To appear in IEEE ICNP 2003, Atlanta, GA, USA.

RR-TCP: A Reordering-Robust TCP with DSACK

Ming Zhang*
Princeton University

Brad Karp*
Intel Research Pittsburgh
Carnegie Mellon University

Sally Floyd
ICSI

Larry Peterson
Princeton University

Abstract

TCP performs poorly on paths that reorder packets significantly, where it misinterprets out-of-order delivery as packet loss. The sender responds with a fast retransmit though no actual loss has occurred. These repeated false fast retransmits *keep the sender's window small, and severely degrade the throughput it attains. Requiring nearly in-order delivery needlessly restricts and complicates Internet routing systems and routers. Such beneficial systems as multi-path routing and parallel packet switches are difficult to deploy in a way that preserves ordering. Toward a more reordering-tolerant Internet architecture, we present enhancements to TCP that improve the protocol's robustness to reordered and delayed packets. We extend the sender to* detect and recover from *false fast retransmits using DSACK information, and to* avoid false fast retransmits proactively*, by adaptively varying* dupthresh*. Our algorithm is the first that adaptively balances increasing* dupthresh*, to avoid false fast retransmits, and limiting the growth of* dupthresh*, to avoid unnecessary timeouts. Finally, we demonstrate that TCP's RTO estimator tolerates delayed packets poorly, and present enhancements to it that ensure it is sufficiently conservative, without using timestamps or additional TCP header bits. Our simulations show that these enhancements significantly improve TCP's performance over paths that reorder or delay packets.*

1. Introduction and motivation

In today's Internet, deployment of systems that introduce packet reordering in their normal course of operation, regardless of their other benefits, is strongly ill-advised. This taboo derives from the poor throughput TCP achieves under reordering, and the predominance of TCP traffic on the Internet. We seek to end this restriction on the Internet architecture by *enhancing TCP to improve its robustness on network paths that reorder packets.* In this paper, we describe a Reordering-Robust TCP (RR-TCP).

To the extent that reordering occurs today, it is generally perceived as a transient malfunction, or as an indication that a technology is maladapted for use with TCP. Many authors have reported on the causes of reordering in today's Internet. Route oscillation for a destination among routes with different round-trip times (RTTs) may cause reordering [16]. Routers have been observed to cease forwarding while processing a routing update, and intersperse the delayed packets with new arrivals, causing reordering [17]. Bennett *et al.* [4] show that MAE-East reordered packets frequently when it striped packets across multiple links between neighboring switches. Satellite links have very long RTTs, typically on the order of several hundred milliseconds. To keep the pipe full, link-layer retransmission protocols for such links must continue sending subsequent packets while awaiting an ACK or NAK for a previously sent packet. Here, a link-layer retransmission is reordered by however many packets were sent between the original transmission of that packet and the return of the ACK or NAK [18].

We wish to make clear that the primary motivation for our work is to enable future deployment of novel, beneficial systems that today cannot be deployed because they introduce *frequent* and/or *severe* reordering. The algorithms we propose also improve TCP's performance when reordering occurs on the present-day Internet. To argue for the applicability of our algorithms in both these cases, we simulate them across a comprehensive range of reordering frequencies and severities. Two examples of important future systems that motivate our work are:

Multi-Path Routing: Routing a TCP flow's packets over multiple routes with distinct bottlenecks will increase the total end-to-end bandwidth available to the flow. Overlay networks are poised to offer this functionality. But the resulting divergent routes can easily have RTTs that differ sufficiently to cause significant packet reordering. While a multi-path routing scheme could still offer benefit by always routing a single flow's packets on one route, this approach doesn't let one flow use the total available capacity on all routes, and requires the router dividing packets among routes to use flow identifier information in making per-packet forwarding decisions.

Parallelism in Packet Forwarding: A promising technique for building inexpensive high-speed routers is to use

* Karp and Zhang began this work while at ICSI.
Authors' email addresses: {mzhang, llp}@cs.princeton.edu, bkarp@cs.cmu.edu, floyd@icir.org

parallel forwarding and/or switching hardware. Successive packets that arrive at a router, even on the same link, may be forwarded and/or switched simultaneously by independent hardware. This simple parallel approach ignores ordering between packets processed simultaneously, and introduces reordering when packets require different processing delays. Enforcing in-order delivery in such architectures significantly increases their complexity [6], and in the case of switching, eliminates much of the cost savings of the parallel hardware approach.

TCP's inability to distinguish reordering from packet loss causes the protocol to perform poorly on paths that deliver packets out of order. Losses, falsely detected or genuine, cause TCP to send more slowly. Yet mistaking reordering for loss is not fundamental to window-based congestion control. Rather, it is an artifact of TCP's fast retransmit mechanism, which arbitrarily concludes that a packet must have been lost if it is still missing at the receiver after *three* packets sent later have arrived at the receiver. On a network path that reorders packets more than minimally, this choice of three is too aggressive in concluding loss; waiting longer before concluding loss might reveal that the packet wasn't lost at all, but only delayed en route.

Clearly, there is a tension between timely detection of loss and being forgiving in case packets arrive out-of-order. We demonstrate in this work that a properly extended TCP sender can achieve significantly improved robustness to reordering, without sending significantly more aggressively in the face of genuine congestion.

We present a control loop for dynamically adapting the trigger for TCP's fast retransmit, based on measurements the sender takes of the reordering behavior of the network, as provided by ACKs, selective ACKs (SACKs) [13] and duplicate SACKS (DSACKs) [8]. The control loop also uses fast retransmit events and timeout events as feedback, and aims to *maximize the throughput* achieved by a connection. A key difference between our control loop and the prior work in this area is that we use information concerning timeouts to avoid making TCP *too tolerant* of reordering; we show in Sections 4 and 5 that excessive reordering tolerance leads to timeouts and *reduced* throughput. The control loop presented in this paper uses more state at the sender than do previous approaches; we seek to demonstrate that this additional state confers greater performance benefits than prior, lower-cost approaches. Finally, we propose a simple change to TCP's RTO estimator that renders it sufficiently conservative on paths that significantly delay packets, *without* requiring TCP header information beyond that provided by standard DSACK TCP. The end result, RR-TCP, offers significantly enhanced throughput on reordering paths, as demonstrated in simulations. Its deployment could substantially loosen the in-order delivery restriction on the Internet architecture.

We proceed in the remainder of this paper as follows: Section 2 describes the phenomenon of *false fast retransmit*, responsible for TCP's poor performance on reordering paths, and reports on other dynamics of fast retransmit that affect performance on reordering paths. Section 3 catalogs related work in TCP and reordering. In Section 4, we present the algorithms used in RR-TCP's reordering adaptation control loop. A detailed performance evaluation of RR-TCP in simulation can be found in Section 5. To conclude, we suggest future avenues of research in Section 6, and summarize our findings in Section 6.

2. Fast retransmit and reordering

The fast retransmit and fast recovery mechanism [2] allows the TCP sender to detect loss without experiencing a retransmit timeout, by retransmitting a packet after receiving three *duplicate acknowledgements* (duplicate ACKs). This value of three for the *dupthresh* parameter is fixed in the fast retransmit specification, which states, "Since TCP does not know whether a duplicate ACK is caused by a lost segment or just a reordering of segments, it waits for a small number of duplicate ACKs to be received." [2] Clearly, this choice assumes that the network hardly ever perturbs a packet's position in the stream sent by the sender by more than three packets' distance; otherwise, fast retransmit would incorrectly conclude that loss has occurred, and halve the congestion window needlessly.

We view reordering as a process that causes a packet or packets to be delayed, such that they arrive later than packets *sent later* by the sender. In the eyes of TCP, delaying a packet and "accelerating" a packet are equivalent; both cause a packet with a higher sequence number to arrive before a packet with a lower sequence number. Throughout this work, we use the convention of referring to reordering as being caused by the delay of packets.

This section describes the interactions between fast retransmit and reordering, and how they affect TCP's performance. We restrict our attention herein to the effects of reordering on the TCP sender's retransmission behavior and window size. The sender enhancements we will propose are robust *both* to reordered data packets and reordered ACKs.[1]

2.1. False fast retransmit

The *false fast retransmit* phenomenon [9] limits TCP's throughput when the network reorders a connection's packets. When a packet in the sender's window is delayed by many packet times, and subsequent packets are not delayed, the delayed packet arrives after the subsequent ones. A cluster of duplicate ACKs will arrive at the sender, triggering fast retransmit of the delayed packet, and causing the window size to be halved, though no loss has occurred. As reorderings recur, this process will repeat, and beat down the sender's window, resulting in a severe throughput reduction

[1] Reordered ACKs have been shown to have other effects outside the scope of this paper, *e.g.*, increasing the burstiness of the sender.

unwarranted by the network's congestion status. On network paths that reorder, a value of *dupthresh* greater than the distance in packets a segment is displaced in the packet stream will prevent false fast retransmits.

The duplicate-SACK (DSACK) extension [8] to SACK TCP [13] is a useful tool for making the TCP sender more robust to reordering. DSACK reports to the sender when duplicate packets (for the same sequence number) arrive at the receiver, but does not specify the sender's actions. Should the sender incorrectly conclude that a packet was dropped rather than reordered, and retransmit that packet, it will later learn so from a DSACK, once both the original packet and the spurious retransmission of it arrive at the receiver. If the receiver doesn't implement DSACK, the sender won't be able to detect false fast retransmits and behaves identically with standard SACK.

In the case of false fast retransmit, upon receiving a DSACK for the retransmitted packet, the sender can undo the window reduction made at retransmission time. Prior work on TCP's response under reordering, described in Section 3, investigates this recovery strategy; our work additionally *avoids* false fast retransmits.

2.2. Risks of increasing *dupthresh*

Unfortunately, increasing *dupthresh* is not without cost. While progressively greater *dupthresh* values prevent the TCP sender from wrongly concluding that progressively longer reorderings are losses, these greater *dupthresh* values make the TCP sender *respond more slowly after real packet drops*. Should *dupthresh* grow large, risks include:

- generation of one-second-minimum timeouts, because insufficient duplicate ACKs return to trigger fast retransmit after a loss.
- significantly increased end-to-end delay for dropped packets—even if enough duplicate ACKs return, the fast retransmit will be delayed until they all arrive; interactive transfers or video over TCP, as done by the popular RealVideo application, are intolerant of spikes in end-to-end packet delay;
- delayed response of TCP to congestion, also because fast retransmit is delayed and limited transmit (described in the next section) may send multiple congestion windows (*cwnd*) of additional packets as duplicate ACKs arrive.

There is a clear tradeoff between avoiding false fast retransmits and the above-enumerated risks. A scheme for adapting *dupthresh* must balance these opposing goals. Increasing *dupthresh* alone is insufficiently adaptive; an algorithm for reducing *dupthresh* is also needed.

2.3. Limited transmit and *dupthresh*

One further detail of fast retransmit bears mention: the *limited transmit* extension, currently a Proposed Standard in the IETF [1], under which the sender is permitted to send one new data packet for each of the first two duplicate ACKs that arrive. This behavior helps the sender accumulate three duplicate ACKs after a loss, so that it can use fast retransmit even when its window is small. While limited transmit sends two packets beyond the sender's current congestion window, it follows the ACK clock, and so does not significantly deviate from TCP's congestion control model.

The proposers of limited transmit specifically avoid considering a *dupthresh* of any value other than three. If we are to consider greater *dupthresh* values, the degree to which the sender sends beyond its current congestion window will increase. Together with the work we present on varying *dupthresh*, we also extend limited transmit to permit sending up to one additional congestion window's worth of packets. Note that this limit only matters when *dupthresh* is greater than the current congestion window size; otherwise, it is *dupthresh* itself that limits the number of packets sent with limited transmit. There are reasons for attention to the number of packets limited transmit allows beyond the current congestion window. They include:

- Flows that do not use limited transmit will send fewer packets after a loss than flows that do.
- The extra transmissions of limited transmit may cause retransmissions by competing flows to be dropped.
- In cases where there is very little statistical multiplexing at the bottleneck, a flow using limited transmit may be substantially responsible for the congestion at the bottleneck, and may send packets under limited transmit that will only congest the bottleneck further.
- Limited transmit can delay fast retransmit's reduction of the congestion window—after a single loss, if *dupthresh* is very great, sending k congestion windows of packets with limited transmit delays window reduction by k RTTs.

These statements all relate to the aggressiveness of a TCP that uses limited transmit, *vs.* that of a TCP that does not. No matter how many packets limited transmit sends, it is always self-clocking. Bansal *et al.* [3] show that under appreciable statistical multiplexing, self-clocking congestion control protocols do not cause high packet drop rates at a bottleneck for protracted periods. Their result applies to protocols that respond slowly to congestion, such as TFRC [7], which requires four to eight RTTs to cut its sending rate by half, even under persistent congestion. Thus, our permitting limited transmit to send up to one ACK-clocked additional congestion window's worth of data when *dupthresh* is great does not make TCP significantly more aggressive; it merely slides TCP in the direction of being a more slowly responding congestion control protocol.

Finally, note that the number of packet transmissions permitted by limited transmit determines the maximum reordering length for which an increased *dupthresh* is useful in improving TCP's throughput. If *dupthresh* becomes greater than the total number of packets that limited transmit is willing to send, the sender will be unable to keep the pipe full during reorderings longer than the bound on limited transmit. Thus, there is a tradeoff between the maximum reordering length for which TCP will be robust, and restricting the delay of TCP's response to loss by limiting the bound on limited transmit.

3. Related work

This section describes prior work in improving TCP's performance on networks that delay or reorder packets, and differentiates our work from this earlier research.

Ludwig and Katz [12] study both spurious timeouts and spurious fast retransmits. The work does not consider adaptation of *dupthresh* to *avoid* spurious retransmissions; it only backs out window reductions that are found to have been made in response to packet delays or reorderings. They do not use DSACK to notify the sender of duplicates. Rather, they propose two alternatives. The first is to use TCP timestamps on every packet [10], such that the different timestamps on the original and retransmission return in ACKs, and reveal to the sender that the ACKs are for distinct transmissions of the same packet. The second is to use a reserved bit in the TCP header, dubbed the RTX bit [12], to mark every packet as either an original or a retransmission. An ACK reflects the RTX bit value of the data packet it acknowledges. Timestamps add twelve bytes to each packet's length, and render present-day header compression schemes ineffective; requiring their use is therefore strongly undesirable. The RTX bit uses one of only three remaining unused bits in the TCP header; primarily for this reason, it has been withdrawn from the IETF standards process. We examine the performance difference between the RTX bit and DSACK in Section 5.1.1.

Blanton and Allman [5] use DSACK information to restore the sender's congestion window size after detecting false fast retransmits, *and* to increase *dupthresh* with the aim of avoiding future false fast retransmits. Their schemes all hold less state at the sender than the ones we propose, but they do not address the negative effects of too great a *dupthresh*. They present no strategy for reducing *dupthresh* other than resetting it to three packets upon a timeout. In their work, they consider six strategies for increasing *dupthresh*; later in the paper, we refer to them with short textual tags of the form DSACK-BL-*x*, where *x* is a different string for each of their six variants.

4. Algorithms

This section describes algorithms for enhancing TCP's robustness to reordering. We begin with a simple scheme that the sender uses to sample the reordering length distribution experienced by the data packets sent on a connection. Next, we show how to use this distribution to increase *dupthresh* in such a way as to avoid false fast retransmits. While increasing *dupthresh* on lossless paths yields improved throughput, this simplistic strategy is problematic on lossy paths, where a dropped packet that could have triggered a fast retransmit with the default *dupthresh* of 3 may instead cause a timeout.

Motivated by this difficulty, we present a strategy for *reducing dupthresh* adaptively in response to timeouts. The combined increase/decrease scheme for *dupthresh* balances the tradeoff between false fast retransmits and timeouts using a cost function that quantifies the reduction in throughput associated with a false fast retransmit, *vs.* the reduction in throughput associated with a timeout. The result is a heuristic for adapting *dupthresh* in response to false fast retransmit and timeout events.

We complement the *dupthresh* adaptation algorithm with an improvement to TCP's RTO estimator that eliminates a sampling bias that causes RTOs to be too aggressive on paths that delay packets. Today's SACK TCP avoids sampling RTTs for all retransmitted packets, in accordance with Karn's Algorithm [11], because the sender cannot know whether an ACK matches a data packet's original transmission or its retransmission. Because delayed (reordered) packets are likely to trigger retransmissions, they are less likely to be included in TCP's RTO estimator, and produce RTO estimates that are too short. We enhance the RTO estimator to include RTT samples for falsely retransmitted packets, *without* requiring the use of timestamps or any other extension to the TCP header. We must omit the details of the enhanced RTO estimator and its evaluation in simulation in this paper in the interest of brevity; the interested reader is referred to [19].

4.1. Reordering-related state: the scoreboard

The SACK TCP scoreboard data structure [14] stores per-packet state at the sender concerning recently transmitted packets. It offers a natural framework for storing per-packet reordering-related information: whether a fast retransmit is false, the duration of false fast retransmit, and the reordering length a packet experiences. We record each fast retransmit's starting time and window reduction amount in the retransmitted packets' scoreboard entry. If the fast retransmit is later identified as false, we record the interval between the start and end of the false fast retransmit, during which the window was unnecessarily halved.

Measurement of reordering length is more nuanced. There are two phases to sampling the distribution of reordering lengths experienced by packets: *measuring* the reordering length for each packet, and *aggregating* these samples into a histogram of reordering lengths recently observed on the connection's path.

It is important to note that the extensions we describe to the scoreboard here *do not change the asymptotic storage or computation requirements of scoreboard maintainance.* The techniques we describe are not significantly greater in cost from SACK TCP, which is already widely deployed [15].

We must elide a detailed discussion because of space constraints, and refer the interested reader to [19].

4.1.1. One packet's reordering length For the sender to avoid a false fast retransmit after a packet is reordered, its *dupthresh* must be greater than the number of duplicate ACKs the reordering generates. When packet i is delayed, one duplicate ACK will arrive at the sender for each packet $i+1 \ldots i+k$ that arrives at the receiver before packet i. Thus, delaying packet i can generate k duplicate ACKs: the difference between the highest packet number ACKed or SACKed so far and the number of the delayed packet.[2] [3]

Intuitively, when a packet is delayed and arrives out-of-order, there is a "hole" in the sender's scoreboard for that packet; the sender receives SACKs for packets sent later than the delayed packet *before* receiving the ACK or SACK for the delayed packet. For the moment, let us assume that ACKs are not dropped or reordered, and that delayed acknowledgement is not used. Here, the arrival of one packet at the receiver triggers one cumulative or selective ACK, that communicates the receipt of that one packet. In this case, a returning ACK or SACK block for a delayed packet must always close exactly a one-packet hole in the scoreboard. This hole must lie between the previously acknowledged packet with the greatest contiguous packet number and the greatest packet number in the newly arriving ACK or SACK. Thus, where i is the greatest packet number in the newly arriving ACK or SACK, a sender measures the reordering length r by scanning the scoreboard as follows:

```
c = greatest contiguously ACKed packet number
m = greatest ACK or SACK number received so far
n = 0
foreach packet k such that c < k ≤ i
    if current ACK newly ACKs or SACKs k
    then
        h = k        // found a hole
        n = n + 1
    endif
end
if n == 1 then
    r = m − h
endif
```

When ACKs are dropped or reordered, a single returning ACK can close more than one hole in the scoreboard. The test for $n == 1$ ignores samples where a returning ACK closes more than one hole, such that erroneous samples are not caused by dropped or reordered ACKs. This test makes the reordering length measurement mechanism robust against reordered or dropped ACKs; in Section 5.3, we demonstrate in simulation that reordered ACKs do not affect our TCP sender's throughput.

[2] We use packet numbers here for clarity of exposition; the correspondence between packet and sequence numbers is abstracted away by the scoreboard.

[3] It's possible to use the number of duplicate ACKs to measure reordering length. But the number of duplicate ACKs is affected strongly by delayed, dropped or reordered ACKs. The method described below is not, and also allows us to measure multiple reordering events within a single window of packets.

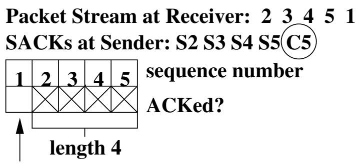


Figure 1. Reordering measurement using the scoreboard.

Figure 1 shows a simple example of the reordering length measurement mechanism. Here, packet 1 is reordered to arrive after packets $2\ldots5$, for which SACKs return to the sender. The scoreboard is shown at the moment the cumulative ACK for packet 5 returns. The sender finds the hole at packet 1, and concludes a reordering length of 4.

When a packet is retransmitted, there is an ambiguity as to whether its ACK corresponds to the original transmission or the retransmission. When no DSACK for the retransmission returns, the sender discards the reordering length sample for the retransmitted packet, because that packet was lost, not reordered. When a DSACK does return, the sender pairs the original transmission with the first ACK to return, and the retransmission with the second, computes the reordering length for each, and takes their mean as a conservative approximation to the reordering length encountered by both packets. Note that this mean has the same value, regardless of which of the two possible pairings of ACKs with data packets is used.

TCP receivers are not intended to use delayed ACKs when they receive out-of-order packets [2], to promote the accumulation of duplicate ACKs at the sender. A look at the FreeBSD 4.3 TCP code reveals that a receiver only uses delayed ACK when *both* the newly arrived segment is contiguous with previously acknowledged data, *and* the reassembly queue (containing packets with sequence numbers greater than those contiguously acknowledged) is empty. We conclude that delayed ACKs are extremely unlikely during a reordering epoch, between receipt of the first non-contiguous packet at the receiver, and the emptying of the reassembly queue.

4.1.2. Aggregating into the reordering histogram Samples of reordering lengths from each transmitted packet are stored in a *reordering histogram* as ACKs return to the sender. The bins in the histogram are reordering lengths; they count the number of packets that have experienced each reordering length between one and a configurable maximum. The histogram tracks the reordering history for a configurable period of time. Each reordering event stored in the histogram holds a timestamp. Periodically, events older than the history period are deleted from the histogram. The histogram provides details of the reordering distribution to our *dupthresh* adjustment algorithms. Blanton and Allman explore alternatives that accumulate less state [5]; our goal is to demonstrate the *best-case* performance improvement that can be attained by using the most accurate and detailed reordering information. We stress that a histogram makes no assumption about the distribution of reordering lengths; for any persistent reordering process, it will provide percentiles of reordering lengths.

Note that for each retransmission of a packet, the sender must wait for the return of both an ACK and a DSACK, as described previously, before being able to determine the packet's reordering length. If no DSACK returns, we assume the original or retransmitted packet was dropped.

4.2. Avoiding false fast retransmits: increasing dupthresh

The reordering histogram summarizes the distribution of reorderings experienced by a connection's packets. A simple strategy for avoiding false fast retransmits is to choose the desired percentage of reorderings for which false fast retransmits are to be avoided, and to set *dupthresh* such that it equals that percentile value in the reordering length cumulative distribution. That is, if 90% of reordering events are of 8 packets or fewer, a *dupthresh* of 9 will avoid 90% of false fast retransmits. Even with a fixed percentile choice, *dupthresh* may vary over time, as the reordering histogram's contents change in accordance with the reordering behavior of the connection's path.

We refer to this algorithm as DSACK-FA, for **F**alse Fast Retransmit **A**voidance, and the percentage of reorderings the algorithm avoids as the *FA ratio*.

4.3. Avoiding timeouts: adapting the FA ratio

As described in Section 2.2, increasing *dupthresh* is not without cost. Potential negative effects of a too-large *dupthresh* include timeouts, long end-to-end delays for packets retransmitted after drops, and a delayed response of TCP to congestion. To avoid these ills, an algorithm for reducing *dupthresh* is also needed.

Rather than directly varying *dupthresh*, we instead propose varying the FA ratio. Increasing the FA ratio will increase *dupthresh*, while decreasing the FA ratio will decrease *dupthresh*. A natural approach to building a control loop that governs adaptation of the FA ratio is to consider the relative costs of false fast retransmits and timeouts, and to set the FA ratio accordingly.

Cost Function: Timeouts Both false fast retransmits and timeouts have opportunity costs in needlessly missed packet transmissions. A false fast retransmit causes a window reduction by half, and this smaller window prevails until DSACKs return, and permit reinstatement of the previous window value. In contrast, timeouts have two main costs: the idle period after the full window of packets has been sent, but before the timer expires; and slow start, during which the congestion window size must grow from one, and will be smaller than half the previous congestion window size for multiple RTTs. We distinguish between two types of timeouts:

- **False timeouts,** for which a DSACK eventually returns, occur when delay, not loss, causes an timeout.
- **True timeouts,** for which no DSACK returns, occur when loss causes an timeout.

Suppose that a TCP connection has a steady state window size W, a smoothed RTT of R, and a retransmission timeout period of T. TCP will send a maximum of $k \times cwnd$ additional packets while duplicate ACKs return under limited transmit [1], which permits TCP to send new data in response to returning duplicate ACKs to ease triggering of fast retransmit.

A true timeout consists of three phases: an idle period, slow start, and linear increase beyond the halved window. Fast retransmits reduce throughput less than timeouts; they consist only of halving the window and linear increase. Thus, the *additional* cost of suffering a true timeout rather than a fast retransmit is only the idle period and slow start.

During the idle period, the sender misses the opportunity to transmit $W\frac{T}{R} - W(1+k)$ packets. During slow start up to $W/2$, the sender misses the opportunity to send $(W-1)+(W-2)+\ldots+(W-W/2+1)+(W-W/2)$ packets, or:

$$\sum_{i=0}^{\log_2 W-1} [W-2^i] = W(\log_2 W - 1) + 1$$

packets. Thus, the total cost of a true timeout is:

$$C(\text{true timeout}) = W(\frac{T}{R} + \log_2 W - k - 2) + 1$$

packets. After a false timeout, when a DSACK returns, the pre-timeout congestion window is restored. Thus, there is no period of opportunity cost during linear increase of the congestion window, and the cost of a false timeout with window restoration under DSACK is roughly *equal* to that of a true timeout.[4]

Cost Function: False Fast Retransmits The transmission opportunity cost after a false fast retransmit depends on the interval required for the sender to receive the DSACK that identifies the fast retransmit as false. Recall from Section 4.1 that the scoreboard measures, for each false fast retransmit, the duration of the wrongly reduced window (between the window reduction and the return of the DSACK, if any). We maintain an exponentially weighted moving average of this false fast retransmit duration, D. When $D = R$, the cost of a false fast retransmit is merely $W/2$; the window was halved unnecessarily for only one RTT. When $D > R$, however, the cost is greater, as the reduced window is in effect for a longer period. Note that each subsequent RTT costs less, as linear increase of the congestion

[4]The costs are only approximately equal; the DSACK information may be delayed in returning, in which case linear increase may begin before the old window can be restored.

window progresses, until after $W/2$ RTTs, when the original window value has been restored. Thus, for $k = \lceil D/R \rceil$, the cost of a false fast retransmit is bounded above by $(W - \frac{W}{2}) + (W - \frac{W}{2} - 1) + \ldots + (W - \frac{W}{2} - (k-1))$, or:

$$C(\text{false fast retransmit}) \leq \sum_{i=0}^{k-1} [\frac{W}{2} - i] = \frac{k(W-k+1)}{2}$$

packets. Note that we limit k to $W/2$ regardless of D, to cap the cost appropriately.

Because D and R are estimated as exponentially weighted moving averages, their values are not instantaneously accurate. The actual cost of a false fast retransmit lies *between* the cost for $k_{\text{high}} = \lceil D/R \rceil$ and the cost for $k_{\text{low}} = \lfloor D/R \rfloor$. Rather than using a discrete single value of k, such that a small change in D or R can provoke a disproportionate change in $C(\text{false fast retransmit})$, we linearly interpolate between k_{low} and k_{high}.

Cost Function: Limited Transmit The bound on limited transmit also introduces an opportunity cost in idle time when the FA ratio (and thus *dupthresh*) are great. In this situation, it may happen that limited transmit is insufficient to accumulate the number of duplicate ACKs to trigger a fast retransmit, and an idle period results. This is not to say that limited transmit is problematic. On the contrary, when *dupthresh* is so large, the idle time provides downward pressure on the FA ratio without incurring the more severe cost associated with a timeout.

More specifically, when a large *dupthresh* is in effect and the RTT is small in relation to the minimum RTO, the sender may remain idle after it exhausts limited transmit. Yet no timeout may occur, as the delayed packet can easily be acknowledged before the timer expires. The idle period indicates *dupthresh* has grown too large.

When the sender exhausts limited transmit, we store the time this event occurs. If an ACK returns that permits the window to advance once again, and no timeout has occurred, the idle period I is the difference between the time the ACK returns and the stored time limited transmit was exhausted. During this period, we count the number of further duplicate ACKs that return, d. These duplicate ACKs partially filled the pipe at the time the idle period began, and are not part of the opportunity cost during the idle period. The cost of this idle period is thus $C(\text{limited transmit}) = \frac{I}{R}W - d$.

Here W is the steady state window size, and R is the smoothed RTT. By reducing the FA ratio based on the cost of this idle period, we risk increasing the number of false fast retransmits experienced. Thus, we only decrease the FA ratio after a limited-transmit-induced idle period if $C(\text{limited transmit}) > C(\text{false fast retransmit})$.

Adapting the FA Ratio: Combined Cost Function Having defined the cost functions associated with timeouts, false fast retransmits, and limited transmit, we now explain how they are used together to vary the FA ratio. Let the parameter S be the fundamental step by which we adapt the FA ratio. In the results presented herein, we use an S of 0.01, chosen to permit fine adjustment of the FA ratio by the control loop. Rules for adapting the FA ratio are:

Upon every false fast retransmit, increase the FA ratio by S.

Upon every timeout, decrease the FA ratio by

$$\frac{C(\text{timeout})}{C(\text{false fast retransmit})}S$$

Upon every limited-transmit-induced idle period, provided $C(\text{limited transmit}) > C(\text{false fast retransmit})$, decrease the FA ratio by

$$\frac{C(\text{limited transmit})}{C(\text{false fast retransmit})}S$$

These rules heuristically adapt the FA ratio (and thus *dupthresh*) in a way that maximizes throughput for a connection experiencing reordering. False fast retransmits cause a gradual increase in the FA ratio. Timeouts and significant idle periods triggered by great *dupthresh* values cause the FA ratio to decrease in proportion to the relative throughput reductions they create, as compared with the throughput reduction associated with a false fast retransmit. *dupthresh* is set to the FA ratio's percentile value in the reordering length cumulative distribution. We refer to the algorithm that uses these cost functions and rules to adapt the FA ratio as DSACK-TA, for **T**imeout **A**voidance.

These collected enhancements to the sender result in a TCP that achieves significantly greater throughput than a standard SACK TCP, but it is important to note that this disparity does not directly imply any fairness difficulties between a sender using these DSACK enhancements and a sender using standard SACK TCP. It is the reordering that causes standard SACK TCP to perform poorly. A DSACK-enhanced sender doesn't *cause* reordering, and so is not responsible for the poor throughput SACK achieves under reordering. In cases where a DSACK-enhanced TCP competes with a SACK TCP on a reordering path, replacing the DSACK TCP with a SACK TCP should not materially improve the performance of the other SACK TCP.

Algorithm Name	Description
SACK	Standard SACK
DSACK-R	DSACK + FFR recovery
DSACK-FA	DSACK-R + fixed FA ratio
DSACK-TA	DSACK-FA + timeout avoidance

Table 1. Algorithms compared in simulation.

5. Experimental evaluation

This section presents simulation results to demonstrate the improvement DSACK-based algorithms make to TCP's performance over paths that reorder or delay packets. We compare the performance of several variants of DSACK,

To appear in IEEE ICNP 2003, Atlanta, GA, USA.

Parameter	Value
Initial FA ratio	90%ile sampled
RTT Histogram ratio	99.8%ile sampled
Minimum *dupthresh*	3 pkts
Maximum *dupthresh*	64 pkts
Maximum sending window (*maxwnd*)	50 pkts
Limited transmit bound	$1\times$ *cwnd*
Reordering length sample lifetime	80 s
α in EWMA of FFR duration	$\frac{1}{8}$

Table 2. Simulation parameters. FFR denotes false fast retransmit.

both those proposed in Section 4 and those proposed by others. Table 1 summarizes the algorithms in simulation. We simulate these algorithms in the *ns-2* network simulator [14], version 2.1b8. To introduce reordering, we extended *ns-2* to delay a configurable percentage of packets that traverse a link. Independently of the delay, we also control the drop rate associated with a link.

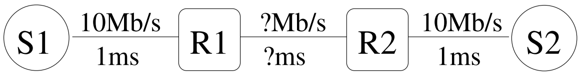


Figure 2. Simulated network topology.

We simulate a wide variety of delay (reordering) distributions to demonstrate the value of our algorithms. Because of space limitations, nearly all the results we present in this paper are for normally distributed reordering lengths. In practice, our algorithms work similarly well for the other distributions we simulate. In fact, its effectiveness does not rely on any assumption about the delay distributions, as explained in Section 4.1.2; we refer the interested reader to [19] for detailed measurements.

Our simulations consist of a single, long-lived TCP flow traversing the network shown in Figure 2. The flow lasts 1000 seconds. All data points in simulation results plots are means of five runs with different pseudorandom number generator seeds for the packet reordering process, except where otherwise noted. Reordering events and packet drops are introduced at bottleneck link (R_1, R_2), whose link speed and propagation delay we vary. In all simulations, the maximum window size M permitted by the sender is fixed at 50 packets. To achieve precise control of the loss behavior of the bottleneck, where the bottleneck link has RTT R, we set the capacity S (in packets per second) of link (R_1, R_2) such that $S = M/R$. Thus, when we don't introduce a controlled packet delay or dropping process at the link, a TCP flow achieves throughput S, and the steady state window size will be exactly $M = 50$. Were M greater, the bottleneck link would periodically cause packet drops as TCP's congestion window varied in saw-tooth fashion bracketing 50 packets. We show other simulation parameters in Table 2.

5.1. False fast retransmit avoidance

We first show how use of DSACK at the sender improves TCP's performance by detecting, recovering from, and avoiding false fast retransmits. Here, the delay of link (R_1, R_2) is 50 ms. Packet delays are normally distributed, with a mean of 25 ms and standard deviation of 8 ms, such that most packets selected for delay are delayed between 0 ms and 50 ms. Note that these parameters represent typical Internet link delays, and relatively mild reordering.

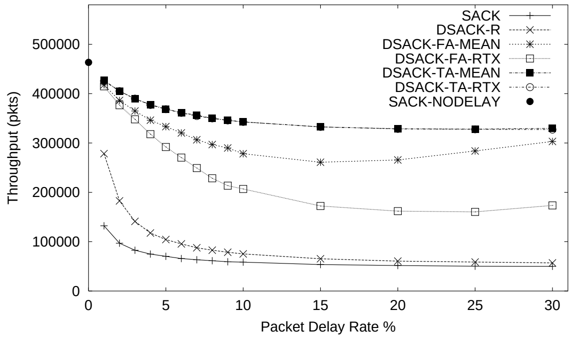


Figure 3. Throughput *vs.* fraction of delayed packets.

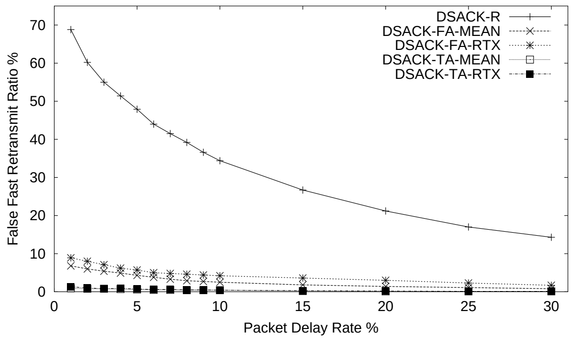


Figure 4. False fast retransmit ratio *vs.* fraction of delayed packets.

5.1.1. Varying packet delay rate First, we vary the percentage of delayed packets from 1% to 30%, without introducing packet drops. As shown in Figure 3[5], as more packets are delayed, the throughput of SACK drops rapidly, but that of DSACK-FA and -TA is better. DSACK-TA performs best, as its throughput decreases much more slowly than that of the other schemes.

DSACK-FA and -TA avoid false fast retransmits by varying *dupthresh*. Figure 4 reveals that the fraction of packets resent with fast retransmit for which retransmission is false under DSACK-FA is less than 10%. DSACK-TA prevents still more false fast retransmits. We do not show SACK's fraction of false fast retransmits here because the SACK implementation does not detect these events.

[5] Note that SACK-NODELAY is a single point plotted at $x = 0$, and that DSACK-TA-MEAN and DSACK-TA-RTX are plotted atop one another.

In cases with virtually no timeouts, such as in this simulation, DSACK-TA adjusts the FA ratio to 99% so that most false fast retransmits are avoided. SACK-NODELAY shows the ideal throughput TCP achieves when there is no packet delay. DSACK-TA can maintain over 71% of the throughput possible without packet delays, even when 30% of packets are delayed.

The -RTX and -MEAN variants of the TA and FA algorithms show a comparison of two different strategies for resolving the ambiguity in matching ACKs with retransmitted packets. The -RTX variant uses the RTX bit [12] to mark retransmitted packets and their ACKs differently. The -MEAN variant uses the technique described in Section 4.1.1, where no RTX bit is used. In all subsequent simulations, plots with no -MEAN or -RTX suffix use the -MEAN variant. The performance of DSACK-TA-MEAN is comparable to that of DSACK-TA-RTX, but the FA-MEAN variant performs a bit better than the FA-RTX variant. DSACK-FA-MEAN averages the two transmitted packets' measured reordering lengths. In this simulation, the reordering length of the original packet is most often shorter than that of the retransmitted packet because the original packet's ACK usually arrives earlier. Thus, DSACK-FA-MEAN tends to measure slightly longer reorderings, and thus selects a slightly larger *dupthresh*, which in turn causes fewer false fast retransmits. The result is higher throughput for DSACK-FA-MEAN.

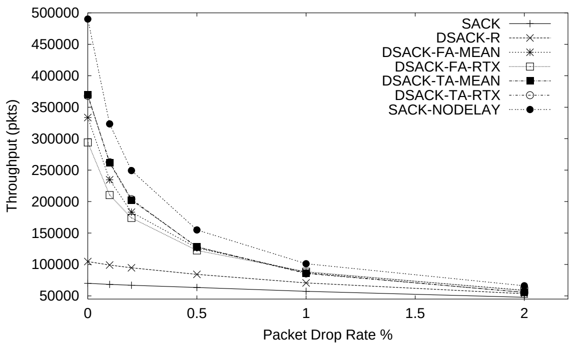


Figure 5. Throughput *vs.* drop rate.

5.1.2. Varying packet drop rate Next, we study the behavior of DSACK when packets are both delayed and lost. In this example, 5% of packets are delayed, and the packet drop rate varies between 0% and 2%. As shown in Figure 5, the throughput achieved by DSACK-TA and DSACK-FA decreases sharply as the loss rate increases. As one expects, all TCP variants suffer reduced throughput under loss. In the case of the DSACK variants, a fast retransmit can be identified as a false fast retransmit *only* when there are no packet losses in that window of packets. As the drop rate increases, it becomes increasingly likely that at least one packet drop occurs within a window. As a result, the percentage of false fast retransmits decreases rapidly, and the performance difference between DSACK and SACK diminishes.

5.2. Timeout avoidance

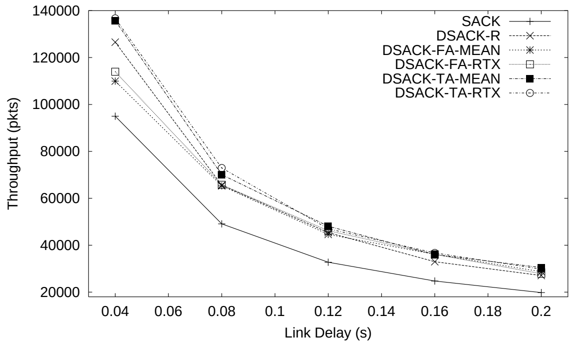
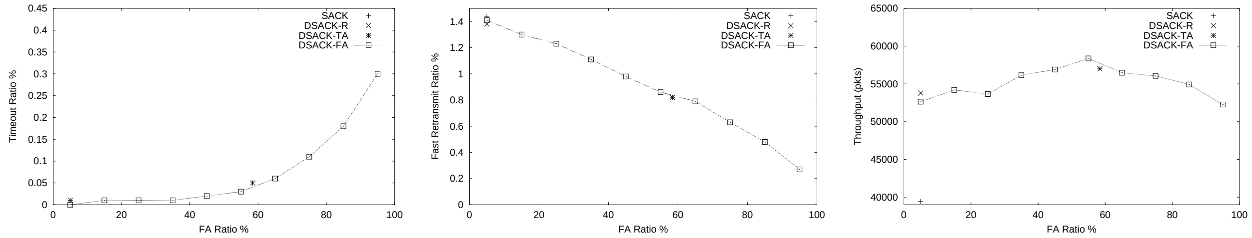


Figure 6. Throughput *vs.* link delay.

Here, we demonstrate the performance benefits of dynamically adapting the FA ratio to balance between false fast retransmits and timeouts. We delay 1.4% of packets and drop 0.6% of packets, and vary the link propagation delay P of (R_1, R_2) between 40 ms and 200 ms. The packet delay time varies uniformly between $[0, 4P]$ (up to 2 RTTs). These parameters represent cases in the upper range of Internet link delays, and moderate packet delay.

As shown in Figure 6, DSACK-TA performs best. But this is not because DSACK-TA causes the lowest percentage of false fast retransmits; DSACK-FA actually causes an even smaller percentage. To examine this relationship more closely, we fix the link delay of (R_1, R_2) at 100 ms, and vary the target FA ratio of DSACK-FA from 95% down to 5%. Figure 7 shows (a) the timeout behavior, (b) the fast retransmit behavior, and (c) the throughput behavior of DSACK-FA under these conditions. In Figure 7a, the fraction of sent packets that encounter timeouts decreases rapidly as the FA ratio decreases from 95% to to 60%, then decreases further only slightly as the FA ratio decreases further. Note that DSACK-TA adaptively chooses an FA ratio of approximately 60%, at this point of diminishing returns below which fewer timeouts are avoided. Figure 7b reveals that as the FA ratio (*dupthresh*) decreases, the actual fraction of fast retransmits will increase. Thus, were DSACK-TA to decrease the FA ratio below 60%, not many timeouts would be avoided, but progressively more fast retransmits would result. Figure 7c shows that DSACK-TA achieves a higher throughput than DSACK-R, which uses a fixed *dupthresh* of 3, and DSACK-FA, which fixes the FA ratio at 90%, because DSACK-TA balances between false fast retransmits and timeouts. Note further in Figure 7c that DSACK-TA adapts *dupthresh* such that it achieves approximately the maximum throughput available among all possible fixed FA ratios.

Figure 6 shows the effect of increasing the propagation delay of link (R_1, R_2). Note that the performance difference between DSACK-TA and DSACK-FA narrows. This phenomenon occurs because as the link delay increases, the idle cost associated with timeout decreases, and the cost difference between a timeout and false fast retransmit does, too.



(a) Fraction of packets that encounter timeouts *vs.* FA ratio.

(b) Fraction of fast retransmitted packets *vs.* FA ratio.

(c) Throughput *vs.* FA ratio.

Figure 7. Timeout avoidance: comparing DSACK-FA and DSACK-TA.

Thus, the performance of DSACK-FA approaches that of DSACK-TA as the link delay increases.

5.3. Robustness to ACK reordering

The mechanisms we propose address the effects of reordered data packets on the sender's window size. Reordering of ACKs may also occur. Because our algorithms use the ACK stream to measure the reordering lengths of data packets, it is important to verify that reordered ACKs do not diminish their effectiveness.[6]

In simulations where we vary the fraction of reordered ACKs between zero and ten percent, we find every DSACK variant offers nearly constant throughput across this entire range of reordered ACK fractions. Thus, reordered ACKs have no significant negative effect on the sender's throughput, for reasons previously explained in Section 4.1.1. There is similarly negligible effect on throughput for the other link delays and data and ACK packet delay distributions we've simulated, as well. Detailed simulation results for ACK reordering appear in [19].

5.4. Robustness for multi-path routing

In Figure 8, we examine DSACK-TA-MEAN's behavior under packet delays similar to those that would be seen if a sender's packets were sent alternately over two paths with different RTTs. If we assume that the RTT of each of the two paths remains fixed, all delayed packets are delayed by the difference between the two paths' RTTs. Here, we examine a case with a 50 ms propagation delay, and simulate 50% of packets being delayed for the same period, representing the RTT difference between the 100 ms RTT path and a longer path. At a delay of zero seconds, all packets are routed on the same path, and there is no reordering. As the delay, and thus the reordering length, increase, DSACK-TA-MEAN continues to offer significantly increased throughput over

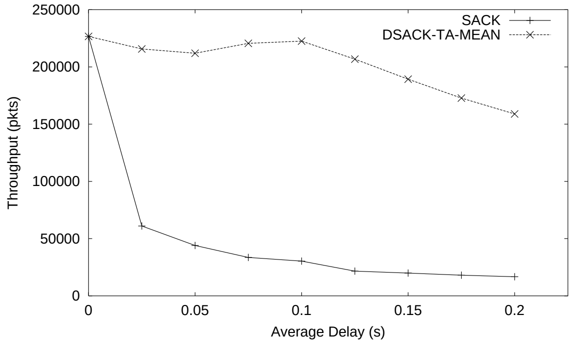


Figure 8. Throughput *vs.* pkt delay.

SACK. Note that the performance advantage of DSACK-TA-MEAN begins to diminish at delays longer than 100 ms; at this point, packets are delayed more than one window's worth. Recall that we restrict limited transmit to one window's worth of packets, to avoid delaying TCP's response to a genuine packet loss. Thus, the performance improvement diminishes because of idle time induced by limited transmit, in accordance with the discussion in Section 2.3. Even with limited transmit of one window and a path RTT difference of two 100 ms RTTs (200 ms), DSACK-TA-MEAN offers a seven-fold throughput improvement over SACK.

5.5. Comparison with prior work

Blanton and Allman propose several techniques for adapting *dupthresh* in response to reordering [5]. They increase *dupthresh* after measuring reorderings, but do not explicitly weigh the tradeoff between false fast retransmits and timeouts. After a timeout, they propose resetting *dupthresh* to 3. This section compares Blanton and Allman's algorithms with our own.

We first characterize the expected behavioral differences between the algorithms. First, in Blanton and Allman's algorithms, *dupthresh* often increases to a great value, often as great as the maximum reordering length seen. This great *dupthresh* value may increase end-to-end delay for dropped packets when the reordering length has a heavy-tailed distribution. When a network path reorders less severely than before, their algorithms without a *dupthresh* decrease strategy

[6] We confine our interest here to the avoidance of false fast retransmits and false timeouts that are our goals in this paper; reordered ACKs have other effects, including increasing the burstiness of the sender, that have been investigated by others previously.

must rely on a timeout to reset *dupthresh* to 3. In comparison, our timeout avoidance algorithm avoids most false fast retransmits, while ignoring rare and extremely long reorderings. Should the reordering length distribution change over time, the histogram reflects any such change, and *dupthresh* changes accordingly.

As timeouts are expensive, Blanton and Allman limit *dupthresh* to 90% of the current congestion window. However, this limit may not always prevent timeouts that could have been avoided with a smaller *dupthresh*—when multiple packets are delayed or lost within a single window, a timeout may be inevitable. The *dupthresh* limit of $0.9 \times$ cwnd can't prevent false fast retransmits in cases where reordering lengths are longer than $0.9 \times$ cwnd, but not long enough to trigger false timeouts with the one-second-minimum RTO. When the congestion window is small, such cases occur frequently.

We use Blanton and Allman's simulator code in *ns-2*.[7] We compare the approaches on a network where link (R_1, R_2) has $P = 50$ ms propagation delay, $S = 4$ Mb/s link capacity, and 1% of packets are delayed according to a normal distribution with mean kP and standard deviation $\frac{k}{3}$. As shown in Figures 9 and 10, we gradually increase k from 0.1 to 4.0, and thus vary the packet delay between 5 ms and 200 ms.

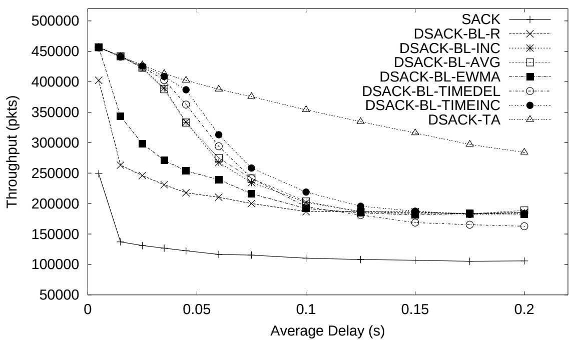


Figure 9. Throughput *vs.* average delay time.

The DSACK-BL-XXX curves give results for Blanton and Allman's algorithms. As shown in Figure 9, when kP is small, all schemes perform similarly better than SACK, but as kP increases, DSACK-TA achieves increasingly higher throughput as compared with all other schemes. Figure 10 shows that the fraction of fast retransmits suffered by DSACK-TA hovers around 0%, whereas the other schemes suffer increasingly from fast retransmits as mean packet delay increases. Here, the $0.9 \times$ cwnd bound on *dupthresh* prevents the other schemes from avoiding false fast retransmits caused by longer reorderings.

We now explore the behavior of Blanton and Allman's algorithms under bursty packet loss. Here, link (R_1, R_2) has

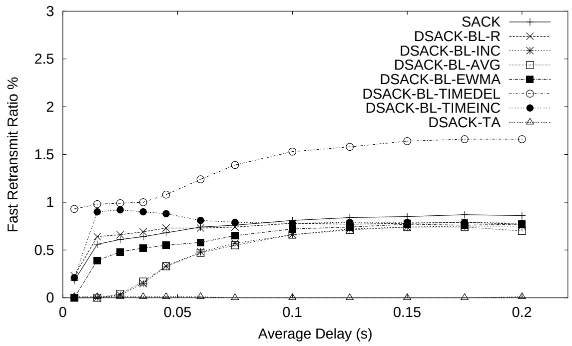


Figure 10. Incidence of fast retransmits *vs.* average delay time.

	No drops Total pkts	Drops Total pkts	No drops FR ratio	Drops FR ratio
DSACK-BL	97184	60708	0.13%	0.48%
DSACK-TA	103770	81916	0.03%	0.19%

Table 3. Throughput and FR ratios, with and without bursty loss.

$P = 200$ ms propagation delay, and 2% of packets are delayed according to a normal distribution with mean 100 ms and standard deviation 33 ms. We further introduce a small fraction (0.02%) of packet drops. Each drop event lasts for a period that varies uniformly in $[300, 400]$ ms, during which all consecutive packets to arrive are dropped. This drop behavior triggers timeouts even with a $0.9 \times$ cwnd bound on *dupthresh*. In Table 3 we see that the throughput of DSACK-BL-INC suffers more than that of DSACK-TA under bursty drops. Each timeout causes DSACK-BL-INC to reset *dupthresh* to 3, so that DSACK-BL-INC loses all its reordering length history. Thereafter, it linearly increases *dupthresh* as it encounters reordering. In contrast, DSACK-TA uses its reordering length histogram to preserves knowledge of the path's characteristics across timeouts.[8] Thus, DSACK-TA suffers fewer false fast retransmits and offers higher throughput.

We have compared all variants of DSACK under an extensive set of network conditions, where we vary the link delay of (R_1, R_2) between $[50, 400]$ ms; the packet drop rate between $[0, 9]$ percent; the fraction of delayed packets between $[0.1, 10]$ percent; and mean packet delays between $[25, 1600]$ ms, using many of the random processes supported in *ns-2*. As expected, DSACK-TA performs best overall because it combines the benefits of false fast retransmit avoidance and timeout avoidance.

6. Conclusion and future work

We have presented RR-TCP, a TCP sender extended to distinguish between reordering and loss, in the interest of

[7] Their code runs in ns-2.1b7, whereas ours runs in ns-2.1b8. In the interest of maximal comparability of results, we used the TCP parameter defaults from 2.1b8 when running their code in 2.1b7.

[8] Plots of *dupthresh vs.* time for the two schemes omitted for brevity; they may be found in [19].

improving TCP's robustness on paths that reorder packets. Our extensions use a histogram of the reordering lengths packets experience to adapt TCP's *dupthresh*, and a control loop to adapt the FA ratio, the fraction of reordering events that the sender should avoid misidentifying as losses. The key novel feature of RR-TCP is its use of *timeout avoidance*; our control loop for varying the FA ratio is mindful not only of the costs of false fast retransmits, but also of the costs of timeouts and idle periods during limited transmit. Our simulations on networks over a wide range of link delays, packet delays, and loss patterns show that RR-TCP consistently improves TCP's throughput significantly in the face of reordering, as compared with both standard SACK TCP and previously published reordering robustness enhancements to SACK TCP.

Our experimental evaluation of RR-TCP reveals much about the nature of the reordering problem. As the loss rate increases, the sender's window is kept small by congestion avoidance, and reordering doesn't limit throughput—congestion does. As the length of reorderings increases beyond the permitted extent of limited transmit, an RR-TCP sender must incur idle periods, and will offer less of a performance improvement over SACK. Limited transmit embodies a fundamental tradeoff between the responsiveness of the sender to congestion and the reordering length a TCP sender can be made to tolerate.

Several avenues bear further investigation. In this paper, we've pursued only sender-side designs that store extra state for each connection in the SACK scoreboard, and in the reordering histogram. On a busy server with many thousands of connections, this additional state may be large. We believe RR-TCP can be built in a receiver-side fashion, whereby the receiver measures reordering, keeps the histogram, applies the *dupthresh* adjustment algorithms, and dynamically informs the sender of this *dupthresh* value, perhaps in a TCP option. This design devolves the reordering-related state requirements from the server to each client, but requires changing the over-the-wire protocol.

We have only considered long-lived flows in the interest of simplifying the evaluation of our algorithms' properties. Many web transfers are short-lived. We believe that sharing reordering state (*i.e.,* the reordering histogram and/or FA ratio) between short-lived flows that occur serially in time will confer the benefits of RR-TCP to short-lived flows. We further believe that there is little to no risk to the network in sharing this state in this way; it is not congestion state, but reordering state, and thus will not cause a sender to send more aggressively than current network conditions permit.

A reordering-robust transport protocol is one step toward viable multi-path routing. But other transport problems in spreading a single flow's packets over multiple paths remain. The different paths packets take may not only have different RTTs, but also different loss rates. Understanding TCP's behavior in such cases will require further study.

RR-TCP is a Reordering-Robust TCP that is safe to deploy. We believe its deployment could substantially loosen the in-order delivery restriction on the Internet architecture. Simulation code for RR-TCP for *ns-2* may be found at `http://www.icir.org/bkarp/RR-TCP/`.

Acknowledgements

Sally Floyd thanks Ethan Blanton and Mark Allman for helpful discussions of the reordering problem. Brad Karp thanks Robert Morris, Mark Handley, and Orion Hodson for their illuminating comments on earlier drafts of this manuscript.

References

[1] M. Allman, H. Balakrishnan, and S. Floyd. Enhancing TCP's loss recovery using limited transmit. *RFC 3042*, Jan. 2001.

[2] M. Allman, V. Paxson, and W. Stevens. TCP congestion control. *RFC 2581*, Apr. 1999.

[3] D. Bansal, H. Balakrishnan, S. Floyd, and S. Shenker. Dynamic behavior of slowly-responsive congestion control algorithms. *Proceedings of ACM SIGCOMM*, Aug. 2001.

[4] J. Bennett, C. Partridge, and N. Shectman. Packet reordering is not pathological network behavior. *IEEE/ACM Transactions on Networking*, 7(6):789–798, Dec. 1999.

[5] E. Blanton and M. Allman. On making TCP more robust to packet reordering. *ACM Computer Communication Review*, 32(1), Jan. 2002.

[6] B. Chen and R. Morris. Flexible control of parallelism in a multiprocessor PC router. *Proceedings of the 2001 USENIX Annual Technical Conference*, pages 333–346, June 2001.

[7] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-based congestion control for unicast applications. *Proceedings of ACM SIGCOMM*, Aug. 2000.

[8] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. An extension to the selective acknowledgement (SACK) option for TCP. *RFC 2883*, July 2000.

[9] J. Hoe. Improving the start-up behavior of a congestion control scheme for TCP. *Proceedings of ACM SIGCOMM*, Aug. 1996.

[10] V. Jacobson, R. Braden, and D. Borman. TCP extensions for high performance. *RFC 1323*, May 1992.

[11] P. Karn and C. Partridge. Estimating round-trip times in reliable transport protocols. *Proceedings of ACM SIGCOMM*, Aug. 1987.

[12] R. Ludwig and R. H. Katz. The Eifel algorithm: making TCP robust against spurious retransmissions. *ACM Computer Communication Review*, 30(1), Jan. 2000.

[13] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP selective acknowledgement options. *RFC 2018*, Oct. 1996.

[14] ns2 (online). http://www.isi.edu/nsnam/ns.

[15] J. Padhye and S. Floyd. On inferring TCP behavior. *Proceedings of ACM SIGCOMM*, Aug. 2001.

[16] V. Paxson. End-to-end routing behavior in the Internet. *Proceedings of ACM SIGCOMM*, Aug. 1996.

[17] V. Paxson. End-to-end Internet packet dynamics. *Proceedings of ACM SIGCOMM*, pages 139–152, Sept. 1997.

[18] C. Ward, H. Choi, and T. Hain. A data link control protocol for LEO satellite networks providing a reliable datagram service. *IEEE/ACM Transactions on Networking*, 3(1):91–103, Feb. 1995.

[19] M. Zhang, B. Karp, S. Floyd, and L. Peterson. RR-TCP: a reordering-robust TCP with DSACK. ICSI Technical Report TR-02-006, July 2002.