Scalable TCP Congestion Control

A thesis presented

by

Robert Tappan Morris

to

The Division of Engineering and Applied Sciences

in partial fullfillment of the requirements

for the degree of

Doctor of Philosophy

in the subject of

Computer Science

Harvard University

Cambridge, Massachusetts

January, 1999

Abstract

Routers in IP packet-switched networks signal congestion to senders by discarding packets. Such discards, as a side-effect, are often the key factor determining the quality of network service perceived by users. For this reason network designers need techniques to explain, predict, and control the packet discard rate.

This thesis explains the discard rate for TCP traffic in terms of the interaction between load and capacity. The key insights are that load should be measured as the number of senders actively competing for a bottleneck link, and capacity as the total network buffering available to those senders. The thesis shows how to predict discard rates using these measures. It also proposes a new queuing method that can limit the discard rate over a wide range of loads.

Acknowledgements

H. T. Kung took me under his wing at a time when my prospects seemed dark. Since then I've profited immensely from his wisdom and high standards.

Mike Smith and Alan Chapman inspire me with their quiet excellence.

Trevor Blackwell, Brad Karp, Dong Lin, and Koling Chang have shared years of ideas, arguments, and good times with me.

Paul Graham, my particular friend, understands how to lead a worthwhile life; would that I had his insight.

Ingrid Bassett has stayed sweet and sane despite everything.

Finally, my parents still love me.

Contents

Chapter 1

Introduction

Buffer space in Internet routers has traditionally been viewed as a way to absorb transient imbalances between offered load and capacity. Choosing the amount of buffer memory has been something of a black art: too little risks high loss rates and low link utilization, too much risks high queuing delay. Current practice favors limiting buffer space to no more than is required for good utilization. The result is that routers use loss to control congestion, by discarding packets when they run out or are in danger of running out of memory. Most Internet traffic sources respond to loss by decreasing the rate at which they send data, making loss feedback a reasonable approach to congestion control.

Loss feedback misses an important factor. The TCP protocol that controls sources' send rates degrades rapidly if the network cannot store at least a few packets per active connection. Thus the amount of router buffer space required for good performance scales with the number of active connections. If, as in current practice, the buffer space does not scale in this way, the result is highly variable delay on a scale perceptible by users. Evidence collected from busy parts of the Internet suggests that this effect might be significant.

The simultaneous requirement of low queuing delay and of large buffer memories for large numbers of flows poses a problem. This thesis suggests the following solution. First, routers should have physical memory in proportion to the maximum number of flows they are likely to encounter. Second, routers should enforce a dropping policy aimed at keeping the actual queue size proportional to the actual number of active flows. This system, referred to here as FPQ (Flow-Proportional Queuing), automatically chooses a good tradeoff between queuing delay and loss rate over a wide range of loads. In particular, FPQ allows a network administrator to set the loss rate to a low constant despite varying load.

FPQ provides congestion feedback using queuing delay, which it makes proportional to the number of flows. TCP's window flow control causes it to send at a rate inversely proportional to the delay. Thus the combination of TCP and FPQ causes each TCP to send at a rate inversely proportional to the number of TCPs sharing a link, just as desired. Under heavy load it turns out that FPQ's delay feedback produces the same overall delay as the timeouts produced by loss feedback. FPQ, however, produces a fairer distribution of delays than loss feedback: every transfer sees the same queuing delay in FPQ, whereas loss feedback sharply segregates transfers into unluckly ones (which see timeouts) and lucky ones (which do not).

Part of the reason that FPQ works is that number of active flows is an important measure of load in TCP networks, and the ability of the network to store packets is an important measure of capacity. These are non-traditional measures: bandwidth is typically used for both load and capacity. Much of this thesis is an exploration of the relationship among number of flows, network storage, and loss rate. These ideas will prove useful not just in the design of FPQ, but also in the analysis and tuning of existing router buffering systems.

The remainder of the thesis starts (in the next chapter) with a tutorial on TCP and router design, along with a review of work in areas related to scalable TCP congestion control.

Chapter 3 describes the simulation environment used by the rest of the thesis.

Chapter 4 presents evidence about how many flows busy Internet links carry, evidence about Internet loss rates, and informal support for the view that number of flows is a good measure of load.

Chapter 5 considers this notion of load more thoroughly, exploring the interactions among number of flows, buffer space, and loss rate. It presents both simulations and predictive formulae. Particular attention is given to setting the parameters of the Random Early Detection (RED) queuing algorithm.

Chapter 6 proposes a new packet dropping strategy called FPQ, designed to help a router adapt its queue length and loss rate to the measured number of active flows.

Chapter 7 uses simulation to show that FPQ causes the same average delay as traditional buffering techniques, but significantly lower delay variation.

Chapter 8 summarizes the lessons learned in this thesis.

Chapter 2

Background and Related Work

This chapter presents the details of TCP and IP router design required to understand the rest of the thesis. It also reviews current research relevant to scalable TCP congestion control. Sections 2.1 and 2.2 are for review and are not intended to be controversial or even interesting. Sections 2.3 through 2.5 touch on areas in which this thesis extends or differs from previous work.

2.1 TCP

The service that an IP network provides is the transport of individual packets of data. These packets consist of a header and at most a few thousand bytes of data. The header contains the address of the destination computer (or "host"). IP networks consist of routers connected by links. The routers contain tables which allow them to forward each packet from source to destination host, potentially along a path consisting of many routers and links. The network usually delivers the packet, but may instead discard it, corrupt it, or deliver it to the wrong host.

Most applications need a higher level of service than this. First, they often need to exchange more data than will fit in a packet. Second, often want reliable in-order delivery of data. Third, they usually want to identify which of many applications running on the destination host the data should be delivered to. Fourth, a pair of communicating applications may want a notion of a "connection" suitable for a sustained conversation.

TCP [46] provides these services in a generic way suitable for many applications. When two applications on different hosts wish to communicate, they each create a TCP endpoint (or "socket") and each tell their local TCP software to create a connection between the sockets. The TCPs on the two host exchange connection setup packets, called "SYN" packets. When the applications indicate that the conversation is over, the TCPs exchange "FIN" (or finish) packets.

During the life of a connection the applications may send each other data. TCP divides this data into packets small enough for the IP network to transport. TCP numbers the packets and sends each packet's number in the packet header. The receiver uses these numbers to reconstruct the original stream of data in order. These are called "sequence" numbers.

When a TCP endpoint receives a data packet, it sends an acknowledgment (or "ACK") packet back to the sender. The ACK contains the lowest sequence number that the receiver has not yet received. Suppose, for example, that a receiver receives packets 1, 2, and 4. On receiving 4, it will return an ACK containing 3. This tells the sender two things. First, that the receiver has packets 1 and 2, so the sender need never think about them again. Second, that the receiver has not received packets 3 and 4, so the sender may need to re-send them.

After the sender sends a packet to the receiver, it sets a "retransmit timer." If the timer goes off before the sender gets an ACK for the packet, the sender retransmits the packet. The sender sets the timer adaptively, by measuring the "round trip time" between each packet transmission and the receipt of the corresponding ACK. In practice, TCPs measure this time with a granularity of half a second, and use a minimum retransmit timer of one second. If successive retransmissions of the same packet fail, TCP doubles the retransmission timeout after each.

2.1.1 Congestion Window

Suppose that a TCP sender waited for an ACK after sending each data packet. This has the valuable effect of causing the sender to send at a rate that is related to the network capacity: if the network is fast, the ACKs will come back quickly and the sender will send quickly; if the network is slow, both processes will proceed slowly. This prevents the sender from sending faster than the network capacity. It may also cause the sender to send far slower than the network capacity. Suppose, for example, that the network can send 1000 packets per second, but has a round-trip speed-of-light propagation delay of 0.1 second. Then a single TCP sender can send 10 packets per second, or just 1% of the network capacity.

To solve this problem, TCP sends a "window" of packets before waiting for an ACK. In the example above, TCP should set its window to 100 packets, which would keep the network busy for the 0.1 seconds until the return of the first ACK. Each time TCP receives an ACK, it sends another data packet. This procedure tends to maintain a window of packets in flight, and to keep the network busy. In general the correct window size is the product of available capacity (or bandwidth) and round-trip propagation delay, or $1000 \cdot 0.1$ packets in this case. Note that this window is mostly stored on the network links–as photons moving through fibers, for example–and mostly not as packets buffered in routers.

A TCP sender actually handles ACKs as follows. Suppose the desired window size is *cwnd*. Whenever TCP receives an ACK, it calculates how much data is still in flight–that is, how much data has been sent but for which the sender has received no ACK. Usually the answer will be *cwnd* less two packets–TCP receivers traditionally send an ACK for every other data packet. Then the sender sends enough new packets so that there are again *cwnd* packets in flight; usually this means sending two new packets.

While using a small window may cause under-utilization, using windows larger than necessary also causes problems. Network routers must buffer the excess packets, causing either excessive delay or buffer overflow and packet loss. Since the "delay-bandwidth" product may vary by orders of magnitude between different network paths, TCP cannot reasonably used a single fixed window size.

However, TCP cannot directly determine either factor of the delay-bandwidth product, so it cannot directly decide an appropriate window. Instead it uses an adaptive "congestion window" algorithm [24], which effectively searches for the maximum reasonable window by increasing it until the network starts dropping packets. The algorithm comes in two parts, called slow start and congestion avoidance.

2.1.2 Slow Start

TCP maintains a guess at the current reasonable window size, called the slow-start threshold (or *ssthresh*). Whenever TCP starts sending after being idle (or timing out), it would like to send with a window of size *ssthresh*. It turns out to be a bad idea to send the entire window in a burst, which might force a nearby router to buffer the whole window; far better to spread the window over a round-trip time, so that they are stored in transit on the links. TCP accomplishes this using this algorithm, called "slow-start:"

1. Initialize the window size, *cwnd*, to one packet.
2. Whenever an ACK that acknowledges new data arrives (a "positive" ACK), increase *cwnd* by one packet.
3. If the resulting *cwnd* is less than *ssthresh*, stay in slow-start. Otherwise, enter congestion avoidance mode, described in the next section.

This doubles *cwnd* every round-trip time, so that TCP opens its window to *ssthresh* in time proportional to $\log ssthresh$ instead of all at once.

A typical initial *ssthresh*, used when a TCP connection is first created, is 64 kilobytes. *ssthresh* is adjusted after packet loss as described below.

2.1.3 Congestion Avoidance

A TCP in congestion-avoidance mode is searching for a reasonable window size. The goal is to increase the window slowly in case available network bandwidth has increased, perhaps because of reduced competition from other connections–but to detect reductions in bandwidth and reduce the window accordingly. Congestion avoidance works as follows:

1. *cwnd* starts out at *ssthresh* due to slow-start.
2. After each entire window of positive ACKs arrive, increase *cwnd* by one packet.
3. After a timeout (that is, a lost packet), set *ssthresh* to $\frac{cwnd}{2}$ and enter slow-start.

This algorithm has the effect of increasing the window by one packet per round-trip time when there is no loss. The rate at which TCP sends data is equal to one window per round-trip time, or $\frac{cwnd}{rtt}$. As long as *cwnd* is less than the delay-bandwidth product, increases in *cwnd* cause increases in send rate, because *rtt* is fixed at the round-trip propagation delay. However, once *cwnd* exceeds the delay-bandwidth product, routers must buffer the excess packets. This buffering increases the round-trip time by one packet transmission time per buffered packet. If we measure *rtt* in units of packet transmission times, we can see that increases in *cwnd* leave $\frac{cwnd}{rtt}$ unchanged once *cwnd* is equal to the delay-bandwidth product.

At some point the congestion avoidance algorithm will increase *cwnd* so much that some router runs out of buffer memory–or perhaps some competing connection will do so. At that point the router much drop one or more packets. Once TCP notes the lost packet, it realizes that *cwnd* was too large and effectively halves it (by modifying *ssthresh* and entering slow-start). All other things being equal, TCP's window varies up and down by a factor of two over time.

Two router design considerations spring immediately from these algorithms. First, a router must provide about one delay-bandwidth of buffering if it wishes to make sure that a single TCP can sustain a send rate equal to the link bandwidth [51]. Otherwise TCP will send at less than the link rate after it cuts its window in half. Second, TCP will tend to keep router buffers full no matter how large they are. This means that building routers with huge buffer memories is an invitation to excessive queuing delay.

2.1.4 Fast Retransmit

TCP traditionally waits at least one second before timing out and re-transmitting a lost packet. This causes packet loss to have a large impact on efficiency; for example, a packet loss rate of 5% would prevent TCP from sending faster than 20 packets per second, no matter how fast the network. TCP uses a mechanism called "fast retransmit" [47] that can recover from a packet loss in a round trip time instead of a second.

After a packet is lost, the TCP receiver sends ACKs for each of the remaining packets in the window; each of these ACKs repeats the lost packet's sequence number to indicate that the packet wasn't received. The sender notes these duplicated ACKs. If it sees three in a row, it retransmits the lost packet, sets *cwnd* to half of *ssthresh*, and enters slow-start.

Note that this fast-retransmit mechanism only works if the window is four or more packets.

2.2 Routers

An IP router consists of a number of ports connected to transmission links, a mechanism to forward each packet from the port it arrives on to the appropriate output port, and memory at each output port to buffer packets that cannot be sent immediately. Routers are interesting because they are where speed mismatches occur. For example, a router may receive packets on a link with higher capacity than the relevant output link. Or a router, all of whose links are the same speed, may have multiple input streams converging on the same output.

In the short run a router can absorb excess traffic by buffering it. This is not sustainable in the long run; routers are built with finite memory, and even with infinite memory the queuing delays would grow without bound. Instead, overloaded routers send feedback to the data sources telling them to slow down. In practice this feedback takes the form of dropped packets, which interact with TCP's congestion window algorithms as described above.

Within this framework two approaches are widely used or accepted: drop-tail routers and RED routers.

2.2.1 Drop-Tail and Random-Drop Routers

Most current routers use "drop-tail" queuing. Each output port has a single first-in first-out (FIFO) queue of packets. When a packet arrives on an input, it is appended to the relevant output's queue. Each output transmits the packets from its queue, in order, as fast as it can. Each queue has a limit on the number of packets allowed, typically from a few dozen to a few hundred. If a packet arrives and the queue is already at its limit, the router discards (drops) the arriving packet.

Drop-tail queuing is simple and efficiently implemented, but causes two problems with TCP traffic. First, when a queue gets full, the router tends to drop a packet from each connection using the queue [22, 45]. This causes "global synchronization" of the connections' window size decreases, leading to under-utilization.

Second, it turns out that the probability that a drop-tail router drops a packet (i.e. has a full queue when the packet arrives) is not independent of which connection the packet is from. TCP's ACK feedback mechanism causes repeating patterns or "phase effects" in packet arrivals, which can cause different connections to experience different loss rates [18].

A variant of drop-tail called random-drop discards a randomly selected packet from the output queue when a packet arrives and the queue is full. The arriving packet is queued. This approach helps eliminate phase effects [35, 18], but does not avoid global synchronization [19].

2.2.2 RED Routers

A technique called Random Early Detection (RED) [19] eliminates drop-tail's global synchronization and phase effects, and treats transient bursts more fairly.

A simplified description of RED's workings will be helpful in understanding this thesis. RED (like drop-tail) maintains a single FIFO queue per output port. It remembers the average queue length q_{avg} over recent time. The network administrator must set three RED parameters: min_{th}, the minimum acceptable queue length; max_{th}, the maximum acceptable queue length; and max_p, the maximum dropping probability. When a packet arrives, one of three things happen. If $q_{avg} < min_{th}$, the router always queues the packet. If $q_{avg} >= max_{th}$, the router always drops the packet. If $min_{th} \leq q_{avg} < max_{th}$, the router drops the packet with probability proportional to the average queue length:

$$max_p(q_{avg} - min_{th})/(max_{th} - min_{th}).$$

RED computes the average queue length q_{avg} from the instantaneous queue length q each time a packet arrives with this filter:

$$q_{avg} = (1 - w_q)q_{avg} + w_q q$$

The recommended value for w_q is 0.002, which averages the queue length over about 500 packet arrival times.

Since RED drops packets with a probability governed by average queue length, rather than instantaneous queue length, it tends to drop each packet with equal probability. This distributes the drops among connections in proportion to their bandwidths. Since RED starts dropping with some low probability as soon as the average queue length exceeds min_{th}, it tends to spread losses out over time, thus avoiding global synchronization. RED's averaging also allows buffering of transient bursts, as long as they are noticeably shorter than the queue averaging interval.

2.3 Router Buffer Provisioning

How much packet buffer memory should router ports contain? Previous work suggests two answers. First, one delay-bandwidth product of packet storage. An abstract justification for this is that it allows for the natural delay at which the end-systems react to signals from the network [31, 29]. A TCP-specific justification is that this is the minimum amount of buffering that will ensure full utilization when a number of TCP connections share a drop-tail router [51]. With any less than a full delay-bandwidth product of router memory, the TCPs' windows would sum to less than a delay-bandwidth product after they all halve their window sizes. At least one major router vendor [49, 48] uses a delay-bandwidth product of memory.

Another answer is that buffering is only needed to absorb transient bursts in traffic [4]. This is true as long as the traffic sources can be persuaded to send at rates that consistently sum to close to the link bandwidth. RED, with its ability to de-synchronize TCP window decreases, should be able to achieve this. This means that utilization with RED should be relatively insensitive to parameters such as buffer size [25]. Since large buffer memories contribute to undesirable queuing delay, the recommended RED buffer size (i.e. max_{th}) is only a few dozen packets [16]. Adding weight to the view that router buffers should be small are studies of long-range dependent traffic that conclude that increasing router buffer space is not an effective way to control loss rate [20, 12].

Both of these answers effectively use packet discard as the only mechanism to control load. As later chapters describe, this can lead to high variation in user-perceived delay. This thesis suggests using a combination of queuing delay and discard instead.

2.4 Load and Congestion

Much of this thesis deals with how a TCP network should cope with high load and congestion. These are vague terms. Intuitively, "load" encompasses legitimate user actions tend to place the network under strain and cause it to behave badly. Congestion is bad behavior caused by load. This thesis will argue that an important source or measure of load in TCP networks is the number of simultaneously active connections sharing a bottleneck, and that congestion amounts to loss rates high enough to force TCPs into timeout.

An early definition of congestion in TCP networks focused on a phenomenon called "congestion collapse" [40]. As the load on the network increased, the throughput, queuing delay, and loss rate also increased. Increases in loss and delay caused increases in retransmission rates. These retransmissions themselves increased loss and delay, as well as consuming bandwidth to no useful purpose. The result was that after a certain point, increased load caused a precipitous decrease in useful throughput. Other window-based flow control systems saw similar phenomena [27].

Congestion collapse was solved for TCP by careful window size [24] and retransmit timer [28] management. TCP now does a much better job of matching its window size (and send rate) to available network resources and of retransmitting only lost packets.

At this point Internet congestion manifests itself in high loss rates. This loss is not catastrophic, as in the days of congestion collapse. Lost packets do, however, waste network bandwidth up to the point where they are discarded. Packet loss rates are hard to characterize globally, but measurements from different times and places are available. Bolot [3] reports loss rates on the order of 9% between France and the US in 1992. Paxson [44] reports a loss rates 2.7% in late 1994 and 5.2% in late 1995, between a variety of pairs of hosts on the Internet. Handley [21] reported loss rates on the order of 10% in 1996 from sites all over the Internet receiving MBone transmissions. Yajnik, Kurose, and Towsley [52] report losses of about 10% in 1995 and 1996, again from sites all over the Internet receiving MBone transmissions.

High loss rates decrease TCP's average window size and thus TCP's send rate [14]. This is all to the good, since high loss rates indicate an excess of traffic. However, high loss rates also tend to push TCP into retransmit timeouts, with deleterious effects described in later chapters.

2.5 Coping with Many Flows

The idea that window flow control in general has scaling limits because the window size cannot fall below one packet has long been known [1]. Villamizar [51] suggests that this might limit the number of TCP flows a router could support, and that increasing router memory or decreasing packet size could help. Eldridge [11] notes that window flow control cannot control the send rate well on very low-delay networks, and advocates use of rate control instead of window control; no specific mechanism is presented. In the somewhat different context of ATM virtual circuit flow control, Kung and Chang [30] describe a system that uses a small constant amount of switch buffer space per circuit in order to achieve good scalable performance.

One drawback of drop-tail and RED routers is that they must drop packets in order to signal congestion, a problem particularly acute with limited buffer space and large numbers of flows. These packets consume bandwidth on the way to the place they are dropped, and they also cause increased incidence of TCP timeouts. An alternate strategy is to explicitly notify senders of congestion, either with a flag in the packet header, or with a special packet. Floyd [15] presents simulations of a TCP and RED network using explicit congestion notification (ECN). Floyd observes that ECN reduces timeouts for interactive flows, and thus provide lower-delay service.

Feng et al. [13] note that ECN cannot prevent routers with limited buffer memory from dropping packets if the number of flows is very large. As a solution, they implement Eldridge's rate-control proposal which allows TCP to send less than one packet per round trip time. The combination of rate-control and ECN increases the number of TCPs that can coexist with limited buffer space. Their rate increase algorithm is multiplicative: every time it sends a packet, it increases the rate by a fraction of itself. This increase policy turns out to have no bias towards fairness [7], so that differences in send rates among connections will be stable. In contrast, TCP's standard window algorithms use an additive increase of one packet per round trip time, and do tend towards fairness.

Previous work also exists investigating modifications to routers to allow them to cope with large numbers of flows. Nagle [41] observes that providing unlimited buffer space in a FIFO router does not help by itself. The fundamental problem is that sources that increase their send rates are rewarded with a larger share of the bandwidth. Nagle proposes a separate queue for each flow, with round-robin scheduling among the queues. In such an architecture, flows that send at their fair share or less are rewarded by low delay, and flows that send faster are penalized by increased delay. The optimum behavior with this kind of fair queuing is for each flow to buffer just one packet in the router. If flows really acted like this, routers could be built with unlimited buffer memory, but only use it in proportion to the number of active flows.

In an effort to achieve the benefits of Nagle's fair queuing without the overhead of maintaining per-connection queues, Lin [34] proposes a fair dropping strategy called FRED. Packets are stored in a FIFO, but no flow is allowed to buffer more than a handful of packets. This bounds the unfairness allowed, and requires only per-flow counting, not queuing. FRED should scale well with the number of flows because it allows a router to built with a large amount of packet memory, but only to use it in proportion to the current number of flows.

This thesis makes two contributions in this area. First, it presents analytical and simulation results about the interaction between number of TCP flows and buffering; these should help in configuring conventional routers as well as in understanding their behavior. Second, it proposes the FPQ mechanism, which provides a good delay/loss/fairness tradeoff with significantly less complexity than previous systems.

Chapter 3

Simulation Environment

Most of the simulations described in this thesis were performed on a uniform configuration designed to highlight buffering and flow count issues. As pictured in Figure 3.1, this configuration involves N TCP senders converging on router A. Router A must send what data it can across its link to router B, and either buffer or discard the rest. Thus router A is the bottleneck router, and the link from A to B is the bottleneck link.

The intent of this configuration is to capture what happens on heavily loaded links. For example, the link from A to B might be the link from an Internet Service Provider's backbone to a customer. Such links usually run slower than either the backbone or the customer's LAN, and thus act as bottlenecks.

The simulations use the NS 1.4 simulator [36]. Unless otherwise noted, they use the parameters given in Figure 3.2.

Note that Figure 3.2 implies that the average round-trip propagation delay is 100 milliseconds. Since a 10 megabit link can send 2170 packets per second, the delay-bandwidth product is 217 packets. This means that a single TCP with a 64 kilobyte window can only use about half the link capacity. In addition, each sender's link to router A only runs at 10 times its fair share; this means that if

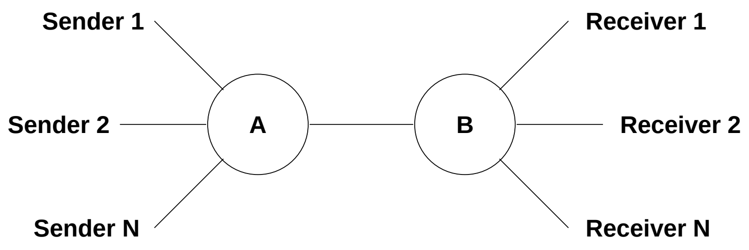


Figure 3.1: Standard simulation configuration. Each of N TCP senders has its own host and its own link to router A. Router A connects to router B over a bottleneck link.

Packet size	576 bytes
Maximum window	64 kilobytes
TCP timer granularity	0.5 seconds
TCP delayed-ACK timer	0.2 seconds
A to B propagation delay	45 milliseconds
A to B bandwidth	10 megabits/second
Sender i to A propagation delay	random, 0 to 10 milliseconds
Sender i to A bandwidth	$10 \cdot (10/N)$ megabits/second
Maximum drop-tail queue length	217 packets
RED max_{th}	217 packets
RED min_{th}	5 packets
RED max_p	2%
Maximum RED queue length	no hard limit
Simulation length	500 seconds

Figure 3.2: Summary of simulation parameters.

more than 90% of the senders are timing out or otherwise idle, the remaining senders won't be able to use the whole link capacity. In practice this doesn't happen.

By default, each TCP sender has an unlimited amount of data to send. This means that most of the simulations focus on the steady-state behavior of TCP, rather than its start-up behavior. Some of the simulations use finite length transfers, which the relevant chapters will describe as they arise.

The simulations include randomness to avoid deterministic phenomena that would never occur in the real world [2]. The connection start times are randomly spread over the first 10 seconds of the simulation and the senders' access link propagation delays are randomly selected over range of about 10% of the total.

Except for the slight randomization, all connections experience about the same round-trip time. This avoids unfairness due to TCP's tendency to use bandwidth in inverse proportion to the round-trip time.

Except when noted, no transmission errors are simulated. Thus all packet loss is caused by router A discarding packets in response to congestion. The simulated drop-tail and random drop routers discard packets when the queue already contains a full 217 packets. The RED router drops according to the RED algorithm; there is no hard limit on the amount of buffer memory that RED can use.

The TCP version used in the simulations is Tahoe [47]. The main difference between Tahoe and the later Reno is that Tahoe invokes slow start after a fast retransmit, whereas Reno uses "fast recovery" [46] to continue in congestion avoidance. Section 5.1 will demonstrate that the behavior of Tahoe and Reno are similar for the purposes of this work.

Many of the graphs in this thesis have number of flows on the x axis, and some simulation result (such as loss rate) on the y axis. Each point typically represents the average of that result over a single 500-second simulation. The upper bound on the number of flows presented is typically about 1000. This is not an unreasonable number of flows to expect a 10 megabit link to support, since it results in the per-flow fair share being somewhat slower than a modem.

Chapter 4

Load and Number of Flows

Much of this thesis revolves around the notion that the number of active flows competing for a bottleneck is the best measure of load in a TCP network. "Load" in this context means some kind of stress that users can legitimately put on a network that may cause the network to behave badly. Stresses that fit this description include speed of users' access links, total amounts of data that users want to send, the number of individual transfers that users wish to perform, the number of active users, and the number of active flows. This chapter presents flow count estimates and then informal analyses motivating the choice of active flow count as the most useful measure of load.

4.1 Flow Definition

The definition of "active flow" that makes sense in this work is an end of a TCP connection that has packets in flight or is in retransmit timeout. This corresponds to the number of distinct instances of TCP algorithms that are placing a load on the network.

Note that the definition includes TCPs in timeout, even though such TCPs aren't sending packets. Most timeouts occur because of congestion losses–that is, they occur because the network has insufficient capacity. Ignoring such flows would misleadingly understate the load on the network.

This definition of flow works badly for transfers not limited by TCP's congestion control algorithms. Examples include the user-to-application direction of interactive applications such as telnet and the X window system. These flows tend to consist of very short packets, with too little data outstanding to be controlled by TCP's window mechanism. Such flows account for no more than about 8% percent of Internet backbone traffic, and probably much less [50].

This definition also ignores TCP ACK packets, which load network links in the opposite direction from the corresponding data. Assuming one 40-byte ACK for each pair of 576-byte packets, the ACK load amounts to less than 4% of the data load. The effects of congestion on ACKs have been investigated elsewhere

[53]. For these reasons this thesis pays no special attention to ACKs.

4.2 Detecting Flows

The most convenient way to count flows on a link is to monitor the packets flowing over the link. One possibility is to increment the count on seeing a TCP connection setup (SYN) packet and decrement the count on a TCP finish (FIN) packet, correcting for retransmitted SYN and FIN packets. This approach has the defect of counting all TCP connections, not just active connections. Let us call this technique a SYN/FIN flow count.

One can target active flows more specifically by counting the number of distinct TCP connection identifiers that appear in the packet headers seen in each interval. The connection identifier consists of the IP addresses of the two hosts involved and the port number on each host. This technique ignores idle connections. However, no choice of interval works perfectly. Intervals less than one or two seconds will miss TCPs in retransmission timeout. Longer intervals may consider TCPs that never overlap in time as simultaneous, and may count mostly-idle TCPs as active. One or two seconds seems a reasonable compromise. Let us call this technique an active flow count.

An intermediate approach avoids problems with lost SYN/FIN packets, idle flows that never terminate, and delicate choices of time interval. It records the times at which each flow first and last sent a packet. Then, for each instant in time, it totals the number of flows whose first and last times straddle that instant. Let us call this technique a known flow count. As with the SYN/FIN flow count, it over-estimates the number of active flows.

Many of the flow counts presented in the following sections are from standard half-duplex Ethernet, in which there is no inherent idea of packets going in one of two directions. That is, even if the Ethernet is used primarily as a link between two routers, the packets in both directions compete for the Ethernet bandwidth. Most WAN links, however, are full duplex: they have physically separate media for the two directions between a pair of connected routers. Flow counts from half-duplex networks should be halved before comparing with counts from otherwise comparable full-duplex networks.

4.3 Flow Counts

How many simultaneous active flows might one expect on a busy Internet link? Three sources of information are available, all using the techniques from Section 4.2: published measurements, publically available packet traces from which measurements may be taken, and measurements taken by the author on nearby networks. The resulting statistics should be viewed as no more than suggestive of conditions on typical Internet links, since they are tiny samples from a huge system. The statistics reported here also span almost ten years of Internet evolution, and particularly the era of rapid growth of Web traffic.

4.3.1 Published Counts

Caceres, Danzig, Jamin, and Mitzel [6] published statistics taken from the University of California at Berkeley's wide-area link in October 1989. These statistics do not include flow counts. They do include a count of about 1000 new flows per hour, counted with a known flow technique. Assuming flows last an average of 20 seconds [50], one might guess that the link in question carried about 5 known flows at any given time.

Claffy, Braun, and Polyzos [9] published active flow counts observed on a T3 (45 megabit) link from the University of Illinois at Urbana-Champaign to the NSFNET. These counts were taken on an afternoon in March 1993. They report median and maximum flow counts for a range of measurement intervals. For 2 seconds, they report a median of 400 flows and a maximum of 500. For 4 seconds, a median of 500 and a maximum of 600.

Newman, Lyon, and Minshall [42] published statistics taken in September 1995 on an FDDI ring connecting the San Francisco Bay area to the Internet backbone. They do not include flow counts. They do include a count of about 140 new TCP flows per second. They used a known flow technique, but used only IP addresses to distinguish flows, not port numbers. Thus their counts are lower than others. They report an average flow duration of about 50 seconds, so one might guess that the link carried about 7000 known flows at any given time.

Thompson, Miller, and Wilder [50] report known flow counts from an OC3 (155 megabit) ATM link in internetMCI's backbone in September 1997. The count varied from 80000 at night to a sustained 200000 flows during the day. These numbers include all flows, of which 75% are TCP. The average flow lasted about 20 seconds, and included about 20 packets.

4.3.2 Counts from Published Traces

Another source of flow counts is publically available packet trace files. The traces are usually generated with the UNIX *tcpdump* [26, 37] program, which makes a record of every packet seen on a local network, including packets not addressed to the host running tcpdump. The trace files include a copy of each packet's header and the time at which the packet was observed. Only packets containing data contributed to the counts, effectively ignoring ACKs.

Analysis of three such traces revealed the following per-second active flow counts. The first trace [38], made available by Digital Equipment Corporation, was collected at 14:00 PST on a day in March 1995 on a 10 megabit Ethernet in Palo Alto that carried most of DEC's traffic to the Internet backbone. Packets from about 130 distinct TCP flows are visible in this trace in each second. The average flow length was 45 packets and 31 seconds.

The second trace [8] was collected on an FDDI ring at the FIX-West [5] interchange point in September 1996. It has packets from about 1300 distinct TCP flows in each second.

The third trace [39] was collected at 14:40 EST on March 13 1997 on a 10

Name	*Year*	*Count*	*Notes*
Caceres &c	1989	5	Known flows. My estimate.
Claffy &c	1993	400	Active flows, 2 seconds.
Newman &c	1995	7000	Known flows. My estimate. Host pairs.
DEC Trace	1995	130	Active flows, 1 second.
FIX-West Trace	1996	1300	Active flows, 1 second.
Harvard Trace	1997	200	Active flows, 1 second.
Harvard Trace	1998	350	Active flows, 1 second.

Figure 4.1: Summary of flow counts from literature and traces.

megabit Ethernet connecting Harvard's main campus to the Internet. This trace has packets from about 200 distinct TCP flows in each second. The average flow length was 15 packets and 11 seconds. A second trace [23] taken on April 16 1998 at the same point in the network, but after a quadrupling in the speed of Harvard's backbone link, showed 350 distinct flows per second during the day.

Figure 4.1 summarizes the flow counts from this and the previous section.

4.4 Harvard Trace Details

Subsequent chapters will use a few flow statistics such as day/night flow count variation and flow durations. The numbers given below come from the Harvard 1997 and 1998 traces, and are generally comparable to published figures [50].

Figure 4.2 shows active flow count as a function of time of day from the two Harvard traces mentioned in Section 4.3.2. Each point is the average of the active flow counts from 60 one-second intervals. The 1997 trace lasted from about 10am until the same time on the next day. The 1998 trace lasted from 4pm on one day until 4pm on the next. Both are shown wrapped around on the graph. The first trace was taken at a time when Harvard's backbone connection was a 10 megabit link, the second when it was a 45 megabit link. Note that the number of flows varies by a factor of 3 to 1 from day to night.

Figure 4.3 shows the distributions of three measures of flow size: bytes, packets, and seconds of duration. These were measured between the first and last packets of each flow. For bi-directional flows, the direction with more bytes was used, and the direction with less was ignored. The average number of bytes was 11600, with median 1700. The average number of packets 22, with median 7. The average duration was 9 seconds, with median 2 seconds.

4.5 Correlation of Loss and Flows

Chapter 5 will present simulation and analytic evidence that the number of simultaneous flows directly drives the loss rate on busy TCP networks. Ideally

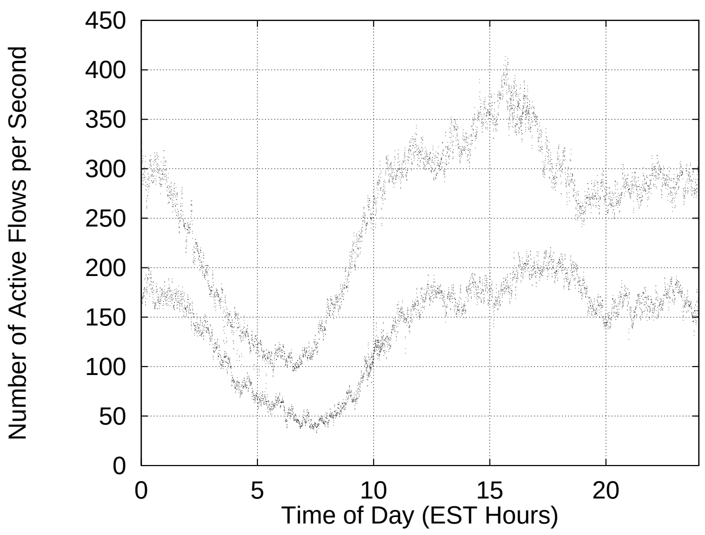


Figure 4.2: Number of active TCP flows in each second, averaged over one-minute intervals. Upper points are from the Harvard April 1998 trace; lower points are from the Harvard March 1997 trace. Note the near-doubling of traffic over a year, and the change from day to night.

experimental evidence could be used to support this idea, perhaps by simultaneously observing router buffer overflows at a bottleneck router along with flow counts. Each of these numbers are available for a few points on the Internet, but not both. The best one can do is examine packet traces.

One can estimate the number of lost packets in a trace by observing the sequence numbers of successive TCP packets from each flow. Doing this correctly is hard [44, 43], since packets may appear out of order for reasons other than loss. The loss estimates in this section are counts of gaps in sequence numbers, corrected for the gap caused by a timeout retransmission, and also corrected for pairs of out-of-order packets. This technique assumes that ACKs and data packets following the same path experience the same loss rate. It assumes that each contiguous gap in the sequence number space corresponds to one lost packet; this probably leads to an under-estimation of the loss rate. Another cause of under-estimation is the possibility that entire windows of packets may be lost, resulting in no visible sequence number gaps.

Figure 4.4 is essentially a scatter plot of the numbers of incoming flows and incoming loss rates for each one-second interval of the Harvard March 1997 trace. Instead of plotting each point, the points corresponding to each number of flows are summarized as a median and a bar from the 25th to 75th percentile of loss rate. The right-hand end of the x axis is the 95th percentile of number of active flows. The graph indicates a clear connection between number of flows and loss rate.

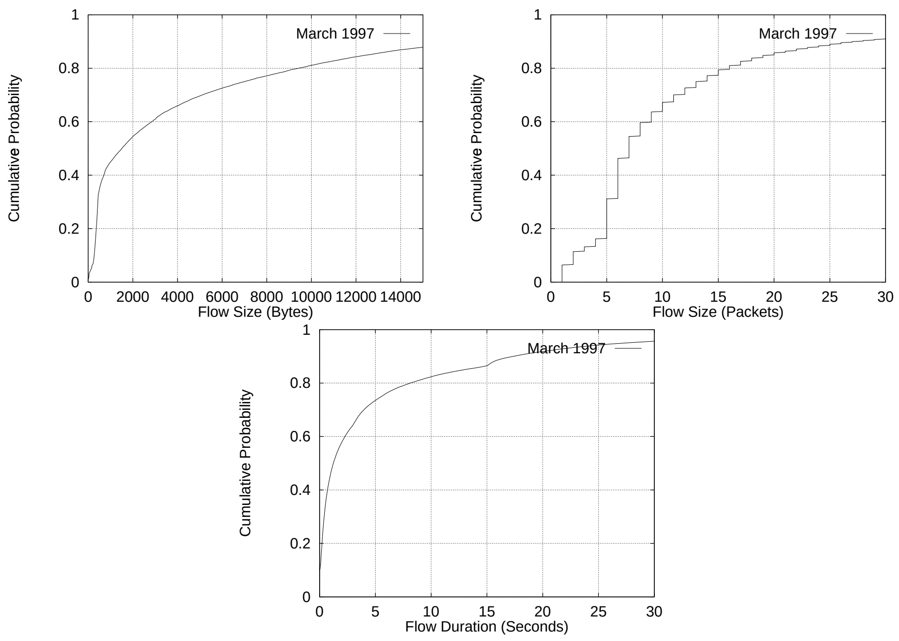


Figure 4.3: Cumulative distributions of bytes, packets, and seconds of duration for flows from the Harvard March 1997 trace.

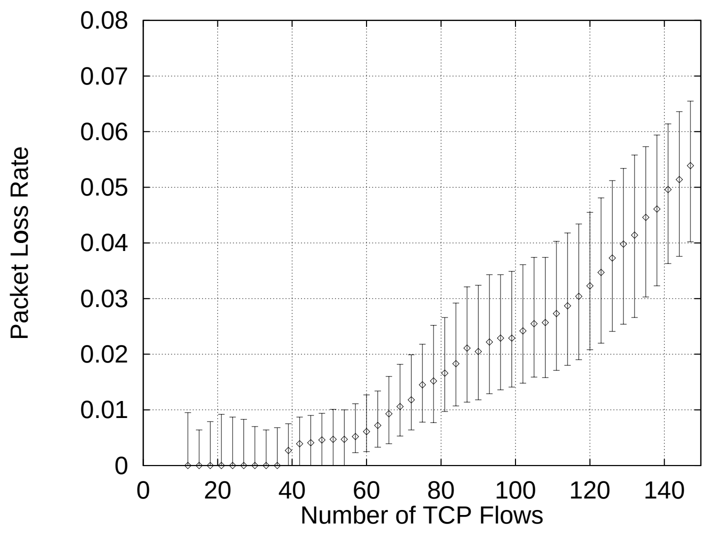


Figure 4.4: Loss rates observed when different numbers of flows were active. Each bar indicates 25th percentile, median, and 75th percentile loss rates for all the one-second intervals with a particular number of flows. From the March 1997 Harvard trace.

Flow count could be correlated with loss but not cause the loss. The most likely way this could happen is if flow count and bandwidth used were correlated, and increases in bandwidth caused increases in loss. For example, each flow might correspond to one modem's worth of bandwidth, and each flow might send at a constant bit rate. To test this possibility, Figure 4.5 shows loss as a function of bandwidth used in a style similar to that of Figure 4.4. The average bandwidth used is 23% of the link rate, with standard deviation of 10%. The right hand end of the x axis is the 95th percentile of utilization. This particular trace exhibits a stronger correlation between flow count and loss than between bandwidth and loss.

These two graphs suggest that number of flows is the more important contributor to loss, and thus the more interesting measure of load.

4.6 Load and Access Link Rate

Suppose network users could control the amount of bandwidth they required. This might be relevant measure of load, since it might affect the network loss rate. Variation in desired bandwidth could happen in a number of ways, the most obvious being changes in the speeds of users' access links. For example, users might upgrade from modems to cable modem links. What can we say about whether such changes might matter more than changes in the number of

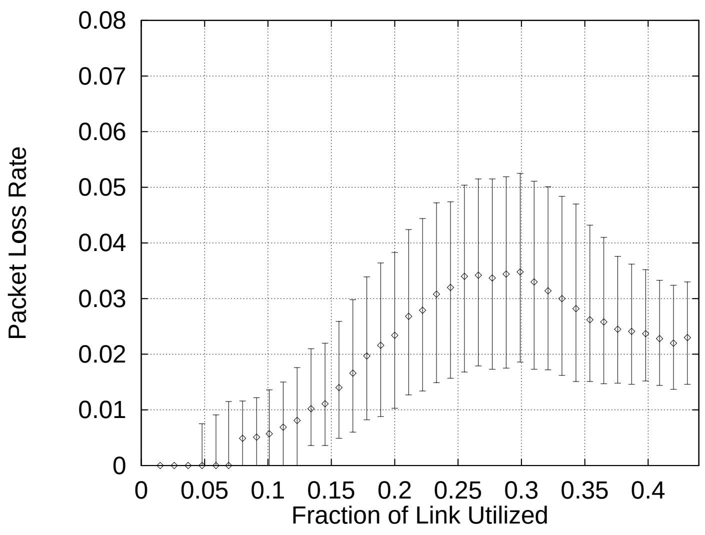


Figure 4.5: Loss rates observed when different amounts of link bandwidth were used. Each bar indicates 25th percentile, median, and 75th percentile loss rates for all the one-second intervals when particular amounts of bandwidth were used. From the March 1997 Harvard trace.

flows?

Figure 4.6 shows simulated loss rate as a function of user access link speed. The simulation involves 300 users, each with one active TCP flow and a separate access link. The simulation configuration involved the standard simulation configuration: a bottleneck link of 10 megabits, a round-trip propagation delay of 100 ms, and RED with a max_{th} of 217. Each user's fair share of the bottleneck is 33 kilobits/second.

If each user can send no faster than his fair share, no loss can occur. As access link speed increases above one fair share, the loss rate also increases. The loss rates quickly level off as TCP's window algorithms start to be the limiting factor. At this point the loss rate depends mostly on the number of flows, as detailed in Chapter 5. This implies that access link speed is an important measure of load only in networks with capacities close to the sum of the access link speeds. In the past such configurations have not been economical; much more common have been networks whose internal capacity is a modest multiple of the speeds of the fastest single access links.

In conclusion, this chapter has informally argued that number of flows is a good measure of TCP network load, and has presented a set of useful flow statistics. The next chapter will provide a more rigorous analysis of the relationship between number of flows and loss rate.

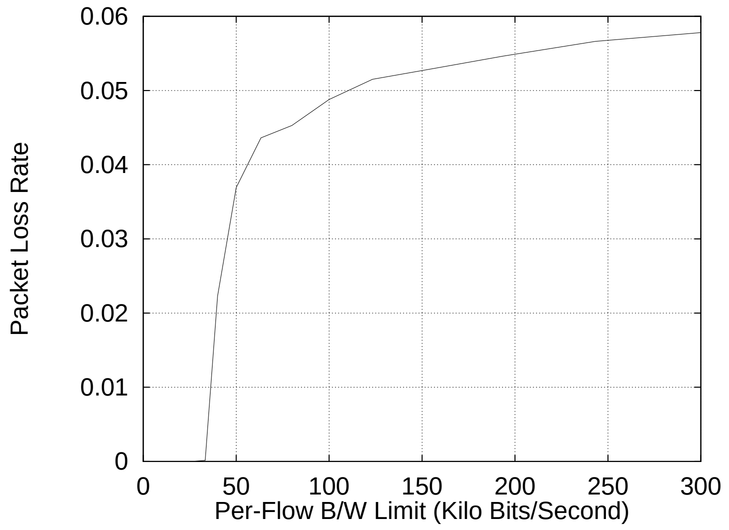


Figure 4.6: Loss rate as a function of users' individual access link rates. Each user has one flow that uses one access link. The bottleneck is a 10 megabit link, the two-way propagation time is 100 ms, and there are 217 router packet buffers. The x-axis shows each user's link rate. Each user's fair share is 33 kbits/second. Note that the access link rate has a modest effect on the loss rate except in a narrow region around the fair share.

Chapter 5

TCP's Response to Load

This chapter explains how TCP networks respond to changing numbers of flows. Half of the story is TCP's response to the packet drop rate imposed by the network. The other half is the way in which the number of flows interact with TCP and router algorithms to determine the drop rate.

The method in this chapter is to derive formulas based solely on TCP and router algorithms, and to compare them with simulations. Since the formulas and simulations are independent, they can be used to verify each other. The formulas are helpful not just in predicting TCP and router behavior, but also in configuring routers; the chapter contains quantitative recommendations for setting RED parameters.

Most of the results in this thesis use the Tahoe [47] version of TCP, as described in Section 2.1. Many Internet hosts use the more recent Reno TCP, which incorporates the "fast recovery" algorithm [46]. Fast recovery allows TCP to use congestion avoidance after a fast retransmit, while Tahoe uses slow start. Section 5.1 will plot simulation results for both versions' response to packet loss. The responses turn out to be nearly identical, since there is little difference between slow start and congestion avoidance when the window size is small. For this reason, subsequent sections will use only the simpler Tahoe.

The analyses in this chapter assume that TCP's behavior is dominated by the congestion window mechanism and single timeouts. The results are not accurate when TCP spends a significant amount of time in retransmit timer backoff. Simulation suggests that the fraction of time that TCP spends in backoff is roughly equal to the loss rate. This means that the results in this chapter are not likely to be accurate for loss rates larger than 5 or 10 percent. The main error that results from a high loss rate is that larger than expected numbers of TCPs are idle, and do not contribute to queue length or loss rate. Thus the formulas in this chapter will tend to over-estimate queue length and loss rate when the load is very high.

The initial result, Equation 5.1, is taken from existing work [14]. The remainder of the chapter is original.

5.1 TCP's Response to Loss

Since packet loss is the primary way that the network communicates congestion information to TCP, it's important to understand TCP's response to packet loss. Of most interest is the number of packets that TCP keeps in flight in response to a given loss rate, as this number governs the bandwidth and buffer space that TCP uses. We can estimate this number by first estimating the average size of the congestion window as a function of the loss rate, and then estimating the fraction of time TCP spends sending rather than pausing in timeout.

5.1.1 Average Congestion Window Size

Suppose that the fraction of packets that the network drops is l, and that the network spaces these losses evenly. TCP's congestion window will go through a repeating pattern, experiencing a loss at some size w_{max}, cutting back to $w_{max}/2$, and growing back to w_{max} again. The window grows by about 1 packet every two windows (it would be 1 packet per window without the "delayed ACK" mechanism). Thus TCP will send about w_{max} windows of packets before the window has grown from $w_{max}/2$ to w_{max}. The average window size is about $\frac{3}{4}w_{max}$, so the window will grow from $w_{max}/2$ to w_{max} after sending about $\frac{3}{4}w_{max}^2$ packets. We also know from the loss rate that the number of packets TCP sends during this cycle is $1/l$. Solving this equation yields $w_{max} = \sqrt{\frac{4}{3l}}$. The average window size is then $\frac{3}{4}w_{max}$, or

$$w_{avg} \approx \frac{0.87}{\sqrt{l}} \tag{5.1}$$

The point here is that the average congestion window size is determined almost solely by the loss rate. This observation is implicit in [24] and was explicitly published in a different form in [14]. Figure 5.1 compares this predicted value with the observed value of the sender's congestion window variable in simulations.

Equation 5.1 over-estimates the number of packets in flight in three ways. First, TCP implementations send only whole packets, which means they truncate non-integer congestion window sizes. This results in an average error of one half packet when the window size is two or more. Second, delayed ACK effectively reduces the number of packets in flight by an average of one half packet. These two errors can be corrected by subtracting one from Equation 5.1, yielding

$$w_{eff} \approx \frac{0.87}{\sqrt{l}} - 1 \tag{5.2}$$

The third discrepancy, due to timeouts, requires more work to fix.

5.1.2 Timeouts

If TCP always kept one congestion window of packets in flight, the discussion above would be enough. However, a high loss rate or an unlucky loss pattern

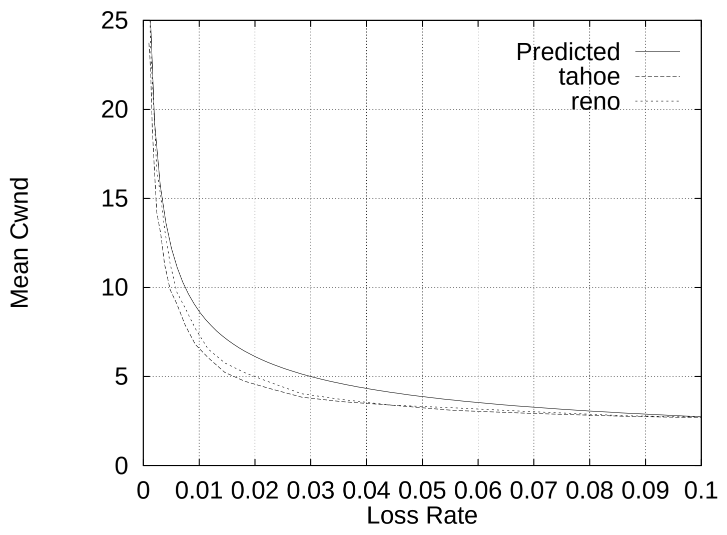


Figure 5.1: Average TCP congestion window as a function of loss rate. From simulations with a single TCP and a uniform probability of each packet being dropped.

may cause TCP to fall into retransmission timeout, causing to stop sending for a second or more. To a first approximation, if TCP Tahoe is to avoid timeouts, losses cannot be spaced less than 13 packets apart: three to generate enough duplicate ACKs to trigger fast retransmit, and another ten to allow slow-start to increase enough that fast retransmit will work for the next loss. The fraction of losses followed by another loss within 13 packets is roughly $1-(1-l)^{13}$, so the timeout rate o measured in timeouts per packet is

$$o \approx l(1-(1-l)^{13}) \tag{5.3}$$

The fraction of time TCP is likely to spend timing out, given an average window size w, a round trip time r, and assuming a timeout interval of one second, is

$$O \approx \frac{1}{1+\frac{r}{wo}} \tag{5.4}$$

These equations only work for window sizes greater than four, since a smaller window may not be able to recover from even a single lost packet. They are not accurate for very large windows, either, since such windows make it easier to recover from losses. In addition, the precise value of 13 was chosen somewhat empirically. Figures 5.2 and 5.3 compare the predictions with TCP simulations.

A word of caution: these timeout results assume uniform distribution of losses. Bursty losses, coupled with TCP's tendency to send bursts of packets, mean that the losses may be concentrated in a small number of TCPs, forcing

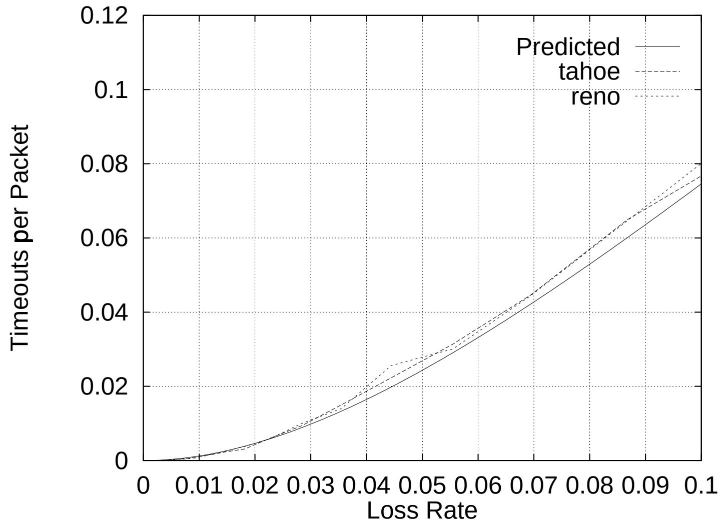


Figure 5.2: Timeouts per Packet (o) as a Function of Loss Rate.

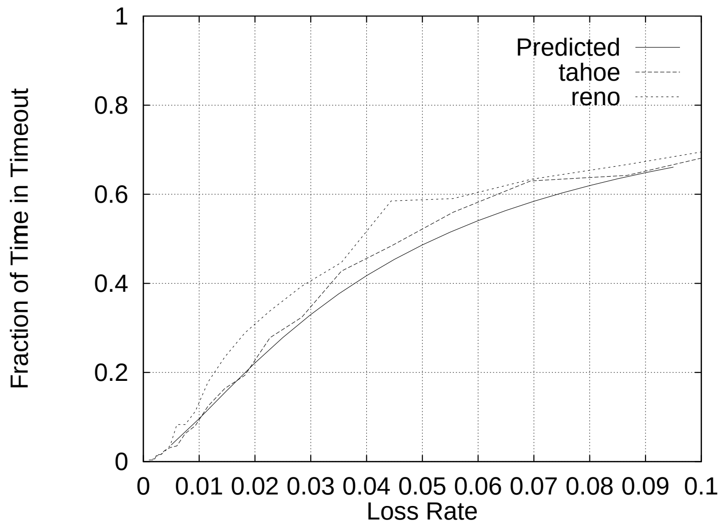


Figure 5.3: Fraction of Time in Timeout (O) as a Function of Loss Rate. RTT = 100ms.

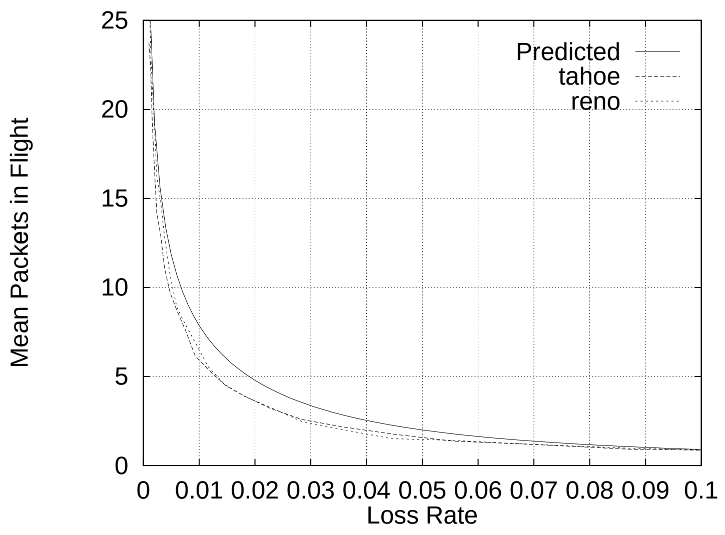


Figure 5.4: Packets in flight as a function of loss rate. RTT = 100ms.

them into timeout. The more bursty the losses, the more timeouts will occur, even if the average loss rate stays the same. Drop-tail gateways tend to produce bursts of loss, whereas one of RED's goals is to spread drops evenly over time.

5.1.3 Average Packets in Flight

Recall that our goal was to find how many packets TCP will keep in flight as a function of the loss rate imposed by the network. Equation 5.1 tells us TCP's window size when it is sending, and Equation 5.4 tells us how much of the time TCP is sending. From them we can predict the average number of packets in flight:

$$(1 - O)w_{avg} \tag{5.5}$$

Figure 5.4 shows the number of packets TCP keeps in flight derived from simulations. These numbers came from a simulated 10 megabit network with a round trip time of 100 milliseconds.

5.2 Drop-Tail and Random Drop Analysis

The basic analysis of a single TCP's response to loss forms one part of a model for the behavior of multiple TCPs sharing a bottleneck. The other key part is the bottleneck router's packet discard policy. This section considers the drop-tail and random-drop policies, which turn out to have similar behavior. Section 5.3 makes a similar analysis for RED. The overall goal is an understanding of the relationship between number of flows and router buffer space.

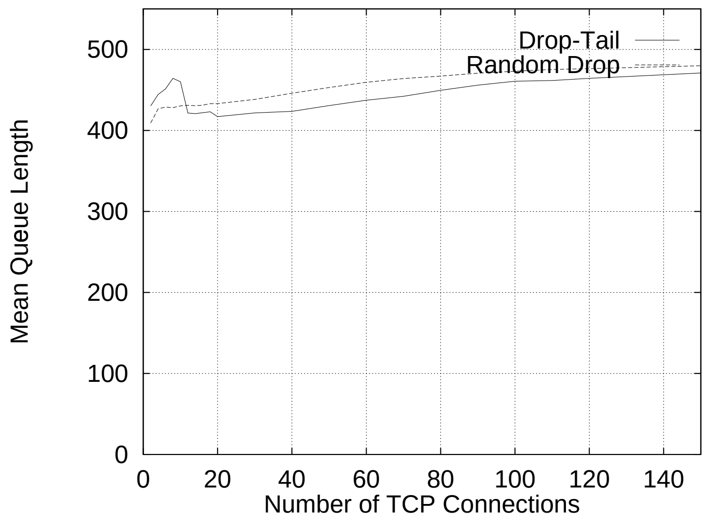


Figure 5.5: Average queue length as a function of number of flows, drop-tail and random drop. From simulations with 500 packet buffers.

Consider n TCPs competing for a 10 megabit bottleneck link, fed by a drop-tail or random drop router with 500 packet buffers, and a 10 millisecond round trip time. Since TCP will expand its window until the router runs out of buffers, we expect the router's queue to be mostly full; the simulation results in Figure 5.5 show this to be correct.

The average queue length in Figure 5.5 for random drop is longer than for drop-tail. Drop-tail forces more TCPs into timeout than random-drop, mostly due to the higher drop rate it imposes (see below), but also because it tends to concentrate each episode of loss in a few unlucky TCPs rather than over all TCPs with packets queued. Figure 5.6 compares fraction of time spent in timeout for drop-tail and random drop. With fewer TCPs active at any given moment, a drop-tail router tends to have a shorter queue.

The drop-tail loss rate can be predicted by multiplying Equation 5.2 by n to yield q, the total buffer space used by n flows at loss rate l:

$$q = n\left(\frac{0.87}{\sqrt{l}} - 1\right)$$

Solving for l as a function of n and q:

$$l = \frac{0.76n^2}{(n+q)^2} \tag{5.6}$$

Random drop differs from this model because it deletes packets from the middle of the queue rather than the end. The TCPs see losses and cut their

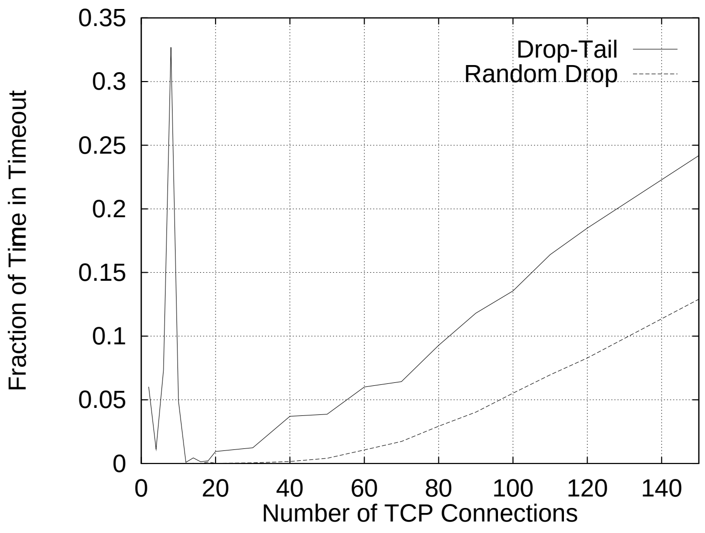


Figure 5.6: Fraction of time in timeout as a function of number of flows, drop-tail and random drop. 500 packet buffers.

windows half a window earlier than with drop-tail, causing the average congestion window to be half a packet smaller. As a result the loss and queue length formulas for random drop are:

$$q = n\left(\frac{0.87}{\sqrt{l}} - 1.5\right)$$

$$l = \frac{0.76n^2}{(1.5n+q)^2} \tag{5.7}$$

Figure 5.7 shows the simulated loss rate as n varies for the example configuration, along with Equations 5.6 and 5.7 for $q = 500$. The predictions stop being accurate as the number of flows approaches 100; at that point too few packet buffers are available to support fast retransmit, so significant numbers of flows fall into timeout.

5.2.1 Drop-Tail and Random Drop Discussion

Drop-tail and random drop scale similarly with the number of flows. Equations 5.6 and 5.7 imply that loss rate is proportional to n^2 when n is substantially less than q. If n is large compared to q, the predicted loss rate approaches some constant as n grows; the real loss rate is less than predicted due to timeouts.

The main difference between the two policies is that random drop effectively sends back congestion notification earlier by dropping from the middle of the queue, resulting in a lower loss rate. Figures 5.8 and 5.9 illustrate this with

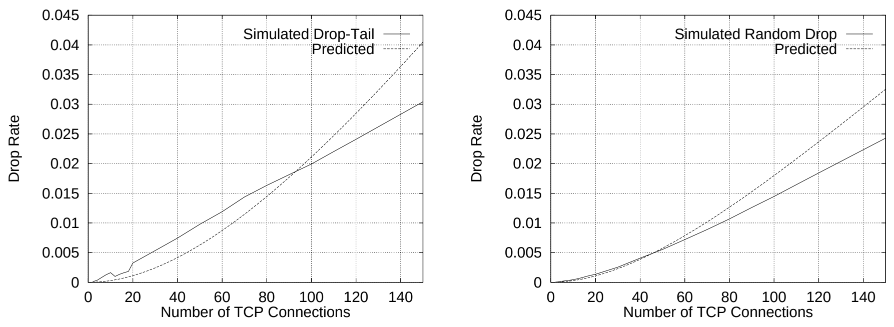


Figure 5.7: Drop rate as a function of number of flows, drop-tail and random drop. 500 packet buffers.

plots of queue length over time for drop tail and random drop simulations. Careful inspection shows that periods of time during which the queue is full (and dropping packets) are longer in the drop-tail simulation. In this respect, random drop acts somewhat like drop-front [32].

A drop tail or random drop router designed to support up to n flows must have about $5n$ packet buffers. Since the queue is likely to be full even with few flows, such a router effectively exhibits best case loss rate and worst case delay. The only way to control delay is to decrease the buffer space. The loss rate, however, is inversely proportional to the square of the buffer space. This tradeoff makes drop tail and random drop unattractive when n is large.

As pointed out in [19], drop tail and random drop routers share a tendency to synchronize the congestion window cycles of the TCPs using them. That is, when the queue fills up, such routers drop packets from many flows at about the same time, causing them all to decrease their windows. Figure 5.9 illustrates the impact of this synchronization on router queue length. The oscillation visible in Figure 5.9 is the reason why the FPQ system presented in Chapter 6 does not use random drop to control the queue length. It is also the reason why drop-tail routers should be equipped with at least one delay-bandwidth product of buffering – otherwise the router may run out of buffered packets to send after all the TCPs halve their windows, causing the link to go idle. Finally, one of RED's primary goals is to prevent TCP synchronization and consequent under-utilization.

5.3 RED Parameter Analysis

While the structure of RED is well motivated [19], little has been published regarding rules for setting RED parameters. Using number of flows as a load metric, we can derive the relationships between load, parameters, and performance, and confirm them with simulations. These rules should be useful when configuring RED equipment for known levels of load. They will also point the

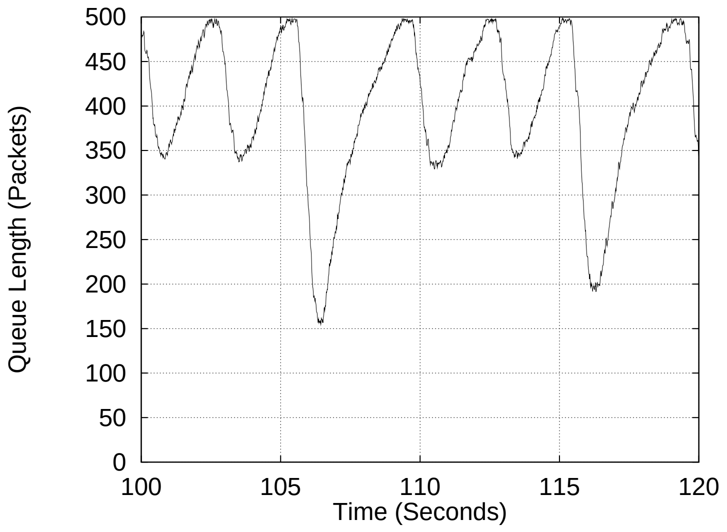


Figure 5.8: Queue length over time for a single drop tail simulation with 30 competing flows. Note the oscillation in queue length caused by synchronized window decreases.

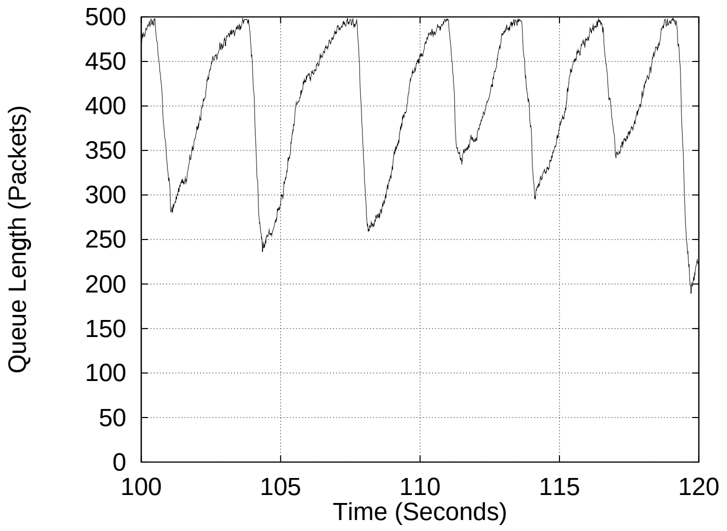


Figure 5.9: Queue length over time for a single random drop simulation with 30 competing flows. Note that the queue stays full for less time than in Figure 5.8.

way to a buffering system that automatically adapts to load, described in Chapter 6.

The RED parameters most subject to tuning are max_p and max_{th}. max_{th} is the highest average queue length (q_{avg}) the router will tolerate; the router drops all incoming packets when q_{avg} exceeds max_{th}. max_p controls the drop rate when q_{avg} is less than max_{th}, in approximately this relationship:

$$l = \frac{2 \cdot max_p \cdot q_{avg}}{max_{th}} \tag{5.8}$$

Existing suggestions for setting max_p have ranged from 0.02 [19] to 0.10 [17], with an implication that max_p should be related either to the actual network loss rate or to the desired network loss rate. Even less advice is available for setting max_{th}, mostly by way of examples in which it has values around a few dozen.

RED's most valuable advantage over drop-tail is its ability to discard randomly chosen packets, rather than just the packets that happen to arrive when the queue is full. The random choice avoids unfair phase effects [18] and bias against bursty traffic. The ability to drop before the queue is full avoids synchronization of TCP window decreases and consequent low utilization. In order to achieve these benefits, a RED router must maintain the following:

- Keep q_{avg} noticeably below max_{th} and the physical memory size. This allows RED to space packet discards out evenly, avoiding drop-tail-like forced drops, TCP window synchronization, and low utilization. A controlled queue size also avoids high delay.
- Keep q_{avg} non-zero so that link bandwidth isn't wasted.

With appropriate parameter choices RED can achieve these goals automatically, by varying the drop rate in proportion to the queue length. But how should one set the parameters?

5.3.1 Derivation of Relationships

We can approximate how the load and the parameters affect RED's queue length and discard rate as follows.

Equation 5.1 approximates how many packets one TCP keeps in flight given the loss rate. We can find how many packets N TCPs will jointly keep in flight at a given loss rate thus:

$$b = \frac{0.87N}{\sqrt{l}} \tag{5.9}$$

We will assume that most of the b packets that the TCPs keep in flight will be buffered in RED's queue, so that $b = q_{avg}$. We can solve Equation 5.8 to find what queue length would be required to produce a given loss rate:

$$b = \frac{l \cdot max_{th}}{2max_p} \tag{5.10}$$

$l \propto N^{2/3}$	$l \propto {max_p}^{2/3}$	$l \propto {max_{th}}^{-2/3}$
$b \propto N^{2/3}$	$b \propto {max_p}^{-1/3}$	$b \propto {max_{th}}^{1/3}$

Figure 5.10: Dependence of RED drop rate l and buffer use b on number of flows N and RED parameters max_p and max_{th}.

Assuming an equilibrium between Equations 5.10 and 5.9 is reached, it must look like this:

$$\frac{l \cdot max_{th}}{2max_p} = \frac{0.87N}{\sqrt{l}} \tag{5.11}$$

We can solve for the equilibrium loss rate l:

$$l = \frac{1.4N^{2/3}{max_p}^{2/3}}{{max_{th}}^{2/3}} \tag{5.12}$$

Substituting back into Equation 5.10 we can find the equilibrium queue size:

$$b = \frac{0.7N^{2/3}{max_{th}}^{1/3}}{{max_p}^{1/3}} \tag{5.13}$$

While these equations ignore a number of details, they show the order of magnitude relationships. Figure 5.10 summarizes.

The most critical lesson from Figure 5.10 is that the loss rate is much easier to control than the queue length. For example, we can keep l constant despite a doubling of N by halving max_p. However, the end result will be a doubling in the average queue length. Keeping control of the queue length is important: most routers have physical limits on buffer memory, and RED abruptly increases the loss rate to 100% when the queue exceeds max_{th}.

5.3.2 RED Parameter Simulations

Equations 5.12 and 5.13 ignore four important details:

- RED abruptly raises the discard probability to 100% when the average queue length exceeds max_{th}. This causes RED queues to be shorter than predicted by the equations, especially under heavy load.
- The network effectively stores one delay-bandwidth product of packets on the wire. The equations assume these packets are stored in the RED queue, so they over-estimate queue length and drop rate, especially when the delay-bandwidth product is large compared to max_{th}.
- Figure 5.1 shows that Equation 5.1 over-estimates the congestion window size, in that case by 20%. This causes Equation 5.13 to over-estimate queue length, and Equation 5.12 to over-estimate the loss rate.

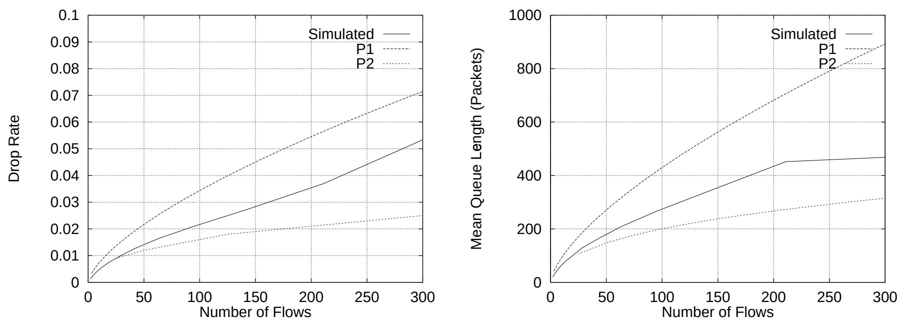


Figure 5.11: Drop rate and queue length as a function of number of flows. $max_{th} = 500$, $max_p = 0.02$.

- The equations don't account for timeouts. TCPs in timeout don't contribute to queue length, so the equations over-estimate queue length and consequently loss rate. This effect is most noticeable with high loss rate. The effect is amplified by small round trip times, which increase the proportion of time spent in timeout.

This over-estimation of queue length and drop rate is apparent in Figures 5.11 through 5.13. These graphs show the results of simulations varying just one parameter at a time. The base simulation configuration involves 100 flows competing for a 10 megabit bottleneck. The round trip propagation time is 10 milliseconds. The default max_{th} is 500 576-byte packets, and the default max_p is 0.02. Each graph includes simulation results labeled Simulated, predictions from Formulas 5.12 and 5.13 labeled P1, and predictions labeled P2 which the next section will explain.

All but two of the graphs conform roughly to the relationships in Figure 5.10. The queue length in Figure 5.11 abruptly stops rising because it reaches max_{th}, at about 200 flows. More puzzling, the predicted drop rate as a function of max_p in Figure 5.12 seems to bear little resemblance to the simulation. Most of the error stems from timeouts, which the predictions don't model. Figure 5.15 shows the average fraction of time a TCP spent timing out in the simulations. For example, when $max_p = 0.08$, half the TCPs were in timeout at any given time; this effectively means there were half as many flows as expected, with a correspondingly lower drop rate and queue length.

Increasing max_p or decreasing max_{th} both have the effect of decreasing queue length and increasing the loss rate. This has a double influence on the fraction of time spend in timeout. First, the higher drop rate increases the probability that TCP will encounter two drops close enough together to force a timeout. Second, the smaller queue decreases the round trip time, increasing the ratio of time spent in timeout to time spent transmitting. These effects conspire to blunt the effectiveness max_p at controlling drop rate.

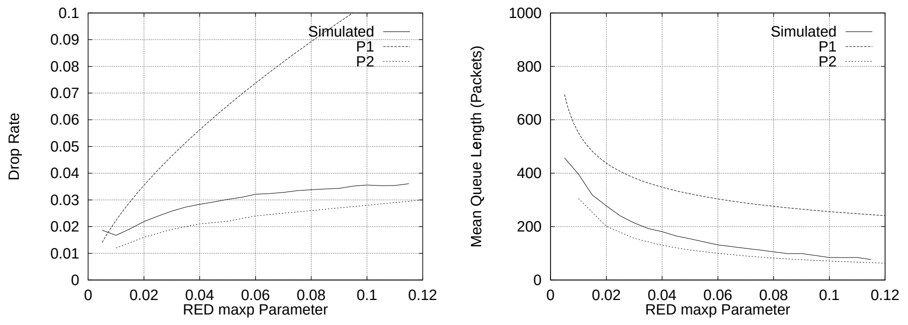


Figure 5.12: Drop rate and queue length as a function of max_p. 100 flows, $max_{th} = 500$.

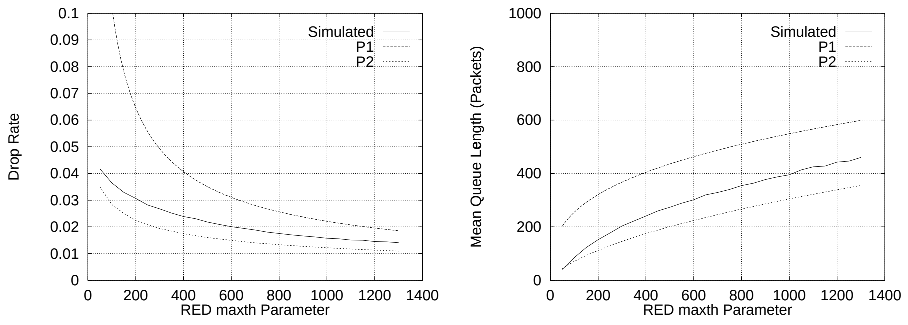


Figure 5.13: Drop rate and queue length as a function of max_{th}. 100 flows, $max_p = 0.02$.

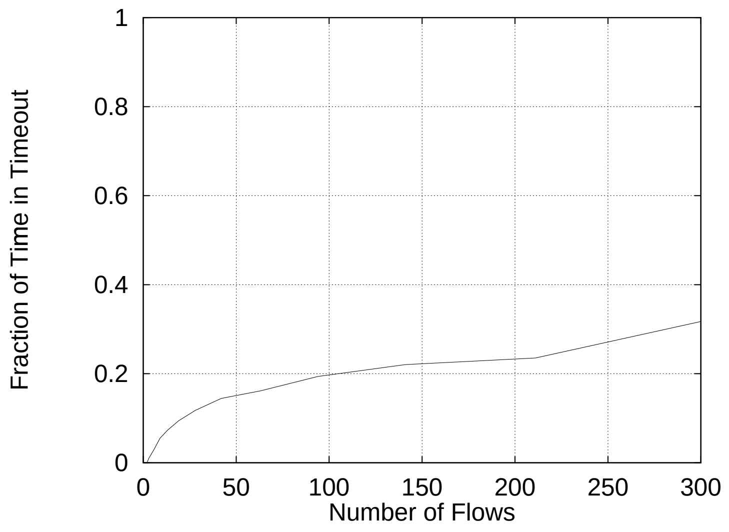


Figure 5.14: Fraction of time in timeout as a function of number of flows. $max_{th} = 500$, $max_p = 0.02$.

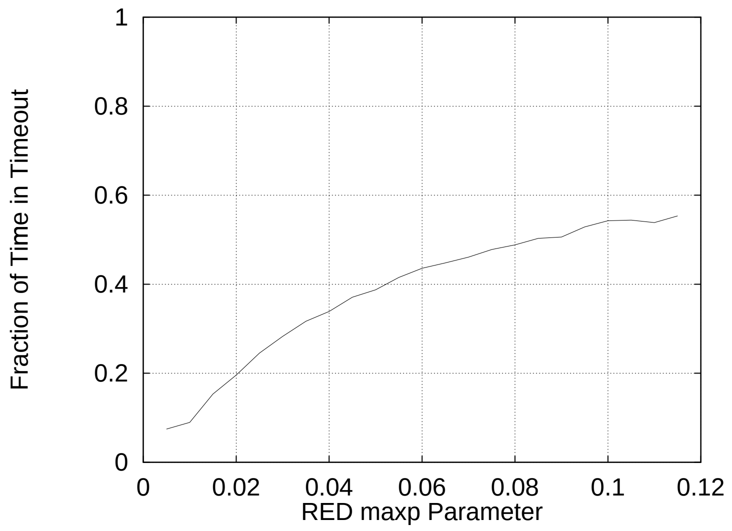


Figure 5.15: Fraction of time in timeout as a function of max_p. 100 flows, $max_{th} = 500$.

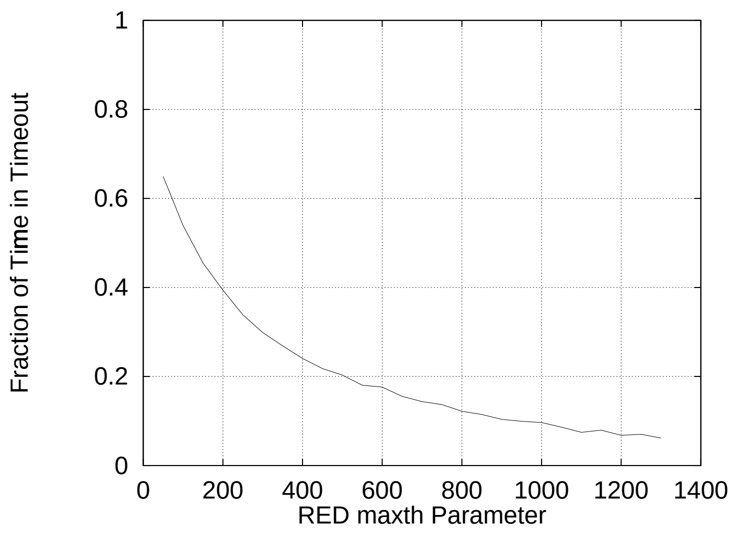


Figure 5.16: Fraction of time in timeout as a function of max_{th}. 100 flows, $max_p = 0.02$.

5.3.3 Improving the Predictions

The previous section noted a number of ways in which its predictions of RED performance were flawed. The predictions can be improved to account for packets stored on the wire and the fact that some TCPs are timing out. The cost, however, is that the resulting equations have to be solved numerically. This limits their intuitive appeal, but they still have use in verifying the simulations, in evaluating the explanation of the errors in the last section's predictions, and perhaps in configuring networks.

The first improvement is to make use of Equation 5.4 to predict the fraction of time spent in timeout:

$$o(l) \equiv l(1-(1-l)^{13})$$

$$O(l,r,w) \equiv \frac{1}{1+\frac{r}{w\cdot o(l)}}$$

This yields a version of Equation 5.5 which we'll call *pif*, for average packets in flight:

$$w(l) \equiv \frac{0.87}{\sqrt{l}}$$

$$pif(l,r) \equiv (1-O(l,r,w(l)))w(l)$$

For convenience we'll turn Equation 5.10 into the function *redq*:

$$redq(l, max_{th}, max_p) \equiv \frac{l\cdot max_{th}}{2max_p}$$

Observing that the number of packets kept in flight by the n TCPs must equal the queue length plus the packets stored on the link, we get this equality:

$$redq(l, max_{th}, max_p) + (pps \cdot r) = n \cdot pif(l, r) \tag{5.14}$$

pps is the bottleneck rate in packets per second, and r is the round trip time; these are 2170 and 0.01 respectively for the simulations on the previous section.

The P2 curves in Figures 5.11 through 5.13 come from numerical solutions of Equation 5.14 for l. These curves are all closer to the simulations than the P1 curves because they correct for timeouts and packets stored on the link.

5.3.4 Setting RED Parameters

The preceding sections show that RED's performance depends on the interaction between the number of flows and the RED parameter settings. Given a known flow count, a network administrator needs to know how to set the RED parameters to achieve a good tradeoff among utilization, delay, and TCP timeout probability.

To avoid excess timeouts an ideal router for TCP should allow somewhat more than four packets of buffering per flow. RED cannot do this in general, because it doesn't provide a linear relationship between flow count and average buffer size. It can, however, be tuned to operate well for a range of flow counts. For example, Figure 5.11 shows a router that works well for between 10 and 40 flows, providing 6 and 4 buffers per flow respectively. Outside that range it either buffers more packets than necessary or imposes an uncomfortably high loss rate.

Figure 5.13 implies that max_{th} can be used to tune a RED router for a target number of flows. For the 100 flows simulated, a max_{th} of 1000 yields about four packets of buffering per flow.

On the other hand, max_p has a narrow useful range. In the example situation, values below about 1% cause the queue to stay close to max_{th}, risking episodes of 100% loss. Values above about 2% cause TCP to spend a substantial amount of time in timeout.

These considerations suggest the following procedure for selecting RED parameters. First, measure the typical number n of simultaneously active TCP flows. Second, choose a maximum desirable loss rate l; this should probably be less than 3% to allow TCP a window of four packets. Third, choose a max_{th} somewhat larger than the number of packets that n TCPs will keep in flight with a loss rate of l:

$$max_{th} = 3/2 \cdot n \cdot pif(l, r)$$

At this point the choice of max_p is limited to a value which will keep the queue somewhat less than full:

$$max_p = \frac{3}{4} l$$

Few users of routers have free choice of max_{th}, since router buffer memory is often physically limited. In such cases the best one can do is pick a max_p

which will cause the queue to be somewhat less than full, given the known n and max_{th}. We can get a reasonable guess at this by equating Equation 5.13 to $\frac{3}{4}max_{th}$ and solving for max_p:

$$max_p = \frac{0.81n^2}{{max_{th}}^2}$$

5.3.5 RED Scaling Discussion

RED's main strength is that, given a known load, it can be tuned for high utilization and low queuing delay. It could still be improved:

- A given set of RED parameters work well only across a limited range of numbers of flows. This means that network administrators must manually tune RED parameters to get good performance.
- RED tempts network administrators to optimize for high utilization alone. For example, increasing max_p in response to higher load should increase utilization by decreasing TCP window synchronization. While some recommend this approach [17, 13], Figure 5.15 shows that its cost is is a high level of TCP timeouts.

One could imagine a manufacturer equipping a RED router with a large max_{th} in an effort to make the router's performance scale gracefully with large numbers of flows. For example, a 10 megabit router port should be able to handle 300 33-kilobit modem flows without any queuing at all. Since the Internet has been remarkably unforgiving about under-estimates of its growth rate, our manufacturer might want to ensure good behavior with up to an order of magnitude more than 300 flows. Support for 3000 flows implies a max_{th} of 15000 packets. The same 10-megabit port might also be used on a LAN with only a few flows. Equation 5.13 implies that with 20 flows, for example, the router's queue length would be about 500 packets. This is a quarter second of queuing delay, intolerably high for many interactive LAN users.

Note that the above example is not in the apparent spirit of RED parameter tuning [17]. In practice RED implementations seem to tend to the opposite extreme: small max_{th}, low queuing delay, and (presumably) many timeouts, but with a high max_p to keep utilization high. The point is that RED presents the network administrator with an uncomfortable choice between queuing delay and TCP timeouts.

This suggests a router buffering system that automatically adapts to the number of flows, with the simultaneous goals of limiting timeouts, limiting queue length, and maintaining high utilization. Chapter 6 proposes just such a system.

Chapter 6

FPQ: Supporting Large Router Queues

The implication of the existence of large numbers of flows and TCP's behavior with small windows is that routers should have large buffer memories. A reasonable target is five packets per flow for the maximum number of flows that a router might ever have to support. Assuming 14 kbit/second modem flows, this works out to one packet of buffering per 2800 bits/second of link bandwidth. The common current practice of providing one delay-bandwidth product of buffering works out to one packet per 46080 bits/second of link bandwidth, assuming a round-trip propagation delay of 0.1 seconds and a packet size of 576 bytes. Provisioning buffers based on flow count would require 16 times as much memory as current practice. The result should be dramatically decreased loss rate, but the cost might be excessive queuing delay.

If the number of flows a router would have to handle were fixed and predictable at the factory, routers could be shipped with parameters set to achieve a reasonable tradeoff between loss rate and queuing delay. However, a router manufacturer must expect any particular model of router to be used in situations ranging from connecting LANs with a few dozen flows to ISP/ISP peering with tens of thousands of flows. A queuing configuration appropriate for a LAN will impose a high loss rate on an ISP/ISP connection; a good ISP/ISP queuing configuration will impose excessive queuing delay in a LAN. An ideal router would automatically adapt its queuing configuration to the load.

What should an adaptive queuing scheme look like? Ideally, it would

- provide five packets of buffering per active flow,
- preserve the simplicity of FIFO queuing,
- preserve RED's resistance to phase effects and TCP window synchronization, and
- require no manual tuning.

This problem divides naturally into three parts. First, a mechanism to count active flows. Second, a choice of target queue length and drop rate based on the flow count. Third, a mechanism to enforce the targets on a FIFO queue while avoiding the global synchronization and phase effects mentioned in Section 2.2.1. The rest of this chapter describes a queue management system that fits these requirements. For convenience we will call it FPQ, for Flow-Proportional Queuing.

6.1 Bit-Vector Flow Counting

One way to count TCP connections might be to have the router participate in an explicit connection setup/teardown protocol. This could be a separate protocol, as in ATM [10] networks, or an additional use of TCP's existing setup (SYN) and teardown (FIN) protocol. One reason this would not work well is that a flow's path through the network may change: the path taken by the setup packets may not the same as that taken by the data packets. Another problem is that many connections are idle some of the time, and only active connections are of interest here.

A better counting scheme would observe all packets rather than just setup and teardown packets, count a connection active if it had sent packets recently, and not count it if idle. The router must remember if it has already counted a flow as active, so that if more packets arrive on that flow, the router doesn't count the flow again. How much state does this require? One option is to remember the identity of each currently active flow, perhaps using IP addresses and port numbers. The router could hash the identity of each packet to index into a table of the identities of flows already included in the count.

Even the above mechanism is more complex than required. The counting mechanism does not need to be precise, and does not need to support any operation other than deciding if a flow has already been counted. For example, it does not need to be able to recover the actual identities of counted flows. Thus it should be sufficient to store just one bit of state per flow.

A router can count flows with just one bit of state per flow as follows. Create a vector of v_{max} bits called v. The index for v is the hash of a packet identifier. Maintain the count of bits set in v in a variable c. When a packet arrives and the bit in v for its identifier isn't set, set it and increment c. Clear bits out of the table incrementally, so that every bit is cleared, and c decremented if the bit was set, after the passage of t_{clear} seconds.

This scheme under-counts due to hash collisions. If packets from f distinct flows have caused bits in v to be set, and the hash function is uniform, the probability of a bit in v being zero is

$$\left(\frac{v_{max}-1}{v_{max}}\right)^f$$

Thus the expected count c of one bits in v is

$$c = v_{max}\left(1 - \left(\frac{v_{max} - 1}{v_{max}}\right)^f\right)$$

Solving for f yields a formula that converts a count of one bits into an expected number of flows:

$$f = \frac{\ln\left(1 - \frac{c}{v_{max}}\right)}{\ln\left(\frac{v_{max}-1}{v_{max}}\right)} \tag{6.1}$$

The router can apply Equation 6.1 to the count of bits at the expense of some CPU time or clever approximating. It turns out that f and c are within 10% of each other as long as v_{max} is at least five times as large as f, so the computation could be eliminated at some expense in memory.

Figure 6.1 contains pseudo-code for this flow counting algorithm. The input is a stream of data packets, and the output is an estimate f of the current number of active flows.

The algorithm has only two tunable parameters. v_{max} should be set to the maximum number of flows expected at the router. The only penalty to setting it too high is the expense of incrementally clearing it. Setting v_{max} too low gradually reduces the accuracy of f.

t_{clear} controls how fast the algorithm forgets about old flows. It should be higher than the maximum round trip time in the network including queuing delay. Setting t_{clear} too low will cause the algorithm to be unstable. Setting t_{clear} too high will increase the number of idle or terminated connections counted in f, which will allow too much buffer space to be used in the router.

The hash function $h(p)$ should be uniform to ensure that Equation 6.1 fully corrects for hash collisions.

6.2 Choosing the Target Queue Length

Once equipped with the current count of flows, f, a FPQ router must choose a target queue length q, drop rate l, and per-flow congestion window size w. This is really one decision: q is $w \cdot f$, and w and l are related by Equation 5.2.

With only one TCP flow, a router can only achieve high throughput by allowing the flow to buffer an entire delay-bandwidth product of packets. This causes the TCP's window to be two delay-bandwidth products just before the router drops a packet. Thus after the drop the TCP's window will be just large enough to use the entire link bandwidth.

More generally, the number of buffers that a small number f of flows requires for full utilization depends on the delay-bandwidth product r (measured in packets) as follows:

$$q = \frac{r}{2f - 1} \tag{6.2}$$

```
Initialization:
    $v(0..v_{max} - 1) \leftarrow 0$
    $c \leftarrow 0$
    $f \leftarrow 0$
    $t_{last} \leftarrow$ current time
as each packet $p$ arrives
    $h \leftarrow h(p)$
    if $v(h) = 0$
        $v(h) \leftarrow 1$
        $c \leftarrow c + 1$
    $t \leftarrow$ current time
    $n_{clear} \leftarrow v_{max} \frac{t - t_{last}}{t_{clear}}$
    if $n_{clear} > 0$
        $t_{last} \leftarrow t$
        for $i \leftarrow 0$ to $n_{clear} - 1$
            $r \leftarrow random(0..v_{max} - 1)$
            if $v(r) = 1$
                $v(r) = 0$
                $c \leftarrow c - 1$
    $f \leftarrow$ Equation 6.1
```

Variables:

$v(i)$	Vector of v_{max} bits. $v(i)$ indicates if a packet from a flow with hash i has arrived in the last t_{clear} seconds.
c	Count of one bits in v.
f	Current estimated flow count.
t_{last}	Time at which bits in v were last cleared.
r	Randomly selected index of a bit to clear in v.

Constants:

v_{max}	Size of v in bits; should be larger than the number of expected flows.
t_{clear}	Interval in seconds over which to clear all of v.
$h(p)$	Hashes a packet's flow identifying fields to a value between 0 and v_{max}.

Figure 6.1: Flow-counting pseudo-code.

$$q = \max\left(\frac{r}{2f-1}, 5f\right)$$

$$l = \min\left(\left(\frac{0.87}{\frac{q+r}{f}+1}\right)^2, 0.016\right)$$

Figure 6.2: FPQ target queue length and loss rate as a function of number f of flows and round trip time r in packets.

The crucial observation for this is that RED spreads out the window decreases over time, so that only one TCP cuts its window at a time. For full utilization, we want the buffer space to equal the amount that one TCP will cut its window. The average window size is $\frac{r+q}{f}$, so each TCP cuts its window by $\frac{r+q}{2f}$ packets at a time. Thus we want

$$q = \frac{r+q}{2f}$$

Solving for q yields Equation 6.2. The corresponding loss rate l can be derived from Equation 5.2:

$$l = \left(\frac{0.87}{\frac{q+r}{f}+1}\right)^2$$

With more than a few flows the main considerations in choosing w, q, and l are minimizing queuing delay and preventing TCP from falling into timeout. The target average window size should be somewhat larger than four so that TCP's fast retransmit works. We use 5, a choice justified in Section 7.1. Figure 5.1 indicates that a loss rate of about 1.6% will produce an average window of 5. The router's target queue size should be $5f$ and its target loss rate should be 1.6%.

A loss rate of 1.6% is neither wonderfully low nor disastrously high. Figure 5.2 indicates that well under one percent of packets will incur timeouts with a loss rate of 1.6%, so the typical web transfer will complete without interruption.

Figure 6.2 summarizes the target q and l for both cases. Figure 6.3 shows the target queue length as a function of f for a network with a delay bandwidth product of 217 packets.

6.3 Achieving the Target Queue Length

Once the flow counting mechanism determines a target queue length q and drop rate l, the router must enforce them. As described above, $q = w \cdot f$, where f is the current flow count and w is the desired per-TCP window size.

One possibility is a dropping scheme with a direct queue limit, such as drop-tail or random drop. Even without dropping or window size changes, however,

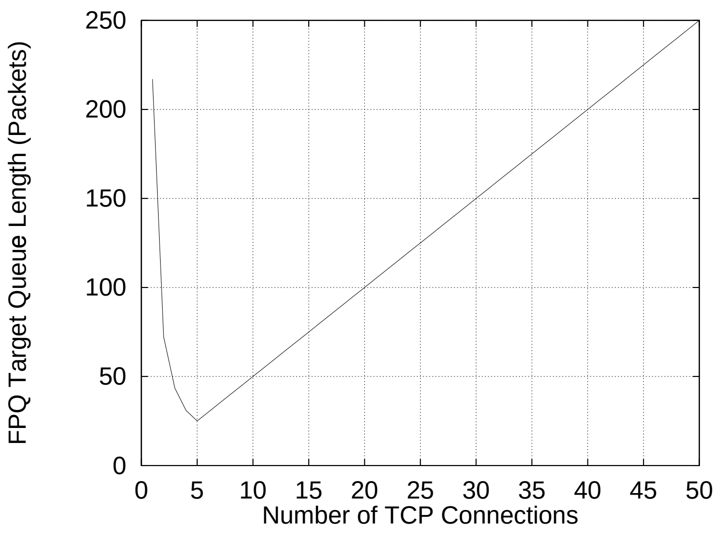


Figure 6.3: FPQ target queue length as a function of number of active flows. RTT of 217 packets.

queue lengths in a TCP network tend to oscillate. A fixed queue limit would clip the peaks from queue length cycles. These drops themselves contribute to oscillation by forcing the affected TCPs to reduce their windows. Further, they force the actual average queue length to be shorter than the maximum by an amount difficult to predict. Thus a simple maximum queue length does not work well to achieve a target average queue length.

A better scheme would use average queue length to decide when to drop. However, it turns out that the idea of a maximum queue length is not appropriate, even with averaging. The desired effect is that all the active TCPs use a particular average congestion window size. The best way to do that is to impose a steady loss rate, as in Equation 5.1. Dropping only when the queue exceeds some limit will cause the loss rate to oscillate. With some cleverness the drop rate could be made to vary around the desired average, but the maximum queue length mechanism seems to have no advantage to offset its complex behavior.

The ideal scheme would drop packets based on average queue length q_{avg} as follows. Recall that the previous section described how to choose a target queue length q and loss rate l based on a target window size w and counted number of flows f. When q_{avg} equals q, impose the loss rate l. This should cause each TCP to use an average window of w. Vary the loss rate around l in proportion to the difference between q_{avg} and q. The combination of averaging and a predictive model for drop rate should keep the queue length steady at the desired target.

This dropping mechanism fits into a RED framework: set RED's max_p to $l/2$, max_{th} to q, and min_{th} to a small value such as five. In this context RED's

$$max_p = \frac{2l}{2} = l,\ max_{th} = 2q,\ min_{th} = 5$$

Figure 6.4: RED parameters as set by FPQ, based on f from Figure 6.1 and l and q from Figure 6.2.

policy of dropping 100% of packets when q_{avg} exceeds max_{th} isn't desirable. Two other issues might also cause trouble. First, the relationship between loss rate and window size isn't linear, so RED's linear queue length to drop rate function isn't quite what FPQ needs. Second, RED averages the queue length over some number of packets. This number probably needs to be longer than any possible queue length in order to provide enough damping, so it may need to be orders of magnitude larger than in existing RED configurations.

Proportionally higher values of max_{th} and max_p allow FPQ to use RED without interference from RED's 100% drop policy. Figure 6.4 shows the RED settings used by FPQ in the rest of this work.

6.4 Validation

This section presents simulation results validating FPQ's ability to maintain the desired queue size and drop rate under a range of loads. The simulation configuration is a 10 megabit bottleneck link. The network has a 100ms round trip propagation delay. The FPQ algorithm is as described in Figures 6.1, 6.2, and 6.4, with parameters $v_{max} = 5000$ and $t_{clear} = 4$ seconds. The goal is to support up to 1400 flows: a 10 megabit link has enough bandwidth to support 700 14.4 kilobit modem flows, and the extra factor of two is to handle inevitable overload.

6.4.1 Queue Length

Figure 6.5 shows how the router's queue length varies with the number of flows. Figure 6.6 displays packets buffered per flow, derived by dividing the queue length by the number of flows. FPQ comes quite close to achieving its target of five packets per flow across a wide range of numbers of flows. The queue length is slightly lower than FPQ's target because of packets stored in flight in the network links. A per-flow packet count that includes packets in flight, as in the left graph in Figure 6.6, comes closer to the target.

6.4.2 Drop Rate

Hand-in-hand with FPQ's low queuing delay is the loss rate it must impose to keep the queue short. The main problem with packet loss at the levels involved here is timeouts. Figure 6.7 shows that FPQ imposes well under one timeout per hundred packets.

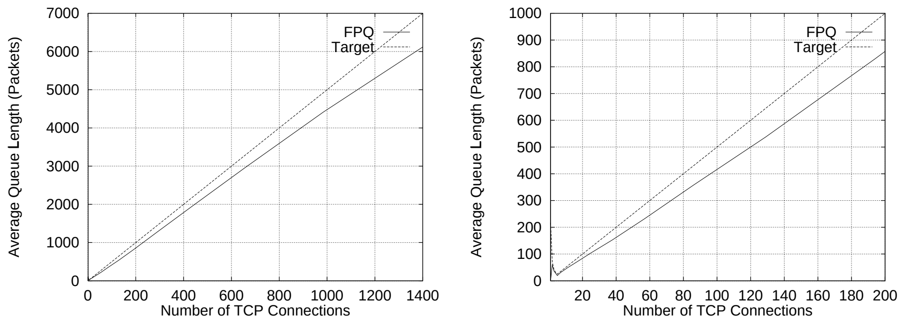


Figure 6.5: Average queue length (with detail) as a function of number of flows. RTT of 217 packets.

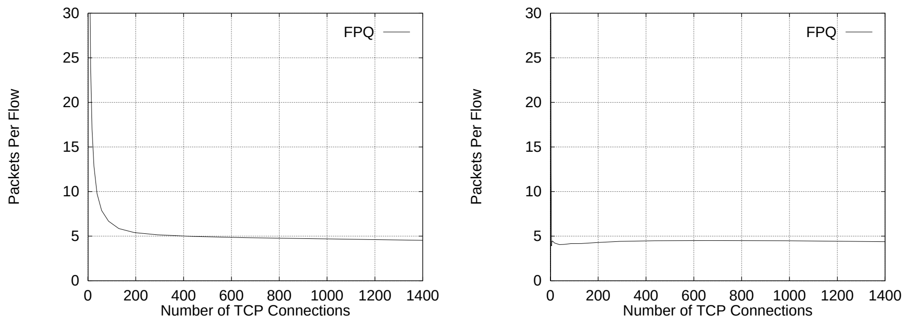


Figure 6.6: Average packets stored per flow as a function of the number of flows. The left graph includes both router buffering and the 217 packets stored on the links. The right graph includes only router buffering. RTT of 217 packets.

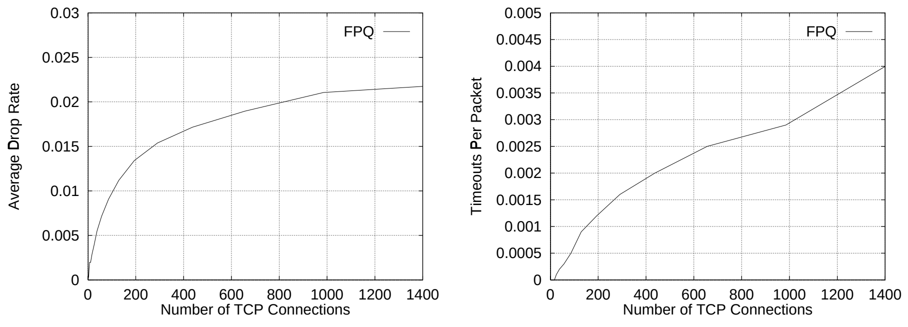


Figure 6.7: Drop rate and timeouts per packet as functions of the number of flows.

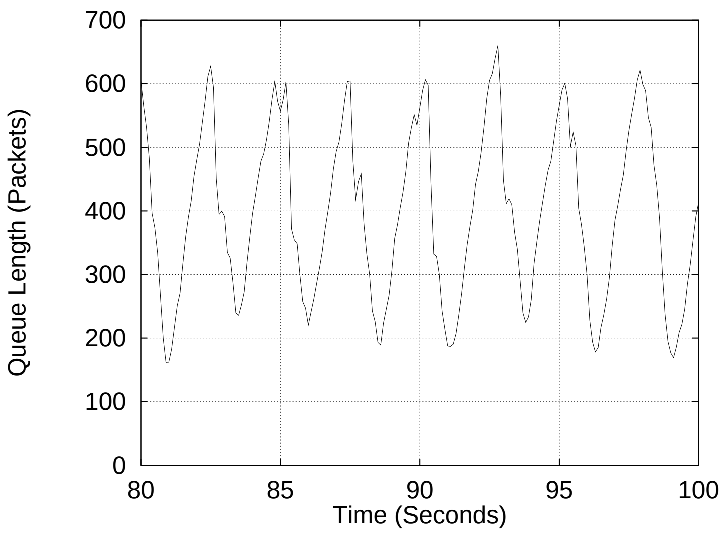


Figure 6.8: Queue length over time from a simulation with 300 flows and $t_{clear} = 0.05$ seconds.

6.5 Sensitivity to t_{clear}

A potential problem with FPQ is that t_{clear}, the flow counting interval, might be set too low. A low t_{clear} might cause FPQ to undercount flows in time-out and flows with low send rates. This section describes the consequences of misconfiguration and explores the range of t_{clear} values.

Figure 6.8 shows a primary symptom of excessively low values of t_{clear}: queue length oscillation. The graph plots queue length over time in a simulation with 300 flows and t_{clear} equal to 0.05 seconds. Packets from at most about 100 flows can arrive at the router in 0.05 seconds, so FPQ will never count more than 100 flows or allow more than about 500 packets of buffering. This is far short of the 1500 packets required by 300 flows. In this configuration, FPQ acts like a RED router with max_{th} set too low or max_p too high. That is, FPQ drops bursts of packets, causing the TCPs to synchronize their windows and create queue length oscillations. The oscillations seen in Figure 6.8 disappear once t_{clear} is greater than about 0.2 seconds.

Even if t_{clear} is large enough to avoid oscillation, it might be so small as to cause FPQ to substantially under-count flows, provide too few buffers, and thus force TCPs into timeout. Figure 6.9 presents the effect of t_{clear} on the accuracy of FPQ's flow counts. Each point is the number of flows counted by FPQ, averaged over the life of a simulation with 300 flows. Values of t_{clear} much below the round trip time (on the order of half a second) or the minimum retransmit timeout (one second) work badly: they cause FPQ to count substantially fewer than 300 flows. Figure 6.10 shows the packet discard rate and the per-packet

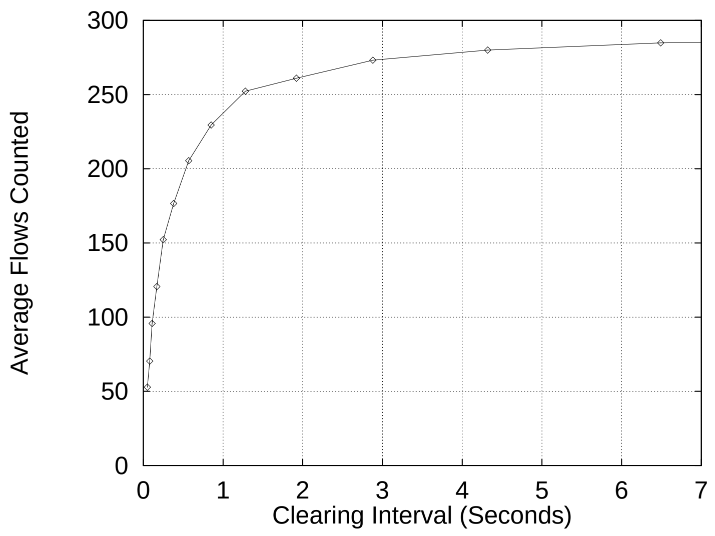


Figure 6.9: Effect of varying t_{clear} on FPQ's ability to count flows. The correct answer is 300 flows.

timeout probability from the same simulations. Increasing t_{clear} much beyond a few seconds has little effect.

It might be the case that the patterns of traffic in a real network could cause FPQ to over-count short flows or under-count low bandwidth flows. Flows that last much less than t_{clear} are a problem because FPQ will allocate buffer space for them for a whole t_{clear} interval, increasing queuing delay needlessly. Such flows are not the common case on WANs: most TCP flows, even most Web flows, last more than 10 seconds [50]. Low bandwidth flows that send less than one packet per t_{clear} interval are less of a problem. FPQ will under-count them, but they don't need as much buffer space as a greedy flow. In any event, Claffy et al. [9] show that changing the counting interval has a relatively small effect on the flow count in real Internet traffic.

Another troublesome possibility is a sustained rapid increase in the number of flows. This should not cause errors in the FPQ flow count, since FPQ counts new flows as soon as they send their first packet. But it might break implicit assumptions about the fraction of counted flows that are genuinely active. This kind of problem is probably best addressed by raising number of packets that FPQ allocates per flow; see Section 7.1 for more discussion.

The most important aspect of t_{clear} is that its value isn't critical as long as it is more than a few seconds. Perhaps it will need to be tuned for best performance, but it can also safely be left at some default value.

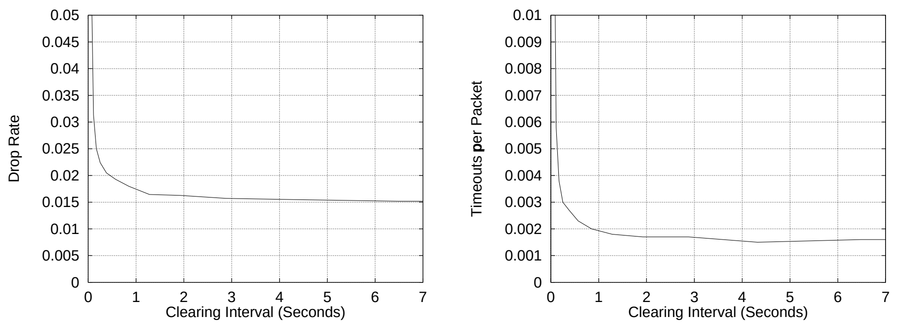


Figure 6.10: Discard rate and timeouts per packet as functions of FPQ's t_{clear} parameter. From simulations with 300 flows. The target discard rate is 0.016.

6.6 FPQ Discussion

The basic motivation for FPQ is that the best tradeoff between queuing delay and TCP performance depends on the number of active flows. Counting flows requires router state, but FPQ's bit-vector mechanism requires an attractively low quantity of per-flow state: one bit. FPQ uses RED to enforce its target queue length and drop rate because of RED's ability to avoid oscillation.

FPQ takes an unusual view of loss rate and buffering. FPQ isn't forced to drop packets as a reaction to overload or incipient overload. Instead it always drops packets at a steady rate calculated to make TCP use an appropriate window size. FPQ only varies the loss rate to the extent that its model of TCP's reaction to loss is not correct, or at the margins of extremely high or low load.

Similarly, a FPQ router expects a persistently long queue: just long enough to let each TCP flow avoid timeouts, and thus as short and low-delay as is reasonable. Since FPQ causes the queuing delay to vary in proportion to the number of flows, and TCP sends at a rate inversely proportional to the queuing delay, FPQ effectively uses the queuing delay to force each TCP to send at its fair-share bandwidth. This approach differs from dropping policies aimed at keeping the queue no longer than is required for high utilization.

While FPQ caters primarily to networks with very large numbers of flows, it should also reduce delay on high-speed links with few flows. Such links are often provisioned with a delay bandwidth product of buffering so that a single flow can use the entire link. Equation 6.2 shows that this is more buffering than required when there is more than one flow. By combining flow counting with RED's de-synchronization, FPQ can cut the queuing delay substantially below one round trip time without decreasing utilization.

Chapter 7

Delay Analysis

The main performance goal of FPQ, along with TCP, is to give users fair and efficient access to the network. For networks like the Internet that are dominated by short interactive transfers, a reasonable performance metric is the total delay incurred by a TCP transferring a short file. The optimum delay is easy to calculate and compare against, the delay distribution captures fairness, and users experience delay more directly than metrics such as bandwidth.

This chapter compares FPQ's delay performance against that of the drop approach; that is, it compares the queuing delays caused by FPQ with the time-out delays caused by the drop approach. FPQ's average performance is shown to be the same as that of the drop approach. FPQ, however, produces substantially less variation in delay than the drop approach. In other words, FPQ improves the fairness and predictability of network service. This improvement is the central point of FPQ.

As the basis for comparison is the delay incurred by finite-length transfers, we need a model for TCP transfer lengths. Since this chapter compares FPQ against existing buffering practice, rather than against a particular network scenario, the model need be no more elaborate than required to effect the comparison. Measurements on the Internet [50] suggest that the average TCP transfer length is about 20 packets. The fact that transfer lengths are distributed around this average will turn out to matter; we use an exponential distribution to approximate the observation [6] that most transfers are shorter than average but most bytes come from longer-than-average transfers.

Some of this chapter's sections compare FPQ against a traditional router configuration: RED buffering with one delay-bandwidth product of buffer memory. This comparison does not reflect on the RED algorithm, but on typical choices for RED parameters. In fact, direct comparison of RED and FPQ may not be meaningful, since RED's performance depends on parameter settings. Published work on RED has explicitly left proper parameter settings for future work, and this thesis is part of that future. Such RED guidelines as exist [19, 17, 51] implicitly assume small numbers of flows, and this thesis is aimed at reconsidering that assumption. Thus the RED results in this chapter should

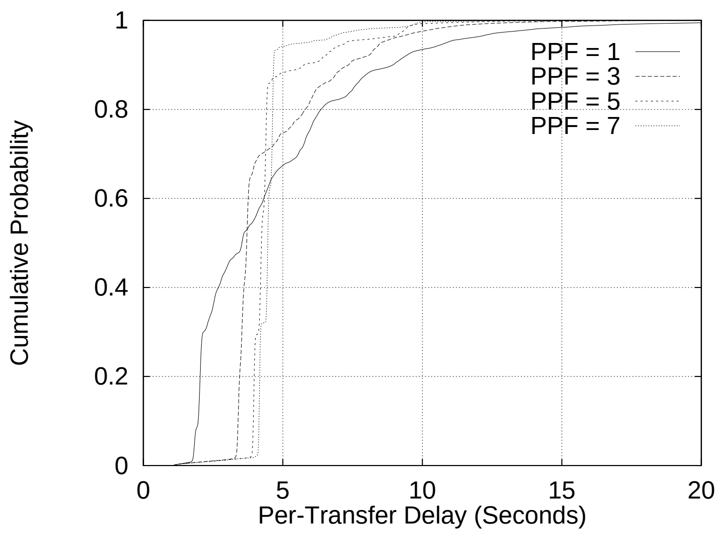


Figure 7.1: Cumulative per-transfer delay distribution, 500 flows performing 20-packet transfers. Shows effect on fairness of different choices of FPQ packets per flow. The goal is for every transfer to take 4.6 seconds.

not be taken as reflecting on RED so much as on the way in which it is used. To emphasize this point, we refer to the RED configuration used in many of this chapter's simulations as RED217, after the fact that its max_{th} is 217 packets.

7.1 Packets-Per-Flow Parameter

FPQ's main tunable parameter is the number of packets per flow (PPF). A low value will provoke TCP timeouts, causing users unfair variations in delay. A high value will encourage unnecessary queuing delay. Varying PPF allows one to choose a tradeoff between delay and fairness.

We use simulation to find the delay involved in transferring a short file with different choices of packet buffers per flow. Each TCP connection transfers a file of 20 576-byte packets, waits for the acknowledgment for the last packet, resets the TCP state to that of a new connection, and starts another 20-packet transfer. 500 such connections share an otherwise standard simulation environment. Figure 7.1 shows the cumulative distribution of per-transfer delays for various different choices of FPQ packets per flow.

The optimum result would be for each transfer to take 4.6 seconds, the amount of time required to send 20 576-byte packets at one 500th of the 10 megabit link bandwidth. The transfers actually segregate into two types: transfers with no timeouts, and transfers with one or more timeouts. The transfers with no timeouts have lower than expected delay because they use the band-

PPF	Loss Rate	Avg Queue	Median Delay	Avg Delay	Std Dev Delay
1	7.3%	270	3.5	4.6	3.5
3	2.4%	890	3.7	4.6	1.9
5	0.9%	1120	4.2	4.6	1.3
7	0.4%	1210	4.5	4.6	0.9

Figure 7.2: Effects of varying FPQ packets per flow. 500 flows and 20-packet transfers. Delays are in seconds.

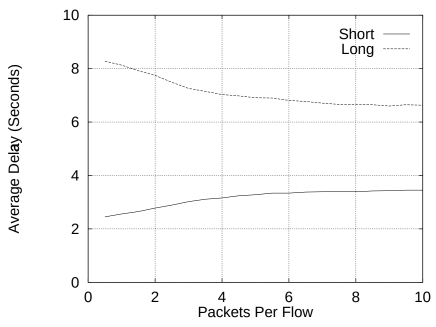
width left idle by the transfers in timeout. For example, Figure 7.1 shows that with 3 packets per flow, about 65% of transfers complete without a timeout. Increasing the number of packets per flow increases the proportion of transfers that complete without timeouts, and thus makes the system fairer.

Figure 7.2 shows statistics from the same simulations as Figure 7.1. Changing PPF does not affect average transfer delay, since this is a function only of link bandwidth and transfer size. Increased PPF increases the median delay because a higher fraction of the transfers actively compete for bandwidth rather than pause in time-out. This increase cannot be considered a defect because it reflects a fairer allocation of an unchanged average. On the other hand, increased PPF decreases the variation in transfer delay by decreasing the incidence of timeouts. This decrease is the main point of FPQ.

In the preceding discussion, the increase in queuing delay caused by increased PPF has no effect on average transfer delay because it merely shifts queued packets from the end systems to the router. This apparent lack of negative effects from increasing PPF is misleading. When transfers have different lengths, increased queuing delay hurts shorter transfers and helps longer transfers. We can see this effect by simulating transfers with exponentially distributed lengths averaging 20 packets, rather than lengths of exactly 20. Figure 7.3 shows the average and standard deviation of transfer delay for transfers of fewer than 20 packets separately from those of 20 or more packets. Each point's Y value is a delay statistic (in seconds) from a simulation with PPF indicated by the X value. Transfers that are too short to keep PPF packets in flight have delays dominated by queuing time, and so experience higher delay when PPF increases. Longer transfers are more sensitive to timeouts, and so experience lower delay when PPF increases. Both kinds experience less variation with increased PPF.

The reason why a PPF of about 5 works well is that TCP's fast retransmit mechanism depends on keeping at least four packets in flight. Variants of TCP that can recover with smaller windows [33] would require correspondingly smaller PPFs.

PPF may have to be adjusted based on experience of what fraction of flows have packets in flight at any given time. The value 5 suggested above is conservative: it assumes that all flows are greedy and need buffering. If significant numbers of flows are low bandwidth or very short, a lower PPF could safely be used. On the other hand, PPF must include some headroom for unexpected



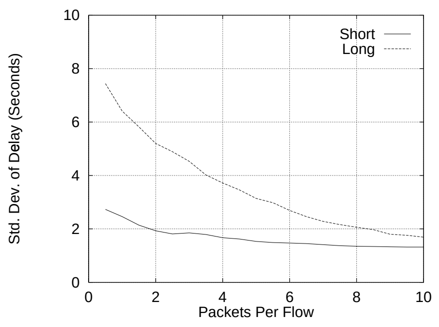


Figure 7.3: Per-transfer delay average and standard deviation for short and long transfers. 500 flows performing transfers with exponential distribution averaging 20 packets. Increased PPF improves average delay for long flows but hurts average delay for short flows.

increases in average flow bandwidth or length.

7.2 Average Delay Comparison

Before comparing the fairness of FPQ and the drop approach, we need to make sure FPQ provides the same average performance as the drop approach. FPQ could only fail in this by reducing the total useful throughput in the network, thus making the average transfer experience higher total delay. This is mostly likely to happen because of a high loss rate. It might also happen in conjunction with a number of flows small enough that their window sizes don't sum to the network's delay-bandwidth product. Similarly it might happen with transfers short enough to limit the per-flow window size.

To explore these possibilities we simulate a network with transfers that average 20 packets, with exponential distribution. The average transfer takes six windows (assuming delayed ACK) and has an average window size of about 3 packets. With 576-byte packets and a 100ms round trip propagation delay, the average flow sends no faster than 140000 bits per second. Thus it takes at least 70 flows to saturate a 10 megabit link. Figure 7.4 shows the total throughput used in this configuration as the number of flows increases. Neither FPQ and RED217 prevent the TCP flows from using all the bandwidth they can. In particular, FPQ provides the same average throughput as RED217.

Average per-transfer delay is a function only of average transfer size, total useful throughput, and number of flows. Since FPQ and RED217 provide the same useful throughput, we should expect them to provide the same average per-transfer delay. They do: Figure 7.5 presents the average per-transfer delay as a function of number of flows for the same simulation configuration as the previous paragraph. FPQ and RED217 do not differ in this respect.

The mechanisms by which FPQ and RED217 produce delays are not the

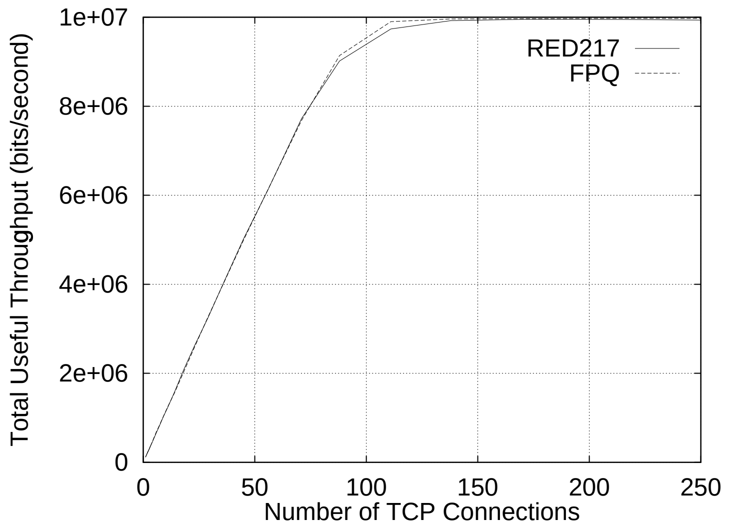


Figure 7.4: Useful throughput as a function of number of flows. Transfer lengths exponentially distributed with an average of 20 576-byte packets. RTT is 100ms.

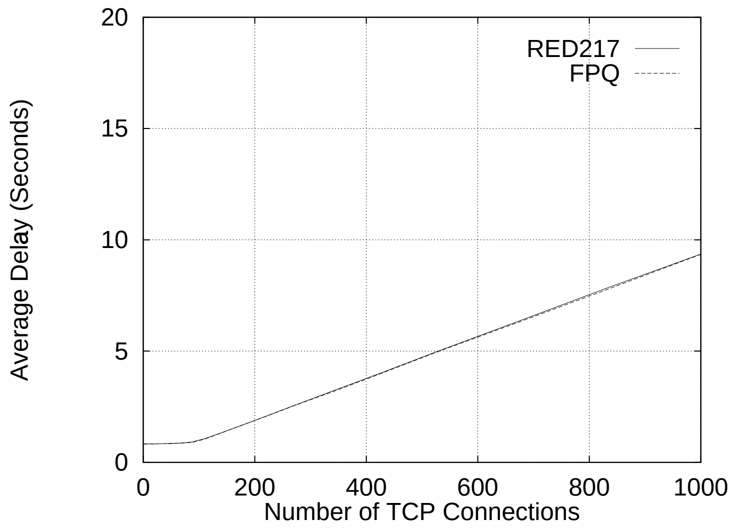


Figure 7.5: Average per-transfer delay as a function of number of flows.

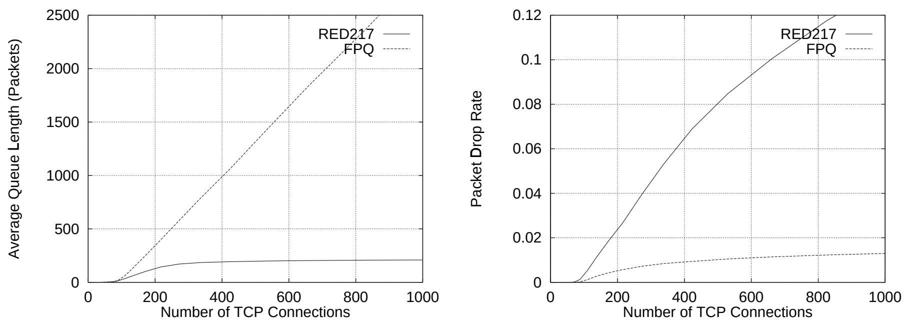


Figure 7.6: Average queue length and drop rate as functions of number of flows.

same. Figure 7.6 compares queue length and drop rate for FPQ and RED217. Clearly FPQ produces delay with queuing, while RED217 produces delay by loss-induced timeouts. For example, with 600 flows FPQ maintains a queuing delay of about 0.74 seconds, for a total round trip time of 0.84 seconds. The average 20-packet flow takes six windows, for a total per-transfer delay of slightly over 5 seconds, just as simulated in Figure 7.5.

In the RED217 simulation with 600 flows, the average 20-packet flow can expect just under two packet losses. Since the average window size with such short transfers is quite small, each packet loss is likely to incur a timeout. Each timeout turns out to last about 2 seconds, rather than the 1.5 seconds typical of long transfers, since the initial timeout is 3 seconds. Thus the typical transfer can expect to spend about 4 seconds in timeout. Because the timeouts reduce the window size to one and take the connection out of slow start, each transfer is likely to use an average window size of roughly two packets. With a round trip time totaling 0.2 seconds (0.1 of propagation delay, 0.1 of queuing), it takes about 2 seconds of RTTs to transfer 20 packets. The total is 6 seconds, again close to the actual simulated value in Figure 7.5.

In general, then, FPQ and RED217 produce similar average per-transfer delays, one with queuing delay, the other with timeouts. However, because they produce this delay with different mechanisms, the distribution of delays produced by the two systems is not the same.

7.3 Delay Fairness Among Transfer Sizes

As Section 7.1 explains, the long queues favored by FPQ tend to increase delays for short transfers and decrease them for long transfers. Figure 7.7 compares FPQ and RED217 delays for short and long transfers as the number of flows increases. The simulation scenario is the same as in the previous section. FPQ reacts to increased number of flows by increasing the queue length, which hurts shorter transfers. RED217 reacts by increasing the loss rate, which primarily

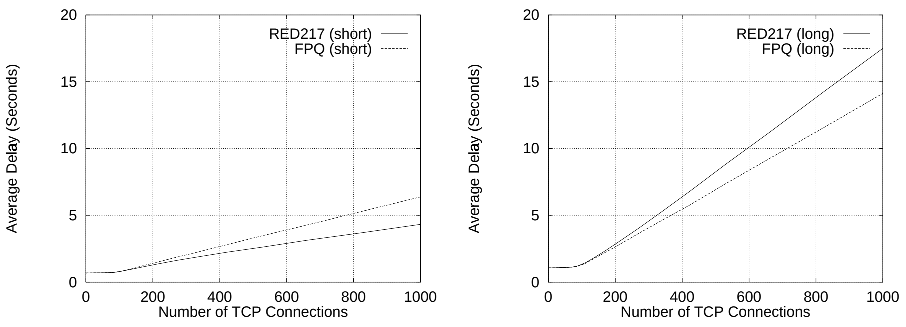


Figure 7.7: Average per-transfer delay as a function of number of flows. Transfer lengths exponentially distributed with average of 20 packets. Left graph shows just transfers of fewer than 20 packets, right graph shows just transfers with 20 or more packets.

hurts transfers long enough to have window sizes limited by loss.

Figure 7.8 shows the relationship between transfer size and transfer delay for a single simulation with 661 connections. Again, FPQ has a bias in favor of longer connections when compared to RED217.

This bias against short transfers arises from TCP itself; FPQ increases it but does not create it. Short transfers limit the window size. Small windows prevent full utilization with long RTTs and prevent TCP's fast retransmit mechanism from working.

7.4 Delay Fairness in General

This section attempts to characterize the improved fairness that FPQ provides in comparison to the drop approach. A fair system should cause identical users' network transfers to complete in the same amount of time. The extent to which identical users experience varied completion times is a basic measure of unfairness. Within this basic framework we present a number of comparisons of FPQ with RED217.

7.4.1 Cumulative Delay Distribution

Figure 7.9 presents the cumulative distribution of per-transfer delays. The simulated network configuration involved 661 connections, each making repeated transfers of exponentially distributed length and average size of 20 576-byte packets. Each graph includes only the transfers of a particular size, to make the delays comparable. The vertical line on each graph marked Perfect represents amount of time it should take to send a transfer in a perfectly fair system: the transfer size divided by 1/661 of 10000000 bits per second.

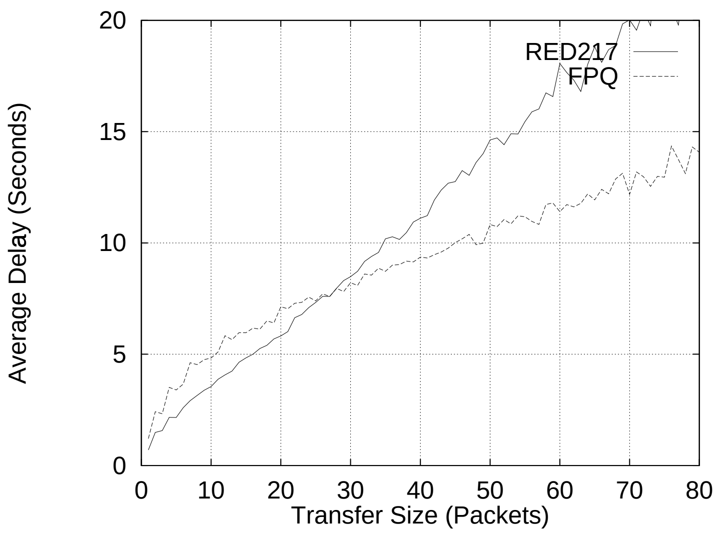


Figure 7.8: Average delay for transfers of each size. From a single simulation with 661 flows, transfer sizes exponentially distributed with an average of 20 packets.

Examine the graph for 20-packet transfers. With FPQ, about 80% of transfers take almost exactly the fair 5.4 seconds. These transfers are all delayed the same amount since they encounter the same queuing delay (averaging 1580 packets). The other 20% of transfers encounter one or more timeouts. This makes sense since the loss rate is about 1%, so the probability of a 20-packet transfer encountering one or more losses is $(1 - 0.99^{20}) = 0.18$.

The 20-packet RED217 transfers in Figure 7.9, however, mostly have either significantly above- or below-fair delays. The large majority of transfers suffer one or more timeout delays, since the loss rate is 9%. We expect $(1 - 0.91^{20}) =$ 15% of transfers to suffer no loss, and indeed somewhat more than this fraction complete in 2 seconds. This is roughly the minimum time in which TCP can send 20 packets with a round trip time of 0.2 seconds.

The 10- and 40-packet graphs have similar explanations. RED217's delay variation is lower with shorter than with longer transfers, because shorter transfers have a smaller probability of encountering a timeout. FPQ provides low variation even for the longer transfers. Note that FPQ's median delay is higher than perfect for short transfers and lower than perfect for long transfers, and that RED217 demonstrates an opposite tendency; this is the effect mentioned in Section 7.3.

FPQ, then, provides less delay variation than RED217. Much of the reason for this is that FPQ has an explicit mechanism (the queuing delay) that makes most transfers experience equal delays. RED217's reliance on dropping, however, tends to produce both transfers with no timeouts (and low delay)

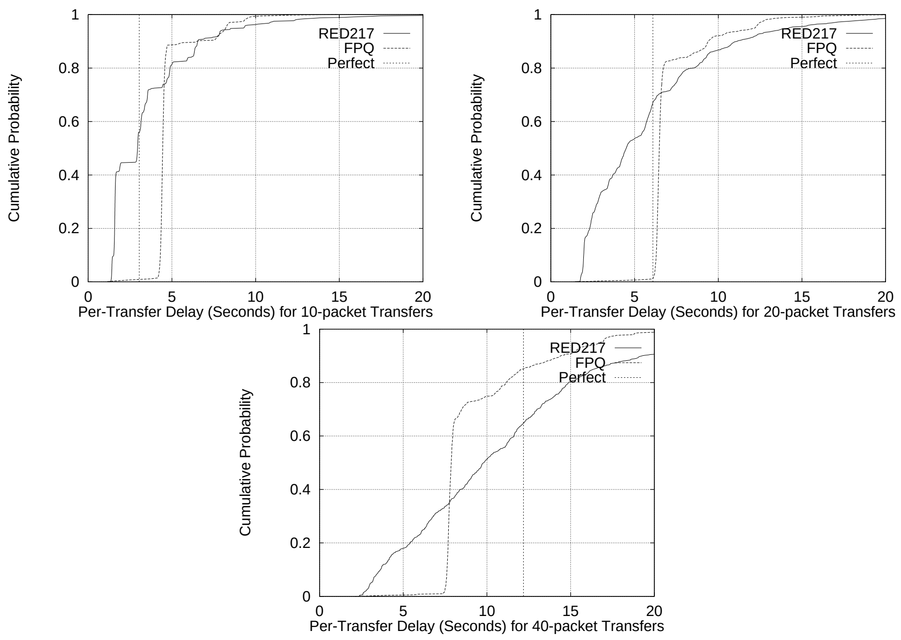


Figure 7.9: Cumulative per-transfer delay distributions from a simulation with 661 flows. Each flow performs repeated transfers with exponentially distributed length and average 20 packets. Each graph includes the transfers of just one size: 10, 20, or 40 packets. The FPQ delays are clustered more tightly than those of RED217.

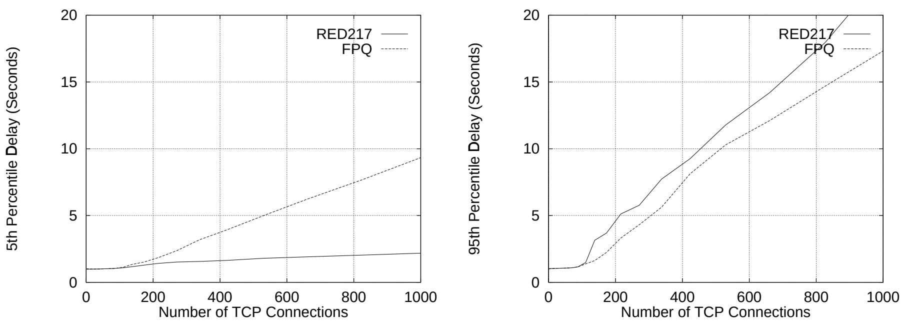


Figure 7.10: 5th and 95th percentiles of per-transfer delay as functions of the number of flows. Just for transfers of 20 packets.

and transfers with multiple timeouts (and high delay), with no particular bias towards the fair delay value.

7.4.2 Percentile Ratios

The previous section discussed fairness at just one level of load. In order to assess fairness across a range of loads we need a single fairness metric subject to comparison. We use the ratio of the 95th percentile of delay to the 5th percentile. This reflects the extent to which unlucky transfers get a worse deal than lucky ones. As above, we compare only transfers of the same length.

Figure 7.10 shows the 5th and 95th percentiles of delay as a function of number of competing flows. The left-most extremes of both graphs are flat because there are too few flows to use the entire link bandwidth, and thus neither queuing nor packet loss. The 5th percentile graph for RED217 slopes up very gradually because only transfers that experienced no timeout are lucky enough to be included. The delay for these transfers depends only on the fixed 0.2 second RTT, not on total number of flows. The 5th percentile graph for FPQ increases with number of flows because all transfers experience the same increasing queuing delay. RED217's and FPQ's 95th percentile delays both increase with the number of flows because in both cases the flows involved are those subject to timeouts. RED217's 95th percentile delay is higher than FPQ's because RED217 produces more timeouts. These graphs suggest that the behavior evident in Figure 7.9 occurs across a wide range of loads.

Dividing the two percentile graphs yields Figure 7.11. The combination of slightly higher 95th percentile delays and unfairly low 5th percentile delays gives RED217 a higher ratio for most loads. Worse, the difference between RED217's and FPQ's ratio increases as the load increases. This reflects RED217's tendency to let a fixed number of flows enjoy most of the bandwidth while the rest time out.

Figure 7.12 shows ratios for 10 and 40 packet flows. As expected, RED217

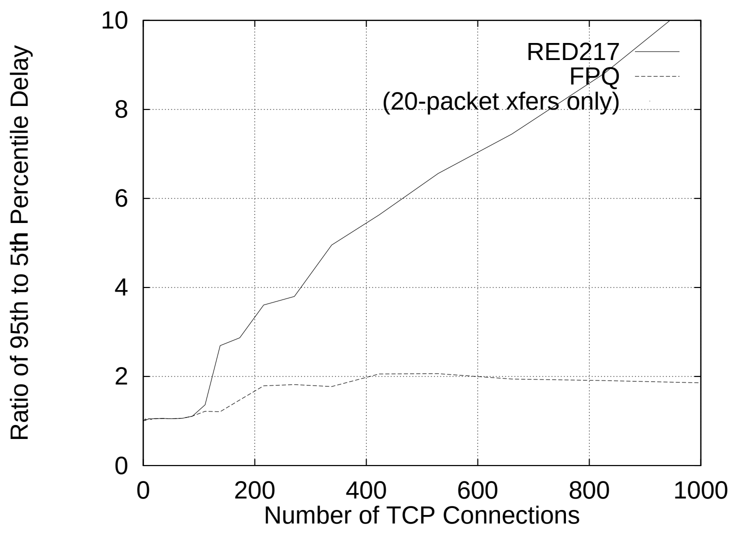
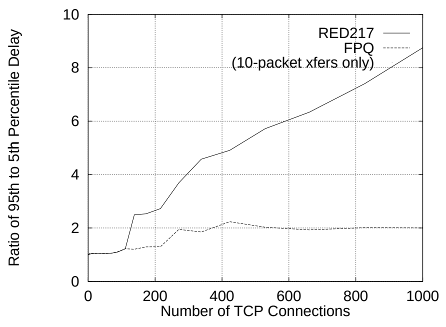


Figure 7.11: Ratio of 95th to 5th percentile of delay as a function of the number of flows. Just for transfers of 20 packets.

has a lower ratio for short flows than for long flows. FPQ's ratio is substantially lower in both cases.



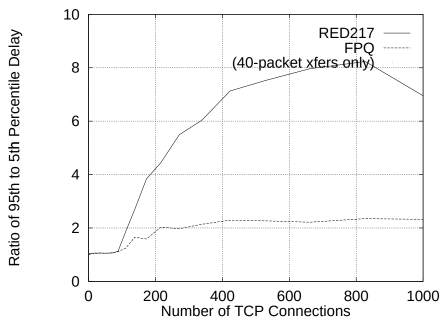


Figure 7.12: Ratio of 95th to 5th percentile of delay as a function of the number of flows. For transfers of just 10 and 40 packets.

Chapter 8

Conclusions

The central contribution of this thesis is the observation that congestion loss in busy TCP networks depends primarily on the number of active flows and the total storage in the network. Though the details vary, the general relationship is

$$l \propto \frac{n^2}{b^2} \tag{8.1}$$

where l is loss rate, n is the number of active flows, and b is the total storage. Total storage includes both router buffer memory and packets in flight on long links.

Holding b constant in Equation 8.1 approximates the behavior of existing IP routers, which have fixed-sized buffer memories. Chapter 5 analyzes some typical router configurations from this point of view, and derives guidelines for adjusting router buffering parameters. A constant b causes routers to signal congestion to senders by varying the loss rate. This approach has the desirable effect of capping the queuing delay at a low value. However, it produces high loss rates as the number of flows increases, causing long and unfair timeout delays.

Chapter 6 proposes a more scalable "Flow Proportional Queuing" (FPQ) system for TCP congestion control. FPQ provides congestion feedback by varying the queue length in proportion to the number of flows. In terms of Equation 8.1, FPQ varies b in proportion to n, keeping the loss rate l roughly constant. The network administrator can arrange for l to maintain a low value despite varying load. Since every user sees low loss and similar queuing delay, this approach should be fairer and more predictable than loss feedback under heavy load.

An FPQ router may cost more than a conventional router. First, FPQ assumes large amounts of buffer memory: a few packets for each of the maximum expected number of flows. Second, FPQ requires that a router count the number of flows. This thesis presents a simple flow-counting algorithm that takes a few instructions per packet and uses just one bit of state per flow. Other than these two costs, an FPQ router could be built with the same architecture that current routers use.

FPQ's intentional use of queuing delay may seem to invite higher user-visible latency, but in fact does not. Simulation of web-like traffic in Chapter 7 shows that FPQ provides the same average transfer latency as loss feedback; FPQ's queuing delay and loss feedback's timeouts have the same average effect. FPQ has an advantage in the delay distribution, however; FPQ distributes delays and bandwidth more fairly than loss feedback, and this advantage increases with the number of competing flows.

FPQ enjoys these advantages over conventional drop-tail and RED queuing:

- FPQ scales automatically with the number of flows, without the manual parameter tuning required by drop-tail and RED routers.
- FPQ allows routers to include very large buffer memories without risking unnecessarily high delay.
- FPQ eliminates most of the timeouts and unfairness caused by drop-tail and RED with large numbers of flows.

For these reasons the performance of TCP networks such as the Internet should benefit from the deployment of FPQ routers.

Bibliography

[1] Dimitri Bertsekas and Robert Gallager. *Data Networks.* Prentice Hall, Englewood Cliffs, New Jersey, 1987.

[2] Trevor Blackwell. *Applications of Randomness in System Performance Measurement.* PhD thesis, Harvard University, 1998.

[3] Jean-Chrysostome Bolot. Characterizing end-to-end packet delay and loss in the internet. In *Proceedings of ACM SIGCOMM*, 1993.

[4] 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, and L. Zhang. Rfc2309: Recommendations on queue management and congestion avoidance in the internet. Technical report, Internet Assigned Numbers Authority, Jon Postel, USC/ISI, 4676 Admiralty Way, Marina del Rey, DA 90292, 1998. http://info.internet.isi.edu/in-notes/rfc/files/rfc2309.txt.

[5] Hans-Werner Braun. What is fix-west? http://oceana.nlanr.net/NA/fixwest.html, 1997.

[6] Ramon Caceres, Peter Danzig, Sugih Jamin, and Danny Mitzel. Characteristics of wide-area tcp/ip conversations. In *Proceedings of ACM SIGCOMM*, 1991.

[7] Dah-Ming Chiu and Raj Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN Systems*, 17:1–14, 1989.

[8] Kim Claffy. Fix-west network traces from the national laboratory for applied network research. ftp://ftp.nlanr.net/Traces/FR+/960926, 1996.

[9] Kim Claffy, Hans-Werner Braun, and George Polyzos. A parameterizable methodology for internet traffic flow profiling. *IEEE Journal on Selected Areas in Communications*, 13(8), October 1995.

[10] ATM Forum Technical Committee. *ATM User-Network Interface Specification.* Prentice Hall Software, 1995.

[11] Charles Eldridge. Rate controls in standard transport layer protocols. *Computer Communications Review*, 22(3), July 1992.

[12] Ashok Erramilli, Onuttom Narayan, and Walter Willinger. Experimental queueing analysis with long-range dependent packet traffic. *IEEE/ACM Transactions on Networking*, 4(2):209–223, April 1996.

[13] Wuchang Feng, Dilip Kandlur, Debanjan Saha, and Kang Shin. Techniques for eliminating packet loss in congested tcp/ip networks. Technical report, University of Michigan, 1997. CSE-TR-349-97.

[14] Sally Floyd. Connections with multiple congested gateways in packet-switched networks part 1: One-way traffic. *Computer Communications Review*, 21(5), October 1991.

[15] Sally Floyd. Tcp and explicit congestion notification. *ACM Computer Communication Review*, 24(5), October 1994.

[16] Sally Floyd. Red: Discussions of setting parameters. http://www-nrg.ee.lbl.gov/floyd/REDparameters.txt, November 1997.

[17] Sally Floyd. Red: Discussions of setting parameters, 1997. http://www-nrg.ee.lbl.gov/floyd/REDparameters.txt.

[18] Sally Floyd and Van Jacobson. On traffic phase effects in packet-switched gateways. *Internetworking: Research and Experience*, 3(3), September 1992.

[19] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, August 1993.

[20] Henry Fowler and Will Leland. Local area network traffic characteristics, with implications for broadband network congestion management. *IEEE Journal on Selected Areas in Communications*, 9(7):1139–1145, September 1991.

[21] Mark Handley. An examination of mbone performance. Technical report, University of Southern California Information Sciences Institute, 1997. ISI/RR-97-450.

[22] Eman Hashem. Analysis of random drop for gateway congestion control. Master's thesis, Massachusetts Institute of Technology, 1989. MIT/LCS/TR-465.

[23] Adon Hwang. Observations of network traffic patterns at an end network: Harvard university. Master's thesis, Harvard College, 1998.

[24] Van Jacobson. Congestion avoidance and control. In *Proceedings of ACM SIGCOMM*, 1988.

[25] Van Jacobson. Notes on using red for queue management and congestion avoidance. http://www.nanog.org/mtg-9806/ppt/vj-nanog-red.pdf, June 1998.

[26] Van Jacobson, Craig Leres, and Steve McCanne. tcpdump. anonymous ftp at ftp.ee.lbl.gov.

[27] Raj Jain. A timeout-based congestion control scheme for window flow-controlled networks. *IEEE Journal on Selected Areas in Communications*, SAC-4(7):1162–1167, October 1986.

[28] Phil Karn and Craig Partridge. Improving round-trip time estimates in reliable transport protocols. In *Proceedings of ACM SIGCOMM*, 1987.

[29] H. T. Kung, Trevor Blackwell, and Alan Chapman. Credit-based flow control for atm networks: Credit update protocol, adaptive credit allocation, and statistical multiplexing. In *Proceedings of ACM SIGCOMM*, 1994.

[30] H. T. Kung and Koling Chang. Receiver-oriented adaptive buffer allocation in credit-based flow control for atm networks. In *Proceedings of IEEE Infocom*, 1995.

[31] H. T. Kung and Alan Chapman. The fcvc (flow controlled virtual channels) proposal for atm networks. In *Proceedings of the International Conference on Network Protocols*, 1993.

[32] T. Lakshman, A. Neidhardt, and T. Ott. The drop from front strategy in tcp and in tcp over atm. In *Proceedings of IEEE Infocom*, 1996.

[33] Dong Lin and H. T. Kung. Tcp fast recovery strategies: Analysis and improvements. In *Proceedings of IEEE Infocom*, 1998.

[34] Dong Lin and Robert Morris. Dynamics of random early detection. In *Proceedings of ACM SIGCOMM*, 1997.

[35] Allison Mankin. Random drop congestion control. In *Proceedings of ACM SIGCOMM*, 1990.

[36] Steve McCanne and Sally Floyd. Ns (network simulator). http://www-nrg.ee.lbl.gov/ns/, June 1998.

[37] Steve McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *Proceedings of the Winter USENIX Conference*, 1993.

[38] Jeffrey Mogul. Dec-pkt-4. http://ita.ee.lbl.gov/, March 1995.

[39] Robert Morris and Shieyuan Wang. Harvard network traces. http://www.eecs.harvard.edu/net-traces/, March 1997.

[40] John Nagle. Rfc896: Congestion control in ip/tcp internetworks. Technical report, Internet Assigned Numbers Authority, Jon Postel, USC/ISI, 4676 Admiralty Way, Marina del Rey, DA 90292, 1984. http://info.internet.isi.edu/in-notes/rfc/files/rfc896.txt.

[41] John Nagle. Rfc970: On packet switches with infinite storage. Technical report, Internet Assigned Numbers Authority, Jon Postel, USC/ISI, 4676 Admiralty Way, Marina del Rey, DA 90292, 1985. http://info.internet.isi.edu/in-notes/rfc/files/rfc970.txt.

[42] Peter Newman, Tom Lyon, and Greg Minshall. Flow labelled ip: A connectionless approach to atm. In *Proceedings of IEEE Infocom*, 1996.

[43] Vern Paxson. Automated packet trace analysis of tcp implementations. In *Proceedings of ACM SIGCOMM*, 1987.

[44] Vern Paxson. End-to-end internet packet dynamics. In *Proceedings of ACM SIGCOMM*, 1987.

[45] Scott Shenker, Lixia Zhang, and David Clark. Some observations on the dynamics of a congestion control algorithm. In *Proceedings of ACM SIGCOMM*, 1990.

[46] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols.* Addison-Wesley, 1994.

[47] W. Richard Stevens. Rfc2001: Tcp slow start, congestion avoidance, fast retransmit, and fast recovery algorithms. Technical report, Internet Assigned Numbers Authority, Jon Postel, USC/ISI, 4676 Admiralty Way, Marina del Rey, DA 90292, 1997. http://info.internet.isi.edu/in-notes/rfc/files/rfc2001.txt.

[48] Cisco Systems. Memory options for cisco 4000 series (product bulletin #419). http://www.cisco.com/warp/public/728/4000/419_pb.htm, 1996.

[49] Cisco Systems. Cisco 12000 series gigabit switch routers (gsrs) packet-over-sonet/sdh line card data sheet. http://www.cisco.com/warp/public/733/12000/gsont_ds.htm, 1998.

[50] Kevin Thompson, Gregory Miller, and Rick Wilder. Wide-area internet traffic patterns and characteristics. *IEEE Network*, November/December 1997.

[51] Curtis Villamizar and Cheng Song. High performance tcp in ansnet. *Computer Communications Review*, 24(5), October 1994.

[52] Maya Yajnik, Jim Kurose, and Don Towsley. Packet loss correlation in the mbone multicast network. In *IEEE Global Internet Conference*, 1996.

[53] Lixia Zhang, Scott Shenker, and David Clark. Observations on the dynamics of a congestion control algorithm: The effects of two-way traffic. In *Proceedings of ACM SIGCOMM*, 1991.