TCP Fast Recovery Strategies: Analysis and Improvements

Dong Lin and H.T. Kung
Division of Engineering and Applied Sciences
Harvard University
Cambridge, MA 02138 USA

Abstract

This paper suggests that, to match an ideal Internet gateway which rigorously enforces fair sharing among competing TCP connections, an ideal TCP sender should possess two properties while obeying congestion avoidance and control principles. First, the TCP sender which under-uses network resources should avoid retransmission timeouts. When experiencing network congestion, a TCP connection should not time out unless it has already reduced its congestion window to one packet but still cannot survive. Second, the TCP sender which over-uses network resources should lower its bandwidth. The congestion window for a connection should decrease each time a lost packet is detected, because an ideal gateway will drop packets, during congestion, with a probability proportional to the bandwidth of the connection.

Following these guidelines, we propose Network-sensitive Reno (Net Reno), a set of optimizations that can be added to a traditional Reno TCP sender. Using TCP's self-clocking property and the packet conservation rule, Net Reno improves Reno and its variants (New-Reno and SACK), in reducing TCP retransmission timeouts (RTOs) and in being conservative in network usage during the fast recovery phase. Through a trace analysis, we have shown that over 85% of RTOs are due to small congestion windows that prevent fast retransmission and recovery algorithms from being effective. This implies that sophisticated recovery schemes such as SACK will have limited benefits for these loads. Net Reno overcomes this problem with a small window optimization.

While being less aggressive than previous approaches, Net Reno can recover any number of packet losses without timeouts as long as the network keeps at least one packet alive for the connection. This scheme thus brings TCP one step further toward the ideal model. Net Reno requires no modifications to the TCP receiver. Simulations and laboratory experiments have shown that they significantly reduce RTOs and improve TCP's goodput.

1. Introduction

With its explosive growth, the Internet backbone faces the challenge of operating at its capacity. Many remote TCP connections experience high loss rates due to gateway congestion during busy hours. Congestion avoidance and control have become critical to the use of the Internet.

In 1988, Jacobson [6] pioneered the concepts of TCP congestion avoidance and control: slow start, congestion avoidance, conservation of packets, and exponential timer backoff. TCP was later augmented with fast retransmission and fast recovery algorithms in 1990 to avoid inefficiency caused by retransmission timeouts (RTOs) [7, 19, 20].

These basic TCP principles were designed based on the assumption that the gateway drops at most one packet per flow when the sender increases the congestion window by one packet per round-trip time. Therefore, TCP's fast retransmission and fast recovery algorithms can quickly recover the loss and adapt to equilibrium using exponential decrease and linear increase of the congestion window. A recent study [16] has suggested that packet loss rates on the Internet have doubled within a year and that burst dropping is common. With the deployment of Random Early Detection (RED) gateways [3], the number of dropped packets per connection will be proportional to its bandwidth usage at the shared link, causing multiple drops for large window connections. Therefore, it is worthwhile to study the interactions between end-systems and gateways at the individual flow level.

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

1.1 Previous Work on Avoiding Timeouts

TCP's fast retransmission and fast recovery algorithms [7, 19, 20] were developed to recover packet losses quickly without RTOs. The fast retransmission algorithm, which first appeared in Tahoe TCP [6], retransmits an unacknowledged segment after receiving three duplicate acknowledgments (ACKs), resets the congestion window to one packet, and begins slow start. The fast recovery algorithm in Reno [7, 20] replaces the slow start with congestion avoidance by reducing the congestion window to one half.

In Reno, the maximum number of recoverable packet losses in a congestion window without timeout is limited to one or two packets in most cases. Under the most optimistic assumptions that the algorithms always be triggered, no more than six, or $\log_2 128 - 1$, losses can be recovered with a maximum window size of 128 packets. This is because Reno TCP cuts the congestion window by half for each recovered loss. With six back-to-back lost packets, the final window size would reduce to two packets. Further losses have to be retransmitted after a long delay when RTO is triggered by a 500ms slow timer, bringing the throughput to its knees.

TCP's selective acknowledgment (SACK) option [12] enables the receiver, when holding non-contiguous data, to inform the sender consecutive blocks that were successfully received. With SACK, the sender is able to identify and

retransmit multiple lost packets within the same round-trip time (RTT) if there are enough ACKs returning to the sender.

Hoe [5] proposed a modification to Reno (New-Reno TCP) that can help the sender recover multiple packet losses. It is suggested that the sender should fall into a slow start immediately after the initial loss is detected and inflate the congestion window by one packet for every two duplicate ACKs. This scheme, however, generates unnecessary retransmissions for packets already cached at the receiver. Fall and Floyd [2] described a modified New-Reno which avoids unnecessary retransmissions and slow start.

In summary, we are interested in four different TCPs:

- Tahoe: The sender implements fast retransmission only.
- Reno: The sender implements both fast retransmission and fast recovery.
- Modified New-Reno: The sender retransmits one lost packet per RTT upon receiving partial ACKs and terminates the recovery phase when the whole window is acknowledged.
- SACK: The sender is able to retransmit multiple lost packets per RTT using additional information in SACK blocks.

1.2 Remaining Issues

Previous studies have assumed that multiple packet loss is the major cause of RTO. By studying an extensive set of traces of Paxson [17], we have found that for these traces the average congestion window is small (12 packets) and that over 85% of the timeouts are due to non-trigger of fast retransmission (see Section 4.1 below). This strongly suggests that no multiple loss recovery schemes can be effective for the Internet load represented by these traces. Other factors that cause RTO including lost retransmissions and limited data are not addressed.

Another shortcoming of some previous work is that TCP's efficiency concerns seem to overwhelm network congestion concerns. That is, refinements to improve TCP's efficiency are provided by adding aggressiveness.

1.3 Results of This Paper

In this paper, we provide insights into Internet congestion avoidance and control in the following two areas:

- We describe an ideal model based on cooperative gateway and end-system strategies.
- Following the ideal model, we propose Network-sensitive Reno (Net Reno) TCP, a set of optimizations that make TCP more resilient to packet losses even under small congestion windows and more conservative in network usage.

We hope that, with end-systems running Net Reno TCP, gateways implementing ideal packet discard algorithms will be able to achieve fair sharing among competing TCP connections.

Our optimizations can be applied to both Reno and SACK TCP. They require no modifications to the TCP receiver. Most of all, they maintain TCP's congestion control algorithms and strictly obey the principles of slow start, congestion avoidance, and conservation of packets [6, 7].

The rest of this paper is organized as follows: Section 2 describes our ideal model for Internet congestion avoidance and control; Section 3 summaries terminologies of this paper; Section 4 describes the problem and solution for small congestion windows; Section 5 describes our conservative loss-sensitive window reduction mechanisms; Section 6 gives additional optimizations to avoid timeouts; Section 7 presents simulation and experimental results.

2. Cooperative Gateways and End-Systems

While efficiency and stability have been the major objectives in the study of TCP congestion avoidance and control algorithms, the interactions among competing TCP connections sharing gateway resources have not been given equal attention.

2.1 Ideal Network Model

With respect to traffic management, an ideal Internet should demonstrate the following properties:

- congestion avoidance and control
- equal sharing
- denial of service avoidance

An ideal gateway should have mechanisms for congestion avoidance. It drops packets in order to signal congestion and trigger end-system's flow control algorithms. These signals must be selectively delivered only to the offending end-systems. A fair gateway should statistically avoid dropping packets from connections that use less than the fair share of the resources. Otherwise the system would not converge to the ideal equilibrium.

The gateway causes denial of service when it cannot provide at least one packet buffer per flow when the number of simultaneous flows increases. While it is possible that TCP's exponential timer backoff scheme allows a subset of the competing flows to share the resources while others are waiting in timeouts, we believe that smooth sharing and less bandwidth variation provide better stability than ON/OFF sharing. Unfair bandwidth distribution within short intervals noticeable by users is particularly biased against short-lived and interactive connections. This idea of one buffer per user was first addressed by Nagle [15]. The ideal gateway should delay denial of service by reducing the bandwidth variation among the competing flows in order to maximize the number of simultaneous users.

In [11], we demonstrated a fair gateway packet discard algorithm, Flow Random Early Drop (FRED), which supports a large number of simultaneous flows fairly by simply adding more buffers. In contrast, fairness worsens when the buffer size increases under other algorithms such as Drop-Tail or RED. Therefore, enforcing fairness not only benefits each individual flow, but also makes the maximum number of simultaneous flows more scalable with respect to the required gateway buffer size.

2.2 Ideal End-System Model

An ideal end-system should deploy multiplicative decrease and additive increase for congestion control and avoidance. As suggested by [6], anything more aggressive would cause collapse or instability.

It is commonly believed that TCP should be robust on recovering packet losses and avoiding timeouts. We claim that a TCP sender should aggressively recover packet losses and avoid timeouts. A timeout is necessary only if TCP cannot survive with a window size of one packet. A fragile TCP connection (which is very sensitive to losses) should not fall into timeouts while other connections are increasing their bandwidth share at the gateway.

Matching aggressive TCP recovery schemes with non-ideal gateway packet discard algorithms deserves further investigation. An ideal gateway signals the offending connections only. RED [3], an approximation to the ideal model, assures that the number of packet losses from a connection is proportional to its bandwidth usage. Thus, offending connections using bandwidths larger than their fair shares will incur more losses than the others. An ideal TCP sender should respond properly to this clue by reducing the congestion window each time a lost packet is detected. A SACK TCP sender which cuts the window by one half, regardless of the number of losses, may be too aggressive and destructive to other connections and the whole network.

3. Terminology and Notations

Throughout this paper, we use the following terms in TCP's protocol control block. These names are taken from NetBSD 1.2's TCP implementation:

- snd_cwnd: the sender's congestion window size
- snd_una: the smallest sequence number of the unacknowledged packets
- snd_nxt: sequence number of the next packet to be sent
- snd_ssthresh: sender's slow start threshold
- dupacks: the number of duplicate ACKs received

The following terms are not in the implementation but appear in our analysis and graphs:

- rcv_rseq: sequence number in the header of a received packet
- rcv_dseq: sequence number of a missing packet observed by the receiver when receiving a non-consecutive packet
- snd_sseq: sequence number in the header of a transmitted packet

Maximum segment size is 512 bytes. All terms described above are converted from sequence numbers to packet numbers (sequence numbers divided by segment size). Delayed ACK is enabled by default. Each TCP session starts with a congestion window of one packet and snd_ssthresh of 128 packets (64KB). TCP's fast timer expires every 200ms and slow timer every 500 ms. A random factor is added to each timeout interval so that no two timers go off at the same time.

4. The Small Windows Problem and Solution

According to a trace study of this section, the most dominant factor that causes TCP to timeout appears to be small congestion windows. Before a lost packet is recovered, the window size limits the number of returning ACKs the sender may receive. Because TCP requires three duplicate ACKs in order to trigger fast retransmission and fast recovery, small windows may prevent these algorithms from being effective.

Paxson [16] has found that out-of-order delivery is common and argued that TCP should not retransmit too early by lowering the received duplicate ACK threshold. In this section, we describe a novel approach that remedies the problem. This algorithm was used in [11] as a demonstration of FRED where we pointed out that this small window optimization improves the robustness of TCP significantly and enables connections with tiny windows to compete fairly with other high bandwidth connections sharing the same gateway.

4.1 Trace Study of Timeouts

To understand the dynamics of RTOs, we analyzed 2,000 actual TCP bulk transfers over the Internet. This is a subset of a large trace collected by Paxson [16, 17] during November and December of 1995. Each transfer delivered 100KB of data. Traces were collected at both senders and receivers.

Our objective is to identify retransmission timeouts from the traces and analyze their causes. We categorize the timeouts into the following three classes:

- Non-trigger: The sender retransmits a packet without previous attempts because the fast retransmission algorithm has not been triggered.
- Multiple losses: The sender retransmits a packet that is different from previous retransmissions sent at the beginning of this timeout period.
- Lost retransmission: The sender retransmits the same packet that has already been resent.

A TCP receiver sends a duplicate ACK when it receives out-of-sequence packets. The sender has to accumulate three duplicate ACKs before it triggers the fast retransmission and fast recovery algorithms in order to avoid unnecessary retransmissions due to out-of-order delivery. The non-trigger case will lead to RTO for all versions of TCP. Under standard Reno, multiple packet loss causes back-to-back recoveries as the sender only retransmits one packet per recovery phase. Previous work enables the sender to handle multiple losses per recovery, improving efficiency during the recovery period. Currently, no TCP that we are aware of can deal with lost retransmissions[1]. By analyzing the distribution of various causes, we hope to identify the most significant factor among these three.

For each timeout, we record the connection's instant congestion window size. We then calculate the distribution for all timeouts that have congestion windows greater than or equal to X packets for all values of X. Figure 1 shows details of the analysis. Astonishingly, this graph shows that over 85% of the timeouts are due to non-trigger. For the remaining, 11% are due to multiple loss and 4% are due to lost retransmissions (from the three dots at X=1). Figure 1 also shows that less than 10% of timeouts have congestion window larger than 10 packets (from the total window distribution curve) and that the distribution for multiple loss is never more than 10% (from the multiple loss curve).

To verify our findings, we measured the congestion windows while the connections are not in timeout. The average window

1. See Section 6.2 and [9] for a recent development.

size is 12 packets. Given such a small window size, it is quite intuitive to expect that non-trigger is common.

These results imply that sophisticated multiple loss recovery schemes such as SACK have limited benefit for TCP connections over the load represented in the trace. Given the small congestion window size, variants of New-Reno might do just as well. SACK can, perhaps, be more effective over long links where connections have large windows and the congestion is not as severe.

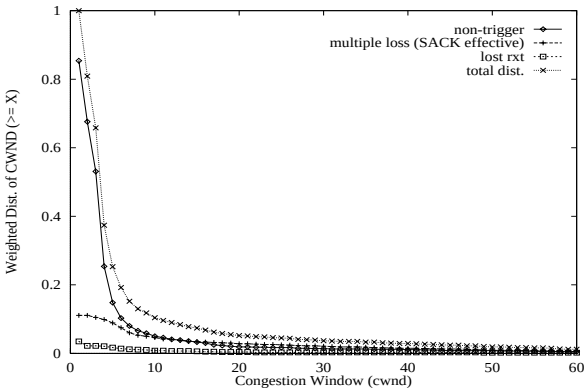

Figure 1. Congestion window sizes distribution and timeout classes distributions. Non-trigger is the dominant factor for timeouts. Multiple losses never contribute to more than 10% of the timeouts.

4.2 Recovery from Small Windows

This section presents a solution to the small window problem, based on the packet conservation rule [7]. In Reno TCP, the sender stops sending when receiving duplicate ACKs. When three dupacks are received, the fast retransmission and recovery phase starts, otherwise the sender waits for an RTO. In our solution, the sender generates a new packet for each duplicate ACK received (only for the first two ACKs). As mentioned in [7], a duplicated ACK received means a packet has left the network. We can inject one new packet so that the total number of in-flight packets is as many as the current congestion window allows. New packets are generated by inflating the congestion window as in the recovery phase. Assume that the network will keep alive at least one packet per connection. Then these new packets injected into the network will cause more ACKs to come back and eventually "force" the sender into the recovery phase when three duplicate ACKs arrive.

When the recovery phase is triggered, the inflated congestion window should be deflated back for computing the slow start threshold. If a positive ACK comes back before dupacks reaches three, then the number of packets acknowledged should be checked against the number of packets injected. If the amount acknowledged is more than or equal to the amount inflated, dupacks is cleared. Otherwise, dupacks should be subtracted by the amount acknowledged as if the next unacknowledged packet is now receiving duplicate ACKs. The reason is that if we have received N ACKs, but can only shift the congestion window less than N packets to the right, this next unacknowledged packet must be either out-of-order or missing.

Figure 2 demonstrates the effect of window inflation for recovering losses for small windows. X-axis is time in the unit of packet time (time to transmit a 512 byte packet at link rate). Y-axis is packet number (sequence number divided by segment size). The connection has a window size of three packets. The outstanding packets were 43, 44 and 45 transmitted at (93, 43), (93, 44) and (99, 45). Packet 43 was lost (marked by the triangle at (97, 43)) and the ACK generated by packet 44 was positive because packet 42 did not cause an ACK due to delayed ACK. The ACK generated by packet 45 set dupacks to one. The congestion window was immediately inflated to four so that packet 46 was sent at (103, 46) and it became the only packet alive. One RTT later the ACK generated by packet 46 came back which incremented dupacks to two and inflated the window further to five and packet 47 was sent at (108, 47). Another RTT later, the new returning ACK finally triggered fast retransmission of packet 43 at (112, 43).

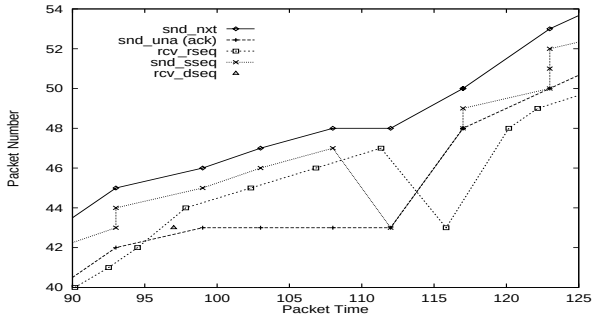

Figure 2. Recovering one packet loss for a window of three packets, under the small window recovery scheme.

This optimization for small windows is of less significance for large windows and may cause extra packet losses under severe congestion. Therefore, pre-recovery window inflation should be dynamically enabled. In our implementation, it is triggered only if the congestion window size is less than ten packets.

A similar approach was simultaneously developed in [18]. However, their approach does not justify why two new packets can be injected into the network or how to account for the two extra packets in the congestion window after three duplicate ACKs or a positive ACK are received. It is also pointed out in [18], through trace analysis of a busy web server, that 90% of the retransmission timeouts are due to non-trigger.

5. Loss-Sensitive Window Reduction

As mentioned earlier, ideal gateways, to which RED and FRED approximate, signal congestion by dropping packets from offending flows. The number of packet drops reflect the over aggressiveness of individual flows. A Reno sender reduces its congestion window by one half for each packet loss. Therefore, the congestion window is $W/2^L$ after L recoveries where L is the number of losses in a window of size W. New-Reno and SACK, however, cut the window by a half regardless of the number of losses. This makes the window to be $W/2$ after one recovery when all losses are retransmitted.

Besides the mismatch with gateway packet discard algorithms, such loss insensitive window reduction also causes packet bursts when recovering multiple packet losses. A partial ACK that only acknowledges some but not all packets in the original window causes a sudden shift of the window allowing multiple packets to be sent. In addition, since the number of in-flight packets will be smaller than half of the original window size if more than one packet is lost, the window reset at the end of the recovery will allow multiple packets to be injected. Fall and Floyd [2] first pointed out this packet bursts problem.

In this section, we described a loss-sensitive window reduction algorithm based on the packet conservation rule. The size of the congestion window after the recoveries will be a function of the number of lost packets. Specifically, the algorithm will reduce the window to $W/2 - c$ where c is proportional to the number of packet losses. In addition, the algorithm does not have the packet bursts problem.

5.1 Loss-Sensitive New-Reno TCP

We describe below the two changes made to the standard Reno TCP implemented in NetBSD 1.2 for our loss-sensitive New-Reno:

1. When a partial ACK is received, i.e., a positive ACK that only acknowledges some but not all the packets in the original window after fast recovery was triggered, immediately retransmit the next lost packet pointed by the ACK and continue the recovery process. In addition, reduce the congestion window so that this partial ACK does not cause more packets to be sent other than the retransmission.
2. If a positive ACK acknowledges all the packets in the original congestion window, terminate the recovery phase. Reset the congestion window to one half or less so that at most one packet can be sent due to the left edge of the window shifting to the right. This will cause the sender to slow start instead of blasting a large number of packets to the network.

Notice from the above two rules that after the number of outstanding packets is reduced by one-half, each received ACK (duplicate or partial) causes exactly one packet to be sent for the rest of the recover phase. In other words, the number of outstanding packets does not re-grow during the recovery phase. In contrast, the method of [2] imposes a maximum burst of four packets for each received ACK and does not adjust the window upon receiving positive ACKs.

5.2 Loss-Sensitive SACK TCP

While our SACK receiver strictly complies with the specification [12], our SACK sender is implemented by the following additions to NetBSD 1.2 Reno TCP:

1. If the inflated congestion window allows sending one packet, retransmit a lost packet and reduce the window by one packet. If all lost packets have been retransmitted, send a new packet as in Reno.
2. When a partial ACK is received, retransmit a lost packet or send a new packet. Reduce the congestion window so that this ACK does not cause more packets to be sent.
3. Use the last rule defined in Section 5.1 to terminate the recovery process and reset the congestion window.

Notice again that each received ACK (duplicate or partial) causes exactly one packet to be sent for the second half of the recovery phase. The authors of [2] suggest that when SACK retransmits a lost packet, another packet should be sent because the number of outstanding packets is at least one less than expected due to the newly detected lost packet. By doing this, the number of outstanding packets would be as close to half of the original window size as possible. We argue that, under some circumstances, the congestion window should actually be reduced by *more* than one-half in view of sudden increase of link sharing at the gateways as indicated by packet losses. Our less aggressive sender allows the number of outstanding packets to be shaped by the exact number of packet losses. Therefore, the number of packets on the fly is half of the original window size minus a constant c, i.e., $(W/2 - c)$, where c is the distance between the first and the last lost packets. During no time does the sender generate more than one packet for each received ACK. At the end of a multiple-packet loss recovery phase, the number of outstanding packets is smaller than half of the original window size, and the connection uses slow start to ramp up. Our more conservative scheme strictly obeys the conservation of packets rule. We believe that the sender should not insist on keeping the number of in-flight packets to be one half of the window during recovery when congestion is detected.

5.3 Dealing with Out-of-order Packets During Recovery

Three duplicate ACKs are required before fast retransmission starts. This is a conservative measure to avoid unnecessary retransmission caused by out-of-order delivery. However, the schemes described in the above two subsections and those in previous work will retransmit other lost packets in the same window upon receiving the first partial ACK. This is inconsistent with the original Reno approach and may cause unnecessary retransmission when out-of-order packets occur. This is demonstrated in the following example:

With a congestion window of W packets (W >> 3), the sender has W packets in-flight. Assume P_1 is lost, P_w and the retransmission of P_1 arrive out-of-order. The incoming packet sequence at the receiver is:

P_2,....., P_{w-1}, P_1, P_w, P_{w+1},....., $P_{3w/2}$

The receiver generates the following ACK sequence accordingly:

A_1,....., A_1, A_w, A_{w+1}, A_{w+2}....., $A_{3w/2+1}$

The outgoing packet sequence on the sender side is:

P_1, P_2,....., P_w, P_1, P_{w+1}, P_{w+2},....., $P_{3w/2}$, P_w

Notice A_w is a partial ACK because it falls into the original window of W packets. This causes the sender to falsely retransmit P_w due to out-of-order by one packet.

Our Net Reno solves the problem with the following modifications to Rule 1 in Section 5.1:

1. When a packet is retransmitted, set a marker at the in-flight packet with the largest sequence number (snd_nxt).
2. When a partial ACK is received, the following value is calculated $D = (M - A)/S$, where M and A are the sequence numbers of the marker and the partial ACK respectively, and S is the segment size. Retransmit the next lost packet if D is at least three. Otherwise, send a new packet and more duplicate ACKs must be received in order to initiate the retransmission. The number of additional duplicate ACKs is $3 - D$.

These two rules are used recursively until a complete ACK is received. This modification assumes that most out-of-order delivery is no more than three packets and therefore is consistent with the original Reno.

6. Additional Recovery Optimizations for Avoiding Retransmission Timeouts

This section describes the rest of our proposed improvements to Reno TCP and its variants in detail.

6.1 TCP's Self-Clocking Property

TCP's round-trip time boundaries can be detected by counting the number of returning ACKs. This is similar to congestion avoidance in which the congestion window is increased by one packet per RTT, except that we use duplicate ACKs during the recovery phase. To explain our idea, for the moment, assume that delayed ACK is disabled, that TCP does not change its congestion window size, and that the network does not drop packets. If the current window size is one packet, then the time between two consecutive ACKs is exactly one RTT (including queueing and processing delays). If the window size is two packets, then the time between every other received ACKs is one RTT. In general, if the window size is W, then the time between ACK_N and ACK_{N+W} is one RTT.

For the purpose of this paper, we only use the ACK-clock during recovery in which the reference congestion window is static. More specifically, if the congestion window is W when the first loss is detected, then we consider W worth of ACKs as the first RTT and W/2 worth of ACKs as each additional RTT. The receiver is required to send one ACK for each non-consecutive data packet received.

6.2 Recovery of Lost Retransmissions Using TCP's ACK-Clock

This section explains how we use ACK-clock to recover lost retransmissions. As depicted by Figure 1, lost retransmissions account for about 5% of the timeouts in the traces we examined. Reno, New-Reno and SACK do not deal with lost retransmissions. Figure 3 shows a simulation trace of a SACK TCP session. When the congestion window reached 34 packets (snd_nxt - snd_una), packets 638, 639, and 640 were dropped, marked by three triangles around point (1530, 638). SACK TCP quickly retransmitted packet 638 after receiving three duplicate ACKs at point (1553, 638). The retransmissions of packets 639 and 640 were delayed until (1583, 639) because TCP had to flush half of the packets on the fly in order to shrink its congestion window by one-half. This was done by waiting for 17 duplicated ACKs (14 plus the original 3 dupacks). However, both packets were sent as soon as the congestion window was open again. Each packet was sent upon receiving one ACK. Once all three lost packets were retransmitted, new packets started to fill the pipe due to the inflated congestion window by the conservation of packets rule starting from (1587, 672). The square points in the figure show the packets actually arrived at the receiver. Notice that the retransmission for packet 638 did not make it, although retransmitted packets 639 and 640 did (near point (1602, 639)). As a consequence, snd_una did not get increased and eventually the slow timer went off (not shown in the figure).

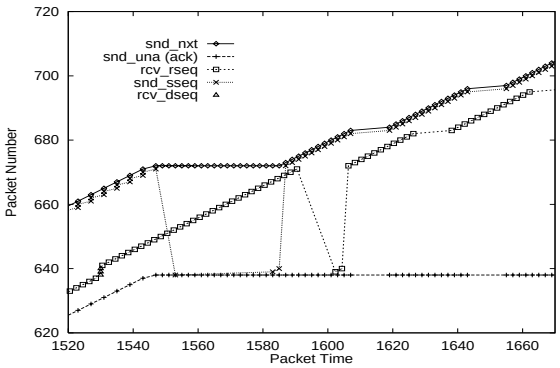

Figure 3. The retransmission for lost packet 638 was dropped by the gateway. The SACK TCP connection fell into a retransmission time out.

We propose using TCP's ACK-clock to time out lost retransmissions. If the clock shows one RTT has gone by and we have received three additional duplicate ACKs since the end of the RTT, the current packet pointed by snd_una should be retransmitted *again*. The three additional dupacks is a conservative measure against out-of-order packets. When a positive ACK comes back and additional lost packets have not been recovered, the ACK-clock value is recorded so that the next unacknowledged packet can be timed.

Implementing the ACK-clock is straightforward. The receiver sends a duplicate ACK for each non-consecutive packet received. Therefore, the value of dupacks represents exactly the ACK-clock ticks. When a data packet is lost in the network, the corresponding ACK, which should have been generated and delivered to the sender, will be missing. This slows down the ACK-clock. Therefore, one additional tick should be generated for each retransmission.

Figure 4 shows a simulation trace over the same TCP connection when the ACK-clock is used. As before, SACK TCP retransmitted packets 638, 639 and 640, but only the latter two packets arrived at the receiver. When dupacks reached 40 (3+RTT+3), the sender retransmitted packet 638 again at point (1623, 638) and the packet successfully arrived at the receiver at point (1642, 638). Consequently, this caused a huge right shift for the sender's congestion window at point (1659, 688) and the recovery phase terminated. At the time dupacks reached 40, 37

duplicated ACKs had been received, with the other three ticks coming from the original three retransmissions.

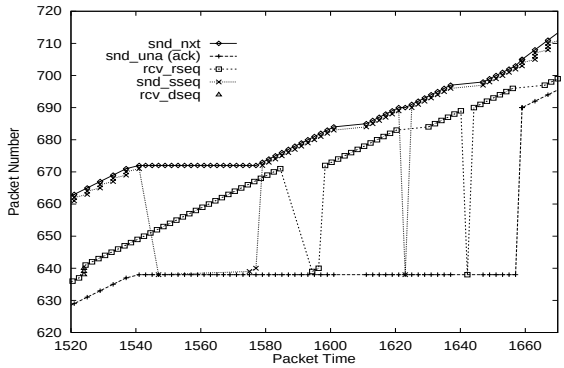

Figure 4. TCP's ACK-clock recovered the lost retransmission which enabled the connection to continue without timeout.

The authors of [9] proposed adding a field to the TCP header that carries a non-decreasing counter generated by the sender and echoed by the receiver. Our approach does not require modifications to the header fields or the receiver. In addition, the scheme of [9] does not consider out-of-order delivery.

6.3 Correcting the ACK-Clock

The ACK-Clock slows down upon packet losses. In the worst case, when TCP loses over half of the packets in the same window *and* the first retransmission, the clock stalls because the Reno sender requires reception of at least W/2 duplicate ACKs in order to send new packets and keep the clock ticking.

To prevent stalling, the sender needs to identify the first RTT boundary without counting returning ACKs. This can be done by inserting markers after the recovery begins. Piggybacked markers are sent by the sender with data packets and echoed by the receiver with ACKs. The only packet sent during the first half of the RTT is the first retransmission. To increase the probability that at least one marker successfully returns to the sender, new packets need to be injected into the network during the first half of the RTT. But this adds aggressiveness to the algorithm, gives less time for the gateway queue to drain, and therefore might cause further congestion. As a compromise, we propose sending one marked new packet upon receiving the fifth and the seventh duplicate ACK respectively.

Packet marking can be done in two different ways without modifying the receiver. The first option is to use time stamps. For each retransmission, a new time stamp should be used to differentiate new packets from old packets. The second approach is specific to SACK TCP. When a new packet (not a retransmitted packet) is sent, the corresponding ACK will carry a SACK block outside the original window. SACK requires that the first block must reflect the change made by the reception of this new packet [12]. Therefore, as long as new packets are injected into the pipe, the returning SACK blocks can serve as non-ambiguous markers.

6.4 Incorporating Back-to-back Recoveries

The recovery process terminates when a positive ACK acknowledges all packets in the original congestion window. Simulations show that the sender may immediately fall into another recovery phase due to the loss of new packets from the inflated window. The connection needs to accumulate another three duplicate ACKs for the second recovery.

As stated in Section 5.2, at the end of a recovery the total number of outstanding packets is about $W = W/2 - c$. If a positive ACK indicates more than $\overline{W}$ packets unacknowledged, then there is at least one more lost packet even if all the packets in the original packets are acknowledged. In SACK TCP, this positive ACK would carry a SACK block. Such information can be used to help start the second recovery early.

When the sender receives an ACK that acknowledges less than the expected number of packets, the sender should retransmit the packet pointed by the incoming ACK. Because this lost packet is sent after the window is reduced by one half, it is a sign of further congestion. The sender should indeed reduce the window again. We set the dupacks counter to two packets and send out a new packet to void structural changes to the implementation. This way, the second recovery will be triggered by the next incoming duplicate ACK. This is demonstrated by a New-Reno TCP connection in Figure 5: the first recovery starts after the sender has received three duplicate ACKs at (150819, 6554) and retransmitted five lost packets. At the end of the recovery when a positive ACK is received at (151349, 6590), the sender sets dupacks counter to two, and transmits a new packet. The second recovery starts when the next ACK comes at (151369, 6578).

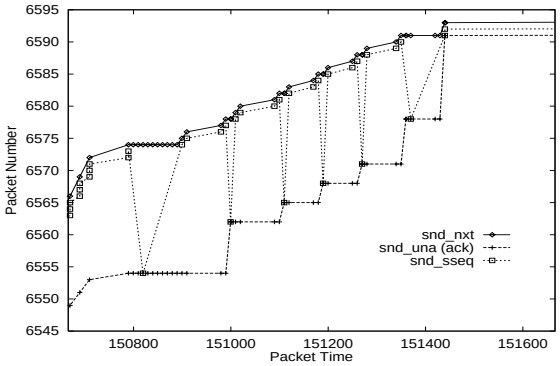

Figure 5. First recovery helps to quickly trigger the second recovery immediately follows.

6.5 Recovering with Limited Data

We have previously demonstrated that, with a window size of W packets and a loss of L packets, SACK is able to recover all losses within one RTT, whereas New-Reno requires L*RTT to finish. During this potentially long period, New-Reno requires data to generate new packets. If the sender does not have enough data or is limited by the receiver's advertised window, then the connection would stall into a RTO.

To prevent this RTO, we propose that the sender should resend the packets starting from the first unacknowledged packet at

snd_una. Although this would cause unnecessary retransmissions in case of out-of-order delivery, it can keep the flywheel going and eventually clock the connection out of recovery. Another approach is to send header-only packets with sequence numbers higher than snd_una. This accomplishes the same goal and preserves link efficiency although it does not contribute to TCP's goodput. Gateways such as RED and FRED that are capable of measuring buffer usage in bytes should favor the second approach.

7. Performance Comparison

We present performance results based on simulations and also on experiments on laboratory test networks.

The simulator used for results of Section 7.1 is a distant descendant of one written for the DARPA/Nortel funded CreditNet ATM Project [10] in 1992. The switch implements pure EPD with no partial packet discard in order to emulate packet switching. We use 512 data-byte TCP packets which are carried in 12 ATM cells. The gateway decides whether to drop a packet upon receiving the first cell in the packet. NetBSD 1.2 TCP Reno source files are used with our modifications described in this paper.

For the rest of this section, we use the terms New-Reno and SACK to represent, respectively, Loss-Sensitive New-Reno and SACK described in Section 5.

7.1 Avoiding RTOs by Net Reno

Senders C1-C4 in Figure 6 each have a (1ms, 103.3 mbps) link to a shared gateway. Sender C-5 has a long link of (16 ms, 31.0 mbps). All five connections go through a shared link (2 ms, 51.7 mbps) to the same host receiver. The five FTP senders always have data in the socket buffers and the congestion windows can grow up to 128 packets. The gateway implements RED [3] (with buffer size = 16 packets, min_{th} = 4, max_{th} = 8, w_q = 0.002, and max_p = 1/50). As discussed in Section 4.1, our trace analysis reveals that the average congestion windows size for TCP connections in the trace is 12 packets. We used small buffers at the gateway so that we can imitate the correct congestion window sizes for both local and cross country connections. RED is used to eliminate the phase effects caused by constant propagation delays and deterministic control algorithms [4].

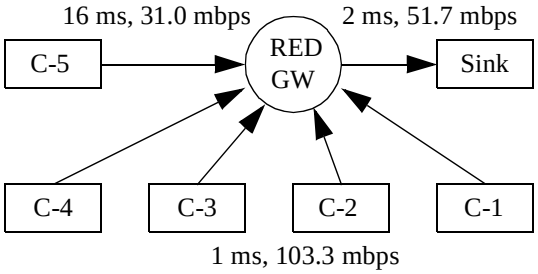

Figure 6. A simulation test network. A long distance TCP competes with four local connections at a RED gateway for the shared link to a sink.

We simulated New-Reno, SACK, and Net Reno (Net-Reno and Net-Reno-SACK). For each of the above TCP implementations, we ran the simulation 10 times, each simulating about 30 seconds of traffic.

Table 1 shows the total number of RTOs each connection has experienced over the ten runs. Notice that the Net Reno optimizations described in this paper have completely eliminated RTOs for both New-Reno and SACK TCP.

	C-1	C-2	C-3	C-4	C-5	total
new-reno	19	33	36	28	13	129
sack	28	35	26	26	8	123
net-reno	0	0	0	0	0	0
net-reno-sack	0	0	0	0	0	0

Table 1. Total number of retransmission timeouts (RTO) over ten runs for the configuration in Figure 6. Columns one to five show the #RTOs for the five connections.

Table 2 shows the total number of packet losses for each connection over the ten runs. Notice that each connection experienced slightly more losses under Net Reno. This is because the connections were constantly generating packets into the network whereas, in the New-Reno and SACK case, the connections were shut off during RTOs. The traffic patterns are smoother with Net Reno than without, due to absence of RTOs. Note that Net Reno did not fall into any RTO even if there were more packet losses.

	C-1	C-2	C-3	C-4	C-5	total
new-reno	2627	2528	2491	2442	708	10796
sack	2691	2645	2562	2504	734	11136
net-reno	2674	2625	2610	2553	695	11157
net-reno-sack	2748	2704	2682	2690	691	11515

Table 2. Total number of packet drops over ten runs for the configuration in Figure 6

7.2 Improving Goodput by Net Reno

In our laboratory at Harvard, we have implemented Net Reno (except packet marking), New-Reno, SACK, and RED under NetBSD 1.2. The SACK implementation was written from scratch, but was partially inspired by a BSDI version [1].

Figure 7 depicts a test setup in our lab. A TCP sender on a 100 mbps fast ethernet sends infinite data to a sink on a 10 mbps ethernet through a RED gateway (buffersize=20, min_{th}=5, max_{th}=10, w_q=1/512, max_q=0.02). A small buffer size is to limit TCP's window size so that the small window optimization can be more effectively demonstrated. All links are a couple of feet long. The maximum segment size is 512 bytes, delayed ACK is enabled, and the maximum TCP socket buffer size is 256KB. All machines are 200 Mhz Intel Pentium Pro with 64 MB RAM. To test the effectiveness of each optimization, we measured the TCP's goodput with various combinations of the optimizations turned on. Each test were run 300 seconds.

Table 3 summarizes average TCP goodput of the experiments. The first column shows the optimizations used for the measurement. "lrxt" stands for lost retransmission. The second and third columns show the goodput for New-Reno. and SACK respectively. For both New-Reno and SACK, the goodput improves as more optimizations are added. The gateway reported loss rate of 1%. For comparison, a standard Reno sender obtained a goodput of 4006 kbps. Surprisingly, SACK fell behind New-

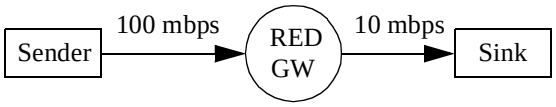

Figure 7. An experimental test network. The TCP sender on a 100 mbps fast ethernet sends infinite data to a sink on a 10 mbps ethernet through a RED gateway. All links are less than two feet.

Reno in all cases. We replaced the sender and the sink machines with two Pentium 100 Mhz machines. Reno and New-Reno provided similar goodput. SACK performance degraded in all cases by about 20%. Our simulation traces of Section 7.1 reveal that New-Reno provided lower total goodput than SACK, but Net-Reno's performance is better than SACK and comparable to that of Net-Reno-SACK.

We speculate from our simulations and real experiments that SACK might not be able to demonstrate significant advantages over Net Reno under moderate congestion and that it will require significantly more CPU power than Reno variants unless it can be efficiently implemented. Previous studies of SACK [2, 13] have all been simulation-based in which end-systems are assumed to have infinite CPU power. Perhaps the necessity of SACK deserves further investigation, given its added complexity. In addition, transporting SACK blocks from the receiver to the sender reduces capacity in the packet payload.

	new-reno	sack
none	5274	5009
smallwnd	6027	5321
lrxt	5782	5406
back-to-back	5584	4985
smallwnd+lrxt	6717	5925
smallwnd+lrxt+back	6739	5842

Table 3. Measured TCP goodput (kbps) for configuration in Figure 7. The first column specifies the optimization(s) used for the measurements in the same row.

8. Conclusions

We have described an ideal network model where TCP end-system algorithms will match ideal gateway congestion control algorithms. We have presented optimizations to TCP's fast recovery strategies to bring end-systems one step further toward the ideal model. These optimizations make TCP more tolerant of packet losses and more robust on avoiding retransmission time-outs. Simulations and experiments have shown that Net Reno will significantly reduce RTOs and improve TCP's goodput.

A Net Reno TCP connection is able to avoid RTOs completely as long as the network keeps at least one packet alive for each round trip time. This requirement is met by FRED gateways [11]. Reducing RTOs also help improve fair resource sharing at gateways. Low bandwidth connections with these improvements are more likely to recover packet losses and avoid throughput deficiency than without them.

While improving efficiency and robustness, Net Reno is less aggressive than New-Reno and SACK. Net Reno strictly obeys the conservation of packets rule, enforces loss-sensitive window reduction, and does not generate packet bursts during recovery. Optimizations of Net Reno TCP are applicable to the SACK option and can also work independently under Reno.

Acknowledgments

This work was supported in part by research funding from Nortel and Sprint. Vern Paxson kindly provided us the N2 trace [16, 17]. His suggestions and the tcpanaly program have saved us a tremendous amount of time.

Reference

[1] Balakrishnan, H., TCP SACK Implementation for BSDI 2.1. ftp://cs.daedalus.berkeley.edu/pub/tcpsack/

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

[3] Floyd, S., Jacobson, V., "Random Early Detection Gateways for Congestion Avoidance," Transactions on Networking, August 1993

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

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

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

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

[8] Jacobson, V., Braden, R., Borman, D. "TCP Extensions for High Performance," RFC1323

[9] Keshav, S. and Morgan S.P., "SMART Retransmission: Performance with Overload and Random Losses," INFOCOM'97

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

[11] Lin, D., Morris, R., "Dynamics of Random Early Detection," SIGCOMM'97

[12] Mathis M., Mahdavi, J., Floyd S., Romanow A., "TCP Selective Acknowledgment Options," RFC2018

[13] Mathis, M., Mahdavi, J., "Forward Acknowledgment: Refining TCP Congestion Control," SIGCOMM'96

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

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

[16] Paxson, V., "End-to-End Internet Packet Dynamics," SIGCOMM'97

[17] Paxson, V., "Automated Packet Trace Analysis of TCP Implementations," SIGCOMM'97

[18] Seshan, S., Stemm. M., Balakrishnan, H., Padmanabhan, V.N., and Randy H. Katz, "TCP Behavior of a Busy Internet Server: Analysis and Improvements," INFOCOM'98

[19] Stevens, W.R., "TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms," RFC2001

[20] Wright, G., Stevens, W. R., TCP/IP ILLUSTRATED VOLUME 2, Addison-Wesley Publishing Co., New York, 1995

[21] Zhang, L., Shenker, S., Clark, D., "Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two-Way Traffic," SIGCOMM'91