Robust TCP Congestion Recovery

Haining Wang and Kang G. Shin

Real-Time Computing Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, MI 48109-2122
{hxw, kgshin}@eecs.umich.edu

ABSTRACT

This paper presents a new robust TCP congestion-recovery scheme to (1) handle bursty packet losses while preserving the self-clocking capability; (2) detect a TCP connection's new equilibrium during congestion recovery, thus improving both link utilization and effective throughput; and (3) make the TCP behavior during congestion recovery very close to that during congestion avoidance, thus "extending" the performance model for congestion avoidance to that for TCP loss recovery. Furthermore, the new recovery scheme requires only a slight modification to the sender side of TCP implementation, thus making it widely deployable.

The performance of the proposed scheme is evaluated for scenarios with many TCP flows under the drop-tail and RED gateways in the presence of bursty packet losses. The evaluation results show that the new scheme achieves at least as much performance improvements as TCP SACK and consistently outperforms TCP New-Reno. Moreover, its steady-state TCP behavior is close to the ideal TCP congestion behavior. Since the proposed scheme does not require selective acknowledgments nor receiver modifications, its implementation is much simpler than TCP SACK.

1. INTRODUCTION

The Transmission Control Protocol (TCP) is a reliable, unicast data-transfer protocol used widely for numerous applications, including electronic mail, file transfer, remote login and WWW. The large-scale deployment of TCP in the Internet is due mainly to its robustness in heterogeneous networking environments. The congestion avoidance and control mechanisms of TCP in [10] have made significant impacts on the performance and behavior of the Internet [18, 20]. As the Internet continues to grow rapidly in size and scope, the increasing demand for network resources has increased packet-loss rate in the Internet, and bursty packet losses are reported to be common [18]. Providing a robust congestion-recovery mechanism is, therefore, an important and difficult task for TCP implementation.

Currently, the most widely-used TCP implementation is TCP Reno, which includes slow-start and congestion avoidance [10], as well as fast retransmit and fast recovery [11, 19]. However, TCP Reno is shown to perform poorly in recovering from bursty losses within a window of data packets [6, 13]. When multiple packets within the same window are lost, the fast-recovery algorithm treats each packet loss in a window as an independent congestion signal, thus halving the congestion window multiple times. The TCP Reno's drastic reduction of congestion window size, plus its over-estimation of data packets in flight, inhibits the transmission of new data packets, losing its self-clocking ability. A retransmission timeout is triggered and then slow-start begins to recover from packet losses, causing a substantial performance degradation. Recent Internet measurements [2] show that the majority of timeouts in TCP Reno are caused by bursty packet losses. Also, it is observed [8] that the performance gain of TCP Vegas [3] over TCP Reno is due mainly to TCP Vegas' new techniques for slow-start and congestion recovery, which are closely related to reduction and recovery of bursty packet losses, not the innovative congestion-avoidance mechanism in TCP Vegas. Thus, a robust TCP mechanism that can recover from bursty packet losses without causing timeouts is key to achieve high TCP performance.

Several enhancements to TCP Reno's congestion-recovery mechanism have been proposed, including the modified fast recovery in TCP New-Reno [9], SACK and FACK TCP [13, 14] for recovering from multiple packet losses within the same window of data. While SACK and FACK TCP can efficiently recover from multiple packet losses in a window, they add significant complexity to TCP implementation, both at the sender and receiver sides. The main weakness of SACK and FACK is that they require "cooperative" receivers. Considering hundreds of millions of clients scattered in the Internet, this requirement makes them practically unattractive for large-scale deployment in the Internet.

Recognizing the packet-loss signal indicated by a partial ACK [1], in TCP New-Reno, upon arrival of a partial ACK, the sender retransmits the packet immediately following the partial ACK without taking TCP out of the fast-recovery phase. Therefore, TCP New-Reno has better capability to recover from multiple packet losses in a window than TCP Reno, and it does not require selective acknowledgments. However, its ability to keep the "flywheel" of ACKs and data packets and prevent the loss of self-clocking, depends on the TCP window size at the time when the first packet loss is detected, as well as on the number of packets lost within a window. TCP New-Reno's ability to recover from packet losses is limited by its inherent weaknesses, including:

- In TCP New-Reno, the number of new data packets sent out per round-trip time (RTT) decreases exponentially due to its policy "one new data packet is sent out upon receipt of two duplicate ACKs" during the entire congestion-recovery period. Since TCP New-Reno can only recover from one dropped packet per RTT, this rapid decrease will eventually stop the flow of returning

[1] A partial ACK acknowledges some but not all of the outstanding packets at the start of the previous round-trip time.

ACKs (hence, loss of self-clocking), and a coarse timeout will follow.

- During congestion recovery, TCP New-Reno only passively recovers from the dropped packets. The exponentially-decreased amount of data transmitted during each RTT lowers link utilization even if it does not cause the loss of self-clocking.

- TCP New-Reno cannot detect further data losses that might occur to the new data packets sent out during congestion recovery. It has to resort to another trigger of fast retransmit or a retransmission timeout to detect such packet losses.

To reduce coarse timeouts and improve the effectiveness of fast retransmit under a tiny window condition, *right-edge* recovery [1] has been proposed, in which "one new data packet is sent out upon receipt of each duplicate ACK, instead of two duplicate ACKs." Similarly, Lin and Kung [12] proposed that a new data packet be generated upon each arrival of first two duplicate ACKs. They retain TCP aggressiveness when there is no network congestion. However, the packet conservation rule [10] does not apply when congestion occurs. TCP aggressiveness should be reduced in order to drain the congestion from the network. These transmitted packets on the verge of detection of a packet loss — indicating network congestion — may add more fuel to the "fire" at the congested bottleneck. Also, these enhancements cannot quickly detect further data losses during congestion recovery either. To reduce the occurrence of bursty losses from a window of data, Smooth-start [21] has been proposed as an optimization of the Slow-start algorithm, which is orthogonal to the enhanced recovery schemes.

In this paper, we propose a robust TCP congestion recovery — called *Robust Recovery* (RR) — algorithm to make a TCP flow more robust to bursty packet losses. The key features of RR include:

- The amount of data in flight is accurately measured, since congestion window size (*cwnd*) over-estimates the number of packets in flight during congestion recovery, stalling data transmission.

- RR treats bursty packet losses within a window as a single congestion signal. Like TCP New-Reno, RR exponentially backs off the sending rate after detecting the first packet loss within a window. However, the exponential decrease in the amount of data injected into the network does not last for the entire recovery period. The exponential decrease is applied only during the first RTT of the recovery period, which is consistent with the treatment of single congestion signal.

- By keeping track of the number of new data packets arrived at the TCP receiver in the previous RTT, the TCP sender can detect any further data loss very quickly without triggering fast retransmit or retransmission timeouts. Upon detection of a further data loss, the TCP sender linearly shrinks the pipe size and extends the exit point of RR.

- After the exponential back-off that happens at the first RTT during congestion recovery, as long as no further packet losses are detected, the TCP sender linearly increases the amount of new data transmitted during each RTT while recovering the dropped packet which is indicated by the arrival of a new partial ACK. During this period, similarly to the *right-edge* recovery [1], a new data packet is transmitted upon receipt of each duplicate acknowledgment.

- Congestion recovery is seamlessly switched to congestion avoidance when all outstanding data packets at the beginning of the last RTT have been acknowledged. The big ACK problem, which causes bursty packet transmissions upon exit of congestion recovery, is eliminated.

In addition to recovering the dropped packets, the RR algorithm probes the new equilibrium of a TCP connection during congestion recovery, so as to achieve higher link utilization while recovering the lost packets. Also, RR makes the TCP behavior during congestion recovery very similar to that during congestion avoidance, thereby enabling the performance model for TCP congestion avoidance [15] to represent that for TCP congestion recovery. This allows for accurate prediction of the TCP-consumed bandwidth even without using selective acknowledgments.

The performance benefits of the RR scheme are demonstrated via extensive simulation experiments with the *ns* [16]. Our simulation results show that the proposed scheme achieves at least as much performance improvements as TCP SACK and consistently outperforms TCP New-Reno. Furthermore, since it requires neither selective acknowledgments nor receiver modifications, its implementation and deployment is much simpler than that of TCP SACK, and only the servers in the Internet need to be modified slightly, while keeping intact millions of TCP clients scattered in the Internet.

One characteristic of TCP is its dependency on the returning ACKs as the trigger of data transmission and congestion window growth. Similarly, RR relies on the returning duplicate ACKs to maintain self-clocking during congestion recovery. We will elaborate on the effect of ACK losses on TCP congestion recovery in Section 2.3. RR also handles retransmission losses by using timeouts, as is usually done.

The remainder of this paper is organized as follows. Section 2 describes the proposed RR algorithm, and Section 3 presents its performance evaluation results in the presence of drop-tail and RED gateways. Section 4 assesses the fitness of the proposed algorithm to the ideal congestion-avoidance model. Section 5 discusses the incremental deployability of the proposed RR algorithm along with the TCP. Finally, the paper concludes with Section 6.

2. ROBUST TCP CONGESTION RECOVERY

To recover bursty packet losses within a window while preserving the self-clocking capability, we propose a new TCP congestion-recovery (RR) algorithm. In RR, the TCP sender not only recovers from packet losses, but also finds the connection's new equilibrium during congestion recovery. It also makes the recovery behavior of bursty losses within a window of data very close to an ideal congestion-avoidance behavior in which only a single packet loss within a window of data occurs periodically. RR is detailed in the next subsections.

2.1 Accurate Estimation of Data in Flight

One of the key problems with current congestion recovery schemes is that congestion window size (*cwnd*), which represents the outstanding packets at the sender side, is still used as the control "pedal" during congestion recovery.

Table 1: TCP parameters in congestion recovery

Name	Meaning of state variables
ndup	the number of duplicate ACKs received.
snd.una	the sequence number of the first byte of unacknowledged data.
snd.nxt	the sequence number of the first byte of unsent data.

During congestion recovery, the outstanding packets at the sender side can be divided into three groups: *active*, *dormant*, and *dropped*. The active group is the set of data packets that are in transit, which also include the retransmitted packets. The dormant group is the set of data packets that were transmitted during the past RTTs and have already arrived and queued at the receiver, but have not yet been acknowledged. Actually, each dormant packet has caused the receiver to send a duplicate ACK to the sender. The dropped group is the set of data packets that were lost during the past RTTs. Clearly, the outstanding packets at the sender side as a whole do not represent the data packets in the path. Only the active group, in which data packets are in "flight," represents the data packets in the path, since the packets in dormant and dropped groups have left the network either normally or abnormally in the previous RTTs, and do not consume network resources any longer.

Thus, as a measure of the outstanding packets at the sender side, *cwnd* over-estimates the number of packets in the path and is no longer adequate for transmission control during congestion recovery. A new state variable *actnum* is thus introduced to measure the amount of data in the path at each RTT of congestion recovery. During congestion recovery, *actnum* plays the usual role of *cwnd* as the means to provide congestion control at the sender side. Once the congestion-recovery phase ends, the congestion-control responsibility is returned to *cwnd*.

A similar variable *pipe* [6] has been proposed in TCP SACK, which counts the number of outstanding packets in the path, not at the sender side. However, the role of congestion control is still played by *cwnd*. The TCP sender can transmit a data packet only when $pipe < cwnd$. The variable *pipe* just passively estimates the number of outstanding packets in the path. By contrast, *actnum* not only represents the number of outstanding packets in the path but also controls the data-transmission rate. In each RTT during congestion recovery, *actnum* linearly grows or shrinks according to the network condition.

Table 2: State variables in the description of RR

Name	Meaning of state variables
maxseq	the highest sequence number sent so far.
seqno	the sequence number indicated by the currently received ACK.
recover	the highest sequence number sent before receiving the latest dup ACKs.

2.2 Description of Robust Recovery

Several TCP parameters [19] are used to describe RR as listed in Table 1. For a better description of RR, we divide it into two sub-phases — *retreat* and *probe* — and three transient states — *entrance*, *exit*, and *occurrence* of a further data loss. Figure 1 shows a high-level organization of the RR algorithm. The detailed flowchart of the algorithm is given in Figure 2. The state variables referred in this figure are explained in Table 2.

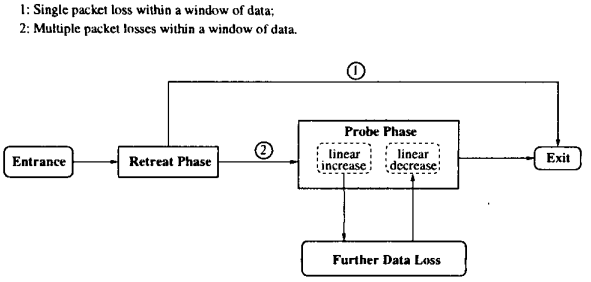


Figure 1: The structure of the RR algorithm

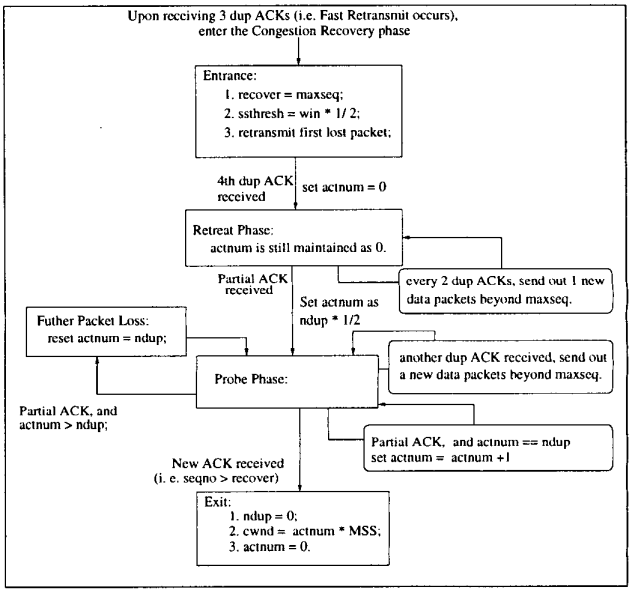


Figure 2: The flowchart of the RR algorithm

As with the fast recovery, RR is triggered by a fast retransmit. However, in RR *cwnd* remains unchanged until the end of congestion recovery, as it is not used for congestion control in RR. At the very beginning of each RTT during congestion recovery, the state variable *ndup* is initialized to 0. As duplicate ACKs arrive within one RTT, *ndup* measures the number of duplicate ACKs received by the sender.

Note upon the arrival of an out-of-sequence data packet at the receiver, the delayed acknowledgment mechanism is off; the receiver immediately sends out an ACK for each received out-of-sequence data packet. The proposed RR algorithm utilizes this to detect further data losses in the probe sub-phase.

If there are no further data losses, the threshold that determines the end of RR's congestion recovery is the same as that of New-Reno and SACK, which is the sequence number of the first byte of unsent data when the first packet loss

was detected by the fast retransmit. The congestion-recovery phase ends as soon as *snd.una* advances to, or beyond, this threshold, indicating that all outstanding data packets at the time of entering the congestion-recovery phase have been acknowledged.

However, if any further data-packet loss is detected, RR advances the threshold used to detect the end of congestion recovery. In particular, the threshold is updated to the value of *snd.nxt* when a further data-packet loss was detected. The congestion-recovery phase ends when *snd.una* advances to, or beyond, this threshold, which may differ significantly from the value of *snd.nxt* when the congestion-recovery phase was entered.

2.2.1 *Retreat Sub-phase*

The retreat sub-phase only covers the first RTT of congestion recovery. In this sub-phase, the TCP sender exponentially backs off its transmission rate. Like in New-Reno, during the retreat sub-phase the sender injects one new data packet for every two duplicate ACKs received. So, the data-transmission rate per RTT is reduced at least by half. The end of the retreat sub-phase is marked by the arrival of the first non-duplicate ACK, i.e., an ACK for a higher-sequence numbered packet; if multiple packets had been lost within the same window, it must be the first partial ACK. During the retreat sub-phase, *actnum* remains to be zero.

When the first partial ACK (i.e., the end of the retreat sub-phase) is received, *actnum* assumes the role of congestion control for the rest of congestion recovery. The variable *actnum* is initially set to $ndup * 1/2$, which is the number of new data packets sent out during the retreat sub-phase.

As can be seen from the above description, the end of the first RTT during congestion recovery is pivotal; the role of congestion control at the sender is transferred from *cwnd* to *actnum*, and the retreat sub-phase ends while the probe sub-phase starts. Note that the first lost packet is recovered in the retreat sub-phase. If only a single packet within a window of data is lost, the TCP sender exits the congestion-recovery phase after the retreat sub-phase. However, if multiple packets in a window of data were lost, all but the first of the lost packets are recovered, one per RTT, in the probe sub-phase, in which the sender linearly adjusts the value of *actnum* according to the network condition. The sending TCP can distinguish the two sub-phases by testing if $actnum = 0$.

2.2.2 *RTT in the Probe Sub-phase*

If there are multiple packet losses within the same window, a key characteristic of the probe sub-phase is that each RTT is distinguished by the receipt of a new partial ACK. At the sender side, the end of the current RTT and the beginning of the next RTT are indicated by the receipt of a new partial ACK. Figure 3 illustrates this feature. Suppose that in a window of data, four packets are dropped and their sequence numbers are 4000, 5000, 7000 and 8000, respectively. The first loss is recovered in the retreat sub-phase, and the rest are recovered in the probe sub-phase, which are represented as 5, 7, 8 in Figure 3.

A packet retransmission is triggered by the arrival of a partial ACK, and this retransmission is acknowledged via the next partial ACK if the packets are delivered in order. If there are no ACK losses, the state variable *ndup* represents the number of new data packets sent out during the last RTT that have been received, because during congestion recovery

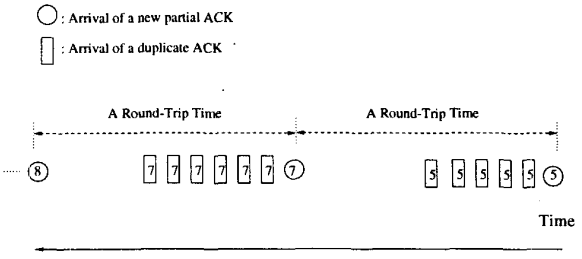


Figure 3: RTT in the probe sub-phase (sender's side)

each received data packet triggers an ACK immediately.

In case of out-of-order delivery, a partial ACK would be generated by the arrival of a new data packet at the receiver, thereby generating one of duplicate ACKs upon arrival of the retransmitted packet. However, since the sender can treat the partial ACK as the ACK of the retransmission, *ndup* still measures the number of new data packets sent during the last RTT that have been received. So, out-of-order delivery does not skew the measurement of the number of new data packets sent during the last RTT that have been received. Unfortunately, this is not valid if there are ACK losses, and we will discuss it later in this section.

2.2.3 *Probe Sub-phase*

At the very beginning of each RTT of the probe sub-phase, the arrival of a partial ACK triggers an immediate retransmission. Upon receiving each duplicate ACK after this partial ACK, the TCP source sends a new data packet. If there are no further data and ACK losses, at the end of this RTT, the value of *ndup* should be equal to that of *actnum*. The reason for this is: *actnum* indicates the number of new data packets that were sent out during the last RTT; and *ndup* represents the number of new data packets sent out during the last RTT that have been received.

Given no ACK losses, at the end of each RTT, a further data loss can be detected by comparing *ndup* with *actnum*. The equality $ndup = actnum$ indicates no further data loss had occurred. However, if $ndup < actnum$, further data losses had occurred during the last RTT. The difference between *ndup* and *actnum* indicates the number of further data losses.

In case of no further data loss, the sender will increment *actnum* by 1 and send one more new data packet for every RTT until the end of congestion recovery, or until a further data loss is detected, which is similar to the congestion-avoidance algorithm. However, if further data losses are detected, *actnum* is reduced to the value of *ndup*. So, reduction of *actnum* linearly depends on the number of further data losses. Since each duplicate ACK triggers the transmission of a new data packet in this RTT, the reduced *actnum* still indicates the number of data packets in flight.

The rationale behind the linear back-off when further data losses are detected is two-fold. The first is to reduce the disturbance caused by ACK losses. The second is to avoid the drastic reduction of in-flight data since the sender exponentially backs off in the retreat sub-phase that happened only a few RTTs ago.

Once further data losses are detected, the exit of the congestion-recovery phase must advance to recover from them. Any further data loss during the congestion-recovery phase can be identified by a new partial ACK beyond the original exit, and

recovered by the subsequent packet retransmission without waiting for two more duplicate ACKs. The exit of congestion recovery extends to the point where all outstanding data packets at the start of last RTT, instead of at the time of entering congestion recovery, have been acknowledged.

After the sender recovers from further data losses, it exits the congestion-recovery phase and enters the congestion-avoidance phase. At that time, the role of congestion control is transferred back to *cwnd*. The sender assigns the current value of *actnum* to *cwnd*. Since *cwnd* is measured in bytes, instead of packets, *cwnd* is set to $actnum \times MSS$. Then *actnum* is set to 0 again. Since the reset value of *cwnd* accurately measures the amount of data packets in flight, the arrival of the new ACK that takes the sender out of congestion recovery only triggers a new packet out, which observes the conservation of packets. So, the big ACK problem that causes sending packets in burst at the exit of congestion recovery has been eliminated.

TCP New-Reno and SACK use a "maxburst" parameter to limit the number of packets that can be sent upon receipt of a single incoming ACK. However, it only limits burstiness but doesn't remove it. Also, it adversely affects bandwidth utilization if the bottleneck has drained all packets; or it causes potential packet losses if the bottleneck is not yet back to its knee area.

2.3 Effect of ACK Losses

Since RR also relies on returning ACKs to inject new data into the network, loss of a string of ACKs will cause RR to lose its self-clocking. Note, however, that RR is more robust to ACK losses than New-Reno. Rare ACK losses cause only a slight negative effect upon congestion recovery. In the probe sub-phase, the transmission rate only linearly decreases when an ACK loss falsely signals a further data loss. Although TCP SACK is less vulnerable to ACK losses, it still has to time out if many ACKs are lost, or the ACK for a retransmission is lost, as shown in [4].

Although data loss on the forward path and the ACK loss on the backward appear uncorrelated in the current Internet [18], we believe that if a fair share is given to each flow at the routers, the loss probability of an ACK packet should be much smaller than that of a data packet. Because the size of ACK packets is usually much smaller than that of data packets — except for those that piggyback other pieces of information — and hence an ACK-packet flow consumes much less network resources than a data-packet flow.

3. PERFORMANCE EVALUATION OF RR

We evaluated the RR algorithm using the *ns-2* [16]. Since New-Reno is known to perform much better than Reno in the presence of multiple packet losses, we focused on the performance comparison among RR, Tahoe, New-Reno, and SACK TCP. The performance evaluation is based on effective throughput, which is a commonly-used metric for end-to-end protocols.

3.1 The Simulation Setup

The simulated network topology is shown in Figure 4, where S_i (K_i) represents a sending (receiving) host, i = 1, ..., n. $R1$ and $R2$ represent two finite-buffer gateways. Different connections from S_i to K_i share the common bottleneck between $R1$ and $R2$. In our simulation experiments each data packet is 1000 bytes long and the size of an ACK packet is 40 bytes.

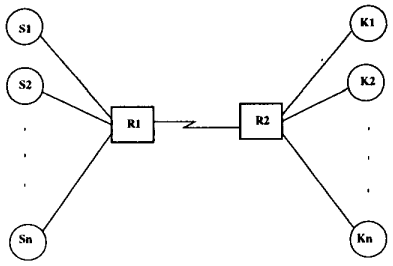


Figure 4: The network topology used for RR experiments

The data traffic in our simulation is generated by FTP. The receiver sends an ACK for every data packet it received. The window size and buffer space at the gateways are measured in number of fixed-size packets, instead of bytes.

3.2 Behavior with Drop-Tail Gateways

The drop-tail gateway with FIFO queueing service has been widely deployed in the Internet, which schedules incoming packets in a FIFO manner and discards incoming packets when the buffer is full. The advantages of the drop-tail gateway are simple, scalable and easy to implement.

Table 3: Simulation configuration

Buffer size	8 packets
TCP maximum window size	30 packets
Bottleneck bandwidth	0.8 Mbps
Bottleneck link delay (1-way)	95 ms
Total delay of side-links (1-way)	5 ms
Side-links bandwidth	10 Mbps

The simulation parameters for RR with drop-tail gateways are summarized in Table 3. There are three TCP connections from S_i to K_i, $i = 1, 2, 3$. Only the first connection is shown in the graphs. The second and the third connections are included to obtain the desired packet loss pattern for the first connection, which only has a limited amount of data to send. Note the buffer size is set to achieve the desired packet loss pattern. If a larger buffer size is set, we can add more background traffic to achieve the same loss pattern. So, the TCP behaviors in each simulation experiment are deterministic, and do not change with different runs as long as the simulation setup and the background traffic remain unchanged.

The simulation results for scenarios with 3 and 6 lost packets within a window of data are plotted in Figure 5, where the effective throughput of the TCP connection during the congestion-recovery period is shown with different TCP recovery schemes.

The RR's effective throughput is significantly higher than that of Tahoe and New-Reno, and slightly higher than that of SACK. Its consistently better performance across 3_drop and 6_drop scenarios indicates RR's resilience to bursty losses within a window of data. Also it is observed that Tahoe is more robust than New-Reno in case of high bursty losses, and achieves a higher effective throughput than New-Reno.

3.3 Behavior with RED gateways

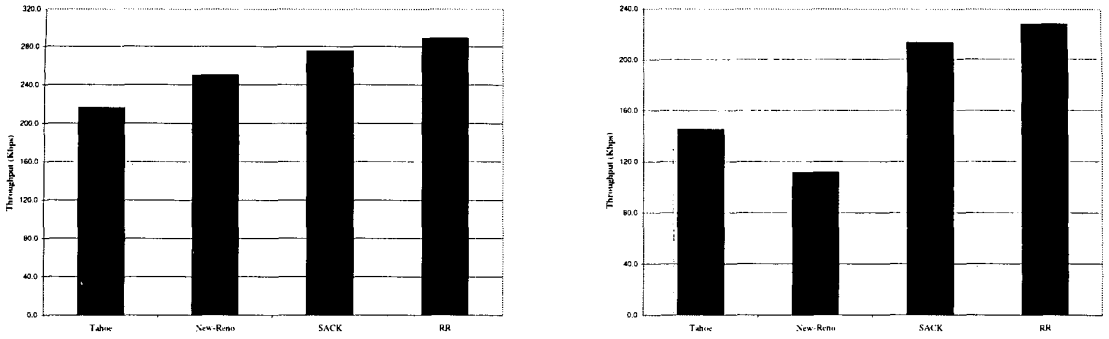


Figure 5: Effective Throughput (Left): 3 packet losses; (Right): 6 packet losses from a window of data

The drop-tail gateway has been shown to arbitrarily distribute packet losses among TCP connections, leading to global synchronization [22], and biasing against bursty connections. The Random Early Detection (RED) [7] gateway has been proposed to alleviate the problems of the drop-tail gateway. The RED gateway detects incipient congestion based on the computation of the average queue size, and randomly drops or marks incoming packets before its buffer is exhausted.

Although RED reduces the frequency of global synchronization and packet-loss rates and minimizes the bias against bursty connections, its performance strongly depends on the behavior of aggregate flows. It is in general difficult to configure a RED gateway into its ideal region as the aggregate flows change with time. Furthermore, RED does not guarantee avoidance of multiple packet losses in a window of data, especially under heavy network congestion. Therefore, given a widely-deployed active queue management mechanisms such as RED, robust TCP congestion recovery is still very important to TCP performance.

Table 4: RED parameter configuration

Minimum threshold	5 packets
Maximum threshold	20 packets
Maximum drop probability	0.02
Weight queue	0.002

The simulation setup for the RED gateway experiments is similar to the previous one, except that RED gateways replace drop-tail gateways and the buffer size is set to 25 packets. The configuration of the RED parameters is summarized in Table 4. Instead of only 3 TCP flows in the drop-tail experiments, 10 TCP flows share the common bottleneck of 0.8 Mbps. The first five TCP flows start at time 0. Then, a new TCP flow starts every 0.5 second. The last one starts at time 2.5s. The duration of the simulation is 6 seconds. All TCP flows have an infinite amount of data to send during the simulation. The purpose of configuring this simulation environment is to generate heavy congestion at the RED gateway. Due to the random drops at gateways, the TCP behaviors in the RED experiments are no longer deterministic. However, except for the case of a random retransmission loss that rarely occurs, randomness does not effect the TCP behaviors in recovering from bursty losses.

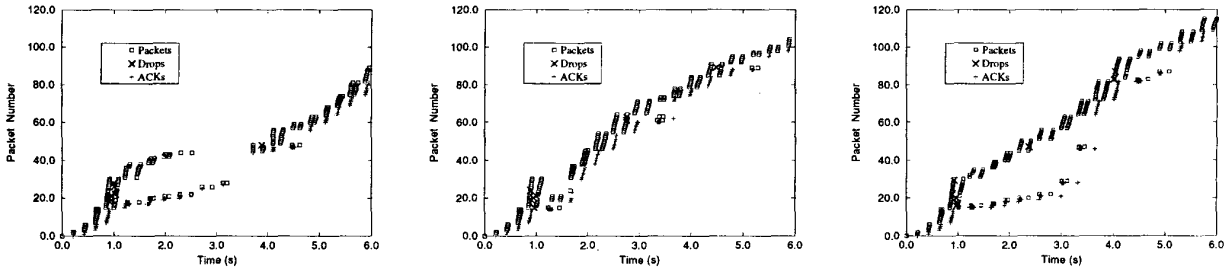
To clearly tell the difference of recovery behavior between the current congestion-recovery mechanisms and RR, the standard TCP sequence number plots are used. In each simulation experiment, all TCP flows use the same congestion-recovery mechanism. Since they experience a similar recovery behavior, only the first one is shown in the graphs. Figure 6 depicts the dynamics of the first TCP flow for different TCP congestion-recovery mechanisms. As with the drop-tail gateway, RR achieves the highest TCP effective throughput when the RED gateway is deployed. As expected, RR's effective throughput is significantly higher than that of Tahoe and New-Reno, and is clearly higher than that of SACK.

As shown in Figure 6, bursty packet losses occur after *cwnd* reaches 16. RR recovers the dropped packets during the next RTT while transmitting new data packets. During the recovery period, a further packet loss occurs at time 2.37s and the lost packet is then retransmitted around time 3.33s. Upon receipt of the new ACK for packet 64 at time 3.65s, the TCP sender leaves the congestion-recovery phase and enters congestion-avoidance phase. However, two packet losses within a window around time 4.0s make the TCP sender enter the congestion-recovery phase again. It will then switch back to congestion-avoidance when the new ACK for packet 98 is received at time 5.30s.

Figure 6 (a) clearly shows that in New-Reno, the exponential reduction in the new data transmission stops the flow of returning ACKs and stalls the transmission of new data packets. Before the receipt of the new ACK that takes the TCP sender out of the recovery phase, only a retransmissions and a new partial ACK flow around the path, which significantly degrades the link utilization.

4. FITTING THE SQUARE-ROOT MODEL

To analytically characterize the throughput of a TCP connection in steady-state as a function of packet-loss rate and RTT, a model has been proposed to describe the macroscopic behavior of the TCP congestion-avoidance algorithm [15]. It assumes that no retransmission timeouts exist with a persistent source and a sufficient receiver window. Under this assumption, the model gives an upper bound on the bandwidth of a TCP connection that can be achieved for a given random packet-loss rate. The estimation of an ideal achievable



(a). New-Reno (b). SACK (c). Robust Recovery

Figure 6: Simulation results for a scenario with RED gateways

throughput is:

$$BW = \frac{MSS}{RTT}\frac{C}{\sqrt{p}}$$

where p is the packet-loss rate, and C is a constant that lumps several factors into one term, including the ACK strategy.

It has been shown in [4, 15] that TCP SACK is much closer to the ideal congestion behavior than TCP Reno. Therefore, in this paper we only compare the fitness of RR with that of TCP SACK against the model. The simulation environment is the same as that in Section 3, except that the simulation length is 100 seconds. Only one TCP connection is active during the simulation, and its start-up phase is ignored. Artificial losses are introduced at the gateway $R1$. The uniform random packet-loss rate is varied in each experiment, while the MSS and RTT are fixed. (MSS is set to 1000bytes and RTT is set to 200ms.) Since the receiver sends an ACK for every data packet received, C is set to $\sqrt{\frac{3}{2}}$.

The simulation results are plotted in Figure 7, where the x-axis represents the random packet-loss rate and the y-axis represents $BW * RTT/MSS$ indicating the window size of the TCP connection. From Figure 7, one can see that RR achieves the same level of fitness to the model as SACK does, and its window size is even slightly closer to the upper bound provided by the model.

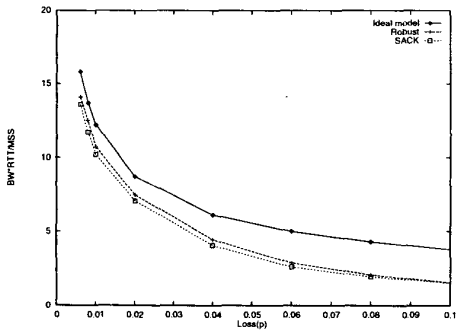


Figure 7: Comparison of fitness to the model

In Figure 7, with the increase of packet-loss rate, the upper bound of window size decreases, and the behavior of both RR and SACK does not fit the model very well. This deviation is due to the timeouts triggered for the following two reasons: 1) the possibility of retransmission loss increases with the increase of packet-loss rate, and the retransmission loss will lead to a retransmission timeout; 2) the small window size reduces the effectiveness of fast retransmit, and hence recovery of some random losses has to resort to retransmission timeouts if three duplicate ACKs are not received.

To predict bandwidth more accurately, Padhye *et al.* [17] proposed a new model, which captures not only the behavior of fast retransmit but also the effect of retransmission timeouts upon throughput, and validated their model over a wider range of loss rates.

5. FAIRNESS

It is clear that RR improves the effectiveness of TCP. Moreover, due to RR's fitness to the ideal TCP congestion model, the stability of TCP is enhanced and the fluctuation of traffic load on the network is reduced. One remaining concern is the fairness of RR.

Owing to its exponential back-off in the retreat sub-phase and linear adjustments in the probe sub-phase, RR strictly follows the AIMD rule [5] and is TCP-friendly. It converges to the optimal point if competing TCP connections have same RTTs. The linear adjustments in the probe sub-phase seamlessly link the congestion-recovery phase with the congestion-avoidance phase, which makes a better use of the bandwidth under-utilized by TCP New-Reno or TCP Reno.

However, to be an incrementally deployable TCP enhancement, RR must interoperate well, in terms of fairness, with existing TCP congestion-recovery strategies, especially TCP Reno. Given less timeouts and linear increase during the probe sub-phase, an obvious concern is that potential unfairness may result from RR when RR competes with other TCP connections whose implementation are TCP Reno.

The simulation environment used for testing fairness is similar to that of testing RR behavior with drop-tail gateways in Section 3.2, except that the buffer size at routers is set to 25 packets. The common bottleneck of 0.8 Mbps is shared by 20 TCP connections. All but one TCP connection have an infinite amount of data to send during the simulation, which are used as the background traffic load, and their starts are staggered. The first TCP connection starts at time 0, then a new TCP connection starts every 0.5 second. All background TCP connections have the same TCP implementation in each experiment. The targeted TCP connection, which has a 100 KBytes file to send from $S20$ to $K20$, starts at 4.8 seconds. The transfer delay and packet loss rate of the targeted TCP connection are measured.

According to the different TCP implementations used by background TCP connections and the targeted TCP connection, the simulation experiments are categorized into four cases. The simulation results of these four cases are summarized in Table 5.

Table 5: Performance of the targeted TCP connection

Case	Targeted	Background	Latency	Loss Rate
I	Reno	Renos	33.3s	15%
II	Reno	RRs	30.1s	12%
III	RR	RRs	25.6s	13%
IV	RR	Renos	18.0s	11%

The simulation results show that when the targeted TCP Reno connection competes with other TCP connections that are implemented with RR, its transfer delay and packet loss rate are lower than those of competing with homogeneous connections that are also implemented with TCP Reno. The performance of the targeted TCP-Reno connection is improved when the background traffic is changed from Reno to RR, instead of being degraded. The reduced global synchronization and traffic fluctuation are the keys to the TCP improvement in Case 2 and 3, which is caused by the background traffic of RR. It demonstrates that RR is TCP-friendly and does not bias against a less aggressive TCP connection. Its fitness to the square-root model shown in Section 4 also confirms its TCP-friendliness.

As shown in Table 5, when a single RR competes with heterogeneous connections that are implemented with TCP Reno, it achieves shorter transfer delay and less packet loss rate. However, the high bandwidth consumption of RR does not result from taking the bandwidth away from TCP-Reno's connections. In the homogeneous scenario, the consumed bandwidth of each TCP Reno connection is only 24 Kbps and the total bandwidth consumption by 20 TCP-Reno connections is 480 Kbps. The unused bandwidth at the bottleneck is 320 Kbps. In Case 4, the achieved bandwidth of RR is 44 Kbps, which is slightly higher than the fair share of 40 Kbps, but there is still a 300 Kbps unused bandwidth. Therefore, RR simply makes a better use of the bandwidth that is underutilized by TCP Reno.

6. CONCLUSION

This paper proposes and evaluates a new congestion-recovery mechanism, called Robust Recovery (RR), to improve the performance of TCP flows in the presence of bursty packet losses. RR is evaluated by simulation, and found to be able to recover from multiple packet losses in a window of data without any significant performance degradation. Also, RR makes the TCP behavior during congestion recovery very close to that during congestion avoidance, thereby extending the performance model for TCP congestion avoidance to represent that for TCP congestion recovery as well. This allows for accurate prediction of the TCP-consumed bandwidth even if selective acknowledgments are not used.

In addition to the performance advantages offered by RR, it offers an implementation advantage. In particular, RR is much simpler than SACK TCP, and does not require selective acknowledgments or receiver modification. Moreover, RR is shown to interoperate well, in terms of fairness, with existing TCP congestion-recovery strategies, making it possible to deploy RR incrementally in the Internet.

REFERENCES

[1] H. Balakrishnan, V. Padmanabhan, S. Seshan, M. Stemm, and R. Katz, "TCP Behavior of a Busy Internet Server: Analysis and Improvement", *Proceedings of IEEE INFOCOM'98*, San Francisco, CA, March 1998.

[2] J. Bolliger, U. Hengartner, and T. Gross, "The Effectiveness of End-to-End Congestion Control Mechanisms" *Technical Report No. 313*, Dept. Computer Science, ETH Zürich, February 1999.

[3] L. Brakmo, S. O'Malley, and L. Peterson, "TCP Vegas: New Technique for Congestion Detection and Avoidance", *Proceedings of ACM SIGCOMM'94*, London, UK, August 1994.

[4] R. Bruyeron, B. Hemon, L. Zhang, "Experimentations with TCP Selective Acknowledgment", *ACM Computer Communication Review*, Vol. 28, No. 2, April 1998.

[5] D. Chiu and R. Jain, "Analysis of the Increase and Decrease Algorithm for Congestion Avoidance in Computer Networks", *Computer Networks and ISDN Systems*, 17:1-14, 1989.

[6] K. Fall and S. Floyd, "Simulation-based Comparisons of Tahoe, Reno, and SACK TCP", *ACM Computer Communication Review*, Vol. 26, No. 3, July 1996.

[7] S. Floyd and V. Jacobson, "Random Early Detection gateways for Congestion Avoidance", *IEEE/ACM Transactions on Networking*, Vol. 1, No. 4, August 1993.

[8] U. Hengartner, J. Bolliger, and T. Gross, "TCP Vegas Revisited", *Proceedings of IEEE INFOCOMM'2000*, Tel-Aviv, Israel, March 2000.

[9] J. Hoe, "Improving the Start-up Behavior of a Congestion Control Scheme for TCP", *Proceedings of ACM SIGCOMM'96*, Stanford, CA, August 1996.

[10] V. Jacobson, "Congestion Avoidance and Control", *Proceedings of ACM SIGCOMM'88*, Stanford, CA, August 1988.

[11] V. Jacobson, "Berkeley TCP Evolution from 4.3-Tahoe to 4.3-Reno", *Proceedings of the Eighteenth Internet Engineering Task Force*, 1990.

[12] D. Lin and H. T. Kung, "TCP Fast Recovery Strategies: Analysis and Improvements", *Proceedings of IEEE INFOCOM'98*, San Francisco, CA , March 1998.

[13] M. Mathis and J. Mahdavi, "Forward Acknowledgment (FACK): Refining TCP Congestion Control", *Proceedings of ACM SIGCOMM'96*, Stanford, CA, August 1996.

[14] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "TCP Selective Acknowledgment Option", Internet Draft, work in progress, May 1996.

[15] M. Mathis, J. Semke, J. Mahdavi and T. Ott, "The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm", *ACM Computer Communication Review*, Vol. 27, No. 3, July 1997.

[16] S. McCanne and S. Floyd, ns-LBNL Network Simulator. Obtain via: http://www-nrg.ee.lbl.gov/ns/.

[17] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose, "Modeling TCP Throughput: A Simple Model and its Empirical Validation", *Proceedings of ACM SIGCOMM'98*, Vancouver, Canada, September 1998.

[18] V. Paxson, "End-to-End Internet Packet Dynamics", *Proceedings of ACM SIGCOMM'97*, Cannes, France, September 1997.

[19] W. Stevens, *TCP/IP Illustrated*, Volume 1. Addison-Wesley Publishing Company, 1994.

[20] K. Thompson, G. J. Miller, and R. Wilder, "Wide-Area Internet Traffic Patterns and Characteristics", *IEEE Network*, Vol. 11, No. 6, November/December 1997.

[21] H. Wang, H. Xin, D. Reeves, and K. Shin, " A Simple Refinement of Slow-Start of TCP Congestion Control", *Proceedings of IEEE Symposium on Computers and Communications'2000*, Antibes, France, July 2000.

[22] L. Zhang, S. Shenker, and D. Clark, "Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two Way Traffic", *Proceedings of ACM SIGCOMM'91*, Zürich, Switzerland, September 1991.