing the optimal parameter settings for FRED differs from RED in that while FRED's primary goal is to offer good performance for responsive flows (i.e. TCP) it also has a secondary goal of encouraging fairness. FRED seeks to divide the network bandwidth fairly between flows, thereby limiting the effect of unresponsive flows. Thus, the first metric examined is the TCP goodput during periods with only TCP and Proshare running. Next, the overall link utilization is considered. If necessary, how *other* traffic is constrained will be examined. Finally, latency will be considered.



MinTh	MaxTh									
	10	15	20	30	40	60	80	100	120	180
2						928	885	912	890	919
3						959	942	929	953	957
4						987	964	897	960	973
5	956	952	964	845	944	975	977	942	969	976
10		943		846	920	955	910	943	961	882
15				926	934	929	942	957	979	950
20				947	944	942	964	977	949	978
25				976	943	933	973	936	948	967
30					965	953	937	893	961	926
60							980	995	993	993
120										960

Figure B.21 TCP Goodput (KB/s) During the Multimedia Measurement Period with FRED for HTTP-Proshare

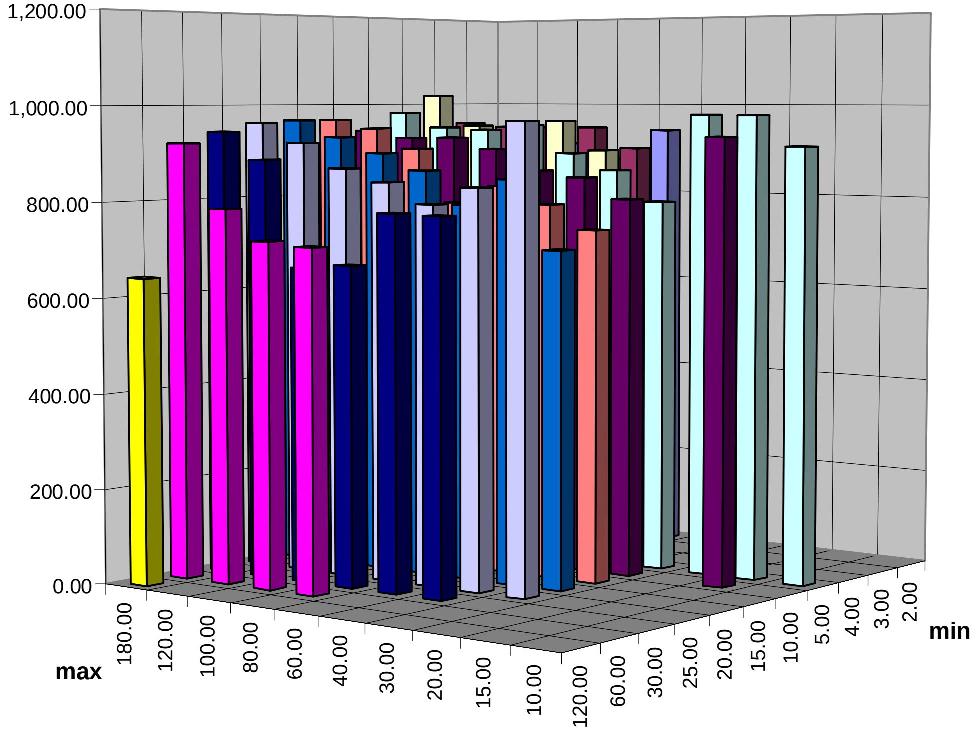
Figure B.21 shows TCP goodput during the multimedia measurement period. As with RED, the performance is fairly uniform. However, a Th_{Min} value of 2 has throughput that is 10% worse than larger settings. A few other scattered data points also offer lower performance and are so indicated with shading. No parameter settings are eliminated based on these parameter settings alone. However, they will be shaded lightly to note their suspect performance thus far. To further refine the parameter settings, next consider overall link utilization.



MinTh	MaxTh									
	10	15	20	30	40	60	80	100	120	180
2						1,111	1,067	1,090	1,082	1,109
3						1,145	1,123	1,120	1,143	1,140
4						1,172	1,150	1,083	1,145	1,158
5	1,144	1,132	1,145	1,033	1,128	1,168	1,166	1,128	1,145	1,153
10		1,132		1,035	1,105	1,142	1,094	1,130	1,151	1,072
15				1,117	1,110	1,111	1,126	1,143	1,161	1,140
20				1,146	1,127	1,130	1,153	1,165	1,135	1,159
25				1,157	1,125	1,117	1,160	1,119	1,136	1,150
30					1,157	1,135	1,135	1,088	1,150	1,113
60							1,178	1,180	1,181	1,179
120										1,146

Figure B.22 Aggregate Throughput (KB/s) during Multimedia Measurement Period with FRED for HTTP-Proshare

As with RED’s link utilization plot, the scale in Figure B.22 has been adjusted to range from 1000-1250 KB/s. The previously noticed performance drop-off Th_{Min} = 2 continues, but no other trends are obvious. Although the differences are still small these parameter combinations are eliminated based on the performance across all of the metrics considered up to this point.



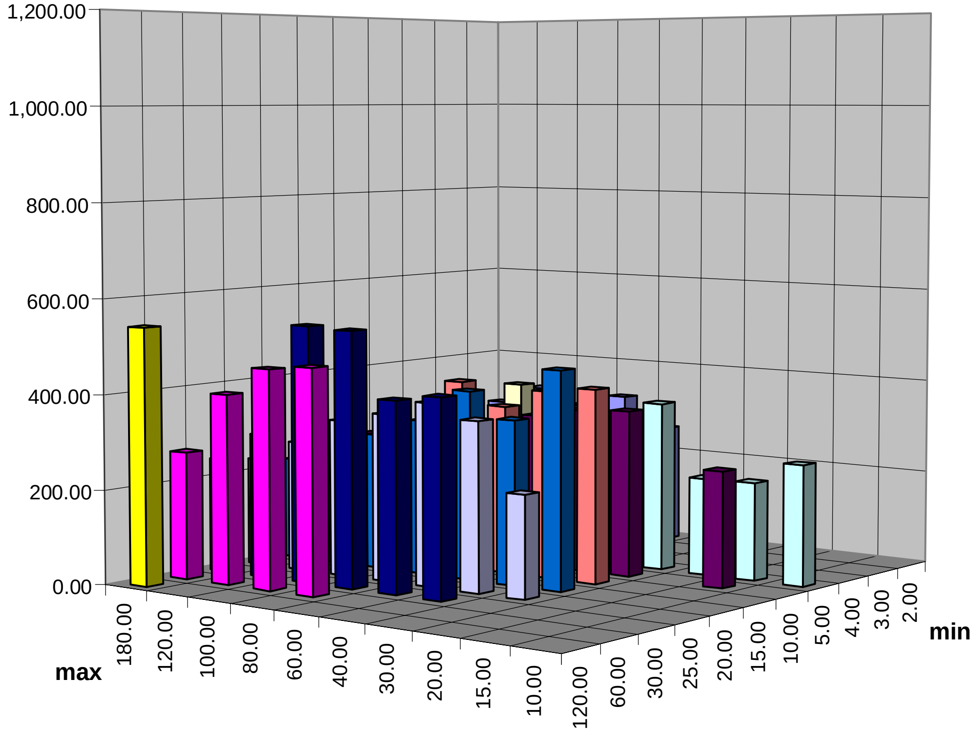
MinTh	MaxTh									
	10	15	20	30	40	60	80	100	120	180
2						941	875	908	869	911
3						902	948	947	948	955
4						899	963	838	951	1,019
5	916	980	981	793	858	894	956	944	948	981
10		936		802	846	859	903	928	926	941
15				741	792	828	790	905	948	966
20				704	847	791	862	897	930	966
25				970	833	796	839	867	920	962
30					780	782	671	661	885	944
60							713	722	785	921
120										642

Figure B.23 TCP Throughput (KB/s) during the Blast Measurement Period with FRED for HTTP-Proshare

Figure B.23 shows the TCP throughput during the blast measurement period. The trend shown indicates that TCP throughput increase as Th_{Max} increases and as Th_{Min} decreases. This is because these parameter settings restrict the percentage of the queue allocated to a given misbehaving flow. That is, each misbehaving flow can use up to Th_{Min}

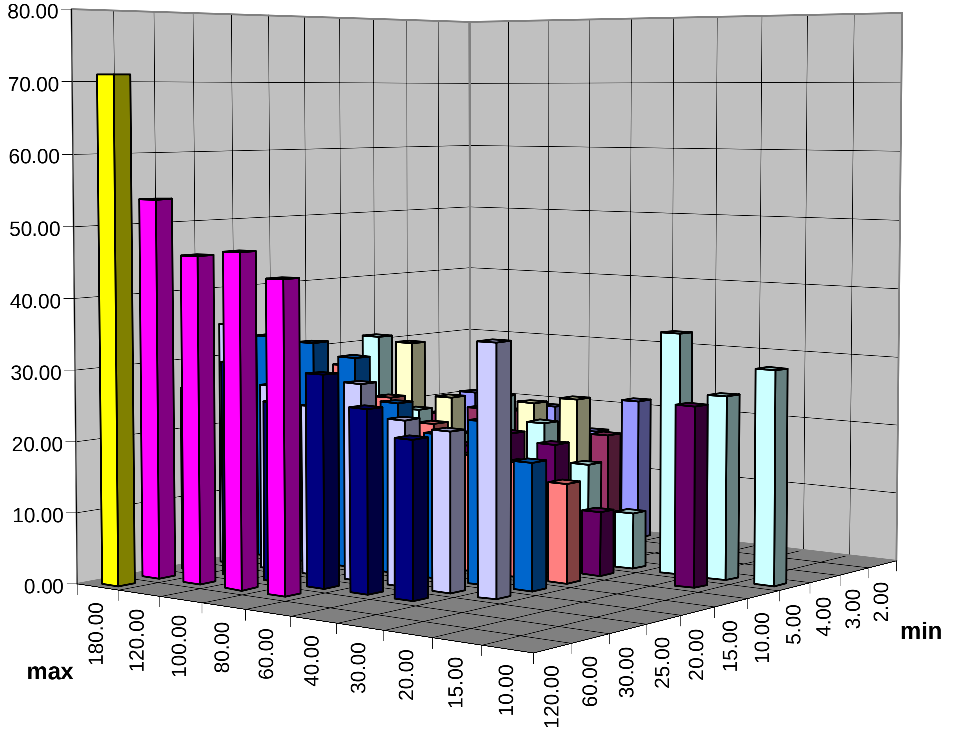
queue slots. The ratio between Th_{Min} and Th_{Max} determines the portion of the link bandwidth a single flow can use. The misbehaving flow is limited to a share of the queue equal to Th_{Min}/Th_{Max}. Thus, as Th_{Min} approaches Th_{Max}, TCP throughput decreases. Note, however, that small values of Th_{Max} are an exception to this trend, with TCP demonstrating good throughput in those cases. One theory for this behavior is that with such a small Th_{Max} the queue behaves like a FIFO queue of length equal to Th_{Max}. Consequently, some TCP flows are able to make progress although others may starve. The aggregate TCP throughput remains high, since the class *other* is constrained. However, the TCP efficiency values presented later will demonstrate that TCP does suffer in these situations. Consequently, at this point the parameter combinations with poor TCP throughput are eliminated. These are the parameter combinations with Th_{Min} as a significant fraction of Th_{Max}, along the diagonal from the upper left to the lower right.

To further refine the selection, consider the throughput of *other* traffic during the blast measurement period. The goal is to minimize the share of bandwidth given to *other* traffic. Figure B.24 shows the overall UDP throughput. Since UDP and TCP are the only traffic types in the network and the overall link utilization was constant during this period, UDP throughput is the compliment of TCP throughput. Clearly, UDP throughput decreases as Th_{Max} increases and as Th_{Min} decreases, with the only exception being when $Th_{Min} = 2$. No new parameter combinations are eliminated.



MinTh	**MaxTh**									
	10	**15**	**20**	**30**	**40**	**60**	**80**	**100**	**120**	**180**
2						256	321	292	326	285
3						291	243	251	244	240
4						292	230	361	239	166
5	255	208	206	359	330	285	232	250	235	199
10		245		353	347	327	288	264	262	252
15				410	397	357	403	284	236	226
20				459	347	401	329	289	260	218
25				215	357	385	353	331	275	230
30					416	401	537	542	303	242
60							470	460	399	269
120										541

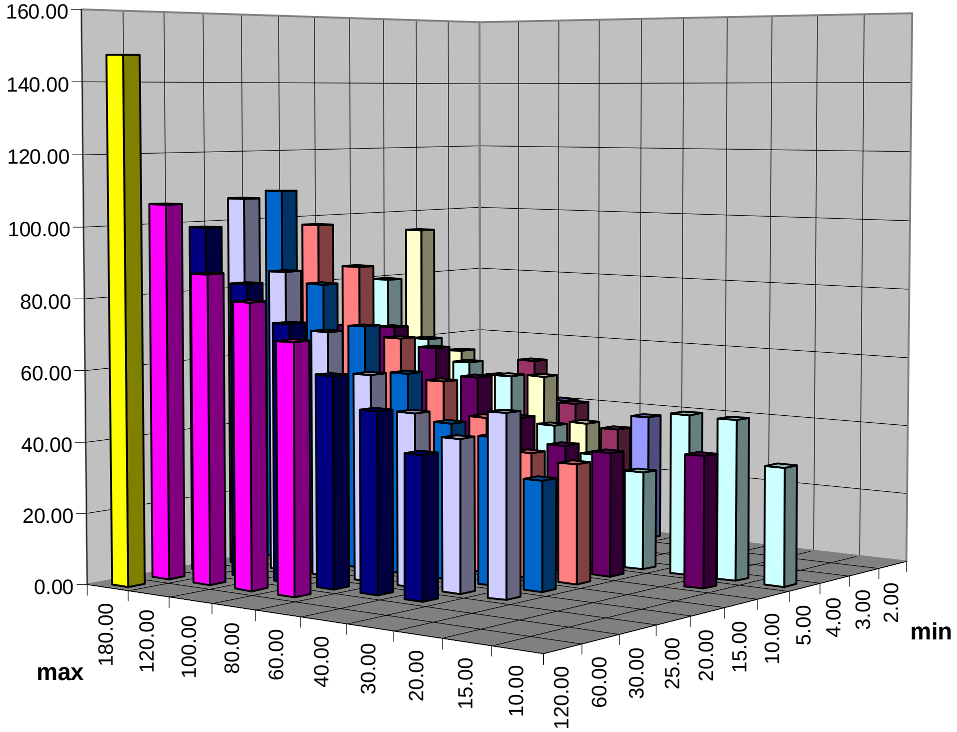
Figure B.24 Aggregate UDP Throughput (KB/s) during the Blast Measurement Period with FRED for HTTP-Proshare



MinTh	MaxTh									
	10	15	20	30	40	60	80	100	120	180
2						21	16	19	16	20
3						17	16	19	19	18
4						23	22	16	22	30
5	30	26	34	8	14	20	23	17	20	31
10		25		9	18	19	17	19	23	18
15				14	16	17	21	24	28	26
20				18	23	21	24	30	32	33
25				35	22	23	28	24	27	35
30					22	26	30	25	30	26
60							43	47	46	54
120										71

Figure B.25 Network Latency (ms) during the Multimedia Measurement Period for FRED with HTTP-Proshare

The next criteria for selecting the parameters is network latency. Figure B.25 shows the network latency for FRED during the multimedia measurement period. The latency increases with Th_{Max}. The trend is more apparent during the blast measurement period, shown in Figure B.26.

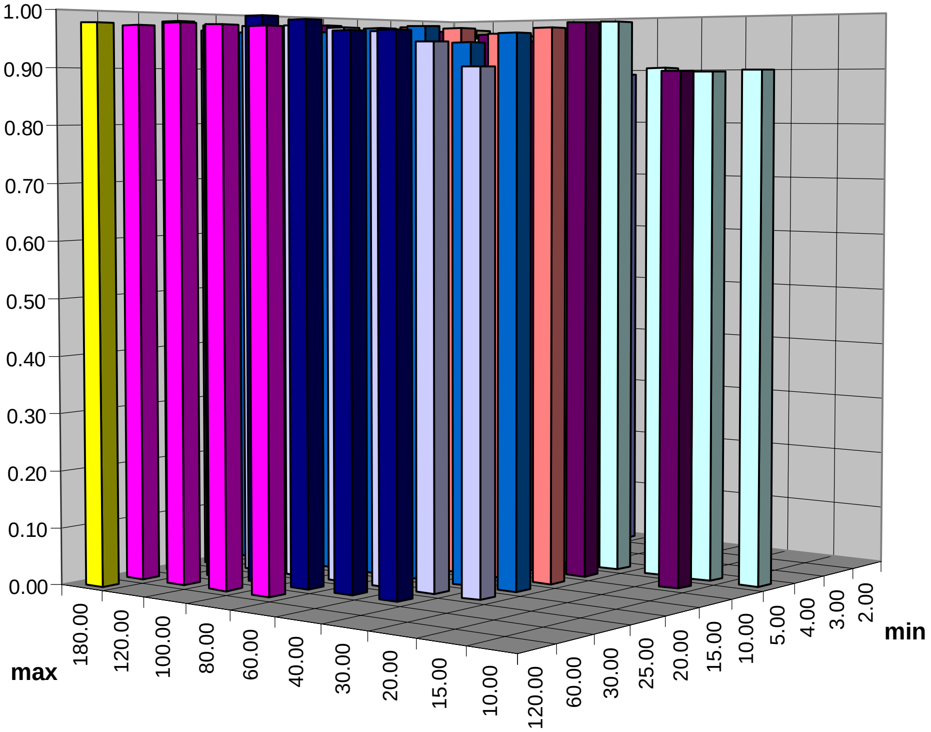


MinTh	MaxTh									
	10	15	20	30	40	60	80	100	120	180
2						37	27	41	29	32
3						35	42	54	45	49
4						39	52	51	58	94
5	33	45	46	28	32	39	53	56	62	80
10		37		35	36	43	54	62	67	67
15				34	36	45	54	66	86	98
20				31	42	44	57	70	82	108
25				51	43	49	58	70	86	107
30					40	51	59	73	83	99
60							70	80	87	106
120										147

Figure B.26 Network Latency (ms) during the Blast Measurement Period with FRED for HTTP-Proshare

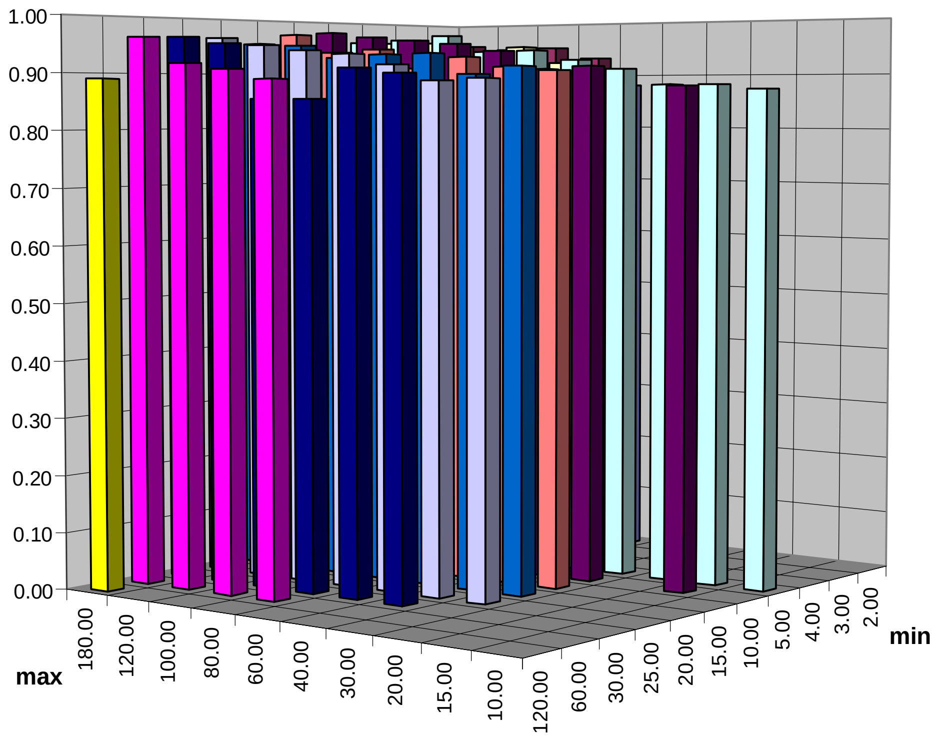
Although latency is generally unrelated to Th_{Min}, very small values of Th_{Min} also lower latency. However, this lower latency is obtained by maintaining a lower average queue size. These small queue sizes come at the price of increased drops. Recall the TCP throughput results for Th_{Min} = 2. Although this setting offers lower latency, as shown in Figure B.25 and Figure B.26, prior results indicated that with Th_{Min} = 2 the throughput and the efficiency values were worse than for the other parameter settings. Based on the data from the multimedia measurement period, many parameter combinations offer reasonable latency.   However, the data from the blast measurement period indicates that the differences are clearer cut. The latency clearly decreases as Th_{Max} decreases and as Th_{Min} decreases. Eliminating all latency values of 40 ms or more leaves only three values with Th_{Max} equal to 60 and two with Th_{Max} equal to 10 and 15.

To make the final decision on optimal parameters, consider TCP efficiency, shown below in Figure B.27. During the multimedia measurement period the efficiency is generally 95-99% uniformly across all the combinations of Th_{Min} and Th_{Max}. This is shown in Figure B.27. The exceptions to this seem to be for small values of Th_{Min} (< 3) and small Th_{Max} (<30). The efficiency values during the blast measurement period support this finding as well. Since the efficiency metric eliminates values with Th_{Max} less than 30, only the parameter combinations: (60,3), (60,4), and (60,5) remain. The combination (60,5) is selected as the optimal parameter combination based on its slightly higher efficiency.



MinTh	MaxTh									
	10	15	20	30	40	60	80	100	120	180
2						0.89	0.90	0.91	0.91	0.90
3						0.95	0.95	0.97	0.96	0.95
4						0.94	0.95	0.97	0.96	0.96
5	0.90	0.89	0.90	0.98	0.95	0.96	0.96	0.96	0.96	0.96
10		0.89		0.98	0.95	0.96	0.97	0.97	0.96	0.98
15				0.97	0.96	0.97	0.97	0.96	0.96	0.98
20				0.96	0.94	0.97	0.97	0.96	0.97	0.97
25				0.90	0.94	0.96	0.97	0.97	0.97	0.97
30					0.96	0.96	0.98	0.99	0.97	0.98
60							0.97	0.97	0.98	0.97
120										0.98

Figure B.27 TCP Efficiency during the Multimedia Measurement Period with FRED for HTTP-Proshare



MinTh	MaxTh									
	10	15	20	30	40	60	80	100	120	180
2						0.88	0.88	0.90	0.89	0.88
3						0.93	0.95	0.95	0.96	0.96
4						0.92	0.95	0.92	0.96	0.96
5	0.87	0.88	0.88	0.91	0.92	0.94	0.94	0.97	0.96	0.96
10		0.88		0.91	0.91	0.94	0.95	0.96	0.97	0.98
15				0.90	0.91	0.93	0.91	0.94	0.94	0.97
20				0.91	0.90	0.93	0.93	0.93	0.95	0.95
25				0.89	0.89	0.91	0.93	0.94	0.95	0.96
30					0.90	0.91	0.85	0.85	0.95	0.96
60							0.89	0.91	0.92	0.96
120										0.89

Figure B.28 TCP Efficiency during the Blast Measurement Period for FRED with HTTP-Proshare

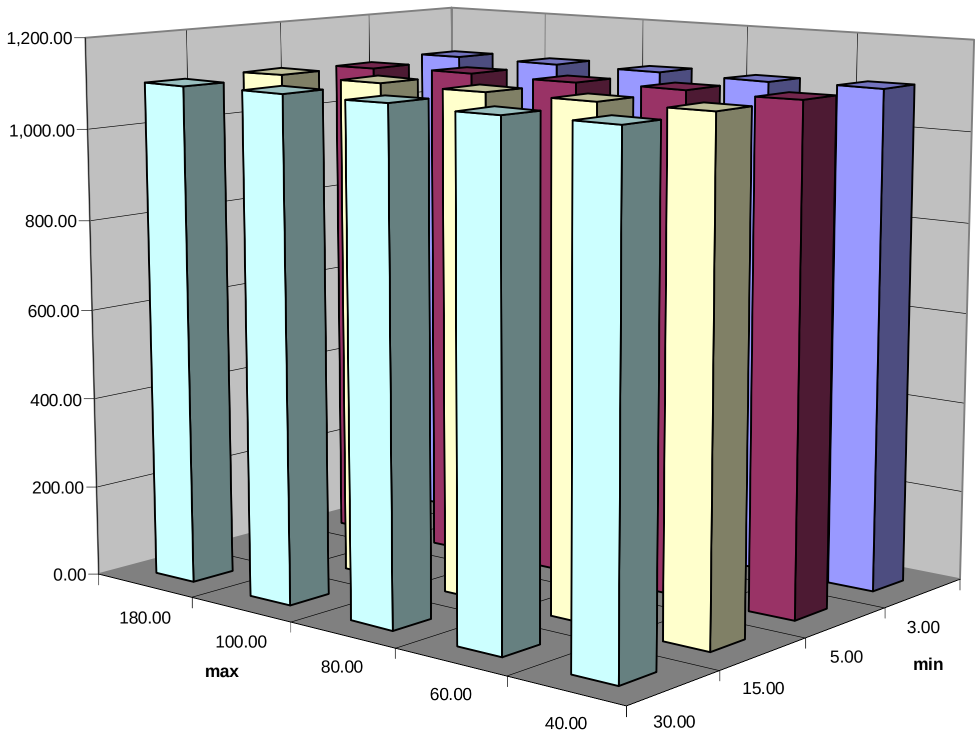
5.1.1. BULK-Proshare

Next, consider BULK-Proshare. As with RED, this set of experiments only considers a subset of the parameter space considered for HTTP-Proshare, simply seeking to confirm that the previously observed relationships continue to hold. If so, the optimal parameter settings established for HTTP-Proshare will be maintained. However, in fact, the relationships previously noted do not hold. Changing the parameter settings has no impact on the performance measured. All parameter settings yield the same results. This behavior can be explained. There are a large number of long lived flows in the BULK traffic type and FRED with support for many flows allows each flow to buffer at least two packets. Since the BULK traffic generators generate sufficient load to saturate the bottleneck link, the per flow buffer allocation overrides the standard RED drop mechanisms and the queue remains full whenever BULK traffic is present. Because these experiments will show that all parameter combinations give the same results, the 3-D bar charts that have been used to illustrate the effects of different parameter combinations hold little value. One chart is included for the first metric considered in order to demonstrate this lack of value. For the remaining metrics, the results are presented only in tabular format and examined briefly. Table B.5 shows the number of runs for each parameter combination.

MinTh	MaxTh				
	40	60	80	100	180
3	5	5	5	5	5
5	5	5	5	5	5
15	5	5	5	5	5
30	5	5	5	5	5

Table B.5 Count of FRED Experiments for Each Parameter Combination with BULK-Proshare

First, consider the TCP goodput during the multimedia measurement period, shown in Figure B.31.   The goodput ranges varies by only 7 KB/s.   Varying the parameters has no effect. Moreover, because there is no variation in results, displaying the data in the chart carries little value beyond that apparent from the table. Since the remaining metrics also have little variation, they will be presented only in tabular format.



MinTh	MaxTh				
	40	60	80	100	180
5	1,104	1,100	1,102	1,103	1,103
15	1,103	1,104	1,101	1,103	1,097
30	1,104	1,103	1,103	1,104	1,103
60	1,103	1,100	1,104	1,103	1,101

Figure B.31 TCP Goodput (KB/s) During Multimedia Measurement Period with FRED for BULK-Proshare

Next, consider total link utilization, shown in Table B.6. Once again, there is very little difference, with all of the considered threshold combinations resulting in link utilization, by all traffic types in aggregate, of approximately 1.2 MB.

MinTh	MaxTh				
	40	60	80	100	180
5	1,188	1,198	1,190	1,190	1,194
15	1,187	1,189	1,187	1,196	1,181
30	1,190	1,188	1,197	1,202	1,187
60	1,199	1,190	1,183	1,188	1,189

Table B.6 Aggregate Throughput (KB/s) During Multimedia Measurement Period with FRED for BULK-Proshare

Given that one of FRED’s goals is to constrain misbehaving flows, TCP throughput during the blast measurement period must be considered. Table B.7 shows TCP throughput on the congested link during the blast measurement period. Since the blast is a single flow, this mechanism is very effective in constraining the blast and, thus, increasing the throughput of TCP. TCP throughput is very high during this period. However, the performance is uniform across parameter combinations so this data indicates no preferred parameter settings.

MinTh	MaxTh				
	40	60	80	100	180
5	1,179	1,175	1,177	1,175	1,177
15	1,177	1,176	1,178	1,179	1,174
30	1,178	1,176	1,179	1,178	1,176
60	1,179	1,178	1,176	1,179	1,177

Table B.7 TCP Throughput (KB/s) During Blast Measurement Period with FRED for BULK-Proshare

FRED's effectiveness constraining the blast is further supported by examining the UDP throughput during the same period as shown in Table B.8. Although the UDP throughput is very low, its performance is again uniform across all parameter

MinTh	MaxTh				
	40	60	80	100	180
5	46	49	48	50	48
15	48	47	47	46	50
30	46	49	46	47	49
60	46	46	48	46	48

Table B.8 Aggregate UDP Throughput (KB/s) during the Blast Measurement Period with FRED for BULK-Proshare

The next metric to consider is latency. Table B.9 shows the network latency during the multimedia measurement period as measured using the instrumented Proshare application. Since queue size determines latency and the queue is always full, latency is very high. Again the latency values are uniform.

MinTh	MaxTh				
	40	60	80	100	180
5	264	257	255	251	256
15	261	253	270	269	249
30	265	257	265	264	263
60	268	262	253	270	263

Table B.9 Network Latency (ms) During Multimedia Measurement Period for FRED with BULK-Proshare

The latency during the blast measurement period, shown in Table B.10, is similar. The latency is limited only by the queue size of 240 packets. Because it takes slightly more than 1 ms to process each packet, the average latency is approximately 265 ms.

MinTh	MaxTh				
	40	60	80	100	180
5	270	247	261	246	257
15	258	261	269	270	255
30	269	252	269	263	266
60	270	267	255	268	268

Table B.10 Network Latency (ms) During Blast Measurement Period for FRED with BULK-Proshare

Finally, consider TCP efficiency (percentage of inbound traffic that reaches the outbound link) during the blast measurement period. Table B.11 shows that the combination of the parameters has no effect on efficiency. The efficiency is consistently between 82-83 %.

MinTh	MaxTh				
	40	60	80	100	180
5	0.83	0.83	0.83	0.83	0.83
15	0.82	0.83	0.82	0.82	0.82
30	0.82	0.83	0.82	0.82	0.83
60	0.82	0.83	0.83	0.82	0.82

Table B.11 TCP Efficiency during Blast Measurement Period with FRED for BULK-Proshare

FRED efficiency during the multimedia measurement period (Table B.12) is similar. Once again, the efficiency ranges between 80-83%.

MinTh	MaxTh				
	40	60	80	100	180
5	0.80	0.82	0.81	0.82	0.82
15	0.81	0.81	0.80	0.81	0.81
30	0.81	0.81	0.81	0.83	0.81
60	0.82	0.82	0.81	0.80	0.81

Table B.12 TCP Efficiency During the Multimedia Measurement Period for FRED with BULK-Proshare

The results of the BULK-Proshare experiments show changing parameter settings has no effect for this traffic mix. Since the BULK-Proshare analysis doesn't indicate an optimal setting, the previously selected threshold values of (Th_{Max}, Th_{Min}) = (60,5) are maintained since HTTP is the most common traffic type.

6. CBT

Unlike the other queue management techniques considered, Class Based Thresholds explicitly allocates resources to different classes of traffic. The other mechanisms sought to manage the queue while CBT seeks to manage the performance classes of traffic receive as they pass through the router. Because CBT does allocate resources, one can calculate the parameter settings that should offer optimal performance based on knowledge of the traffic types involved. As a result, selection of optimal parameters for CBT focuses on confirming that the calculated parameters are optimal rather than probing a wide range of parameters. This confirmation is necessary despite the availability of information about the traffic, such as the average load generated by different traffic classes over a measurement period. This is because the average load over shorter intervals, on the order of a few seconds, is unknown. Such intervals may be important for traffic types, such as multimedia that demonstrate variability in generated load over those time scales. The process used to calculate and evaluate the optimal values is discussed below. But first, the choice of settings for those parameters held constant across all experiments is discussed.

6.1. Constant Parameters

CBT takes a large set of parameters. In addition to the maximum queue size, each class of traffic has a weight (*w*), a maximum drop probability (*maxp*), and minimum (Th_{min}) and maximum (Th_{Max}) thresholds. To limit the number of variables considered this analysis focuses on only ranging the parameters that should have the greatest effect on performance: the maximum thresholds. It is possible that the other parameter settings could be fine-tuned to offer even better performance but those effects are expected to be secondary.

The weighting factors used in tracking each class's average queue occupancy are one of the parameters that held fixed across all experiments. In order to have efficient calculations of the average queue occupancy a weighting factor that is a power of two is used so that the algorithm can use shift operations instead of multiplication. This limits the options for the weighting factor value. Next, since the queue occupancy for each class is sampled when a packet of that class arrives at the queue, and, since the classes have different arrival rates, the averages for the classes may have different sample rates. For two classes of traffic with the same arrival rates, the weighting factors effectively determine how much effect new samples have on the average and, in some sense, determine the period represented by the average [Floyd93]. A higher weighting factor leads to an average that reflects the effect of recent packet arrivals on the queue while a lower weighting factor leads to an average that reflects the effect of older packet arrivals on the queue. However, if packet arrival rates differ across class, in order for all of the averages to indicate behavior over the same approximate time scale, the weighting factors must be adjusted to achieve this effect. However, it is desirable to have some classes be more sensitive to recent queue occupancy than others. Because most of the arrivals at the queue should be TCP and because TCP should receive feedback only when there is sustained overload, but yet allow bursts, a weighting factor of 1/256 is used for TCP. Conversely, multimedia, with its periodic frame transmission should have a much lower arrival rate at the queue. Moreover, the goal is to constrain multimedia to its allocated share. As a result multimedia's weighting factor is set to 1/16. This limits the burstiness accommodated but this is acceptable since multimedia's transmission has limited variability. The desire to accommodate some burstiness to allow for large frames fragmented across many packets accounts for the setting of 1/16 instead of a lower weighting factor that would be more sensitive to bursts. Finally, the traffic class *other* may demonstrate extremely aggressive behavior generating a very high arrival rate at times. This argues for a small weighting factor (as more samples arrive during a given period). However, the goal is to very strictly constrain *other* traffic to its fair share. As a result, each new sample has a very high weighting factor, ¼. Using this weighting factor insures that *other* traffic is tightly constrained. The process of selecting weighting values is discussed in IV.4.2.1 and the potential for additional work in this area in VI.5.1.1.

Additionally, there are some parameters, *maxp* and Th_{Min}, that can be eliminated for some classes of traffic. Recall that the goal is only to constrain, not provide feedback to the traffic classes of multimedia and *other*. Consequently the random drop mode of the RED drop algorithm is not used. Instead, the minimum thresholds for the unresponsive traffic classes of traffic is set equal to their maximum thresholds, leaving the drop algorithm to operate in two modes, no drops or forced drop. As a result, the average queue occupancy for those classes has a fixed limit equal to the maximum threshold. It may be possible to provide effective feedback to responsive multimedia using Th_{Min}; however, that issue is left for future work. As long as the multimedia and *other* classes do not exceed the maximum threshold, no packets are dropped. Further, because the minimum and maximum thresholds are equal, the maximum drop probability isn't used for those classes. However, to provide feedback to responsive traffic the minimum threshold and maximum drop probability must be set to meaningful values for TCP traffic. For TCP the values were a maximum drop probability of 10% and a minimum threshold of 5. These values are both the recommended values for RED and the values determined as part of the optimal parameter set in the RED experiments. Table B.13 shows the settings held constant for the CBT algorithm.

Traffic Class	***w***	***maxp***	Th_{Min}
TCP	1/256	1/10	5
Multimedia	1/16	n/a	$=Th_{Max}$
Other	1/4	n/a	$=Th_{Max}$

Table B.13 Constant Parameter Settings for CBT

Finally, the desired latency was, arbitrarily, held fixed at 100 ms. Although latency is not actually a parameter of the CBT algorithm, is it a factor in the calculation of the maximum threshold parameters. As already demonstrated in IV.5.1.2, the parameter settings do control the maximum average latency independent of the bandwidth allocations. As such, the desired latency value was selected and then CBT's performance confirmed.

6.2. Determining Threshold Settings from Bandwidth Allocations

Consider the parameters that are varied between experiments: the bandwidth allocations for each class. Multimedia's performance is the key to determining whether or not the parameter settings are optimal. This translates to allocating multimedia sufficient bandwidth to maintain its desired frame-rate with low loss. Similarly, *other* traffic should be constrained to a small share of the link's capacity. 150 KB/s was arbitrarily chose as the *other* bandwidth allocation. The goal is simply to demonstrate that CBT is effective in constraining *other* traffic to a desired allocation. In contrast, the allocation for multimedia is carefully determined. Multimedia's bandwidth allocation is based on the expected load generated by the multimedia applications. For these experiments the actual load was determined by running the traffic mixes. As documented in Appendix A Proshare generates an average load of approximately 160 KB/s while MPEG generates an average load of approximately 190 KB/s so these are the respective multimedia bandwidth allocations. Once these values are established, the rest of the link's capacity is allocated to TCP. This is acceptable since TCP has no minimal acceptable allocation – it simply needs as much bandwidth as possible, ideally with little fluctuation in the available capacity so that the load generated by different TCP streams can stabilize in aggregate at the available capacity.

The threshold setting for a given class (Th_i) can be calculated using the desired latency (L), the desired bandwidth allocation for the class (B_i), and the average packet size for that class (P_i) as shown in equation B.4 (see IV.4).

$$Th_i = \frac{B_i L}{P_i} \tag{B.4}$$

CBT could, and should, be implemented by monitoring the average number of bytes enqueued instead of packets. Doing so would alleviate the need to know the average packet size for each class and make the bandwidth allocations more accurate. However, this implementation used the average queue occupancy in packets so that information is included here. In addition to measuring the average load generated for each traffic mix, the average packet size for each traffic class was also measured during each traffic mix combination. When the calculations of the thresholds for each traffic combination are presented below the observed average packet sizes for each traffic class are also indicated.

The threshold settings expected to provide optimal performance were calculated using the formula above, the desired bandwidth allocations for multimedia and *other*, and allocating the remaining capacity to TCP. To confirm these values, bandwidth allocations in a series of small increments (+/- 5 KB/s) on either side of these settings were also considered. For example, Table B.14 shows the initial bandwidth allocations considered for HTTP-MPEG in bold.

B_{other}	**B_{mm}**	**B_{TCP}**
150	*220*	*855*
150	**200**	**875**
150	**195**	**880**
150	**190**	**885**
150	**185**	**890**
150	**180**	**895**
150	*160*	*915*
150	*140*	*935*

Table B.14 Initial Bandwidth Allocations for CBT with HTTP-MPEG

As the initial results were considered, it became apparent that the more parameters needed to be considered. First, the initial increments in the allocations were too small to reveal the relationship between allocations and resulting performance. Second, in some cases the computed optimal bandwidth allocations were not actually optimal. In response to both issues, additional allocations (indicated in italics in Table B.14) were considered. Although the original process was iterative, for clarity of presentation here all parameters are analyzed as a single group.

6.3. Evaluating Parameter Settings

To actually determine which parameter settings offered the optimal performance a number of metrics were monitored in order to determine how well CBT met its goals of good multimedia performance, isolation for TCP, and constraint of *other* traffic. For more information on these performance metrics see Appendix A. The first consideration was multimedia performance. Three metrics are considered here. First, consider the accuracy of the allocation of the multimedia class. To do this, compare B_{mm} to the observed multimedia throughput on the bottleneck link during the blast measurement period. Because the allocations may not be sufficient for the multimedia load at all times the multimedia loss-rate and, for MPEG, the true and playable frame-rates relative to the B_{mm}, were also considered. If the multimedia allocation is sufficient, the frame-rate should be high. If it is inadequate, the frame-rate should suffer. Last, a latency value of 100 ms or less was confirmed for all runs.

To determine the effect the multimedia allocation has on TCP, the TCP throughput was also considered as a function of B_{mm}. TCP throughput is expected to be inversely related to B_{mm} since the TCP allocation decreases as the multimedia allocation increases. Finally, CBT's effectiveness constraining the class *other* is considered. CBT does achieve this goal but the analysis of throughput for class *other* requires some additional discussion. As a result, throughput for *other* is examined across all the traffic mixes at the end of this section.

Because bandwidth is allocated for each traffic class based on its expected load and because the average packet sizes change with the traffic mix, separate optimal parameter settings must be selected for each traffic mix in CBT. The procedure for selecting and confirming the parameter settings is the same for each traffic type. To explain the selection process, the experiments for HTTP-MPEG are presented in detail and then the results for the other three traffic mixes are summarized. In all cases, experiments were repeated five times for each parameter setting and report average values over both the measurement period and across all runs with that parameter combination and traffic-mix.

6.4. HTTP-MPEG

The first traffic mix considered is the HTTP-MPEG mix. Since MPEG has an average load of 190 KB/s, an allocation of 190 KB/s should be optimal. However, a range of multimedia bandwidth allocations above and below that setting was also considered. The allocation for *other* was fixed at 150 KB/s and TCP was allocated the remaining link capacity.

HTTP	MPEG	Other
1053	889	1075

Table B.15 Average Packet Sizes (Bytes) Including Packet Headers for HTTP-MPEG

Using the observed average packet sizes shown in Table B.15 and the desired bandwidth allocations the corresponding threshold values were calculated as shown in Table B.16. For example, consider a target bandwidth allocation for MPEG of 185 KB/s. The value for Th_{Max} for multimedia for that bandwidth allocation was calculated using equation B.4, above. Using the average packet size of 889 bytes and a desired latency of 100ms leads to:

$$Th_{Max} = \frac{\frac{185KB}{\sec} * .1\sec}{\frac{889 Bytes}{Packet} * \frac{1KB}{1024 Bytes}} = 21.3 Packets \qquad \text{(B.5)}$$

Exp.	B_{other}	B_{mm}	B_{TCP}	Th_{Max}		
				other	mm	TCP
1	150	220	855	14.28	25.33	83.15
2	150	200	875	14.28	23.03	85.09
3	150	195	880	14.28	22.45	85.58
4	150	190	885	14.28	21.87	86.07
5	150	185	890	14.28	21.30	86.55
6	150	180	895	14.28	20.72	87.04
7	150	160	915	14.28	18.42	88.98
8	150	140	935	14.28	16.12	90.93

Table B.16 CBT Parameter Settings for HTTP-MPEG with 100 ms of Latency

To determine which of these parameter settings is optimal, the accuracy of the multimedia bandwidth allocation (B_{mm}) is the first metric considered. To do this, the average multimedia throughput during the blast measurement period is compared against the multimedia bandwidth allocation in Figure B.32. At the standard scale it is difficult to discern much beyond the fact that settings for B_{mm} of 180 KB/s and higher offer slightly higher throughput.

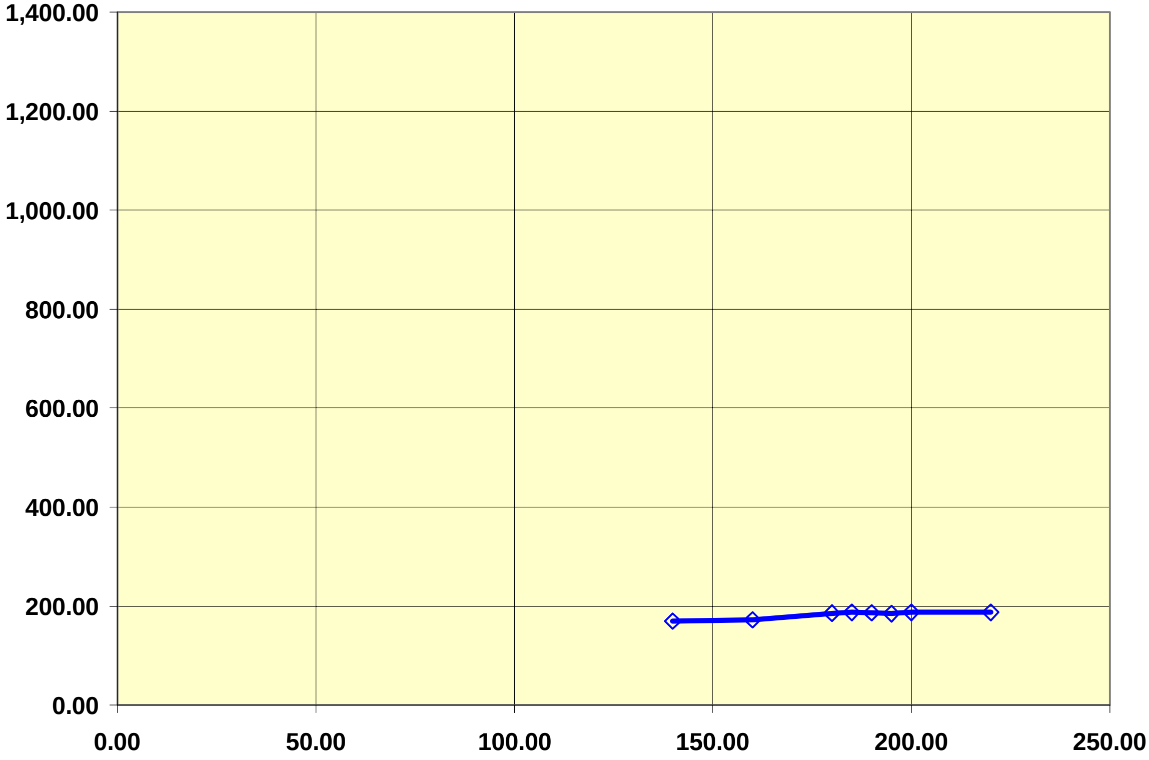


Figure B.32 Multimedia Bandwidth Allocation (KB/s) vs. Multimedia Throughput (KB/s) during Blast Measurement Period for HTTP-MPEG

To clarify the difference, Figure B.33 shows the same data at a finer scale. If the allocations are accurate, the data points would follow a line with a slope of one and a intercept at the origin. However, the values of 180 KB/s and above offer better performance with realized multimedia throughput of 184 KB/s and above. Note that the multimedia throughput is able to exceed its allocation for the settings of 140 and 160 KB/s. This is because of borrowing. Since TCP's congestion control algorithm causes oscillations in TCP's load, TCP's temporarily unused allocation is available for borrowing. Because the average load for MPEG is 190 KB/s over the duration of the movie, the suspicion is that the throughput levels off and fails to use its full allocation beyond B_{mm} of 180 KB/s simply because the throughput observed represents all of the available load. This can be seen more clearly by examining the loss-rate and frame-rates.

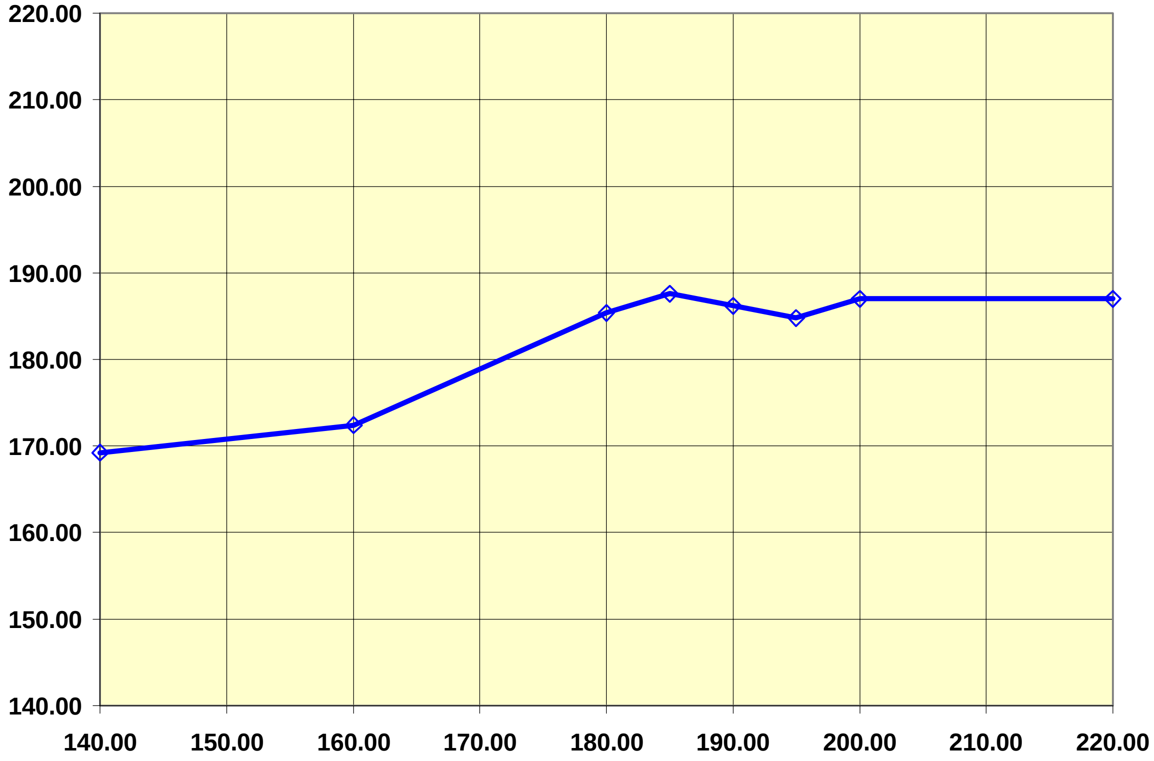


Figure B.33 Multimedia Bandwidth Allocation (KB/s) vs. Multimedia Throughput (KB/s) during the Blast Period with HTTP-MPEG on a One-to-One Scale

Figure B.34 shows the multimedia packet loss-rate during the blast measurement period. As expected, the loss-rate is on the order of 5 packets/second for the restricted B_{mm} settings of 140 and 160 KB/s. There is still a small amount of loss for settings of 180-190 KB/s and there appears to be almost no loss for the settings of 200-220 KB/s. However, because MPEG can be sensitive to packet loss (e.g., if the loss is biased against large frames, like I-frames) the best way to evaluate the effectiveness of these settings is by examining the frame-rates.

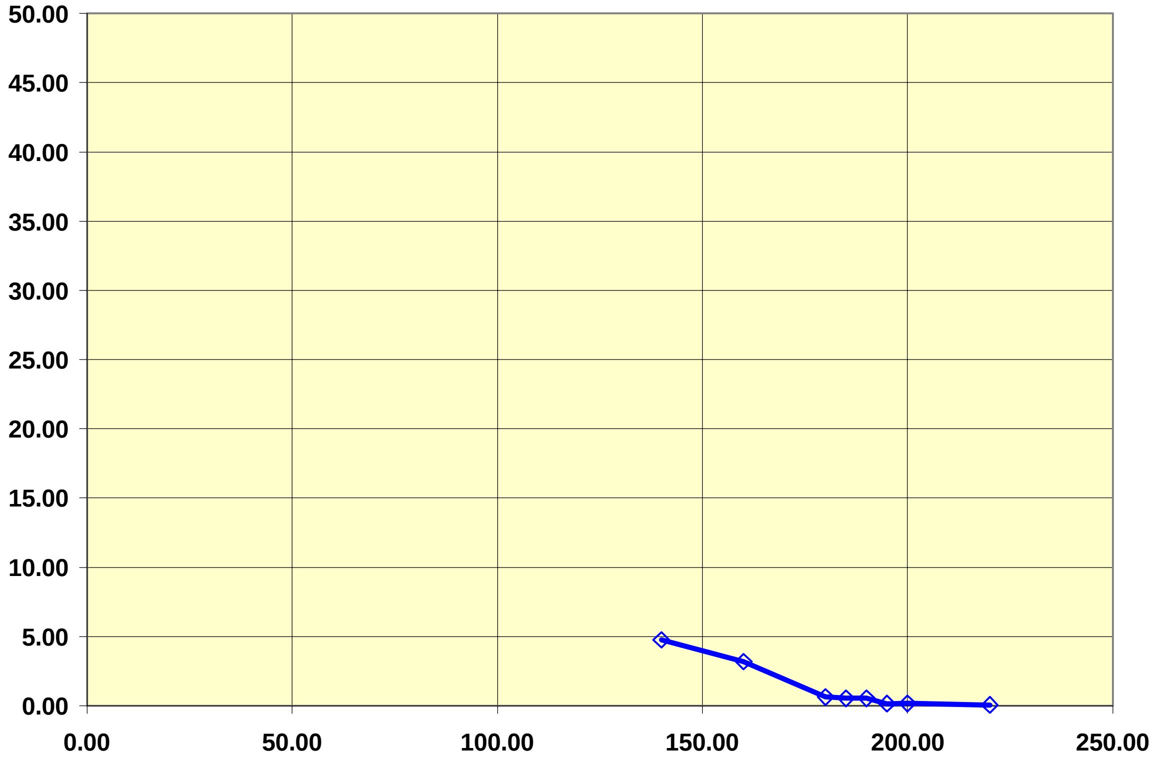
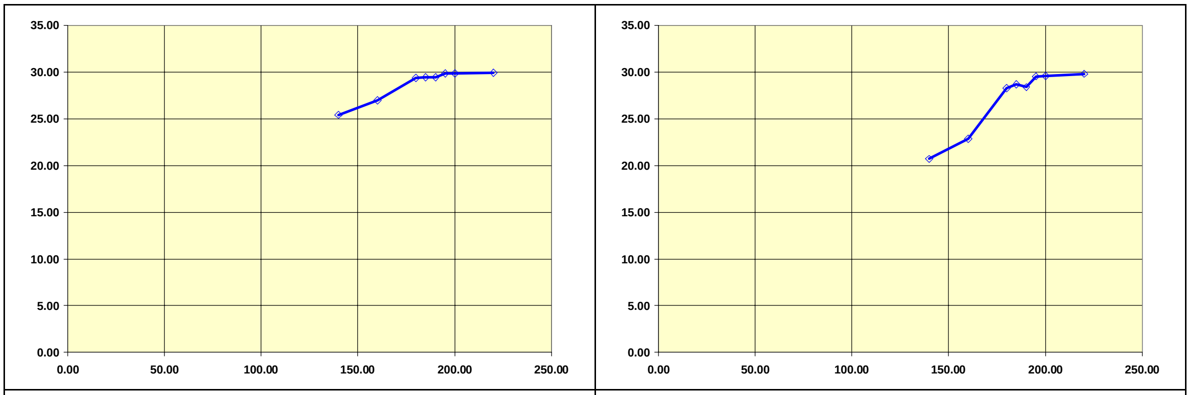


Figure B.34 Multimedia Bandwidth Allocation (KB/s) vs. Packet Loss (packets/s) during the Blast Measurement Period for HTTP-MPEG

Figure B.35 shows the true and playable frame rates for MPEG during the blast measurement period. The goal is a frame rate of 30 frames per second in both cases. Although the true-frame rate is essentially the complement of the loss-rate, the playable frame-rate shows greater variation. Because of interframe dependencies, the relatively low drop-rates do have a significant impact on the playable frame-rate as other frames cannot be decoded due to packet losses in prior or subsequent references frames. This plot limits the options for the optimal parameter settings to the B_{mm} settings of 195-220 KB/s.



a) True Frame Rate

b) Playable Frame Rate

Figure B.35 Bandwidth Allocation for Multimedia (KB/s) vs. Frame Rate (Frames/s) during the Blast Measurement Period for HTTP-MPEG

Examining multimedia performance should also confirm that CBT is managing the latency as desired. Figure B.36 shows the end-to-end latency for MPEG during the blast measurement period. Although there is a bit of variation, all of the values are between 73-85 ms, well within the 100ms limit. The latency is lower than 100 ms because the HTTP traffic is constantly oscillating to attempt to stabilize at the available capacity and rarely using all of its allocated queue capacity. Said another way, the calculated latency bound is a worst case value that should only be realized if when all classes transmit at levels approaching (or exceeding) their allocation.

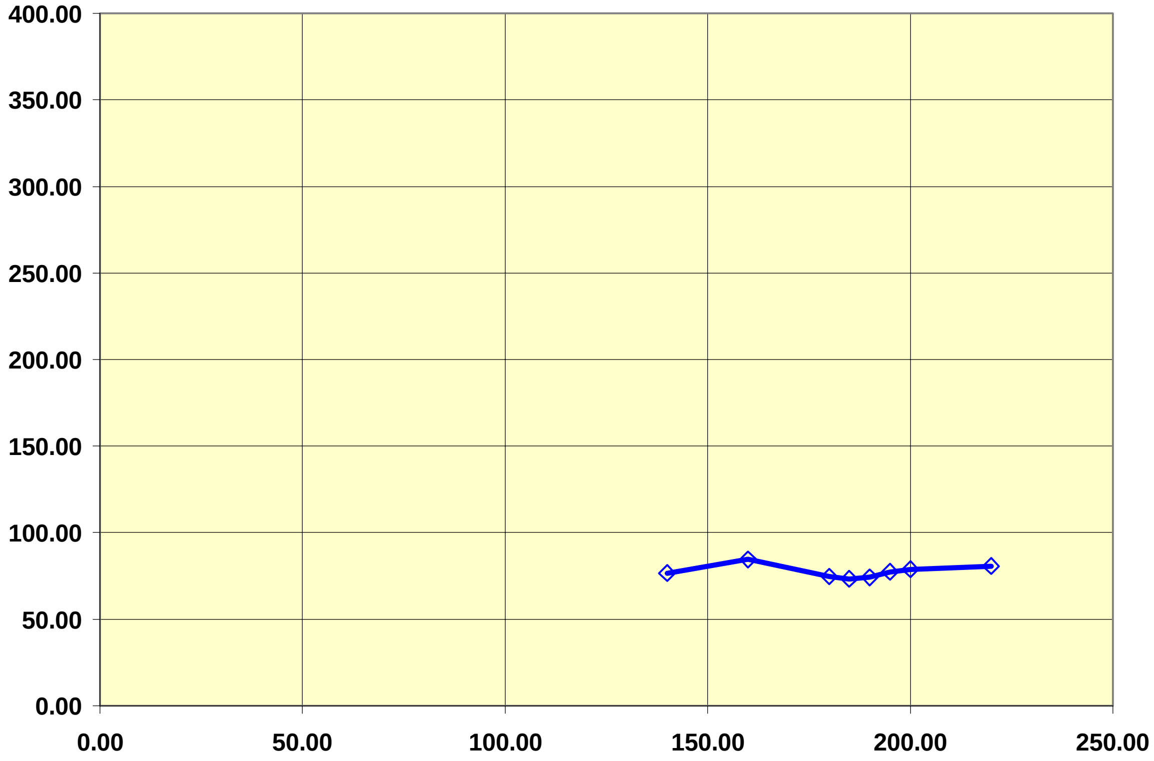


Figure B.36 Multimedia Bandwidth Allocation (KB/s) vs. Latency (ms) during the Blast Measurement Period for HTTP-MPEG

Next, consider TCP throughput to confirm that the optimal multimedia allocations do not have any unnecessarily adverse effect on TCP performance. TCP performance is expected to decrease as B_{mm} increases since TCP's allocation is the complement of the multimedia allocation. However, TCP's performance should be near its allocation. Figure B.37 shows the TCP throughput during the blast measurement period relative to the B_{mm} setting. This scale is chosen to make it easy to compare these results with trends in other plots. The throughput varies very little (753-795 KB/s).

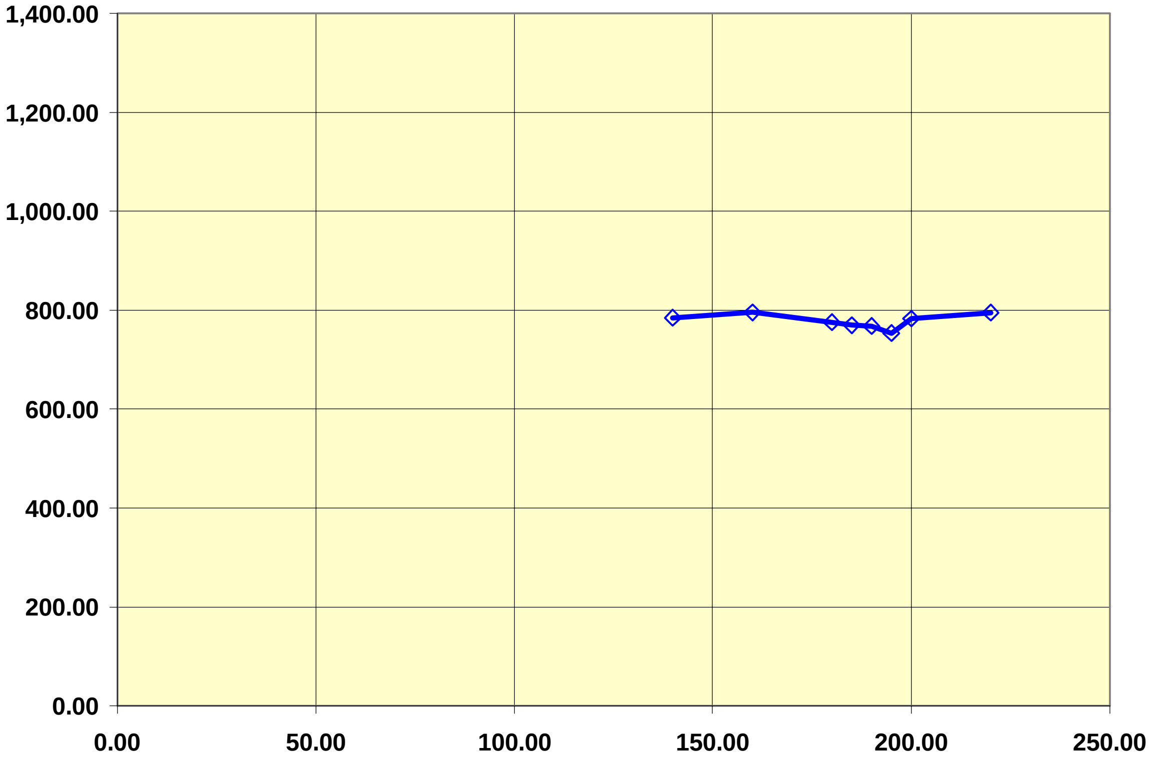


Figure B.37 Multimedia Bandwidth Allocation (KB/s) vs. TCP Throughput (KB/s) during the Blast Measurement Period for HTTP-MPEG

Figure B.38 shows the same data at a finer scale and plotted against B_{TCP}. Moreover, if the throughput matched the allocation precisely the line should have a slope of one and a y-intercept at the origin. Instead, HTTP consistently has throughput lower than its allocation. However, this is consistent with the observed performance of HTTP traffic under other queue management algorithms. The combination of the TCP congestion control mechanism's oscillating nature and the lightweight and short-lived flows in HTTP, seldom lead to full utilization of available capacity for HTTP. It is interesting to note however, that TCP's throughput seems the lowest when the multimedia seems most precisely allocated (at B_{mm} of 195 above or B_{TCP} of 875 below). This is easy to explain. When multimedia is overallocated, TCP is able to borrow the unused capacity and when multimedia is more constrained there is additional capacity allocated to TCP because of the complementary allocations. Thus TCP gets its worst throughput when it is constrained and there is no unused capacity to borrow.

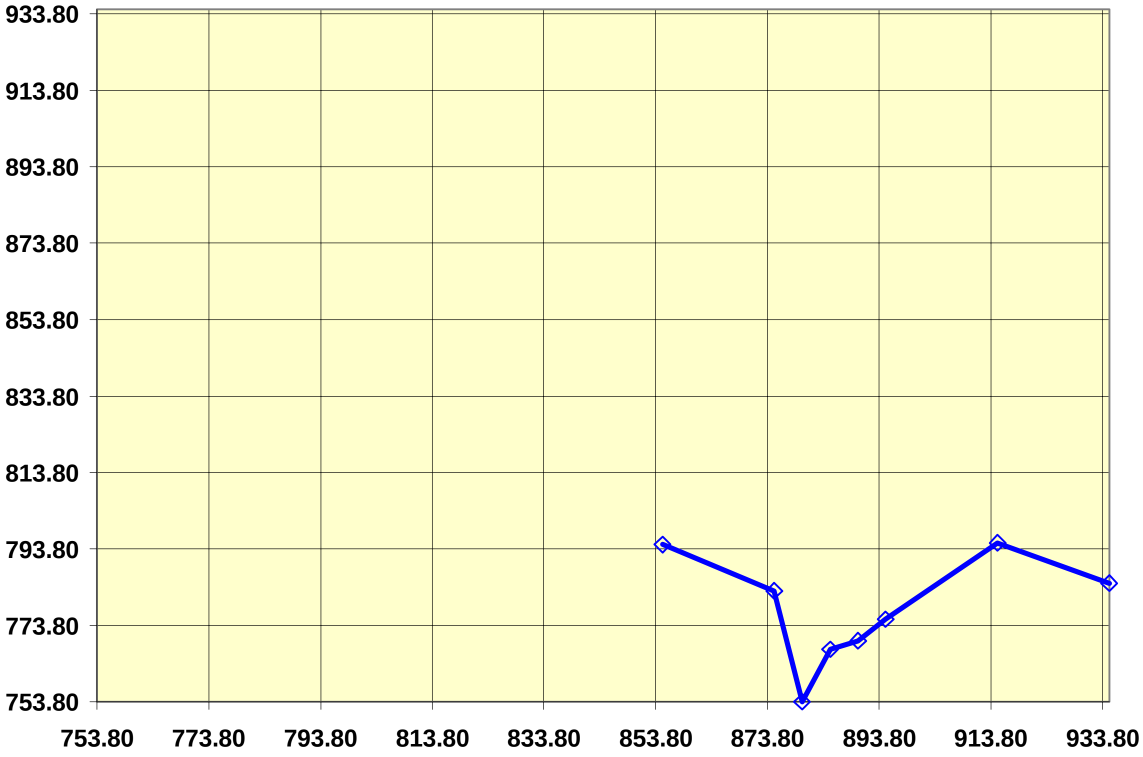


Figure B.38 TCP Bandwidth Allocation (KB/s) vs. TCP Throughput (KB/s) during the Blast Measurement Period for HTTP-MPEG

Based on the results shown here, the optimal settings are those in experiment 3 from Table B.16 (B_{mm}=195, B_{TCP}=880, and B_{other}=150 KB/s). These settings allowed multimedia to minimize loss-rate and maximize playable frame-rate. Moreover, this was the lowest multimedia allocation (and, thus the highest TCP allocation) that offered the desired multimedia performance. Note that this was not the expected optimal setting. Since MPEG's average load during the blast measurement period had been measured at 190 KB/s, 190 KB/s for B_{mm} was assumed to be the optimal setting. However, there was some medium term variation in the load that resulted in periods where the multimedia load reached 195 KB/s. To provide good multimedia performance the allocation had to be increased accordingly.

6.5. BULK-MPEG

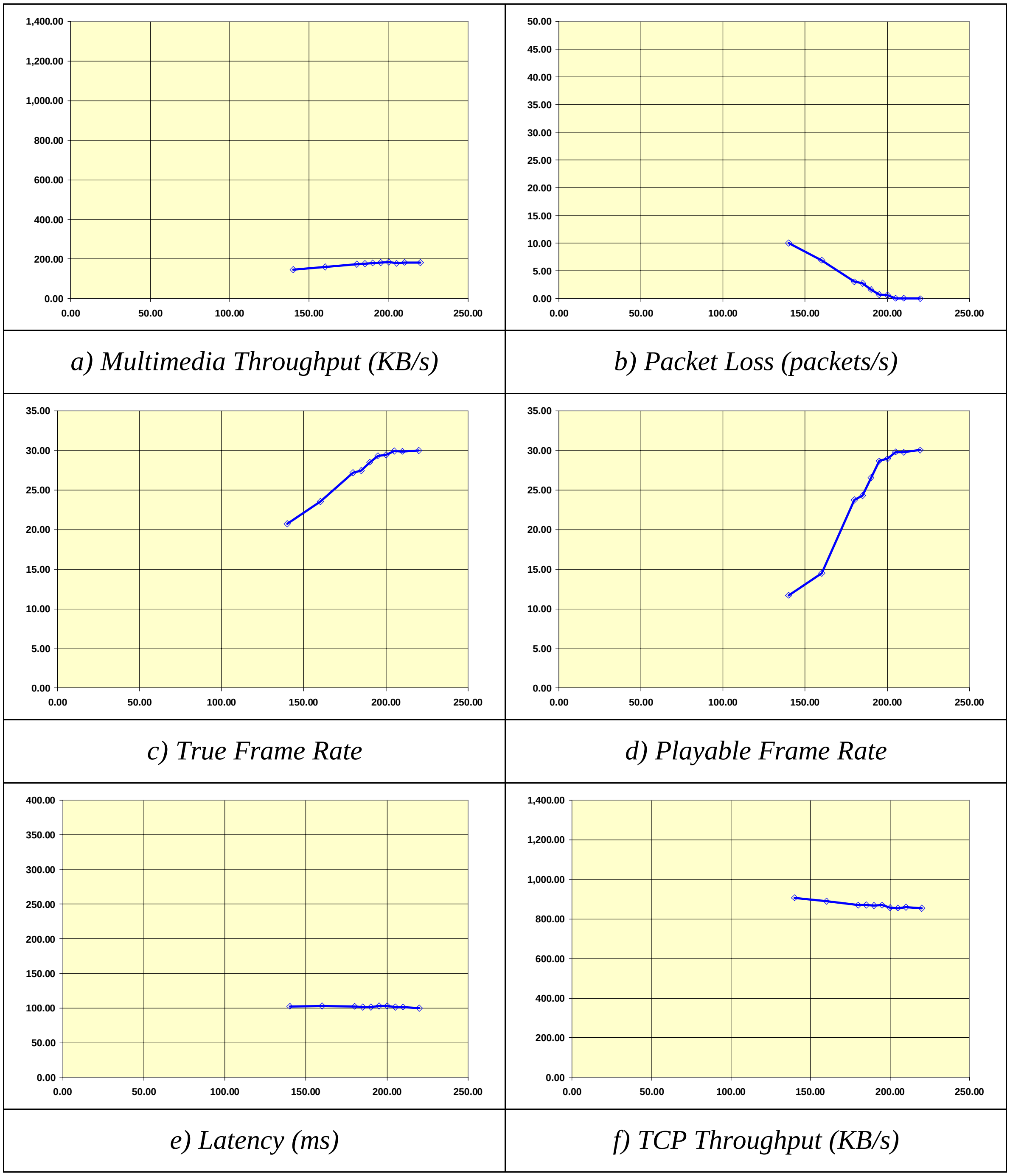
BULK	MPEG	Other
1507	807	1075

Table B.17 Average Packet Sizes (Bytes) Including Packet Headers for BULK-MPEG

The evaluation of BULK-MPEG closely parallels the evaluation of HTTP-MPEG. The measured average packet sizes shown in Table B.17 were used along with the desired bandwidth allocations to compute the threshold settings shown in Table B.18. The most likely optimal value should have a multimedia bandwidth allocation of 190 KB/s. However, the optimum allocation proved to be slightly higher, leading to the examination of the range 200-210 KB/s in smaller increments.

Exp.	B_{other}	B_{mm}	B_{TCP}	Th_{Max}		
				other	mm	TCP
1	150	220	855	14.29	27.93	58.09
2	150	210	865	14.29	26.66	58.77
3	150	205	870	14.29	26.06	59.11
4	150	200	875	14.29	25.39	59.45
5	150	195	880	14.29	24.76	59.79
6	150	190	885	14.29	24.12	60.13
7	150	185	890	14.29	23.49	60.47
8	150	180	895	14.29	22.85	60.81
9	150	160	915	14.29	20.31	62.17
10	150	140	935	14.29	17.77	63.52

Table B.18 CBT Parameter Settings for BULK-MPEG with 100 ms of Latency



a) Multimedia Throughput (KB/s)

b) Packet Loss (packets/s)

c) True Frame Rate

d) Playable Frame Rate

e) Latency (ms)

f) TCP Throughput (KB/s)

Figure B.39 Multimedia Bandwidth Allocation (KB/s) vs. Selected Metrics during the Blast Measurement Period with BULK-MPEG

Figure B.39 shows the metrics examined for BULK-MPEG plotted using B_{mm} as an index along the horizontal axis. All of these measurements are from the blast measurement period. These results are all similar to those for HTTP-MPEG. The multimedia throughput (Figure B.39a) increases with allocation up to the range of B_{mm} set to 180

KB/s and above. Examining the packet loss rate (Figure B.39b) offers a clearer indication that the allocation is insufficient below 195 KB/s. The true-frame (Figure B.39c) reflects the same finding as the packet loss while the playable frame-rate (Figure B.39d) only reaches 30 frames/second at a value of B_{mm} equal to 205 KB/s. As expected, the latency (Figure B.39e) values do not vary significantly, remaining almost entirely centered on 100 ms. Note, however, that this is an increase compared to the latency values with HTTP-MPEG. This is because the BULK traffic type does come much closer to fully utilizing its allocated capacity. If all classes are operating at capacity the latency should reach the limit of 100ms. The TCP Throughput (*f*) shows that the TCP load is higher with this traffic mix than observed with HTTP-MPEG. This data is also shown in Figure B.40. The figure shows TCP throughput during the blast measurement period versus the TCP bandwidth Allocation (B_{TCP}). Notice that the throughput comes much closer to following the desired line with slope 1.0 than with HTTP-MPEG. BULK is almost able to consume its entire allocation.

The values of experiment 3 from Table B.18, a B_{mm} value of 205 KB/s, B_{TCP} of 870, and B_{other} of 150 KB/s are chosen as the optimal parameter settings for HTTP-MPEG. This is based primarily on the playable frame-rate, and, secondarily, on the fact that none of the settings demonstrate an unreasonable impact on TCP.

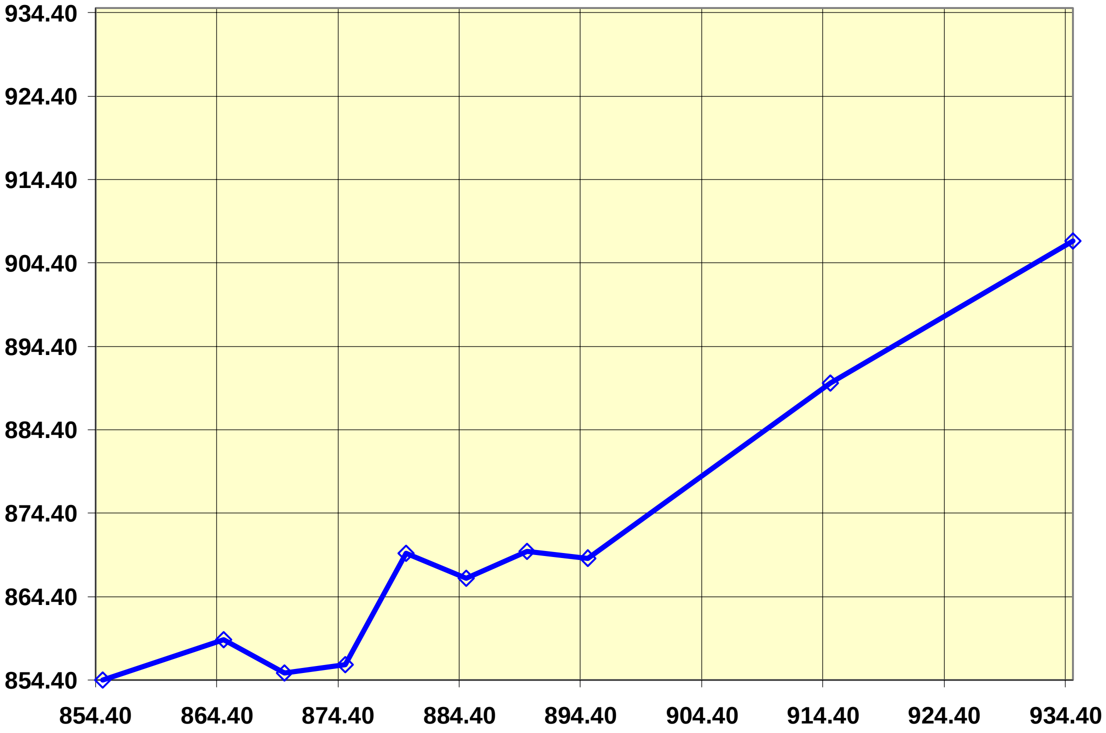


Figure B.40 TCP Bandwidth Allocation (KB/s) vs. TCP Throughput during the Blast Measurement Period with BULK-MPEG

6.6. HTTP-Proshare

The approach for selecting optimal parameters remains largely the same for HTTP-Proshare. However there are a few small changes. First, the observed average multimedia load for Proshare is 160 KB/s so some smaller bandwidth allocations for multimedia will be examined. Second, there is no frame-rate information for Proshare so the analysis will not include those two metrics. This is acceptable because Proshare does not have inter-frame dependencies to allow a single packet drop to effect multiple frames.

HTTP	Proshare	Other
1061	738	1075

Table B.19 Average Packet Sizes (Bytes) Including Packet Headers for HTTP-Proshare

As with the other traffic types, observed average packet sizes (Table B.19) were used along with the desired bandwidth allocations to calculate the threshold values (Table B.20).

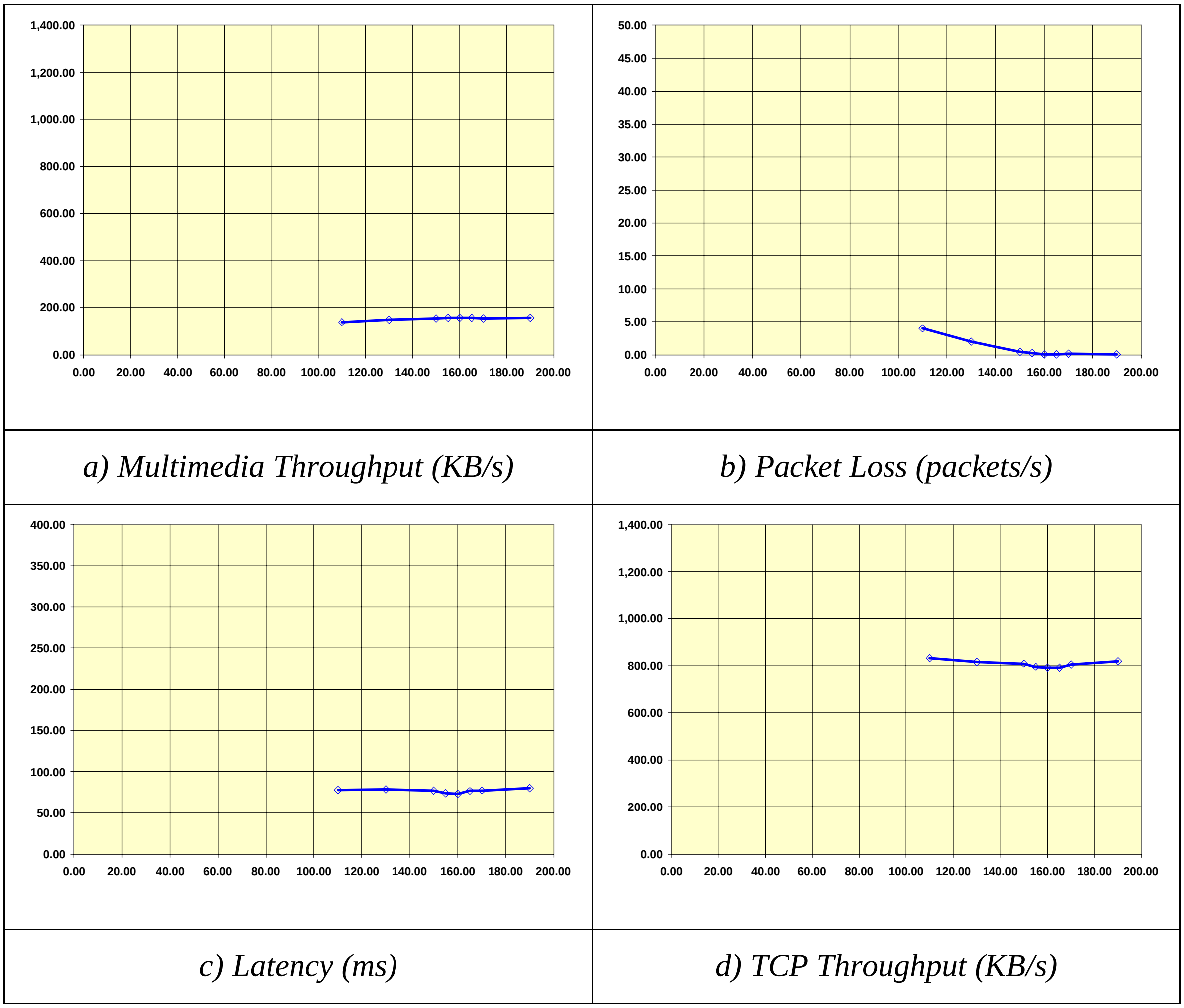
Exp.	**B_{other}**	**B_{mm}**	**B_{TCP}**	**Th_{Max}**		
				other	**mm**	**TCP**
1	150	190	885	14.28	26.35	85.45
2	150	170	905	14.28	23.58	87.38
3	150	165	910	14.28	22.88	87.87
4	150	160	915	14.28	22.19	88.35
5	150	155	920	14.28	21.50	88.83
6	150	150	925	14.28	20.80	89.32
7	150	130	945	14.28	18.03	91.25
8	150	110	965	14.28	15.26	93.18

Table B.20 CBT Parameter Settings for HTTP-Proshare with 100 ms of Latency

Figure B.41 shows the metrics examined for HTTP-Proshare plotted using B_{mm} as an index along the horizontal axis. All of these measurements are from the blast measurement period. The results are all similar to those for HTTP-MPEG. The multimedia throughput (Figure B.41a) increases slightly with allocation up to B_{mm} set to 150 KB/s and above. Examining the packet loss rate (Figure B.41b) offers a clearer indication that the allocation is insufficient below 160 KB/s. As expected, the latency (Figure B.41c) values do not vary significantly, remaining almost entirely centered on 75 ms. As noted previously with HTTP-MPEG, this is because HTTP does not consume its entire allocation. Instead the TCP throughput (Figure B.41d) is near 800 KB/s.

The values from experiment 4 from Table B.20, a B_{mm} value of 160 KB/s, B_{TCP} of 915, and B_{other} of 150 KB/s are selected as the optimal parameter settings for HTTP-Proshare.

This is based primarily on the packet loss-rate, and, secondarily, on the fact that none of the settings demonstrate unreasonable impact on TCP.



a) Multimedia Throughput (KB/s)

b) Packet Loss (packets/s)

c) Latency (ms)

d) TCP Throughput (KB/s)

Figure B.41 Multimedia Bandwidth Allocation (KB/s) vs. Selected Metrics during the Blast Measurement Period with HTTP-Proshare

6.7. BULK-Proshare

BULK	Proshare	Other
1468	757	1075

Table B.21 Average Packet Sizes (Bytes) Including Packet Headers for BULK-Proshare

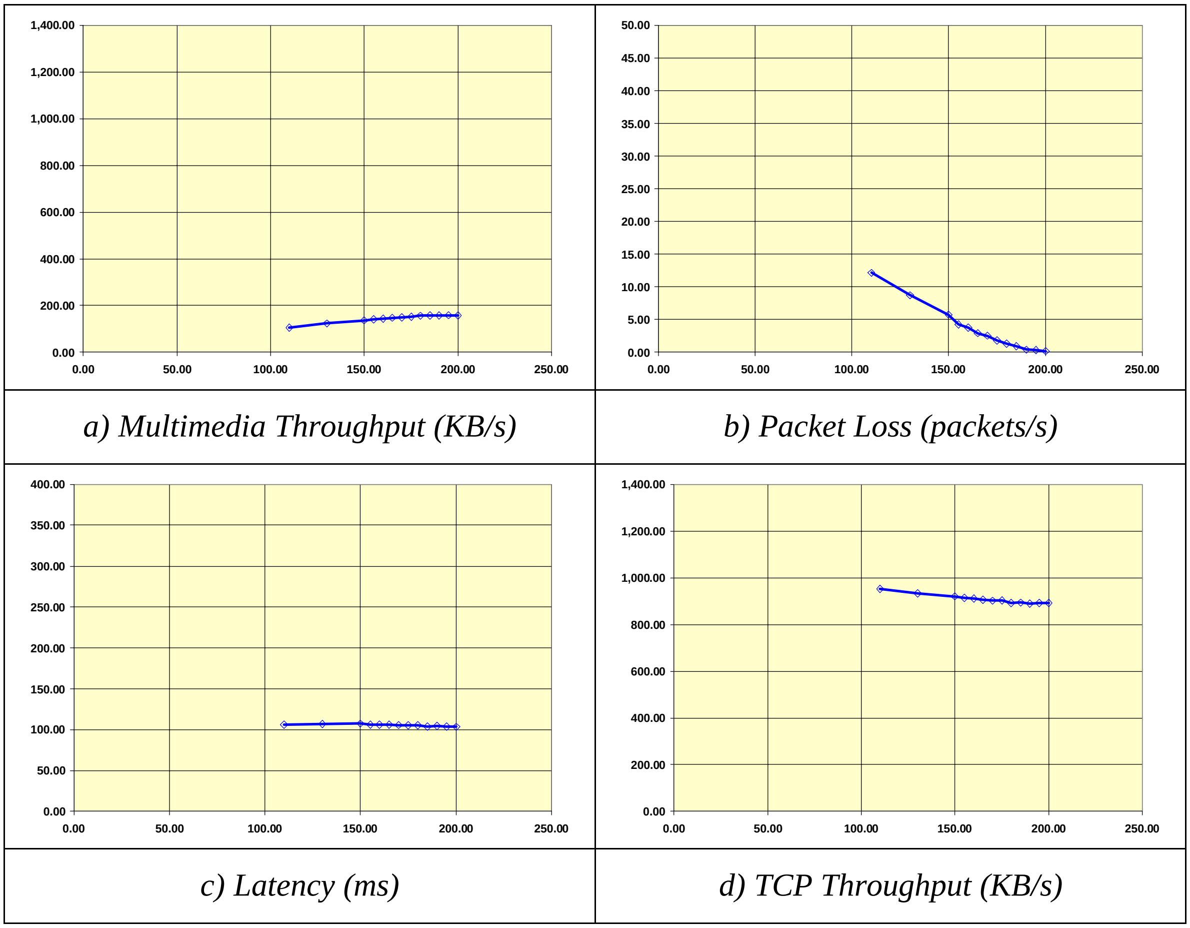
The approach for selecting optimal parameters remains largely the same for BULK-Proshare as it was for HTTP-Proshare. As with the other traffic types observed average packet sizes (Table B.21) were used along with the desired bandwidth allocations to calculate the threshold values (Table B.22).

Exp.	**B_{other}**	**B_{mm}**	**B_{TCP}**	**Th_{Max}**		
				other	**mm**	**TCP**
1	150	200	875	14.29	27.04	61.04
2	150	195	880	14.29	26.36	61.39
3	150	190	885	14.29	25.69	61.74
4	150	185	890	14.29	25.01	62.09
5	150	180	895	14.29	24.33	62.44
6	150	175	900	14.29	23.66	62.79
7	150	170	905	14.29	22.98	63.14
8	150	165	910	14.29	22.31	63.49
9	150	160	915	14.29	21.63	63.83
10	150	155	920	14.29	20.95	64.18
11	150	150	925	14.29	20.28	64.53
12	150	130	945	14.29	17.57	65.93
13	150	110	965	14.29	14.87	67.32

Table B.22 CBT Parameter Settings for BULK-Proshare with 100 ms of Latency

Figure B.42 shows the metrics examined for BULK-Proshare plotted using B_{mm} as an index along the horizontal axis. All of these measurements are from the blast measurement period. These results are all similar to those for HTTP-Proshare. The multimedia throughput (Figure B.42a) increases slightly with allocation up to the range of B_{mm} set to 185 KB/s and above. Examining the packet loss rate (Figure B.42b) offers a clearer indication that the allocation is insufficient below 190 KB/s. As expected, the latency (Figure B.42c) values do not vary significantly, remaining almost entirely centered on 100 ms. As noted previously with BULK-MPEG, this is because BULK does consume its entire allocation. This is reflected by the TCP throughput (Figure B.42d) values in the range 855-910 KB/s.

The values of experiment 3 from Table B.22, a B_{mm} value of 190 KB/s, B_{TCP} of 885, and B_{other} of 150 KB/s are selected as the optimal parameters for BULK-Proshare. This is based primarily on the packet loss-rate, and, secondarily, on the fact that none of the settings demonstrate unreasonable impact on TCP. Note that the B_{mm} setting for BULK-Proshare is much higher than that for HTTP-Proshare. This is because multimedia is able to borrow sufficient link capacity from HTTP in the HTTP-Proshare case to be successful with a lower bandwidth allocation. With BULK, which uses most of its utilization, multimedia must have a larger allocation to accommodate the medium-term variations in load.



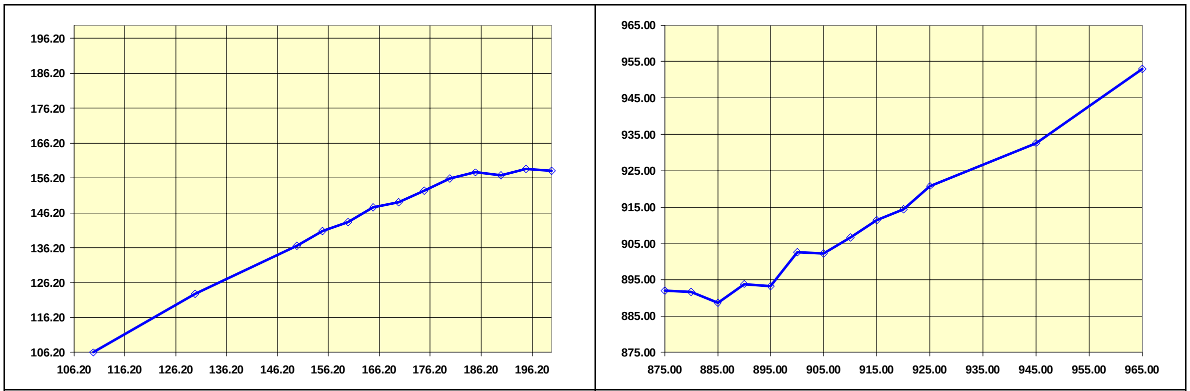
a) Multimedia Throughput (KB/s)

b) Packet Loss (packets/s)

c) Latency (ms)

d) TCP Throughput (KB/s)

Figure B.42 Multimedia Bandwidth Allocation (KB/s) vs. Selected Metrics during the Blast Measurement Period with BULK-Proshare



a) Multimedia Throughput (KB/s) vs. B_{mm} *(KB/s)*

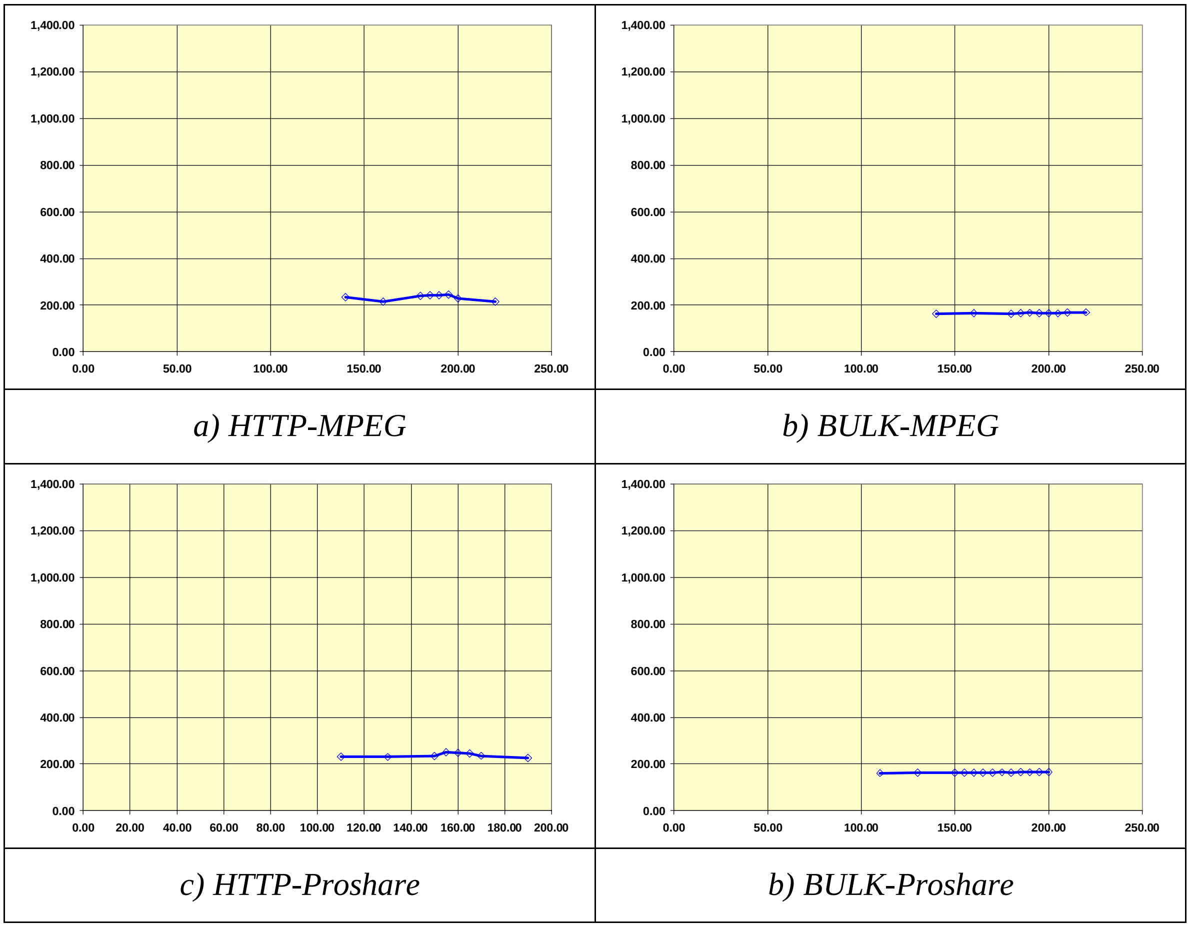
b) TCP Throughput (KB/s) vs. B_{TCP} *(KB/s)*

Figure B.43 Multimedia and TCP Throughput vs. Allocations on a 1:1 scale during the blast measurement period with BULK-Proshare

Figure B.43 presents the throughput for multimedia (*a*) and TCP (*b*) vs. their respective allocations. Perfect allocations again corresponds to a line with slope one and a y-intercept at the origin. These two traffic types come very close to following this line. Multimedia tails off when the traffic stream can no longer generate sufficient load to use the available capacity. Conversely, TCP is able to exceed its allocated capacity in the lower left because it is able to borrow that capacity unused by multimedia.

6.8. Constraining Other

In addition to selecting the optimal parameters for each traffic mix, CBT's effectiveness constraining the traffic class *other* must also be confirmed. Figure B.44 shows the throughput for the traffic of type *other* during the multimedia measurement period. Each plot shows *other* throughput for one traffic mix. The use of B_{mm} on the horizontal axis simply serves to separate different experiments. The primary point to note is that *other* does exceed its allocation (150 KB/s) most of the time. This happens for three reasons. First *other* is able to borrow unused capacity from multimedia or TCP. The most significant amount of borrowing occurs when the TCP type is HTTP. The nature of the HTTP traffic and TCP's oscillating load (due to responsiveness) prevents it from using its full allocation. However, even in the case of BULK, *other* traffic is able to exceed its allocation slightly (~170 KB/s). This is due to the fact that the UDP blast is maintaining its load so aggressively that its average occupancy is always equal to Th_{Max} (during the blast measurement period). As a result, any time that multimedia or TCP display any burstiness, reducing their queue occupancy, *other* is immediately able to borrow this unused capacity. Finally, the allocations of bandwidth leave 25 KB/s of link capacity unaccounted for. As a result, that capacity is also available for borrowing.



a) HTTP-MPEG

b) BULK-MPEG

c) HTTP-Proshare

b) BULK-Proshare

Figure B.44 B_{mm} (KB/s) vs. Other Throughput during the Blast Measurement Period Across All Traffic Mixes

6.9. Summary

To determine optimal parameter settings for CBT, these experiments focused on the parameters with the greatest effect on performance: the maximum threshold settings (Th_{Max}). To keep the number of necessary experiments reasonable the other parameters: Th_{Min}, maxp, queue size, and weights were held constant across all runs. After calculating expected optimal parameter settings for each traffic mix, experiments using those bandwidth allocations as well as allocations above and below the expected optimal settings were run. For each traffic mix the parameters were evaluated by examining the performance of multimedia, TCP, and *other*. Multimedia's throughput, loss-rate, frame-rates, and latency were also examined. Frame-rate and loss-rate were the primary criteria for choosing between settings. The throughput for TCP and *other* were also considered to be sure that the parameter settings had no unexpected extreme effects on those classes of traffic. Table B.23 shows the bandwidth allocations and parameter settings determined to be optimal.

Traffic Mix	**Weight**			**Th_{Max}**			**KB/s**		
	Mm	**other**	**TCP**	**other**	**Mm**	**TCP**	**B_{other}**	**B_{mm}**	**B_{TCP}**
HTTP-MPEG	1/16	1/4	1/256	14.28	22.45	85.58	150	195	880
BULK-MPEG	1/16	1/4	1/256	14.29	26.06	59.11	150	205	870
HTTP-Proshare	1/16	1/4	1/256	14.28	22.19	88.35	150	160	915
BULK-Proshare	1/16	1/4	1/256	14.29	25.69	61.74	150	190	885
maxp = 1/10, maxq = 240									

Table B.23 Optimal Parameter Settings for CBT

7. CBQ

Finally, optimal parameter settings must be chosen for the packet-scheduling algorithm, Class-Based Queueing (CBQ). Because CBQ parameter settings are calculated directly as a function of the desired bandwidth allocation these experiments are intended largely to simply confirm that the calculated optimal parameters do in fact offer the optimal performance. However, there are factors, such as medium-range (on the order of seconds) variability in offered multimedia load which do lead to adjustments of the allocations. In this case, increasing the multimedia allocation to match the maximum sustained load, instead of the long-term average, helps to minimize losses. Moreover, metrics such as latency are not among the inputs used to calculate the parameters for CBQ so the effect these parameters have on latency is not considered.

In most of the active queue management policies, with the exception of CBT, the parameters were simply tuned to minimize latency and fine tune how effectively each algorithm gave feedback and managed queue size. All of those factors were relatively independent of the specific traffic mix. With CBQ, this is not the case. The different loads generated in each traffic mix are important considerations in determining the optimal allocations for each traffic mix. Because CBQ allocates bandwidth instead of simply managing congestion it is important the allocations accurately reflect the expected load for each traffic mix. As a result, optimal CBQ parameters must be chosen for each of the four traffic mixes. This section focuses on the details of this process for one traffic mix and then summarizes the results for the others. A full analysis of how the optimal parameters were selected for the HTTP-MPEG traffic mix is presented, followed by the summaries for each of the other traffic mixes.

7.1. HTTP-MPEG

CBQ is parameterized by the percentage of the link's capacity allocated to each class of traffic. Recall the goal of providing good performance for multimedia while constraining *other* traffic and insuring that TCP is isolated from the behavior of other traffic types. Given this goal, the amount of bandwidth needed by multimedia must be determined, a low limit must be set on the bandwidth *other* traffic can use, and TCP should be allowed to have the remainder of the link's capacity. To make comparisons straightforward, the limit on *other* traffic is the same as in the CBT evaluation, namely150 KB/s. However, because the implementation of CBQ only supported bandwidth allocations in increments of one percentage point the allocation of 150 KB/s had to be represented as an integral percentage. As a result the bandwidth allocation for *other* traffic is 12% of the link or 147 KB/s. This allocation is held fixed while the multimedia and TCP allocations are varied. Because the load generated by the MPEG traffic generators varies between 140 KB/s and 190 KB/s, 190 KB/s was the bandwidth allocation for MPEG. Thus, the desired allocation of 190 KB/s is represented as 15% of the link capacity. Allocating 12% of the link for *other* and 15% for multimedia left 73% of the link available for TCP. This setting is shown in experiment 3 in Table B.24. Since the goal is to explore a range of parameters and confirm the optimal setting, two settings that under allocate multimedia (experiments 1 and 2) and two settings that over allocate for multimedia (experiments 4 and 5) were also considered.   The TCP allocations are adjusted correspondingly to maintain the total allocation of 100% in all cases.

Experiment	**MM %**	**Other %**	**TCP %**	**KB/s**		
				B_{mm}	**B_{other}**	**B_{TCP}**
1	13%	12%	75%	159	147	919
2	14%	12%	74%	172	147	907
3	**15%**	**12%**	**73%**	**184**	**147**	**894**
4	16%	12%	72%	196	147	882
5	17%	12%	71%	208	147	870

Table B.24 CBQ Parameter Settings for HTTP-MPEG

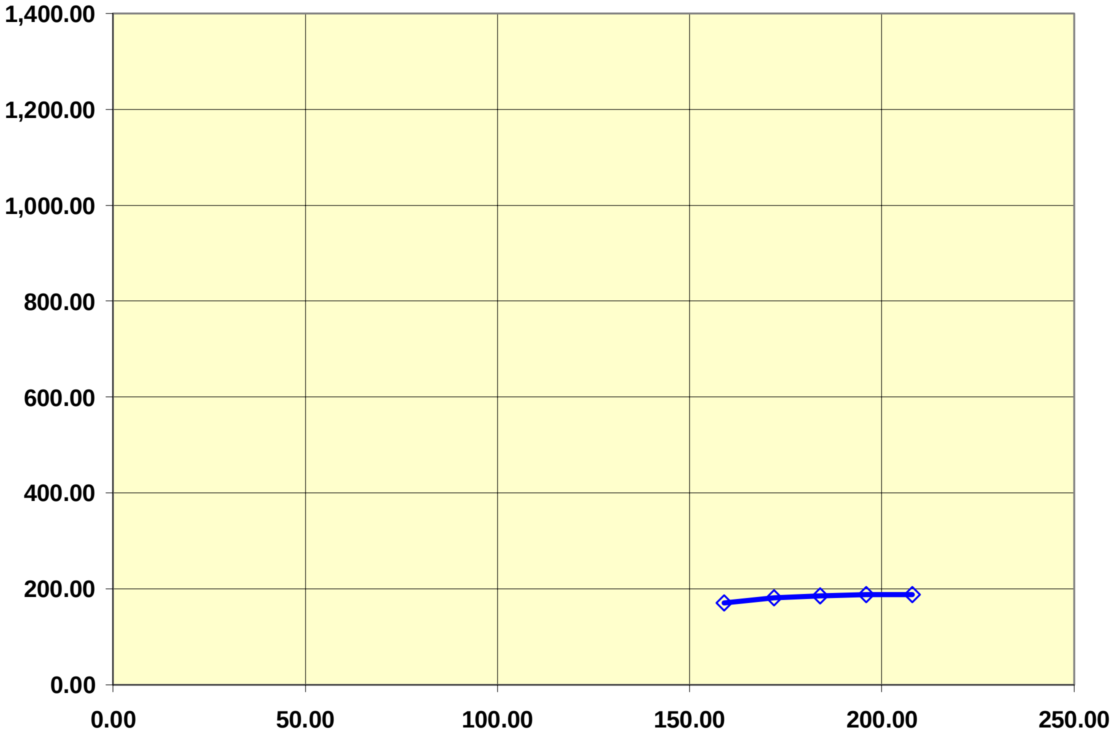


Figure B.45 Multimedia Bandwidth Allocation (KB/s) vs. Multimedia Throughput (KB/s) during the Blast Measurement Period for HTTP-MPEG

Figure B.45 shows the average multimedia throughput as the multimedia bandwidth allocation ranges. The actual changes in throughput as a fraction of the overall link capacity appear rather small. Indeed, each experiment simply increments the bandwidth allocation by one percentage point of the overall link's capacity (i.e. 12 KB/s) so small changes in the throughput are expected. It is helpful to look at this data at a different scale. Figure B.46 zooms in on the same data. If the allocations translated precisely to the throughput measurements they should be along a line from the point (159, 159) to (208, 208). In fact, multimedia has higher average throughput than its allocations of 159 KB/s and 172 KB/s should allow. The allocation is almost precise for the allocation of 184 KB/s, and multimedia is unable to use the full allocations for settings of 196 and 208 KB/s. Borrowing explains multimedia's ability to exceed its allocation in the case of the lower bandwidth allocation. Recall that TCP's responsive nature prevents it from fully utilizing its allocated capacity as the sources oscillate in response to congestion. In the CBQ algorithm when one class of traffic fails to use its allocated capacity the other classes of traffic may borrow this excess capacity in proportion to their allocation. As a result, the multimedia traffic is able to exceed its allocated capacity as long as it generates sufficient load and another class (e.g., TCP) fails to use its full allocation. In contrast to the borrowing at the lower allocations, multimedia fails to match its allocated capacity for the higher allocations because it simply does not maintain sufficient load to use that capacity. From this data is appears that a bandwidth allocation of 196 KB/s for multimedia may be the best. This is based on the fact that throughput does not improve when the allocation is set to 208 KB/s and overallocating is wasteful and imprecise. Moreover, the unused part of an allocation is shared between other classes. If bandwidth is allocated accurately, the reallocation of this "excess" can be controlled.

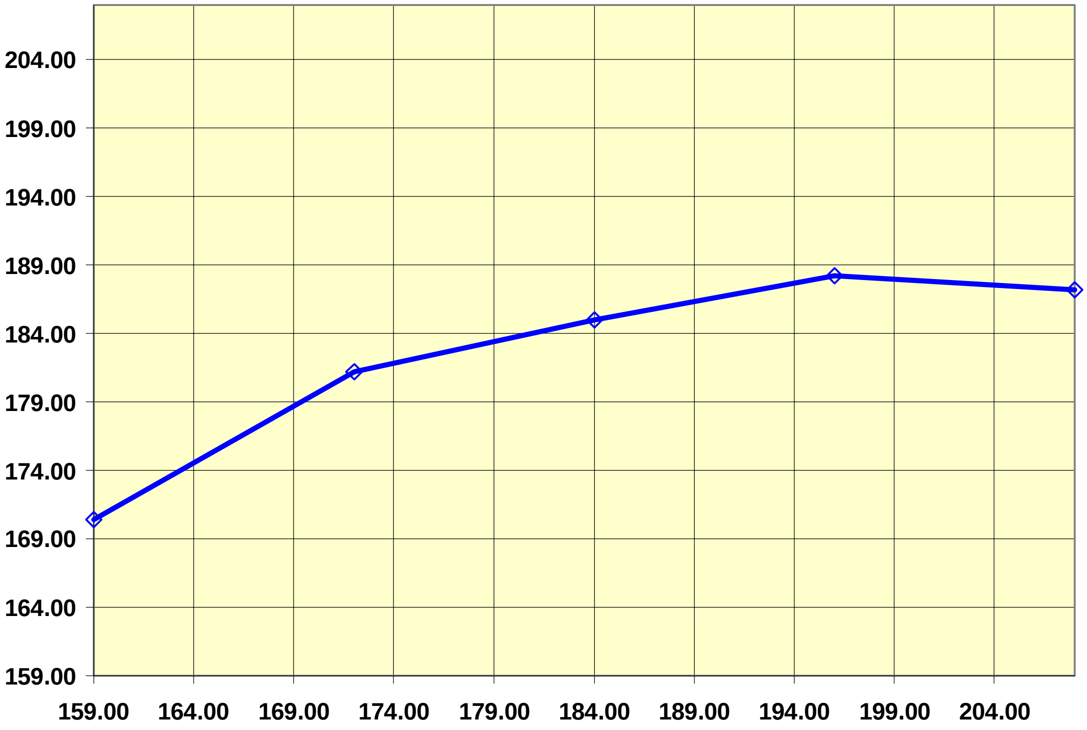


Figure B.46 Multimedia Bandwidth Allocation (KB/s) vs. Multimedia Throughput (KB/s) during the Blast Period with HTTP-MPEG on a One-to-One Scale

To fully evaluate the effectiveness of these settings one could compare the multimedia throughput to the multimedia load for each set of parameter settings. However, other metrics, packet loss rate and frame-rate measurements, address the same performance concerns for the multimedia stream. If the throughput fails to match the load, the frame-rate and loss-rate will degrade. Moreover, these metrics offer an expression of the quality of the multimedia stream that may include the effects of other factors such as a bias against bursty traffic. For example, if an algorithm were most likely to drop single fragments of large packets, this may show up as a relatively small decrease in the overall throughput for multimedia. However, since the largest packets in MPEG are I-frames and those are the reference frames for other packets, such a drop distribution could result in extremely poor playable frame-rates. Given this consideration, packet loss and frame-rates are the next metrics considered.

Figure B.47 shows the packet loss rate for these experiments. The multimedia bandwidth allocations are used as indices for each experiment. Clearly, the packet loss rate decreases as the multimedia bandwidth allocations increase and the two lowest settings do not give good performance, as previously observed.

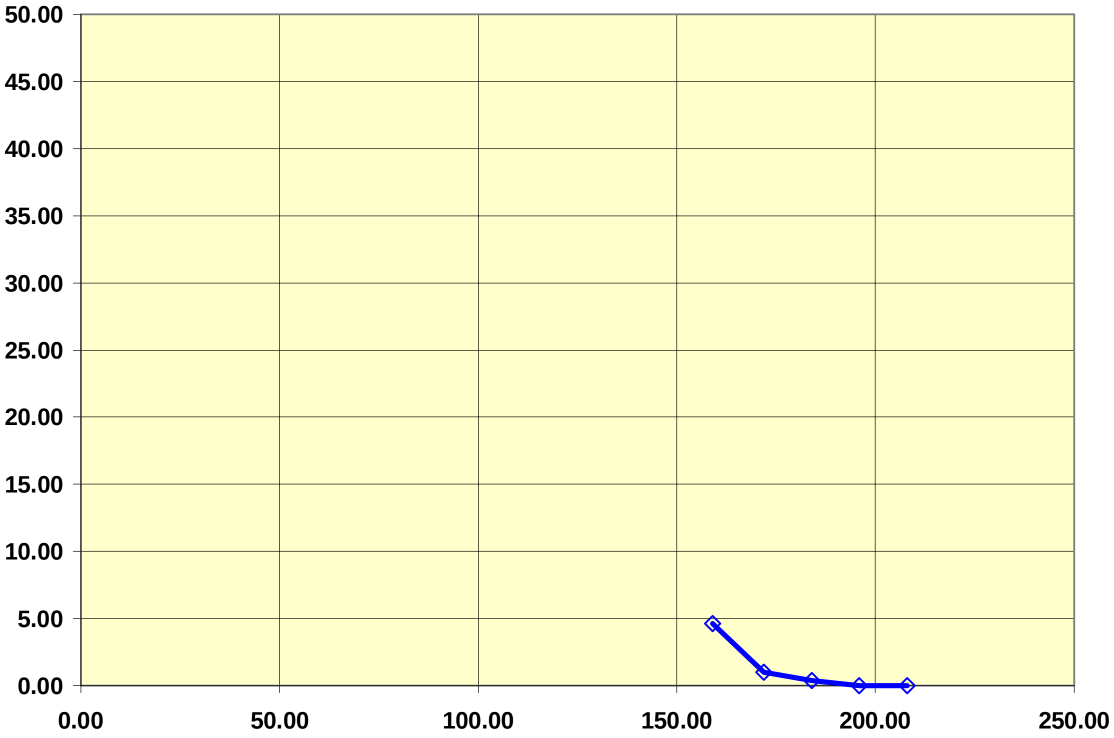
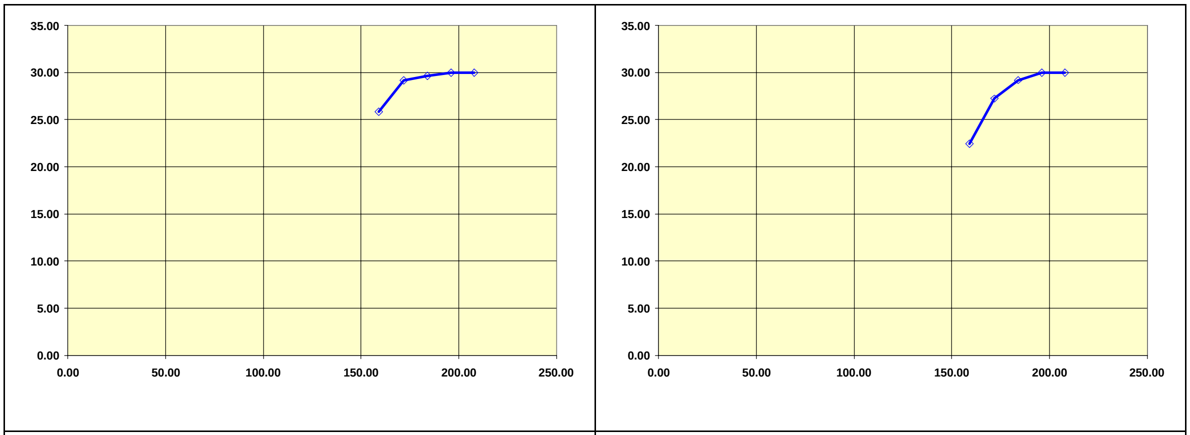


Figure B.47 Multimedia Bandwidth Allocation (KB/s) vs. Packet Loss (packets/s) during the Blast Measurement Period for HTTP-MPEG

However, to discern the difference in quality resulting from the higher bandwidth allocations, consider the frame rate measurements. Figure B.48 shows the multimedia frame rates at the receiver during the blast measurement period. Recall that actual frame rate (Figure B.48a) is based on the number of frames that arrive intact at the receiver, regardless of dependencies between frames, while playable frame rate (Figure B.48b) is the number of frames that arrive intact and are decodable (i.e., their reference frames also arrived intact.) Focusing on the higher allocations which seemed promising based on packet loss, it is apparent that bandwidth allocations of 196 and 208 KB/s offer frame rates of 30 frames/second.



a) Actual Frame Rate

b) Playable Frame Rate

Figure B.48 Bandwidth Allocation for Multimedia (KB/s) vs. Frame Rate (Frames/sec) during the Blast Measurement Period for HTTP-MPEG

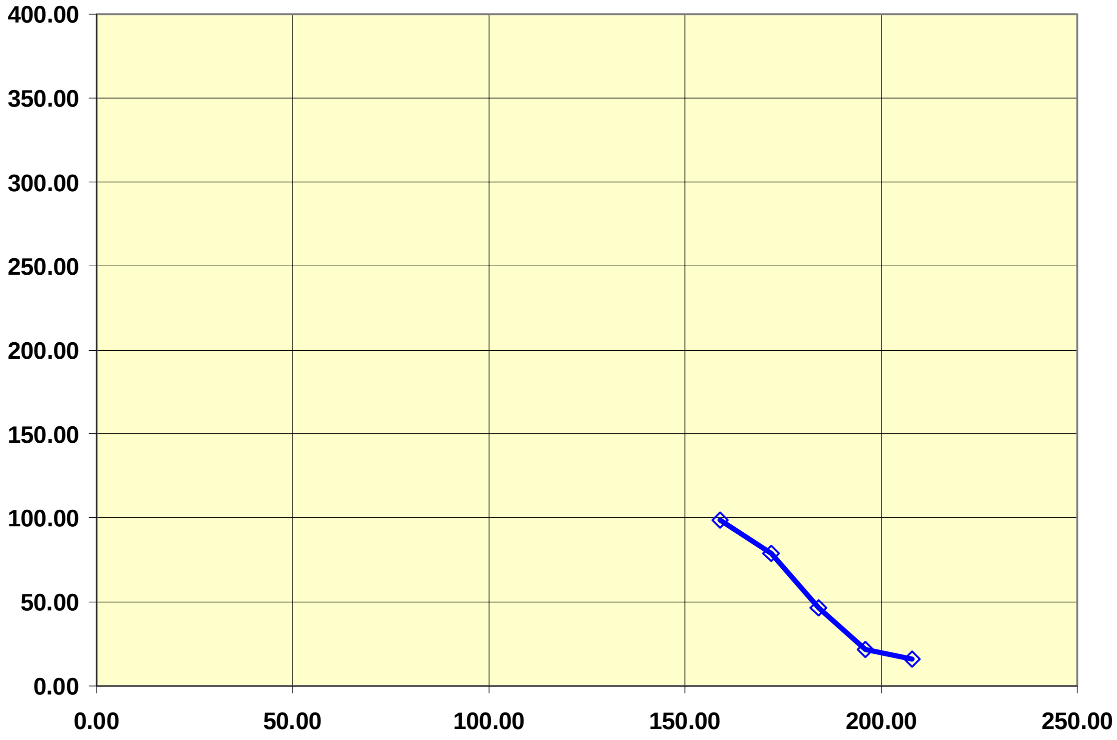


Figure B.50 Multimedia Bandwidth Allocation (KB/s) vs. Latency (ms) during the Blast Measurement Period with HTTP-MPEG

Finally, consider the effect the bandwidth allocations have on latency. Figure B.50 shows the end-to-end latency for MPEG during the blast measurement period. Unlike the active queue management algorithms, latency is not uniform across all classes or bandwidth allocations. Because in CBQ classes have their own queues that are serviced at different rates, one class of traffic may experience significant queue-induced latency while another class experiences little or no latency. However, this is not a significant concern as TCP is tolerant of some latency and there is not attempt to provide any type of performance guarantees for the class *other*. Thus, the focus is on multimedia latency. Any class that is generating load less than its allocated capacity should experience minimal queue induced delay as queues only build up when load exceeds allocated capacity. This is apparent in Figure B.50. Bandwidth allocations of 196 and 208 KB/s have low latency (less than 30 ms). However, when a class exceeds its allocation latency increases relative to the intensity of the overload. Thus, lower bandwidth allocations have latency on the order of 50 ms or more. Llatency concerns also demand that multimedia be allocated sufficient bandwidth to have minimal drops.

The final concern is TCP throughput. Recall that there is an inverse relationship between multimedia bandwidth allocation and TCP bandwidth allocation. Increasing the allocation for multimedia requires a decrease in the allocation for some class. In these experiments the allocation for the *other* traffic was held constant. Hence, the allocation for TCP was decreased. This decision was arbitrary. The primarily goal of this experiment is simply to confirm that TCP performance is equally predictable for all allocations. Figure B.51 shows the TCP throughput during the blast measurement period. The TCP throughput does decline as the multimedia bandwidth allocation increases, as expected, but the degradation is small. Note, however, that the TCP throughput is lower for a multimedia allocation of 208 KB/s than for a multimedia allocation of 196 KB/s, despite the fact that Figure B.45 showed that media throughput did not increase for an allocation of 208 KB/s. This behavior is explained by borrowing. Since multimedia doesn't use its full allocation other classes are able to borrow from multimedia's unused allocation. In this case *other* and TCP share the excess capacity. However, when the multimedia allocation is more precise at 196 KB/s, TCP's allocation is also larger than it is when B_{mm} is 208 KB/s. Consequently, instead of sharing multimedia's excess bandwidth with *other*, TCP is able to use that bandwidth as it is allocated to TCP initially. This illustrates that point that it is important to allocate multimedia all of the bandwidth it needs, but no more.

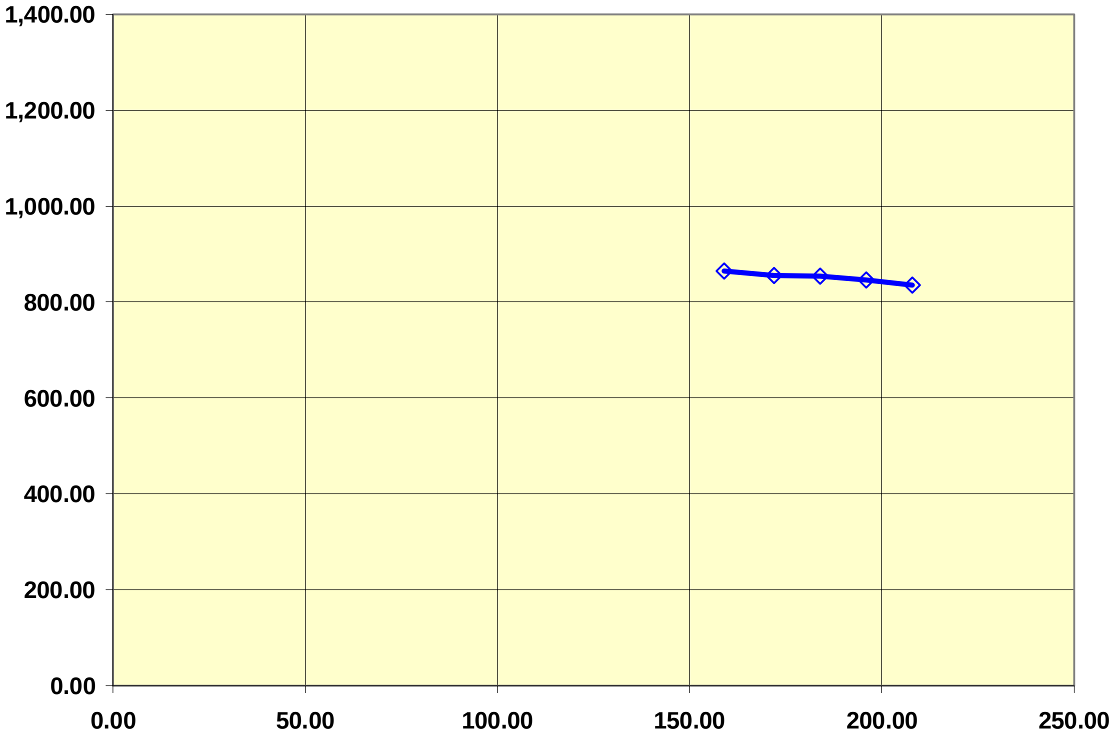


Figure B.51 Multimedia Bandwidth Allocation (KB/s) vs. TCP Throughput (KB/s) during the Blast Measurement Period with HTTP-MPEG

For the HTTP-MPEG traffic mix, multimedia throughput, packet loss, frame-rates, and latency were considered in an effort to determine which parameter settings offer optimal performance. In all of the multimedia metrics, multimedia allocations of 196 KB/s and 208 KB/s offered superior performance. Moreover, the 208 KB/s allocation offered little improvement over the 196 Kb/s allocation. Since multimedia did not benefit from the larger allocation and TCP's throughput did degrade when multimedia was over allocated, the allocations used were 196 KB/s (16%) for multimedia, 147 KB/s (12%) for *other*, and 882 KB/s (72%) for TCP. Note that this is not the allocation originally calculated as the optimal setting. Because the load resulting from the MPEG media stream was variable over medium time-scales (seconds), a larger multimedia allocation had to be used to accommodate brief periods when multimedia's load was greater than 184 KB/s. A larger queue length for CBQ could have avoided some drops in this scenario, but at the cost of increased latency. Increasing the allocation minimizes loss and latency.

The optimal parameters for each of the other three traffic types were determined using the same methodology as before. The findings for those cases are summarized below.

7.2. BULK-MPEG

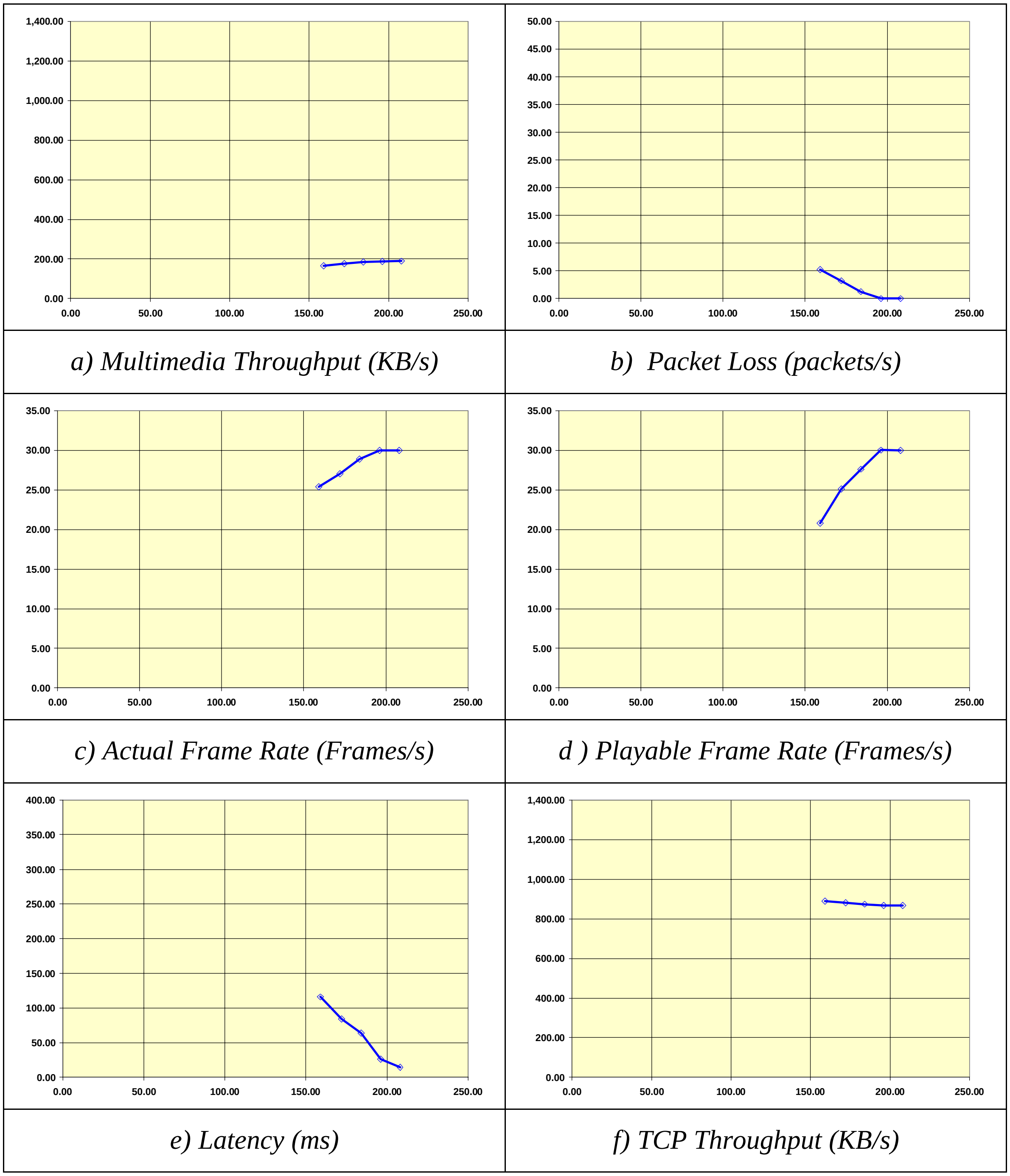
The analysis for BULK-MPEG very closely parallels that of HTTP-MPEG, including the parameter space explored. Since the allocation for *other* traffic is fixed, and the TCP allocation is simply the leftover allocation, the expected multimedia load and the resulting allocation determines all of the parameter settings for these experiments. As a result, the same parameter settings are considered for BULK-MPEG as with HTTP-MPEG (shown in Table B.25). However, because the behavior of BULK traffic is different from HTTP traffic, results with those traffic types must be examined to determine the optimal parameter settings.

Experiment	**MM %**	**Other %**	**TCP %**	**KB/s**		
				B_{mm}	**B_{other}**	**B_{TCP}**
1	13%	12%	75%	159	147	919
2	14%	12%	74%	172	147	907
3	**15%**	**12%**	**73%**	**184**	**147**	**894**
4	16%	12%	72%	196	147	882
5	17%	12%	71%	208	147	870

Table B.25 CBQ Parameter Settings for BULK-MPEG

Figure B.52 shows the plots of (Figure B.52a) multimedia throughput, (Figure B.52b) packet loss, (Figure B.52c,d) frame rates, (Figure B.52e) latency, and (Figure B.52f) TCP throughput. As with HTTP-MPEG, the telling metrics are the frame rates (Figure B.52c,d). The media stream sustains both an actual and playable frame-rate of 30 frames per second for both of the higher multimedia bandwidth allocations (196 KB/s and 208 KB/s). The latency (Figure B.52e) for those two settings is less than 30 ms in both cases (26 ms and 14 ms respectively). The goal is to use the minimum multimedia allocation that offers acceptable performance in order to allocate as much bandwidth as possible to

TCP. As a result, setting 4 in Table B.25, 16% for Multimedia, 12% for *other*, and 72% for TCP, is chosen



a) Multimedia Throughput (KB/s)

b) Packet Loss (packets/s)

c) Actual Frame Rate (Frames/s)

d) Playable Frame Rate (Frames/s)

e) Latency (ms)

f) TCP Throughput (KB/s)

Figure B.52 Multimedia Bandwidth Allocation (KB/s) vs. Selected Metrics during the Blast Measurement Period with BULK-MPEG

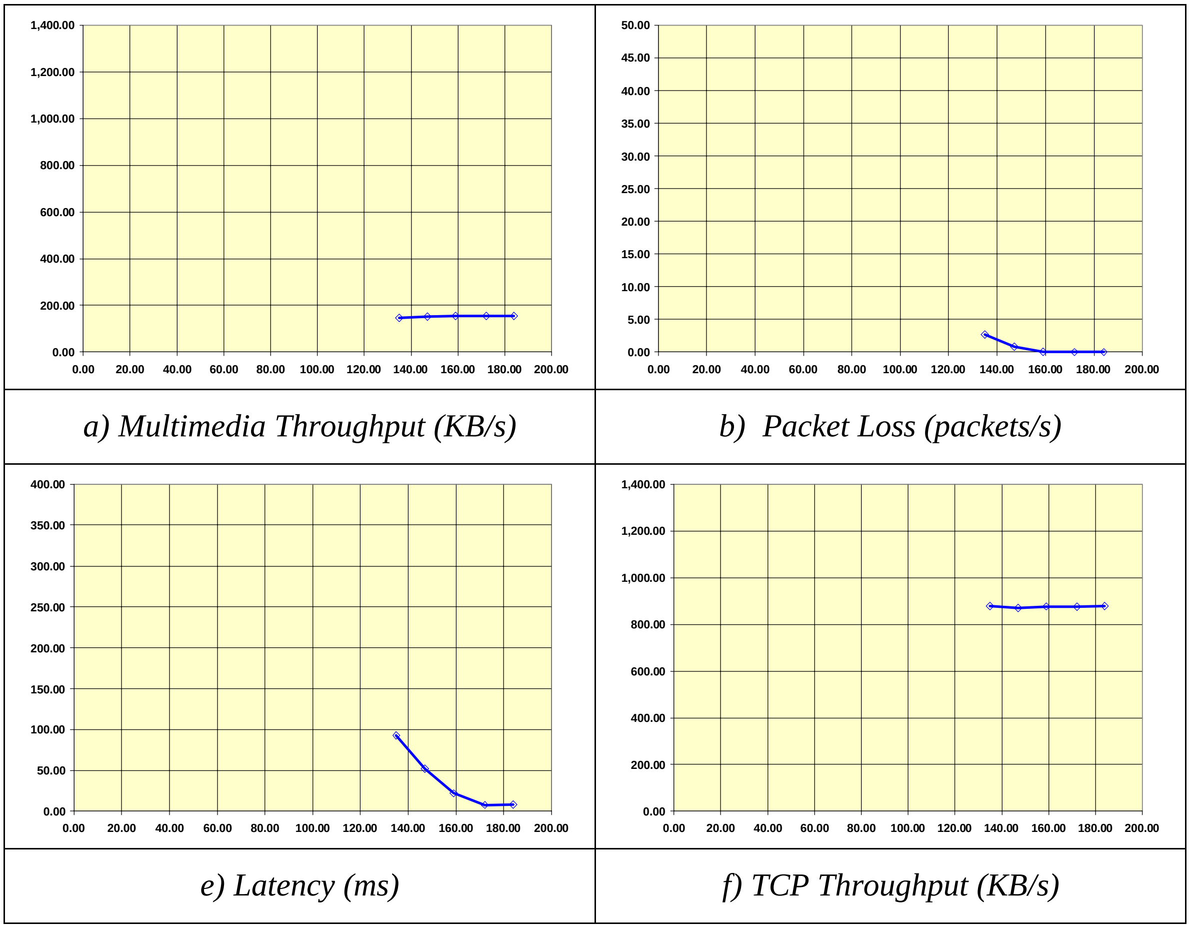
7.3. HTTP-Proshare

Next, consider the HTTP-Proshare traffic mix. The Proshare media stream is both less variable in the medium term (seconds) and has a lower average bandwidth than the MPEG media streams. As a result, the optimal parameter setting should allocate ~160 KB/s for multimedia. This allocation is experiment 3 in Table B.26. As with the other experiments multimedia parameter settings are considered in increments of one and two percentage points above and below the expected optimal settings.

Experiment	MM %	Other %	TCP %	KB/s		
				B_{mm}	B_{other}	B_{TCP}
1	11%	12%	77%	135	147	943
2	12%	12%	76%	147	147	931
3	**13%**	**12%**	**75%**	**159**	**147**	**919**
4	14%	12%	74%	172	147	907
5	15%	12%	73%	184	147	894

Table B.26 CBQ Parameter Settings for HTTP-Proshare

The evaluation of HTTP-Proshare is limited to only four metrics because frame-rate information for Proshare is not available (see Appendix A). However, unlike MPEG, Proshare's encoding mechanism does not use interframe encoding so packet loss is a reasonable indicator of media play-out quality. These metrics are shown in Figure B.53. The multimedia throughput (Figure B.53a) is unremarkable, simply increasing slightly as a result of the increased allocation. However, the packet loss (Figure B.53b) is clearly better for the higher multimedia allocations (159-184 KB/s) as they lead to loss of less than one percent of the packets. However, the allocation of 159 KB/s does lead to slightly more latency (21 ms) (Figure B.53c) than with the two higher allocations (7ms). Note that the combination of latency and very low loss indicates that the Proshare traffic is bursty enough to exceed the 159 KB/s allocation for very brief intervals and build a queue but that the allocation is rarely exceeded long enough to force any overflow of that class's queue. Since the goal is to choose the minimum acceptable multimedia bandwidth allocation, the allocation from experiment 3 in Table B.26, 13% for multimedia, 12% for *other*, and 75% for TCP, are chosen as optimal.



a) Multimedia Throughput (KB/s)

b) Packet Loss (packets/s)

e) Latency (ms)

f) TCP Throughput (KB/s)

Figure B.53 Multimedia Bandwidth Allocations (KB/s) vs. Selected Metrics during the Blast Measurement Period forHTTP-Proshare

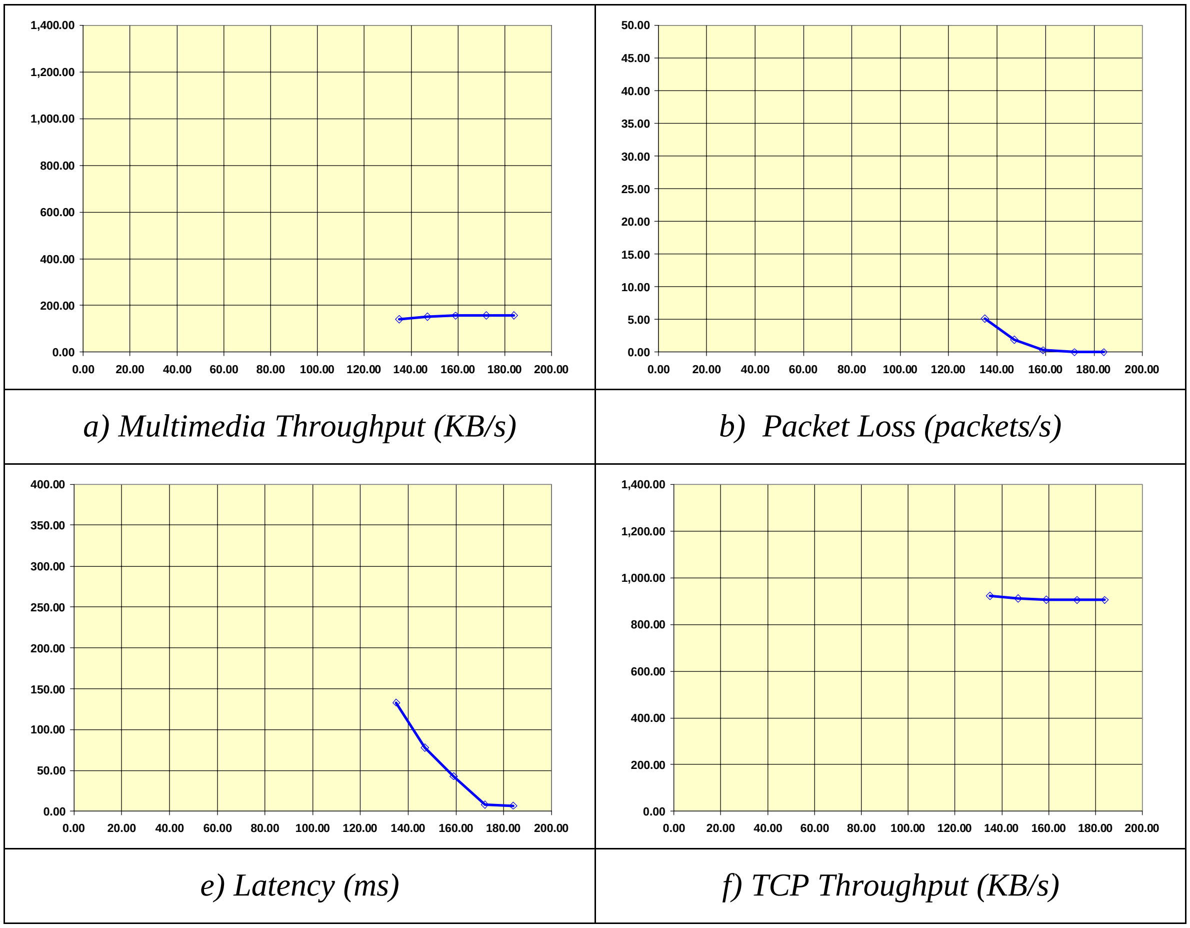
7.4. BULK-Proshare

The evaluation of BULK-Proshare is almost identical to that of HTTP-Proshare. Since the traffic mixes have the same multimedia traffic type, the initial allocations examined are identical. The allocations for BULK-Proshare are shown in Table B.27.

Experiment	**MM %**	**Other %**	**TCP %**	**KB/s**		
				B_{mm}	**B_{other}**	**B_{TCP}**
1	11%	12%	77%	135	147	943
2	12%	12%	76%	147	147	931
3	**13%**	**12%**	**75%**	**159**	**147**	**919**
4	14%	12%	74%	172	147	907
5	15%	12%	73%	184	147	894

Table B.27 CBQ Parameter Settings for BULK-Proshare

Figure B.54 shows the four metrics examined for BULK-Proshare with CBQ. The packet loss (Figure B.54b) metric indicates multimedia allocations greater than or equal to 159 KB/s give minimal packet loss. However, the latency (Figure B.54c) observed indicates the setting of 159 KB/s is unacceptable as the average latency is 42 ms. Both of the higher allocations (172 and 184 KB/s) are acceptable with latency of 7ms. It interesting to note that the latency behavior here is slightly different than that for HTTP-Proshare even though both have the same multimedia traffic type. The difference is that the BULK TCP traffic is more aggressive and more fully uses its allocated bandwidth, leaving no opportunity for multimedia to borrow unused capacity. As a result, the brief periods when Proshare exceeds its average load of 160 KB/s show up more dramatically when combined with BULK than with HTTP. As a result, for BULK-Proshare a multimedia bandwidth allocation of 172 KB/s, 14%, *other* of 12%, and TCP of 74% is selected as optimal.



a) Multimedia Throughput (KB/s)

b) Packet Loss (packets/s)

e) Latency (ms)

f) TCP Throughput (KB/s)

Figure B.54 Multimedia Bandwidth Allocations vs. Selected Metrics during the Blast Measurement Period with BULK-Proshare

7.5. Summary

Traffic Mix	MM %	Other %	TCP %	KB/s		
				B_{mm}	B_{other}	B_{TCP}
HTTP-MPEG	16%	12%	72%	196	147	882
BULK-MPEG	16%	12%	72%	196	147	882
HTTP-Proshare	13%	12%	75%	159	147	919
BULK-Proshare	14%	12%	74%	172	147	907

Table B.28 Optimal Parameter Settings for CBQ

Table B.28 shows the optimal CBQ parameter settings for each traffic mix.

8. Summary

All of the optimal parameter settings are presented in Table B.29, Table B.30, and Table B.31, below, for ease of reference.

Algorithm	maxq	w	maxp	Th_{Min}	Th_{Max}	minq
FIFO	60	n/a	n/a	n/a	n/a	n/a
RED	240	1/256	1/10	5	40	n/a
FRED	240	1/256	1/10	5	60	2

Table B.29 Optimal Parameter Settings for FIFO, RED, and FRED Across all Traffic Mixes

Traffic Mix	Weight			Th_{Max}			KB/s		
	Mm	other	TCP	other	Mm	TCP	B_{other}	B_{mm}	B_{TCP}
HTTP-MPEG	1/16	1/4	1/256	14.28	22.45	85.58	150	195	880
BULK-MPEG	1/16	1/4	1/256	14.29	26.06	59.11	150	205	870
HTTP-Proshare	1/16	1/4	1/256	14.28	22.19	88.35	150	160	915
BULK-Proshare	1/16	1/4	1/256	14.29	25.69	61.74	150	190	885
maxp = 1/10, maxq = 240									

Table B.30 Optimal Parameter Settings for CBT

Traffic Mix	MM %	Other %	TCP %	KB/s		
				B_{mm}	B_{other}	B_{TCP}
HTTP-MPEG	16%	12%	72%	196	147	882
BULK-MPEG	16%	12%	72%	196	147	882
HTTP-Proshare	13%	12%	75%	159	147	919
BULK-Proshare	14%	12%	74%	172	147	907

Table B.31 Optimal Parameter Settings for CBQ

REFERENCES

[Allman99] M. Allman and A. Falk, *On the Effective Evaluation of TCP*, ACM Computer Communications Review, October 1999

[Balakrishnan99] Hari Balakrishnan, Hariharan Rahul, and Srinivasan Seshan, *An Integrated Congestion Management Architecture for Internet Hosts,* Proceeding ACM SIGCOMM, Cambridge, MA, September 1999

[Baker95] F. Baker, Editor, *Requirements for IP Version 4 Routers*, RFC 1812, June 1995

[Berners97] T. Berners-Lee, R. Fielding, and H. Nielsen, "Hypertext transfer protocol – HTTP/1.0", Internet RFC 1945, http://ds.internic.net/ds/rfc-index.html, May 1996, [September 1997]

[Bhola98] Bhola, Sumeer and Banavar, Guruduth and Ahamad, Mustaque, *Responsiveness and Consistency Tradeoffs in Interactive Groupware*, Proceedings of ACM Conference on Computer-Supported Cooperative Work, 1998.

[Bhushan72] A. K. Bhushan, *File Transfer Protocol (FTP) Status and Further Comments*, RFC 414, December, 1972

[Blake97] S. Blake, *Some Issues and Applications of Packet Marking for Differentiated Services.* Internet Draft, Internet Engineering Task Force, Dec. 1997, Work in progress

[Braden89] R. Braden, Ed., Requirements for Internet Hosts-Communication Layers, RFC-1122, October 1989.

[Braden94] R. Braden, D. Clark, S. Shenker, *Integrated Services in the Internet Architecture: an Overview*, Internet Request For Comments: 1633, work in progress, 1994

[Braden98] B. Braden, D. Clark, J. Crowcroft, B. Davie, S. Deering, D. Estrin, S. Floyd, V. Jacobson, G. Minshall, C. Partridge, L. Peterson, K. Ramakrishnan, S. Shenker, J. Wroclawski, & L. Zhang, *Recommendations on Queue Management and Congestion Avoidance in the Internet*, Internet draft, work in progress, 1998.

[Brakmo94] L.S. Brakmo et al., *TCP Vegas: New Techniques for Congestion Detection and Avoidance,* in SIGCOMM'94, pp. 24-35, August 1994.

[Cen98] S. Cen, C. Pu, J. Walpole, *Flow and Congestion Control for Internet Streaming Applications*, Proc. SPIE/ACM Multimedia Computing and Networking '98, San Jose, CA, January 1998, pages 250-264.

[Cho98] K. Cho, *A Framework for Alternate Queueing: Towards Traffic Management by PC-UNIX Based Routers*, In USENIX ‘98, Annual Technical Conference, New Orleans, LA, June 1998.

[Chung00] J. Chung, M. Claypool, *Dynamic-CBT – Better Performing Active Queue Management for Multimedia Networking,* submitted to NOSSDAV’00.

[Claffy98] K. Claffy, G. Miller, K. Thompson, *The Nature of the Beast: Recent Traffic Measurements from an Internet Backbone,* http://www.caida.org/outreach/papers/Inet98/

[Clark97] D. Clark, J. Wroclawski, “An Approach to Service Allocation in the Internet”, Internet Draft, work in progress, draft-clark-diff-svc-alloc-00.txt, July 1997, Int-Serv Working Group at the Munich IETF, August, 1997

[Clark99] M. Clark, K. Jeffay, *Application-Level Measurements of Performance on the vBNS,* Proceedings of the IEEE International Conference on Multimedia Computing and Systems, Vol. II, pp. 362-366. Florence, Italy, June 1999

[Charzinski 00] J. Charzinski, *Internet Client Traffic Measurement and Characterisation Results,* Proc. ISSLS 2000, Stockholm, 18. 23.6.2000, Paper 12:1.

[Christiansen00] M. Christiansen, K. Jeffay, D. Ott, F. D. Smith, *Tuning Red for Web Traffic*, ACM SIGCOMM 2000, Stockholm, Sweden, August, 2000.

[Delgrossi93] Delgrossi, L., Herrtwich, R., Vogt, C., Wolf, L., *Reservation Protocols for Internetworks: A Comparison of ST-II and RSVP*, Proceedings of Network and Operating System Support for Digital Audio and Video, Sept. 1993.

[Delgrossi95] L. Delgrossi, L. Berger, Editors, *Internet Stream Protocol Version 2 (ST2) Protocol Specification - Version ST2+*, Request for Comments, RFC 1819, Internet Engineering Task Force, August 1995.

[Delgrossi93] Delgrossi, L., Halstrick, C., Hehmann, D., Herrtwich, R., Krone, O., Sandvoss, J., Vogt, C., 1993. *Media Scaling for Audiovisual Communication with the Heidelberg Transport System*, Proc. ACM Multimedia T93, Anaheim, CA, August 1993, pp. 99-104.

[Eleftheriadis95] A. Eleftheriadis and D. Anastassiou, *Meeting Arbitrary QoS Constraints Using Dynamic Rate Shaping of Coded Digital Video*, Proceedings, 5th International Workshop on Network and Operating System Support for Digital Audio and Video, Durham, New Hamshire, April 1995, pp. 95-106

[Feng] W. Feng, W. Feng, "The Impact of Active Queue Management on Multimedia Congestion Control," Seventh International Conference on Computer Communications and Networks (IC3N'98), Lafayette, Louisiana, October 1998

[Feng99] W. Feng, D. Kandlur, D. Saha, K. Shin, "A Self-Configuring RED Gateway", INFOCOM '99, March 1999.

[Feng99] W. Feng, *Improving Internet Congestion Control and Queue Management Algorithms*, Ph.D. Dissertation, University of Michigan, Ann Arbor, MI, 1999.

[Ferguson97] P. Ferguson, *Simple Differential Services: IP TOS and Precedence, Delay Indication, and Drop Preference*, Internet draft, work in progress, 1997

[Ferrari90] D. Ferrarri, *Client Requirements for Real-Time Communication Services*, IEEE Communications, November, 1990, pp. 65-72.

[Firoiu00] V. Firoiu, M. Borden, *A Study of Active Queue Management for Congestion Control*, INFOCOMM 2000

[Floyd91a] S. Floyd & V. Jacobson, *On Traffic Phase Effects in Packet-Switched Gateways*, Computer Communications Review, April 1991

[Floyd91] Floyd. S., *Connections with Multiple Congested Gateways in Packet-Switched Networks Part 1: One-way Traffic*, Computer Communications Review, Vol. 21, No. 5, October 1991, p. 30-47

[Floyd93] S. Floyd, & V. Jacobson, *Random Early Detection gateways for Congestion Avoidance*, IEEE/ACM Trans. on Networking, V.1 N.4, August 1993, p. 397-413.

[Floyd94] S. Floyd, *TCP and Explicit Congestion Notification*, ACM Computer Communications Review, 24(5):10-23, Oct. 1994.

[Floyd95a] S. Floyd & V. Jacobson, *Link-Sharing and Resource Management Models for Packet Networks*, IEEE/ACM Transactions on Networking, V.1, N.4, August 1995, pp. 365-386.

[Floyd95b] S. Floyd, V. Jacobson, S. McCanne, C. Liu, and L.Zhang, *A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing*, ACM SIGCOMM ’95, pp. 342-355.

[Floyd97a] S. Floyd, *RED: Discussion of Setting Parameters*, http://www.aciri.org/floyd/REDparameters.txt

[Floyd97b] S. Floyd, *RED: Discussions of Byte and Packet Modes*, March 1997. e-mail archive, With additional comments from January 1998 and October 2000. http://www.aciri.org/floyd/REDaveraging.txt

[Floyd97c] S. Floyd, *RED: Optimum functions for computing the drop probability?*, http://www.aciri.org/floyd/REDfunc.txt

[Floyd98] S. Floyd, S., & K. Fall, *Promoting the Use of End-to-End Congestion Control in the Internet*, IEEE/ACM Transactions on Networking, August 1999

[Gaynor96] M. Gaynor, *Proactive Packet Dropping Methods for TCP Gateways*, October 1996, URL http://www.eecs.harvard.edu/~gaynor/final.ps

[Guerin98] R. Guerin, S. Kamat, V. Peris, and R. Rajan, *Scalable QoS Provision Through Buffer Management*, Proceedings of SIGCOMM'98, (to appear).

[Gupta99] P. Gupta and N. McKeown, *Packet classification on multiple fields*, in Proceedings of ACM SIGCOMM'99, ACM, August 1999.

[Hashem89] E. Hashem, *Analysis of Random Drop for Gateway Congestion Control*, Report LCS TR-465, Laboratory for Computer Science, MIT, Cambridge, MA, 1989, p. 103.

[Hof93] Hoffman, D., Speer, M., Fernando, G., *Network Support for Dynamically Scaled Multimedia Data Streams*, Fourth International Workshop on Network and Operating Systems Support for Digital Audio and Video, Lancaster, UK, November 1993, pp. 251-262.

[ISO93] ISO/IEC: Coded representation of picture, audio and multimedia/hypermedia information (MPEG-2). Video 4th Working Draft, Sept. 9, 1993

[Jacobs96] S. Jacobs and A. Eleftheriadis, *Providing Video Services Over Networks Without Quality of Service Guarantees.* In RTMW'96, Sophia Antipolis, France, Oct. 1996.

[Jacobson88] V. Jacobson, *Congestion Avoidance and Control*, ACM Computer Communications Review, 18(4):314-329, Proceedings of ACM SIGCOMM '88, Stanford, CA, August 1988.

[Jain88] Jain, R. and Ramakrishnan, K.K., *Congestion Avoidance in Computer Networks with a Connectionless Network Layer: Concepts, Goals, and Methodology*, Proc. IEEE Computer Networking Symposium, Washington, D.C., April 1988, pp. 134-143.

[Kent87] C. Kent and J. Mogul, *Fragmentation Considered Harmful,* SIGCOMM Symposium on Communications Architectures and Protocols, pp. 390-401, Aug. 1987

[Kent88] C. Kent, K. McCloghrie, J. Mogul, and C. Partridge, *IP MTU Discover Options,* Request for Comments, RFC 1063, Internet Engineering Task Force, July 1988.

[Keshav97] S. Keshav, *An Engineering Approach to Computer Networks: ATM Networks, the Internet, and the Telephone Network*, Addison-Wesley, professional computing series, 1997

[Laksham96] T.V. Lakshman, A. Neidhardt, T. Ott, *The Drop From Front Strategy in TCP Over ATM and Its Interworking with Other Control Features*, Proc. Infocom 96, pp. 1242-1250.

[Le Gall91] D. Le Gall, *MPEG: A Video Compression Standard for Multimedia Applications*, Communications of the ACM, 34(4), April 1991

[Leland94] W. Leland, M. Taqqu, W. Willinger, and D. Wilson, *On the Self-Similar Nature of Ethernet Traffic (Extended Version)*, IEEE/ACM Transactions on Networking, 2(1), pp. 1-15, February 1994

[Lin97] D. Lin & R. Morris, *Dynamics of Random Early Detection*, In Proceedings of ACM SIGCOMM '97, pages 127--137, Cannes, France, October 1997.

[Mah97] B. A. Mah, *An Empirical Model of HTTP Network Traffic*, in Proceedings of the Conference on Computer Communications (IEEE Infocom), (Kobe, Japan), pp. 592-600, Apr. 1997.

[Mahdavi97] J. Mahdavi and S. Floyd, *TCP-Friendly Unicast Rate-based Flow Control*, June 1997. Technical Note, available from http://ftp.ee.lbl.gov/floyd/papers.html

[Mankin91] A. Mankin, K.K. Ramakrishnan, editors for the IETF Performance and Congestion Control Working Group, "Gateway Congestion Control Survey", RFC 1254, August 1991, p. 21

[Mathis97] M. Mathis, J. Semke, J. Mahavi, and T. Ott. *The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm.*, IEEE Network, 11(6), November/December 1997.

[McCreary00] S. McCreary and K. C. Claffy, *Trends in Wide Area IP Traffic Patterns*, Tech. Rep., CAIDA, Feb. 2000

[Rosolen99] V. Misra, W. Gong, D. Towsley, *A Fluid-based Analysis of a Network of AQM Routers Supporting TCP Flows with an Application to RED,* Proceedings of ACM SIGCOMM'00, (Stockholm, Sweden, September 2000)

[Morris97] R. Morris, *TCP Behavior with Many Flows*, IEEE International Conference on Network Protocols, October 1997, Atlanta, Georgia.

[Nagle84] J. Nagle, *Congestion Control in IP/TCP*, RFC 896, January 1984.

[Nee97] P. Nee, *Experimental Evaluation of Two-Dimensional Media Scaling Techniques for Internet Videoconferencing*, Master's Thesis, University of North Carolina – Chapel Hill, 1997

[Nichols97] K. Nichols, V. Jacobson , & L. Zhang, *A Two-bit Differentiated Services Architecture for the Internet*, Internet draft, work in progress, 1997.

[Nichols98] K. Nichols, S. Blake, F. Baker, D. Black, *Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers*, RFC 2474, December 1998

[Ott97] T. Ott, J. Kemperman, and M. Mathis. *Window Size Behavior in TCP/IP with Constant Loss Probability*. In the Fourth IEEE Workshop on the Architecture and Implementation of High Performance Communication Systems (HPCS' 97), Chalkidiki, Greece, June 1997.

[Pan00] R. Pan, B. Prabhakar, K. Psounis, *CHOKe: A Stateless Active Queue Management Scheme for Approximating Fair Bandwidth Allocation*, INFOCOMM'00

[Paxson96] V. Paxson, *End-to-End Routing Behavior in the Internet.* In IEEE/ACM Transactions on Networking 5(5), pp. 601-615

[Paxson97] V. Paxson, *Automated Packet Trace Analysis of TCP Implementations*. In Proceedings of ACM SIGCOMM, September 1997

Apparently indicates Loss-rate is currently increasing in the Internet (as referenced in [Feng]).

[Postel80] J. Postel, *Unreliable Datagram Protocol,* RFC 768, August, 1980

[Postel81] J. Postel, *Internet Control Message Protocol*, RFC 792, September 1981

[Postel82] J. Postel, *Simple Mail Transfer Protocol,* RFC 821, August, 1982

[Ramakrishnan99] K.K Ramakrishnan, S. Floyd, and D. Black, *The Addition of Explicit Congestion Notification (ECN) to IP*. Internet draft draft-ietf-tsvwg-ecn-04.txt, work in progress, June 2001

[Rizzo97] L. Rizzo, *Dummynet: a simple approach to the evaluation of network protocols*, ACM Computer Communication Review, January 1997, URL:http://www.acm.org/sigcomm/ccr/archive/1997/jan97/ccr-9701-rizzo.pdf

[Romanow95] A. Romanow and S. Floyd, *Dynamics of TCP Traffic over ATM Networks*, IEEE Journal on Selected Areas in Communications, 13(4), 1995. URL: http://www.nrg.ee.lbl.gov/nrg-papers.html

[Rosolen99] V. Rosolen, Bonaventure, O., and G. Leduc, *A RED discard strategy for ATM networks and its performance evaluation with TCP/IP traffic,* ACM Computer Communication Review, July 1999, URL:http://www.acm.org/sigcomm/ccr/archive/1999/jul99/ccr-9907-leduc.pdf

[Salim00] J. Hadi Salim, and U. Ahmed, *Performance Evaluation of Explicit Congestion Notification (ECN) in IP Networks*, Request for Comments, RFC 2884, July 2000

[Saltzer84] J. H. Saltzer, D. P. Reed, and D. D. Clark, *End-to-end Arguments in System Design*, ACM Transactions on Computer Systems, pages 277-288, 1984

[Schulzrinne96] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. *RTP: A Transport Protocol for Real-time Applications*. Technical Report RFC 1889, Internet Engineering Task Force, Jan. 1996

[Schneiderman98] Shneiderman, Ben. Designing the User Interface: Strategies for Effective Human-Computer Interaction, Third Edition. 1998. Addison Wesley. p. 360

[Sisalem97] D. Sisalem, H. Schulzrinne, F. Emanuel. *The Direct Adjustment Algorithm: A TCP-Friendly Adaptation Scheme.* Technical Report, GMD-FOKUS, Aug. 1997. Available from http://www.fokus.gmd.de/usr/sisalem.

[Sisalem98] D. Sisalem, H. Schulzrinne, *The Loss-Delay Based Adjustment Algorithm: A TCP-Friendly Adaptation Scheme,* Eighth International Workshop on Network and Operating Systems Support for Digital Audio and Video, Cambridge, UK, July 1998, pp. 215-226.

[Srinivasan99] V. Srinivasan, S. Suri, and G. Varghese. Packet Classification using Tuple Space Search. In Proc. of SIGCOMM '99, pp. 135--146.

[Stevens94] W. R. Stevens, *TCP/IP Illustrated, Vol. 1: The Protocols*. Addison-Wesley, Reading, Mass. 1994.

[Stone95] D. L. Stone, *Managing the Effect of Delay Jitter on the Display of Live Continuous Media*, Ph.D. Dissertation, University of North Carolina at Chapel Hill, Chapel Hill, NC, July, 1995.

[Talley97] T. M. Talley, *A Transmission Control Framework for Continuous Media*, Ph.D. Dissertation, University of North Carolina at Chapel Hill, Chapel Hill, NC, May 1997.

[Thompson97] K. Thompson, G.J. Miller, and R. Wilder. *Wide-area Internet Traffic Patterns and Characteristics.* IEEE Network, 11(6):pp. ??-??, November/December 1997.

[Varghese96] G. Varghese. *On Avoiding Congestion Collapse*, Technical report, Nov. 19, 1996, view-graphs, Washington University Workshop on the Integration of IP and ATM.

[Villamizar94] C. Villamizar and C. Song, *High Performance TCP in ANSNET*. Computer Communications Review, V. 24, N. 5, October 1994, pp. 45-60. URL ftp://ftp.ans.net/pub/papers/tcp-performance.ps

[Wakeman95] I. J. Wakeman, *Congestion Control for Packetised Video in the Internet,* Ph.D. Dissertation, University of London, London, 1995.

[Waldvogel97] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, *Scalable High Speed IP Routing Lookups,* ACM SIGCOMM '97. PalaisdesFestivals, Cannes, France, pp.25-36

[Willinger95] W. Willinger, M. S. Taqqu, R. Sherman, D. V. Wilson, *Self-Similarity Through High-Variability: Statistical Analysis of Ethernet LAN Traffic at the Source Level*, ACM SIGCOMM ’95, pp. 100-113, August 1995.

[Wilson93] F. Wilson, I. Wakeman, and W. Smith. *Quality of Service Parameters for Commercial Application of Video Telephony*. In *Human Factors in Telecommunications Symposium*, Darmstadt, Germany, Mar. 1993.

[Wolf82] C. Wolf, *Video Conferencing: Delay and Transmission Considerations*, in Teleconferencing and Electronic Communications: Applications Technologies, and Human Factors, L. Parker and C. Olgren (Eds.), 1982

[Ziegler99] T. Ziegler, S. Fdida, & U. Hofmann, *A Distributed Mechanism for Identification and Discrimination of non-TCP-friendly Flows in the Internet*, Submitted to ICNP99.

[Zakon99] R. H. Zakon, *Hobbes' Internet Timeline v5.0* http://www.isoc.org/zakon/Internet/History/HIT.html, January 2000