Dynamics of Random Early Detection*

Dong Lin and Robert Morris
Division of Engineering and Applied Sciences
Harvard University
Cambridge, MA 02138 USA

Abstract

In this paper we evaluate the effectiveness of Random Early Detection (RED) over traffic types categorized as non-adaptive, fragile and robust, according to their responses to congestion. We point out that RED allows unfair bandwidth sharing when a mixture of the three traffic types shares a link. This unfairness is caused by the fact that at any given time RED imposes the same loss rate on all flows, regardless of their bandwidths.

We propose Flow Random Early Drop (FRED), a modified version of RED. FRED uses per-active-flow accounting to impose on each flow a loss rate that depends on the flow's buffer use.

We show that FRED provides better protection than RED for adaptive (fragile and robust) flows. In addition, FRED is able to isolate non-adaptive greedy traffic more effectively. Finally, we present a "two-packet-buffer" gateway mechanism to support a large number of flows without incurring additional queueing delays inside the network. These improvements are demonstrated by simulations of TCP and UDP traffic.

FRED does not make any assumptions about queueing architecture; it will work with a FIFO gateway. FRED's per-active-flow accounting uses memory in proportion to the total number of buffers used: a FRED gateway maintains state only for flows for which it has packets buffered, not for all flows that traverse the gateway.

1. Introduction

Thousands of flows may traverse an Internet gateway at any given time. Ideally, each of these flows would send at exactly its fair share, and the gateway would not need to make any decisions. In practice, the load tends to fluctuate and the traffic sources tend to be greedy. Thus a gateway queuing policy must allow buffering of temporary excess load but provide negative feedback if the excess load persists. Such a policy must prevent high delay by limiting the queue size; it must avoid forcing queues to be too short, which can cause low utilization; and it must provide negative feedback fairly.

* This is a slightly revised version of the paper appeared in the Proceedings of SIGCOMM'97.

A number of powerful approaches to this problem are known. If a gateway keeps a separate queue for each flow, it can use Round-Robin Scheduling [9] to ensure fair shares of bandwidth at short time scales. A network of gateways can also use hop-by-hop flow control [15,23] to provide feedback to sources indicating how much data they are allowed to send. Such systems can guarantee that no data is ever lost due to congestion. While very effective, these mechanisms are rarely used in datagram networks. The main reason is that they typically require a gateway to keep a separate queue for every flow that traverses it, including short-lived and idle flows.

In order to reduce complexity and increase efficiency, IP gateways usually maintain as little state as possible: they keep no per-flow state, use a single first-in first-out (FIFO) packet queue shared by all flows, and do not use hop-by-hop flow control. They provide feedback to senders by discarding packets under overload. A gateway estimates load by observing the length of its queue; when the queue is too long, it drops incoming packets. Senders react to packet loss by slowing down; TCP, for example, decreases its window size when it detects packet loss. The simplest form of packet discard, called Drop Tail, discards arriving packets when the gateway's buffer space is exhausted.

Drop Tail gateways often distribute losses among connections arbitrarily [6]. Small differences in the round trip times of competing TCP connections, for instance, can cause large differences in the number of packets a Drop Tail gateway discards from the connections. Drop Tail gateways also tend to penalize bursty connections.

The gateway algorithms Early Random Drop (ERD) [10] and Random Early Detection (RED) [5] address Drop Tail's deficiencies. ERD and RED use randomization to ensure that all connections encounter the same loss rate. They also try to prevent congestion, rather than just reacting to it, by dropping packets before the gateway's buffers are completely exhausted. Neither ERD nor RED requires per-flow state; both should be easy to add to an existing IP gateway and have little impact on its packet forwarding efficiency.

The first half of this paper analyzes a number of imperfections in RED's performance. The main observation is that dropping packets from flows in proportion to their bandwidths does not always lead to fair bandwidth sharing. For example, if two TCP connections unevenly share one link, dropping one packet periodically from the low speed flow will almost certainly prevent it from claiming its fair share, even if the faster flow experiences more packet drops. Also, TCP connections with large window sizes are more tolerant of packet loss than those with small windows. It might take only one round trip time (RTT) for a large window connection to recover from multiple packet losses, whereas a timeout of one second or more may be needed for a flow with a tiny window to recover from a single packet loss.

We also note that RED is designed with adaptive flows in mind. TCP responds to small increases in loss rate with large decreases in its sending rate. A source that sends too fast regardless of loss rate would gain an unfair fraction of the bandwidth through a RED gateway.

We present Flow Random Early Drop (FRED), a modification to RED that improves fairness when different traffic types share a gateway. FRED is more effective in isolating ill-behaved flows, provides better protection for bursty and low speed flows, and is as fair as RED in handling identical robust flows such as large bulk-data transfers. FRED provides these benefits by keeping state for just those flows that have packets buffered in the gateway. The cost of this per-active-flow accounting is proportional to the buffer size and independent of the total number of flows, except to the extent that buffer use depends on the number of active flows. As with RED, FRED can easily be added to an existing FIFO-based gateway.

FRED can be extended to help support gateways with large numbers of buffers; this in turn may be helpful when supporting large numbers of active flows. Gateways with simple FIFO queuing and large numbers of buffers risk high queuing delay under low load and unfairness under high load. FRED, augmented with a “two-packet-buffer” mechanism, allows use of large numbers of buffers without these hazards.

In this paper, we use the terms flow and connection interchangeably to represent a flow identified by its source/ destination addresses, port numbers, and protocol id.

2. Traffic Categories

Internet traffic is a mixture of various kinds. Some sources use congestion control mechanisms such as TCP; others, such as constant-bit-rate (CBR) video, do not react to congestion. Although TCP traffic is the most dominant component overall, different versions and implementations of TCP react differently to congestion. Even when the implementations are identical, two TCP connections may behave differently if they have different congestion window sizes or round-trip times. In this paper we categorize traffic into the following three groups, according to the way the traffic handles congestion:

1. Non-adaptive: Connections of this sort take as much bandwidth as they require and do not slow down under congestion. Some audio and video applications fall into this category.
2. Robust: These connections always have data to send and take as much bandwidth as network congestion allows; they slow down when they detect congestion. Robust flows are capable of retransmitting lost packets quickly. They ramp up bandwidth usage quickly when they detect spare network capacity. Robust flows have enough packets buffered at the gateway that they can obtain at least the fair share of the bandwidth.
3. Fragile: These connections are also congestion aware, but they are either sensitive to packet losses or slower to adapt to more available bandwidth. They have less packets buffered than robust flows at the gateway. For example, most interactive terminal applications such as telnet do not have data to send most of the time, but would like to send a small number of clustered packets from time to time.

This paper assumes that non-adaptive traffic may need to share resources with ordinary best-effort data traffic. Non-adaptive connections do not stop sending packets under congestion. Unless they are subjected to very high loss rates, they compete unfairly with adaptive sources for buffer space and bandwidth.

There is no clear boundary between robust and fragile traffic. Each TCP connection starts with a congestion window of one packet during slow start. Some connections are able to expand the congestion window and become robust, whereas others are limited by data availability, receiver window constraints, or long propagation delays. Therefore, robust connections usually have larger windows or shorter RTTs than fragile connections.

When TCP connections with different numbers of buffered packets compete at the gateway, proportional packet dropping does not always guarantee fair bandwidth sharing. The reason for this is that TCP treats packet loss as congestion indication, regardless of the number of lost packets. A TCP connection that uses less than the fair share will reduce its congestion window on a single packet loss. In general, TCP connections with fewer buffers are at a disadvantage competing with other connections. Buffer discrepancy is caused by congestion window size and propagation delay variations. Connections with large windows and small RTTs consume the most buffers at the gateway.

3. Random Early Detection

A RED gateway drops incoming packets with a dynamically computed probability when the average number of packets queued exceeds a threshold called min_{th}. This probability increases with the average queue length and the number of packets accepted since the last time a packet was dropped. This approach controls the queue length more effectively than other existing algorithms, as demonstrated in [5]. RED is simple to implement because it drops only incoming packets and allows FIFO queuing.

RED’s goal is to drop packets from each flow in proportion to the amount of bandwidth the flow uses on the output link [5]. It does this by dropping each arriving packet with equal probability (provided the average queue size does not change significantly). Therefore, the connection with the largest input rate will have the biggest drop percentage among total dropped packets. Assume that the average queue size does not change for a short period δ, so RED drops incoming packets with a fixed probability p; also assume that $connection_i$’s current input rate is λ_i (or $\lambda_i \cdot \delta$ packets per δ). The percentage of dropped packets from $connection_i$ is:

$$\frac{\lambda_i p}{\sum \lambda_i p} = \frac{\lambda}{\sum \lambda_i} \tag{1}$$

For a FCFS service discipline, $connection_i$’s output rate is proportional to its buffer occupancy, which is determined by the percentage of accepted packets:

$$\frac{\lambda_i (1-p)}{\sum \lambda_i (1-p)} = \frac{\lambda}{\sum \lambda_i} \tag{2}$$

The above two equations imply that RED drops packets in proportion to each connection’s output usage under FCFS scheduling.

From each connection's point of view, however, the instantaneous packet loss rate during a short period δ is $\frac{\lambda p}{\lambda} = p$, which is independent of the bandwidth usage. If the congestion is persistent, which means the average queue length used by RED has a minimum value above min_{th}, the drop probability has a non-zero minimum and therefore causes a minimum loss rate for all connections, regardless of their bandwidth usage. This unbiased proportional dropping contributes to unfair link sharing in the following ways:

1. Although RED performs better than Drop Tail and Random Drop, it still has a bias against fragile connections. The fact that all connections see the same loss rate means that even a connection using much less than its fair share will experience packet loss. This can prevent a low-bandwidth TCP from ever reaching its fair share, since each loss may cause TCP to reduce its window size by one half.
2. Accepting a packet from one connection causes higher drop probability for future packets from other connections, even if the other connections consume less bandwidth. This causes temporary undesirable non-proportional dropping even among identical flows.
3. A non-adaptive connection can force RED to drop packets at a high rate from all connections. This contributes to RED's inability to provide a fair share to adaptive connections in the presence of aggressive users even if the congestion is not severe.

In the following sections, we investigate RED's behavior with a mixture of different traffic types.

3.1 Fragile vs. Robust

To demonstrate the limitations of RED with fragile connections, we simulated the configuration shown in Figure 1. A TCP connection with a round-trip time (RTT) of 36 milliseconds and a maximum window of 16 512-byte packets competes with four TCP connections with RTTs of 6ms and unlimited windows (up to 64 KB). The TCP implementation is a modified version of Reno that can recover from multiple packet losses in a window, much like the New-Reno TCP described in [11] and [4]. Each sender always has data to send. The packet payload size is 512 bytes. Delayed ACK is enabled. Further details of the simulator may be found in Section 11.1 and Section 11.2.

This experiment is similar to that found in [5], with a few differences. The four local connections have windows with no limit, rather than a limit of 12 1K-packets. Delayed ACK is enabled, which makes sources more likely to send back-to-back packets. We use 512 byte packets, as opposed to 1K bytes. TCP-Tahoe was used in [5].

The long RTT connection can send no more than 16*512 bytes each 0.036 seconds, or 4.05% of the bottleneck link capacity[1]. This is well below the 20% fair share in the presence of the four local connections. Therefore, if the network were fair, the bursty connection would be limited only by its window size and RTT, and would get 4.05% of the bottleneck link.

1. Each 512 byte IP packet is carried by 12 cells in our ATM simulator. The total ATM+AAL5 bandwidth overhead is 13.2%.

Our simulations vary the gateway buffer size from 16 to 56 packets. For each buffer size, we ran 10 simulations, each for 50 simulated seconds. Each TCP connection started at a random time within the first five seconds; we took measurements only from the second half of each simulation. Table 1 summarizes the results of the simulations. Each column shows averaged performance results for a specific gateway buffer size. The first body row shows the ratio of bandwidth achieved by the long RTT connection divided by the 4.05% maximum possible. Notice that the connection was only able to run at about 70% of the maximum speed. This is caused by the uniform loss rate experienced by all five connections. The second body row shows the ratio of the percentage of achieved bandwidth over the percentage of dropped packets for the long RTT connection. Notice that the ratios are close to 1.0, which implies that RED managed to keep the dropping distribution proportional to the bandwidth distribution. However, the output link capacity is not shared fairly. This phenomenon exists regardless of the buffer size used. In all simulations, the average queue size varied slightly about the RED minimum threshold (BS/4). Increasing buffer size improved the link utilization and raised the buffer occupancy for the four adaptive connections, but did not increase fairness. The performance numbers for a buffer size of 16 packets are substantially lower than for other cases, because RED degraded to tail-dropping when the average queue length reached max_{th} given the limited buffering capacity. The second half of Table 1 shows the average loss rates for the fragile and one of the robust connections. A connection's loss rate is calculated by dividing the number of dropped packets by the total number of arrivals. The loss rates agree with our claim that RED provides the same loss rate for all connections regardless of the resource usage. In addition, the long delay connection suffered more loss than others when the buffer sizes are small which made RED similar to drop-tail.

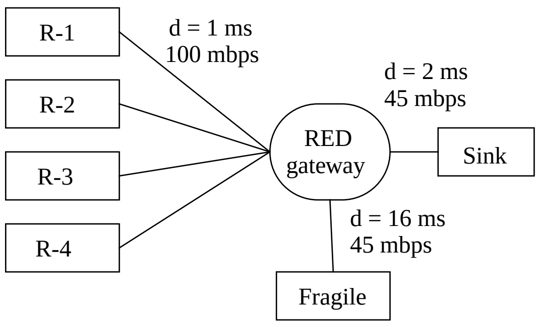

Figure 1. A long delay TCP competes with four local TCP connections through a RED gateway with buffer sizes BS = 16, 24, 32, 40, 48, 56 packets and min_{th} = BS/4, max_{th} = BS/2, w_q = 0.002, max_p = 0.02.

	BS=16	BS=24	BS=32	BS=40	BS=48	BS=56
BW/MAX	45%	67%	69%	70%	70%	71%
BW%/DROP%	0.456	1.05	1.03	0.982	0.911	0.940

Table 1. Performance of a long RTT TCP connection with limited windows under RED (Second row shows the ratio of bandwidth allocated over the 4.05% maximum possible. Third row shows ratio of percentage of achieved bandwidth over percentage of dropped packets. Third and Fourth rows show the loss rates of the fragile and one of the robust connections.)

	BS=16	BS=24	BS=32	BS=40	BS=48	BS=56
Fragile's Loss Rate	1.13%	0.55%	0.42%	0.40%	0.37%	0.32%
R-1's Loss Rate	0.49%	0.44%	0.40%	0.38%	0.35%	0.33%

Table 1. Performance of a long RTT TCP connection with limited windows under RED (Second row shows the ratio of bandwidth allocated over the 4.05% maximum possible. Third row shows ratio of percentage of achieved bandwidth over percentage of dropped packets. Third and Fourth rows show the loss rates of the fragile and one of the robust connections.)

During another experiment, we increased the maximum window allowed for the long delay TCP connection to 64 KB. Our simulations show that, in this configuration, the connection is able to sustain 39% of the link capacity without the presence of the four local TCP connections. This measurement agrees with our estimation (64KB/(45mbps*36ms)). Table 2 summarizes the performance results when the local traffic is present. Notice that the fragile connection was severely penalized by the competing traffic due to its long RTT value and failed to obtain a 20% share of the link rate. RED failed to protect this connection even though it successfully kept the drop distribution equal to the bandwidth allocation. Once again, the second half of the table shows unbiased loss rates among all connections.

The two experiments in this section demonstrate that proportional dropping does not guarantee fair bandwidth sharing. The combination of FIFO scheduling, RED's unbiased proportional dropping and TCP's binary feedback treatment of packet loss produce unfair bandwidth sharing at the bottleneck gateway.

	BS=16	BS=24	BS=32	BS=40	BS=48	BS=56
BW/Link Rate	2.7%	3.9%	4.3%	4.5%	4.6%	4.6%
BW%/DROP%	0.522	0.989	1.01	1.04	0.995	0.907
Fragile's Loss Rate	0.97%	0.51%	0.43%	0.46%	0.37%	0.39%
R-1's Loss Rate	0.51%	0.45%	0.40%	0.37%	0.35%	0.34%

Table 2. Performance of a long RTT TCP with unlimited windows under RED (Second row shows the percentage of bandwidth allocated over the link capacity. Third row shows ratio of percentage of allocated bandwidth over percentage of dropped packets.)

3.2 Between Symmetric Adaptive TCPs

We simulated two symmetric adaptive TCP connections in a simpler configuration shown in Figure 2. The RED gateway has 32 packet buffers and uses the following parameters: min_{th} = 8, max_{th} = 16, w_q = 0.002, max_p = 0.02. Figure 3 shows the congestion windows of the two connections taken from a segment of the simulation. The X-axis measures time in seconds; the Y-axis shows the congestion window size in packets. Each line segment of the curves represents a congestion avoidance phase. Each downward pulse between two line segments is a fast recovery phase. As shown in the figure, while connection 1 is increasing its window in congestion avoidance, connection 2 is suffering packet drops repeatedly, reducing its congestion window in half at each recovery. When the window size drops below four packets, connection 2 falls into a one-second retransmission time-out (RTO). This figure demonstrates that RED can accidentally pick the same connection from which to drop packets for a short period of time, causing temporary non-uniform dropping among identical flows.

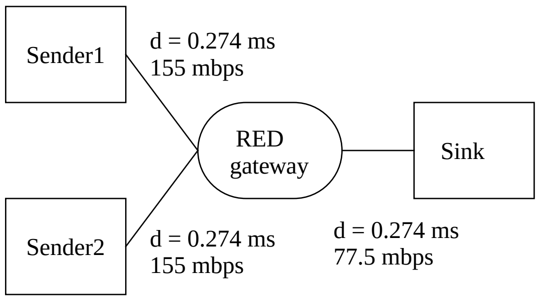

Figure 2. Two identical TCP connections compete over a RED gateway with buffer size 32 packets and (min_{th} = 8, max_{th} = 16, w_q = 0.002, max_p = 0.02).

Figure 4 shows the number of packets queued at the gateway for each connection during the same period. Notice that connection 2 has relatively few packets queued before it falls into the RTO at time 0.38 second. This figure demonstrates that RED may drop a packet with non-zero probability even if the connection has no packets queued and other connections are consuming a larger portion of the buffers and bandwidth.

We simulated the above configuration multiple times. This phenomenon is found in a small percentage of the traces. But the probability is large enough such that we can always find such a case within 20 to 50 simulations.

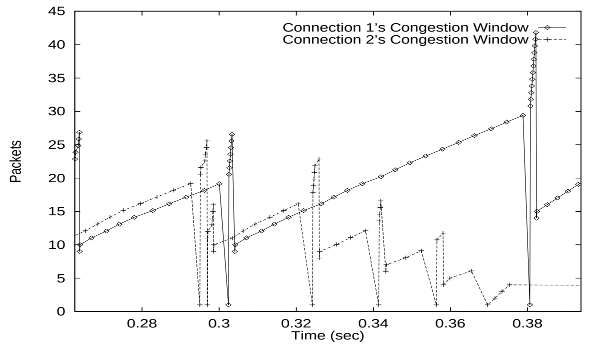

Figure 3. Congestion windows of the two TCP connections in Figure 2.

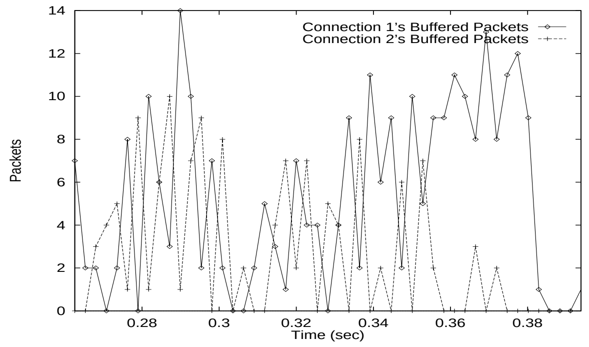

Figure 4. Packet counts at the gateway for the two connections in Figure 2.

3.3 Adaptive TCP vs. Non-adaptive CBR

In the experiment depicted in Figure 5, a TCP connection shares a RED gateway with a non-adaptive constant-bit-rate (CBR) UDP application which persistently sends data at 8 mbps, more than its 5 mbps fair share of the bottleneck 10 mbps link. Such a situation might arise when TCP competes against one or more fixed-bandwidth audio or video streams. The CBR connection starts at time 0, and the TCP connection starts one second later.

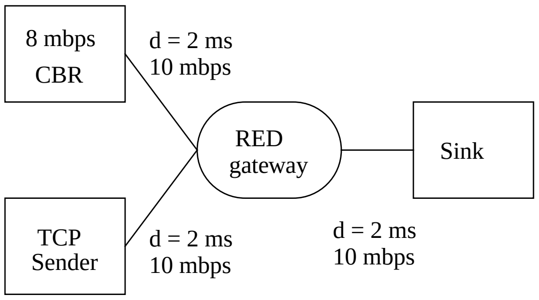

Figure 5. Adaptive TCP competes with a CBR UDP over a RED gateway with buffer size 64 packets and (min_{th} = 16, max_{th} = 32, w_q = 0.002, max_p = 0.02).

Figure 6 shows the bandwidth usage of the two connections at the shared link. Notice that the UDP sender gets 8 mbps almost all the time, whereas the TCP connection can only get the leftover 2 mbps. Because the gateway is not congested initially, no queue builds up during the first second. TCP's slow start causes a large number of packets to be queued at the gateway. Both connections suffer severe packet losses due to the sudden exponential bandwidth increase. New-Reno TCP is able to survive for two fast recovery phases, reducing its congestion window by half for each phase. However, it falls into a time-out when its window size becomes too small. From then on, the congestion avoidance phase is able to utilize the 2 mbps available bandwidth leftover by the CBR connection. The congestion window keeps increasing by one packet for each RTT. After awhile, the aggregate input rate exceeds the output link capacity and extra packets are queued at the gateway. The reason the TCP sender cannot obtain its fair share is due to the unfair FCFS scheduling which distributes the output capacity according to queue occupancy. Since the input rate ratio is 4:1, the CBR connection gets four times as much output bandwidth as the TCP connection. The RED gateway drops packets from both connections even if the TCP connection is using less than its fair share. Eventually the TCP sender slows down after detecting the packet losses. This bandwidth reduction corresponds to the dips in its bandwidth curve. This reduction of input rate causes the queue to drain. At the same time, the CBR connection temporarily gets over 8 mbps because it has packets buffered at the gateway. This bandwidth shift corresponds to the spikes in its bandwidth curve.

One extreme case of this phenomenon is that when the non-adaptive sender transmits at 100% of the network capacity, the competing TCP connection is completely shut out. In general, RED is ineffective at handling non-adaptive connections. Similar results were observed by Zhang in [26] for Random Drop gateways.

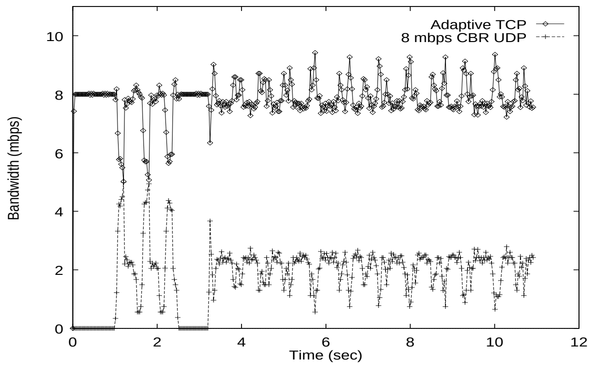

Figure 6. CBR UDP gains more bandwidth share than TCP.

4. Flow Random Early Drop

In this section we present Flow Random Early Drop (FRED), a modified version of RED. Our goal is to reduce the unfairness effects found in RED. Instead of indicating congestion to randomly chosen connections by dropping packets proportionally, FRED generates selective feedback to a filtered set of connections which have a large number of packets queued.

In brief, FRED acts just like RED, but with the following additions. FRED introduces the parameters min_q and max_q, goals for the minimum and maximum number of packets each flow should be allowed to buffer. FRED introduces the global variable avgcq, an estimate of the average per-flow buffer count; flows with fewer than avgcq packets queued are favored over flows with more. FRED maintains a count of buffered packets qlen for each flow that currently has any packets buffered. Finally, FRED maintains a variable strike for each flow, which counts the number of times the flow has failed to respond to congestion notification; FRED penalizes flows with high strike values.

4.1 Protecting Fragile Flows

FRED allows each connection to buffer min_q packets without loss. All additional packets are subject to RED's random drop. Under perfect fair queueing and scheduling, a connection which consumes less than the fair share should have no more than one packet queued. With single FIFO queueing, this condition does not hold. But the number of backlogged packets for low bandwidth connections should still be small. FRED uses min_q as the threshold to decide whether to deterministically accept a packet from a low bandwidth connection. An incoming packet is always accepted if the connection has fewer than min_q packets buffered and the average buffer size is less than max_{th}. Normally, a TCP connection sends no more than 3 packets back-to-back: two because of delayed ACK, and one more due to a window increase. Therefore, min_q is set to 2 to 4 packets.

4.2 Managing Heterogeneous Robust Flows

When the number of active connections is small (N << min_{th}/min_q), FRED allows each connection to buffer min_q number of packets without dropping. Some flows, however, may have substantially more than min_q packets buffered. If the queue averages more than min_{th} packets, FRED will drop randomly selected packets. Thus it may demonstrate the same kind of unfairness as RED: FRED will impose the same loss rate on all the connections that have more than min_q packets buffered, regardless of how much bandwidth they are using. Although this might be a low probability event for a large back-bone gateway (where $N > c*min_{th}/min_q$), it may occur frequently in a LAN environment where RTT is 0 and the gateway has hundreds of buffers.

FRED fixes this problem by dynamically raising min_q to the average per-connection queue length (avgcq) when the system is operating with a small number of active connections. For simplicity, we calculate this value by dividing the average queue length (avg) by the current number of active connections. A connection is active when it has packets buffered, and is inactive otherwise.

4.3 Managing Non-adaptive Flows

With a reasonable amount of buffering, an adaptive connection should be able to adjust its offered load to whatever bandwidth the network provides. Eventually, its packet arrival rate should be no more than the departure rate. A TCP sender exhibits this property. A non-adaptive connection, in contrast, can consume a large portion of the gateway buffers by injecting more packet arrivals than departures. Because of unfair FIFO scheduling, the output bandwidth distribution equals the buffer occupancy distribution. Therefore, fairness cannot be maintained without proper connection level buffer usage policing.

FRED never lets a flow buffer more than max_q packets, and counts the number of times each flow tries to exceed max_q in the per-flow strike variable. Flows with high strike values are not allowed to queue more than avgcq packets; that is, they are not allowed to use more packets than the average flow. This allows adaptive flows to send bursts of packets, but prevents non-adaptive flows from consistently monopolizing the buffer space.

4.4 Calculating the Average Queue Length

The original RED estimates the average queue length at each packet arrival. The low pass filter w_q used in the algorithm prevents the average from being sensitive to noise. Sampling on arrival, however, misses the dequeue movements when there are no packet arrivals. For example, if one packet arrives at time 0 when both the instant and average queue length equal 500 packets, and the next packet arrives 250 packet times later, the instant queue length is 250. RED, however, would leave its calculated average at nearly 500. Such miscalculation could result in low link utilization due to unnecessary packet drops.

In FRED, the averaging is done at both arrival and departure[1]. Therefore, the sampling frequency is the maximum of the input and output rate, which helps reflect the queue variation accurately. In addition, FRED does not modify the average if the incoming packet is dropped unless the instantaneous queue length is zero. Without this change, the same queue length could be sampled multiple times when the input rate is substantially higher than the output link rate. This change also prevents an abusive user from defeating the purpose of the low pass filter, even if all his packets are dropped. Rizzo independently identified similar problems with RED's averaging algorithm in the presence of non-adaptive flows [22].

1. We omit the detail of avoiding double sampling at the same instant, one at output and the other at input.

4.5 The Algorithm

The detailed FRED algorithm is given below.

```
Constants:
  w_q = 0.002;
  min_th = MIN(buffer size / 4, RTT);
  max_th = 2*min_th;
  max_p = 0.02;
  min_q = 2 for small buffers;
          4 for large buffers;

Global Variables:
  q: current queue size;
  time: current real time;
  avg: average queue size;
  count: number of packets since last drop;
  avgcq: average per-flow queue size;
  max_q: maximum allowed per-flow queue size;

Per-flow Variables:
  qlen_i: number of packets buffered;
  strike_i: number of over-runs;

Mapping functions:
  conn(P): connection id of packet P;
  f(time): linear function of time;

for each arriving packet P:
  if flow i = conn(P) has no state table
      qlen_i = 0;
      strike_i = 0;

  if queue is empty
      calculate average queue length avg

  max_q = min_th;
  // define the next three lines as block A
  if (avg >= max_th) {
      max_q = 2;
  }

  identify and manage non-adaptive flows:
  if (qlen_i >= max_q ||
        // define the next line as line B
        (avg >= max_th && qlen_i > 2*avgcq) ||
        (qlen_i >= avgcq && strike_i > 1)) {
        strike_i++;
        drop packet P;
        return;
  }

  operate in random drop mode:
  if (min_th <= avg < max_th) {
      count = count + 1;

      only random drop from robust flows:
      if (qlen_i >= MAX(min_q, avgcq)) {
        calculate probability p_a:
        p_b = max_p(avg-min_th)/(max_th-min_th);
        p_a = p_b/(1 - count * p_b);

        with probability p_a:
            drop packet P;
            count = 0;
            return;
      }
  } else if (avg < min_th) {
      no drop mode:
      count = -1;
  } else {
      // define this clause as block C
      drop-tail mode:
      count = 0;
      drop packet P;
      return;
  }
  if(qlen_i == 0)
      Nactive++;
  calculate average queue length
  accept packet P;

for each departing packet P:
  calculate average queue length

  if (qlen_i == 0) {
      Nactive--;
      delete state table for flow i;
  }

calculate average queue length:
  if (q || packet departed)
      avg = (1-wq)*avg + wq*q;
  else {
      m = f(time - q_time);
      avg = (1-wq)m * avg;
      // original RED missed the following
      // action
      q_time = time;
  }

  if (Nactive)
    avgcq = avg / Nactive;
  else
    avgcq = avg;
  avgcq = MAX(avgcq, 1);

  if q == 0 && packet departed
    q_time = time;
```

FRED's ability to accept packets preferentially from flows with few packets buffered achieves much of the beneficial effect of per-connection queuing and round-robin scheduling, but with substantially less complexity. This also means that FRED is more likely to accept packets from new connections even under congestion. Notice that RED's unfair bias against low bandwidth connections cannot be avoided by per-class queueing because each TCP connection always starts with a congestion window of one packet initially or after each retransmission time-out. Dropping packets from small windows will cause the same problem between connections within the same class. Because TCP detects congestion by noticing lost packets, FRED serves as a *selective* binary feedback congestion avoidance algorithm [14].

5. Simulation Results

We repeated the simulations described in Section 3 using FRED with min_q equal to two packets. Table 3 shows the performance results of the configuration in Figure 1. As the first body row shows, the long RTT connection is able to run at the maximum possible speed as if there were no competition. The lower half of the table shows the actual loss rates of the fragile connection and one of the robust connections. The low loss enables the fragile connection to ramp up to its maximum possible rate.

Table 4 shows the performance of the same connection when its congestion window is allowed to grow up to 64 KB. Note that it achieves a bandwidth close to the 20% fair share, except when the buffer size is only 16 packets. In that case, FRED degrades to tail-dropping, just like RED. In Section 6, we describe an extension to solve the problem of supporting more flows than the number of buffers. The loss rates in Table 4 show that FRED relates packet drops with buffer usage. Even if the long distance connection got the same bandwidth as others, its loss rates, however, are much less than others due to its moderate buffer consumption.

To demonstrate FRED's ability to manage non-adaptive flows more effectively, we augmented the experiment described in Section 3.3 and Figure 5. A CBR UDP connection generates 8 mbps of traffic starting from time 0. At the end of each 20 second interval, a new TCP connection starts to share the same gateway. Figure 7 shows FRED's bandwidth allocation among the five connections for a period of 100 seconds. Unlike the RED bandwidth distribution shown in Figure 6, each TCP connection is able to ramp up from a window size of one packet to its fair share of the link capacity. FRED limits the UDP connection's bandwidth by preventing it from using more than the average number of buffers used by any of the flows.

For the symmetric TCP case described in Section 3.2, we ran more than 10,000 simulations over the configuration in Figure 2. None of them produced a retransmission time-out. FRED's ability to protect small windows has completely prevented the TCP connections from losing packets during ramp up. New-Reno TCP's ability to recover multiple packet losses without resetting the congestion window has also contributed to the outcome.

6. Supporting Many Flows

The simulations presented so far all involve networks that have more packets of storage than there are flows. There is, however, good reason to believe that there are often more flows than packets of storage in the Internet, and that TCP behaves differently under such conditions [2,3,20]. For this reason, we now present simulations with relatively large numbers of flows, as well as an extension to FRED that improves its performance in such situations.

Every active TCP connection tries to keep at least one packet in flight. A large enough number of connections will tend to keep gateway buffers full and will force some of the connections into time-out. A gateway should allocate buffers fairly under these conditions. A gateway should also allow network operators the option of adding buffer memory proportional to the number of flows without incurring unnecessary queuing delay.

The specific danger for a RED or FRED gateway is that under heavy load the average queue size might often exceed max_{th}, causing the gateway to operate as an unfair Drop Tail gateway. We tested FRED under these conditions. In Figure 8, sixteen TCP connections from the same source host share a FRED gateway with parameters (BS=RTT=16, max_{th}=8, min_{th}=4, w_q=0.002, max_p=0.02, min_q=2). The link capacity between the gateway and the sink host is one tenth of that of the link between the source host and the gateway. 2000 retransmission timeouts occurred in a 300 second simulation. Figure 9 shows the output bandwidth allocation for the first eight connections during a segment of the simulation. Notice that at any moment only a subset of the connections share the output link. Other connections are waiting for retransmission timeouts. This is due to the fact that the gateway has only enough buffers to support four to six connections given the small min_q and max_{th} values. FRED lets the connections take turns, giving inactive

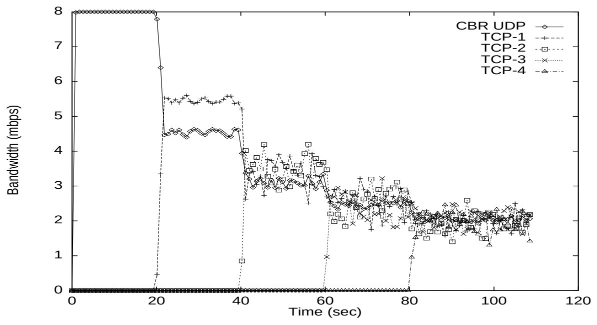

Figure 7. FRED prevents a CBR UDP connection from taking an unfair share of the bandwidth.

	BS=16	BS=24	BS=32	BS=40	BS=48	BS=56
BW/MAX	99%	98%	98%	98%	97%	97%
Fragile's Loss Rate	0.04%	0.01%	0.00%	0.00%	0.00%	0.00%
R-1's Loss Rate	0.98%	0.85%	0.93%	0.88%	0.86%	0.76%

Table 3. Performance of a long RTT TCP connection under FRED (Second row shows the ratio of bandwidth allocated over maximum possible. The lower half shows the loss rates of the fragile connection and one of the robust connections.)

	BS=16	BS=24	BS=32	BS=40	BS=48	BS=56
BW/Link Rate	6.9%	19%	20%	20%	16%	15%
Fragile's Loss Rate	0.26%	0.08%	0.06%	0.07%	0.05%	0.04%
R-1's Loss Rate	1.35%	1.18%	1.21%	1.06%	0.86%	0.75%

Table 4. Performance of a long RTT TCP under FRED (Second row shows the percentage of bandwidth allocated over the link capacity. The lower half of the table shows the loss rates of the fragile connection and one of the robust connections.)

connections priority each time a buffer needs to be reallocated. The long term bandwidth distribution is very close to the fair share. However, this non-smooth ON/OFF sharing is unfair for short periods of time and causes large delay variations. Short-lived connections might be treated unfairly due to scarce resources. Furthermore, the gaps between successive ONs for the same connection are determined by TCP's exponential backoff mechanism, which often imposes very high delays.

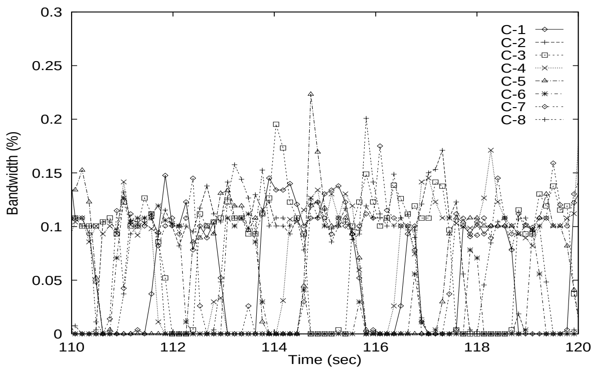

Figure 9. Connections in Figure 8 alternate ONs and OFFs to share the scarce buffers under basic FRED.

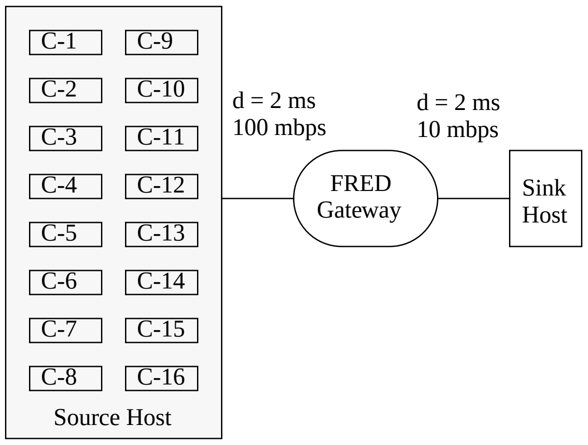

Figure 8. Sixteen TCP connections share a FRED gateway with 16 packet buffers (max_{th}=8, min_{th}=4, max_p=0.02, w_q=0.002, min_q=2).

The timeouts and associated high delay variation could potentially be eliminated by adding buffer memory. This approach risks high delay and unfair buffer use. Nagle described a "one packet switch" model to solve this problem and demonstrated that a well-behaved flow only needs one packet buffer at the switch in order to receive fair service provided by a round-robin scheduler [21]. In this section, we extend FRED with a "two packet buffer" mechanism to achieve a similar effect with FIFO scheduling.

We propose that when the number of simultaneous flows is large, the gateway should provide exactly two packet buffers for each flow. This should provide fair bandwidth sharing even with FIFO queuing, since each flow will get bandwidth proportional to the number of packets it has buffered. Therefore, as long as each flow maintains one or two outstanding packets at the gateway, the largest buffer occupancy, and thus bandwidth allocation discrepancy should be no more than two to one. The following pseudo code implements the FRED "two packet buffer" extension:

```
FRED many flow extension:
// delete block A and line B

// replace block C with the following
two packet mode (avg >= max_th):
if(qlen_i >= 2) {
    count = 0;
    drop packet P;
    return;
}
```

This modification may cause TCPs to operate with small congestion windows. Some TCPs, such as Vegas [1], can recover quickly from lost packets even with a small window. Unfortunately, other TCP implementations such as Tahoe or Reno have to wait for three duplicate ACKs before a lost packet is retransmitted, in order to avoid unnecessary retransmissions caused by out of order delivery [12,13]. They will be forced to use time-outs to recover from packet losses with tiny windows. To overcome this weakness, we propose a modification to these TCPs in Appendix Section 11.3. This change allows a sender with a congestion window smaller than three packets to trigger the fast retransmission and fast recovery without lowering the three duplicate ACK threshold.

To test Figure 8 with FRED's "two packet buffer" extension, we added 16 packet buffers to the gateway and used the same parameters as before. Only 20 retransmission timeouts occurred during the simulation, rather than 2000. Figure 9 shows the bandwidths of eight of the connections during a segment of the simulation. With two packet buffering and the tiny window loss recovery enhancement, each TCP connection is able to maintain a smooth stream of packets through the bottleneck.

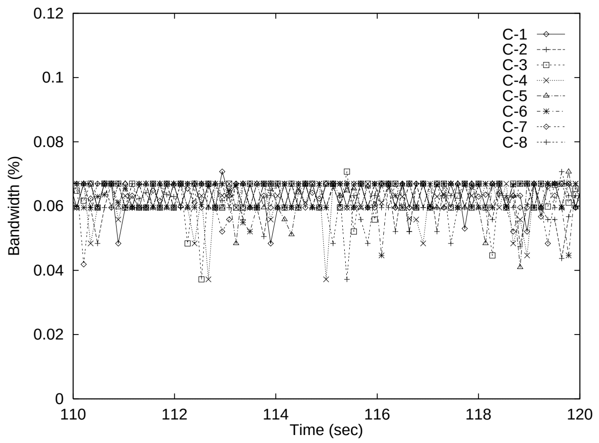

Figure 10. Connections in Figure 8 smoothly share the bandwidth under the extended FRED with two RTT worth of buffering.

For comparison, we also simulated RED for both cases (BS=16 and BS=32) and basic FRED (BS=16), all with the small window enhancement enabled. Table 5 summarizes performance results from all the simulations in this section. RED-16 and FRED-16 produced the most timeouts for lack of buffering. FRED-16 performed slightly better than RED-16 due to the protection for fragile connections. Adding more buffers to RED-32 did not reduce the number of timeouts significantly even with the small window enhancement. In contrast, FRED-32 allowed each connection to buffer two packets and maintained the balance among all competing connections.

	RED-16	RED-32	FRED-16	FRED-32
total RTOs	2349	2006	2196	20
link utilization	95.7%	99.7%	99.3%	100%
TCP goodput (kbps)	496	500	499	503
per-flow throughput / link-rate	6.17%	6.23%	6.21%	6.25%

Table 5. Performance measurements for RED and FRED with or without large buffers. We used tcp-newreno with small window recovery enhancement for all four tests. The TCP goodput measurements do not include 13.2% ATM+AAL5 overhead.

Figure 11 shows the per connection bandwidth distribution. For each connection, we measured the bandwidth usage at the bottleneck at constant intervals (140ms) and created one data set for all connections in the same simulation. Each curve in the figure represents the frequency distribution of one data set. FRED-32 has a small spike at zero and a large spike near the fair share, which means almost perfect smooth sharing. For RED-16, RED-32, and FRED-16, there is a large spike at zero, representing large number of timeouts, and scattered small spikes between zero and eleven times the fair share. FRED-16's spikes are higher than RED-16 and RED-32 near one and lower elsewhere.

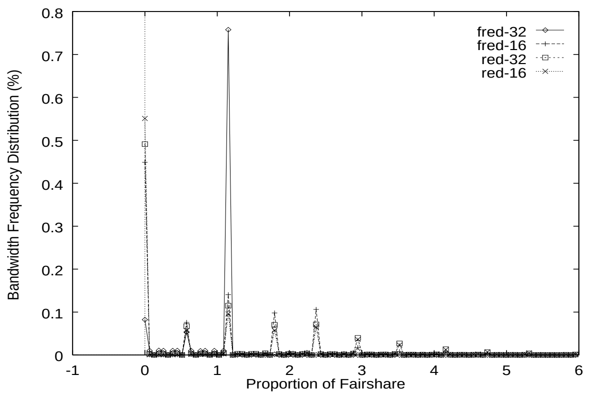

Figure 11. For each connection, we measured the bandwidth usage at the bottleneck at constant intervals (140ms) and created one data set for all connections in the same simulation. Each curve in the figure represents the frequency distribution of one data set.

7. Operating in Byte Mode

Like RED, FRED can be configured to operate in byte mode. To do this, all queue length related constants and variables (min_{th}, max_{th}, min_q, max_q, qlen, avg, avgcq) have to be redefined in the units of bytes. All constant expressions in the algorithm should also be multiplied by MaximumPacketSize. The probability calculation in byte mode is suggested in [5]:

```
Calculate probability $p_a$ in byte mode:
$p_b$ = $max_p$ (avg - $min_{th}$)/($max_{th}$ - $min_{th}$)
$p_b$ = $p_b$ * PacketSize(P)/MaximumPacketSize
$p_a$ = $p_b$/(1 - count * $p_b$)
```

8. Conclusions

We have demonstrated that discarding packets in proportion to the bandwidth used by a flow does not provide fair bandwidth sharing at a gateway. A selective discard mechanism is needed to protect flows that are using less than their fair share, and to prevent aggressive flows from monopolizing buffer space and bandwidth.

FRED, a modified version of RED, provides selective dropping based on per-active-flow buffer counts. FRED keeps this extra state only for flows that have packets buffered in each gateway, and is compatible with existing FIFO queuing architectures. Simulation results show that FRED is often fairer than RED when handling connections with differing round trip times and window sizes. FRED also protects adaptive flows from non-adaptive flows by enforcing dynamic per-flow queueing limits.

9. Acknowledgments

We thank our thesis advisor, Professor H.T. Kung, for his support and guidance. His energy and insight at all levels are a constant inspiration. We also appreciate the valuable comments and suggestions from Trevor Blackwell, Koling Chang, Brad Karp, Vera Gropper, Allison Mankin, and the anonymous SIGCOMM reviewers.

10. References

[1] Brakmo, L., O'Malley, S., Peterson, L., "TCP Vegas: New Techniques for Congestion Detection and Avoidance," SIGCOMM'94

[2] Claffy, K., Braun, H-W., Polyzos, G., "A Parameterizable Methodology for Internet Traffic Flow Profiling," IEEE Journal on Selected Areas in Communications, March 1995.

[3] Eldridge, C., "Rate Controls in Standard Transport Protocols," ACM Computer Communication Review, July 1992.

[4] Fall, K., Floyd S., "Simulation-based Comparisons of Tahoe, Reno, and SACK TCP," Computer Communication Review, July 1996

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

[6] Floyd, S., Jacobson, V., "On Traffic Phase Effects in Packet-Switched Gateways," Computer Communication Review, April 1991

[7] Floyd, S., "TCP and Explicit Congestion Notification," Computer Communication Review, October 1994

[8] Floyd, S., "Connections with Multiple Congested Gateways in Packet-Switched Networks Part 1: One-way Traffic," Computer Communications Review, October 1991

[9] Hahne, E., Gallager, R., "Round Robin Scheduling for Fair Flow Control in Data Communications Networks," IEEE International Conference on Communications, June 1986

[10] Hashem, E., "Analysis of Random Drop for Gateway Congestion control," MIT-LCS-TR-465

[11] Hoe, J., "Improving the Start-up Behavior of a Congestion Control Scheme for TCP," SIGCOMM'96

[12] Jacobson, V., "Congestion Avoidance and Control," SIGCOMM'88

[13] Jacobson, V., "Modified TCP congestion avoidance algorithm," April 30, 1990, end2end-interest mailing list

[14] Jain, R., Ramakrishnan, K.K., Chiu, D., "Congestion Avoidance in Computer Networks With a Connectionless Network Layer," DEC-TR-506

[15] Kung, H. T., Blackwell, T., Chapman, A., "Credit-Based Flow Control for ATM Networks: Credit Update Protocol, Adaptive Credit Allocation, and Statistical Multiplexing," SIGCOMM '94

[16] Kung, H.T., Chapman, A., The CreditNet Project, http://www.eecs.harvard.edu/cn.html

[17] Lin, D., Kung, H.T., "TCP Fast Recovery Strategies: Analysis and Improvements," work in progress

[18] Mankin, A., Ramakrishnan, K., "Gateway Congestion Control Survey," RFC1254

[19] Mankin, A., "Random Drop Congestion Control," SIGCOMM'90

[20] Morris, R., "TCP Behavior with Many Flows," IEEE International Conference on Network Protocols, October 1997, Atlanta

[21] Nagle, J., "On Packet Switches with Infinite Storage," IEEE Transactions on Communications, Vol. 35, pp 435-438, 1987

[22] Rizzo, L., "RED and non-responsive flows," end2end-interest mailing list, June, 1997

[23] Tanenbaum, A., *Computer Networks*, Prentice Hall, 2nd Edition, 1989

[24] Turner, J.S., "Maintaining High Throughput during overload in ATM Switches," INFOCOM'96

[25] Villamizar, C., Song, C., "High Performance TCP in ANSNET," Computer Communications Review, October 1994

[26] Zhang, L., "A New Architecture for Packet Switching Network Protocols," MIT-LCS-TR-455

11. Appendix

This section describes our simulator and the modifications we made to TCP Reno for this work. Our detailed study of TCP's fast retransmission and fast recovery is described in [17].

11.1 The Simulator

Our simulator is a distant descendant of one written for the DARPA/BNR funded CreditNet Project [16] in 1992. The simulated ATM switch consists of N full duplex portcards. Each output interface is capable of receiving and buffering one cell from each input on each cell cycle. The switch implements pure EPD with no partial packet discard in order to emulate packet switching. We use 552-byte TCP/IP packets (including IP and TCP headers). Each packet is carried in 12 ATM cells. RED/FRED decides whether to drop a packet upon reception of the first cell in the packet.

The simulator uses the NetBSD 1.2 TCP Reno source files with a handful of modifications described below. The TCP receivers consume data immediately.

11.2 TCP New-Reno

The version of TCP New-Reno used in this paper is similar to that described elsewhere [4,11], but differs in some details. In this section, we compare the New-Reno used in this paper with the standard TCP Reno implemented in NetBSD 1.2.

When the sender is in recovery and receives a "partial" ACK for some but not all of the original window, the sender immediately retransmits the packet after the one acknowledged by the partial ACK, and then continues with recovery.

The recovery process terminates when an ACK acknowledges all the packets in the original window and the acked sequence number plus half of the original window size (snd_ssthresh) is greater or equal to the right edge of the current window. This ensures that the sender will not immediately fall into a new recovery phase. In addition, at the end of recovery the congestion window is decreased to the number of packets in flight plus one. This prevents bursts of packets from being sent at the end of recovery.

11.3 TCP with Tiny windows

This section describes the TCP enhancement used in the simulations at the end of Section 6. The enhancement allows TCP to recover from losses even when the window is small.

When operating with a congestion window smaller than about 10 packets, a TCP sender should send a new packet for each of the first two duplicate ACKs. These two new packets will generate more duplicate ACKs, increasing the sender's chances of entering fast retransmission. For example, if the current window is two packets and the first packet is lost in the network, then packet 2 would cause one duplicate ACK for packet 1. At that moment there are no packets in flight. The sender should transmit packet 3 upon receiving the first duplicate ACK, which in turn prompts a second duplicate ACK for packet 1. The sender should then transmit packet 4, which will cause a third duplicate ACK and trigger fast retransmission. This process takes three RTTs, but, as long as there is at least one packet in flight, the connection will not fall into time-out. This approach should not itself increase congestion, since it follows the conservation of packets rule [12]. A duplicate ACK received means a packet has left the network. By injecting one new packet, the total number of packets in flight is still no more than the congestion window allows. If the receiver's advertised window does not allow injecting more packets, the sender should re-send the last packet in the window or simply the packet that is causing duplicate ACKs. If the recovery phase is triggered, the inflated congestion window should be restored back for computing slow start threshold and the two extra packets should be accounted for the total number of outstanding packets.