RR-TCP: a reordering-robust TCP with DSACK

Conference Paper · November 2003

DOI: 10.1109/ICNP.2003.1249760 · Source: DBLP

4 authors, including:

Ming Zhang
Tsinghua University
107 PUBLICATIONS 7,402 CITATIONS
SEE PROFILE

Sally Floyd
University of California, Berkeley
139 PUBLICATIONS 40,232 CITATIONS
SEE PROFILE

INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center St. • Suite 600 • Berkeley, California 94704-1198 • (510) 666-2900 • FAX (510) 666-2956

RR-TCP: A Reordering-Robust TCP with DSACK

Ming Zhang[*] **Brad Karp**[†] **Sally Floyd**[‡]

Larry Peterson[§]

TR-02-006

July 2002

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. Persistent reordering occasionally occurs on present-day networks. Moreover, TCP's requirement of nearly in-order delivery complicates the design of such beneficial systems as DiffServ, multi-path routing, and parallel packet switches. Toward relaxing this constraint on 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 adaptively balances increasing *dupthresh*, to avoid false fast retransmits, and limiting the growth of *dupthresh*, to avoid unnecessary timeouts. Finally, we demonstrate that delayed packets negatively impact the accuracy of TCP's RTO estimator, and present enhancements to the estimator 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.

[*] mzhang@cs.princeton.edu, ICIR/Princeton University

[†] bkarp@icsi.berkeley.edu, ICIR

[‡] floyd@icir.org, ICIR

[§] llp@cs.princeton.edu, Princeton University

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. 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. Examples include:

Measured Transient Behavior: When the route to a particular destination oscillates among multiple alternatives, each of which may have a different round-trip time (RTT), reordering may result [19]. Routers have been observed to cease forwarding while processing a routing update, and intersperse the delayed packets with new arrivals, causing reordering [20].

High-Speed Switches: Bennett [5] *et al.* show that MAE-East reordered packets frequently. They argue that striping of packets across multiple links between neighboring switches combined with work-conserving switch architectures must produce reordering.

Satellite Links: 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 [22].

A reordering-robust TCP would perform better in these cases. But the most compelling reasons to improve TCP's robustness to reordering are the beneficial systems that *cannot be deployed*, because they introduce reordering, or are *restricted in their utility* because of the in-order delivery requirement. Such systems include:

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 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 [8], and in the case of switching, eliminates much of the cost savings of the parallel hardware approach.

Differentiated Services: Packets sent by a host may be given different per-hop behaviors (PHBs) under DiffServ [6]. PHB markings cause routers to treat packets with different drop behaviors or forwarding precedences, among other per-hop behaviors. A single TCP flow's packets generally cannot be marked with different PHBs, because the different forwarding behaviors may cause reordering. This restriction is no minor limit on the usefulness of DiffServ. It is entirely possible that a single flow's sending rate may exceed that permitted by a traffic profile enforced by a traffic conditioner for a "premium" PHB. In such a case, the conditioner may either drop out-of-profile packets, or assign them a different PHB—and thus risk reordering the flow's packets. Both alternatives reduce TCP's throughput. To avoid these pitfalls, *none* of that flow's packets may use the premium PHB if only a *few* are out-of-profile. The sender loses the utility of the premium PHB entirely, whereas it could realize (for example) a lower overall drop rate if *some* of the packets in the flow were marked with the Assured Forwarding PHB.

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).

It is TCP's inability to distinguish reordering from packet loss that 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 this mistaking of reordering for loss is not fundamental to window-based congestion control. Rather, it is an artifact of the design 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. Herein lie two of the fundamental costs when improving TCP's throughput under reordering:

- End-to-end delay for packets dropped and subsequently retransmitted will increase, because reordering tolerance means the sender must wait longer before determining a loss has occurred.
- Congestion response will be less timely, because detection of loss is the primary feedback in detecting network congestion.

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.

The DSACK extension [11] to SACK TCP [16] 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. 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 that the receiver returns once both the original packet and the spurious retransmission of it arrive. DSACK doesn't specify the sender's actions; it only provides duplicate receipt information to the sender. While the sender only learns of a spurious retransmission from DSACK after at least a full RTT, we show that this information is still quite useful in adapting the sender's behavior, both in "backing out" the recent incorrect window reduction, and in avoiding future spurious retransmissions.

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. The control loop uses reordering behavior learned from SACKs and DSACKs, 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 DSACK.

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 algorithmic components of RR-TCP's reordering adaptation control loop. A detailed performance evaluation of RR-TCP in simulation can be found in Section 5. We conclude by suggesting future avenues of research in Section 6, and summarizing our findings in Section 7.

2 Fast Retransmit and Reordering

The fast retransmit and fast recovery mechanism, first implemented in 4.3BSD Reno TCP and described in RFC 2581 [2], allows TCP to detect loss without experiencing a retransmit timeout (RTO). After the loss, packets that arrive at the receiver cause it repeatedly to acknowledge the *same* highest contiguous sequence number received thus far. A sender that implements fast retransmit uses the return of these *duplicate acknowledgements* (duplicate ACKs) to detect loss. Using fast retransmit allows the sender to reduce its congestion window by half without going idle until the one-second-minimum retransmit timer expires, or having to slow-start from a window of a single packet.

Fast retransmit obviously requires a choice of *how many* duplicate ACKs the sender must receive before it concludes that the network has dropped a packet. This parameter, *dupthresh*, is fixed at three duplicate ACKs 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.

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

2.1 False Fast Retransmit

The *false fast retransmit* phenomenon limits TCP's throughput when the network reorders a connection's packets. We demonstrate the phenomenon in Figure 1. Here, the first packet in the first window shown is delayed by 0.45 seconds, while subsequent packets are not delayed, such that the first packet arrives after the remainder of the packets in the window. Just after one second, a cluster of duplicate ACKs arrives 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 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.

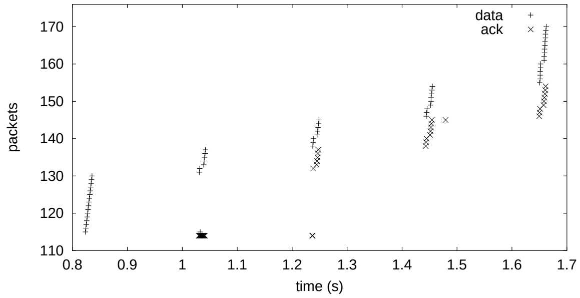


Figure 1: Example of a false fast retransmit; packet number *vs.* time. Times are at sender. The first packet sent is delayed by 0.45 s.

Figure 1 has one other important aspect: it demonstrates the use of DSACK information in *recovering* from a window reduction caused by a false fast retransmit. Should the sender learn from a DSACK it receives that a retransmitted packet was not dropped, it can undo the window reduction made at the time of the retransmission. Here, the DSACK arrives at approximately time 1.45. Note that the sender restores its old window of sixteen packets thereafter. Prior work on TCP's response under reordering, described in Section 3, investigates this recovery after learning of spurious retransmission; our work extends recovery by additionally *avoiding* 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 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;
- drastic lengthening of transfer times for short transfers, when a loss occurs near the end of a short transfer, because there are insufficient packets left to send to provoke a fast retransmit, and a one-second-minimum timeout will result, instead.

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 limited transmit, the sender is permitted to send one new data packet for each of the first two duplicate ACKs that arrive. This behavior aids the sender in accumulating three duplicate ACKs after a loss, even when its window is small. On the majority of paths, the RTT is far shorter than the one-second-minimum RTO. Limited transmit allows senders on these paths with even a window of three packets to recover from a single packet loss with fast retransmit, rather than with a timeout. While limited transmit permits the sender to send two packets beyond its current congestion window, it only sends these packets in response to 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 times the congestion window size 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. Note that they apply to *all* incarnations of limited transmit, regardless of the *dupthresh* value, and whether limited transmit will send up to $dupthresh - 1$ packets. Yet the number of packets sent by limited transmit affects the degree to which these arguments are a concern.

Nevertheless, no matter how many packets limited transmit sends, it is always self-clocking. Bansal *et al.* [4] show that under conditions of 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 [10], which requires four to eight RTTs to cut its sending rate by half, even under persistent congestion. Thus, our choice of 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.

Most importantly, we stress that the work we present is *independent* of the number of packets limited transmit may send. Our algorithms for varying *dupthresh* in response to reordering confer quantitatively similar performance improvements whether we permit *no* limited transmit, one congestion window's worth, or two congestion window's worth.

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.

2.4 Delay and Reordering

The phenomena of delay and reordering are related. It is possible to have delay without reordering (*e.g.*, a router buffers all packets for a period in-order, then continues forwarding), but this case is out-of-scope of our investigation. Instead, 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. The model we use for delay provides for a percentage of packets to be delayed and a distribution of delay times for packets selected to be delayed. Each packet is delayed independently, except where noted otherwise.

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.

Floyd [9] proposes using an extended version of SACK information to back out a window reduction after learning of a false fast retransmit. This proposal evolved into DSACK [11], which offers the sender information about duplicate packets that reach the receiver, but doesn't specify how the sender uses the information to recover from spurious retransmissions.

Ludwig [15] studies the spurious retransmission problem, both for 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. Ludwig does not use DSACK to notify the sender of duplicates. Rather, he proposes two alternatives. The first is to use TCP timestamps on every packet [13], 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 [15], 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. Both the timestamp and RTX bit proposals have two advantages over DSACK. First, both timestamps and the RTX bit fully disambiguate whether an ACK corresponds to the original packet transmission or a retransmission; DSACK does not. Second, DSACK cannot notify the sender of duplicates until *both* the original and retransmitted packets reach the receiver, whereas either of Ludwig's schemes may notify the sender up to one RTT earlier that a retransmission was spurious. We examine the performance difference between the RTX bit and DSACK in Section 5.1.1.

Blanton and Allman [7] 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. They do not study the potential negative effects of an increased *dupthresh*, and present no other strategy for reducing *dupthresh* than resetting it to three packets upon a timeout. In their work, they consider six strategies for increasing *dupthresh*; we name them with short textual tags to ease referring to them later. DSACK-BL-R detects and recovers from false fast retransmit but doesn't vary *dupthresh*. Three strategies directly manipulate *dupthresh*: increasing *dupthresh* by a constant value (one in their simulations) every time a false fast retransmit occurs (DSACK-BL-INC); averaging *dupthresh* with the number of duplicate ACKs that caused a false fast retransmit, each time such an event occurs (DSACK-BL-AVG); and setting *dupthresh* to an exponentially weighted moving average of the number of duplicate ACKs received at the sender (DSACK-BL-EWMA). These schemes limit *dupthresh* to a maximum of 90% of the current congestion window. Their other two strategies involve the use of a timer to delay fast retransmit. One sets the timer's value to the delay between the arrival of the first duplicate ACK and the return of the first ACK for the delayed packet (DSACK-BL-TIMEDEL). The other increments the timer's value by a constant value (10 ms in their simulations) each time a false fast retransmit is encountered (DSACK-BL-TIMEINC). For the timer-based schemes, the maximum value the timer can reach is half TCP's smoothed round-trip time estimate, and the timer is never reduced in its length. These 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*, or too long a timer.

4 Algorithms

This section describes our 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 the reordering length 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 that the sender experiences.

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 [14], whether by fast retransmit or timeout, 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.

4.1 Reordering-Related State: The Scoreboard

The SACK TCP scoreboard data structure [17] 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 length of the reordering a packet experiences. We record each fast retransmit's starting time and window reduction amount in the scoreboard entry for the retransmitted packet. 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 fields we add to the scoreboard affect the bytes per packet recorded in the scoreboard, not the number of packets stored in the scoreboard. And the reordering length histogram is small, as it only stores events for reordered packets. Thus, the techniques we describe are not significantly different in cost from SACK TCP, which is already widely deployed [18].

4.1.1 One Packet's Reordering Length

For the TCP 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, the worst-case number of duplicate ACKs generated by the delay of packet i is k: the difference between the highest packet number ACKed or SACKed so far and the number of the delayed packet.[1]

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. Measuring the reordering length thus amounts to detecting when a returning ACK or SACK fills in a hole in the scoreboard corresponding to a packet number *greater* than the highest contiguous packet number previously acknowledged; the difference between the packet number of the "hole" and the greatest ACK or SACK number received so far is the reordering length.

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 $c < k \leq i$
    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
```

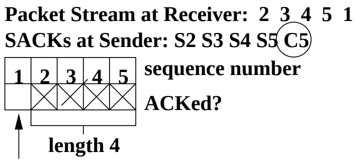

Packet Stream at Receiver: 2 3 4 5 1
SACKs at Sender: S2 S3 S4 S5 C5
1
2
3
4
5
sequence number
ACKed?
length 4

Figure 2: An example of reordering measurement using the scoreboard.

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. It is important to note that this test makes the reordering length measurement mechanism robust against reordered ACKs; in Section 5.4, we demonstrate in simulation that reordered ACKs do not affect the throughput achieved by our TCP sender.

Figure 2 shows a simple example of the reordering length measurement mechanism. Here, packet 1 is displaced four packets later in the stream of data packets that arrive at the receiver. SACKs return to the sender for packets $2 \ldots 5$. The scoreboard in the figure 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 are 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

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

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 from these facts that delayed ACKs are extremely unlikely to occur during a reordering epoch, between receipt of the first non-contiguous packet at the receiver, and the emptying of the reassembly queue.

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 above is not, and also allows us to measure multiple reordering events within a single window of packets.

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. At the cost of the associated state per connection, the histogram provides details of the reordering distribution to our *dupthresh* adjustment algorithms. The choice of a histogram for storing state related to reordering is one of many possible; Blanton and Allman explore alternatives that accumulate less state [7]. Our goal is to demonstrate the *best case* performance improvement that can be attained by using the most accurate and detailed reordering information—a histogram trades off increased accuracy and detail for increased state size. We stress that a histogram makes no assumption about the distribution of reordering lengths packets experience; for any persistent reordering process, a histogram will provide percentiles of reordering lengths.

Note that reordering length samples for retransmitted packets are delayed in their insertion into the histogram. 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.

4.1.3 Storage and Computational Costs

The standard SACK sender implementation keeps a scoreboard for each open TCP connection. The reordering length measurement scheme we've described keeps additional state in two places for each connection: in the scoreboard and in the reordering length histogram. The standard SACK scoreboard already measures whether a packet has been retransmitted and whether it's been acknowledged. Reordering length measurement adds little further state; for each packet index in the scoreboard, the additional fields used are only a reordering length field and a DSACK block counter field. A single byte of storage per packet index suffices for reordering lengths of up to 63 packets (6 bits) and counting DSACK blocks (2 bits). Because the sender must buffer each unacknowledged packet (1500 bytes for Ethernet-MTU-sized packets), adding a byte of storage to the scoreboard per packet is truly insignificant. The reordering length histogram stores up to 1000 reordering events, each of which contains a timestamp (4 bytes provide sufficient resolution and magnitude) and pointer (another 4 bytes). This histogram can be allocated dynamically as reordering is encountered. It costs no storage on connections that do not encounter reordering, and storage proportional to the degree of reordering on connections that do, up to an 8K maximum. These storage requirements restrict cost to the connections that benefit. Again, our goal is to explore the best throughput attainable under reordering; other schemes may keep less state, but can be compared against this one, which uses detailed reordering information.

The computational cost of our reordering length measurement scheme involves scanning the scoreboard on receipt of each ACK, SACK, and DSACK, and modifying the reordering length histogram each time a reordered packet is found. The BSD/OS SACK implementation [3], as a representative SACK implementation, keeps a list of holes to track regions of sequence number space sent but not yet cumulatively ACKed or SACKed. That implementation scans this list to update it with every SACK it receives. Our reordering length measurement algorithm can easily be implemented on this data structure, and requires the same computation: a scan of this list on every received SACK, and additionally on every received new cumulative ACK. Note that each such scan costs computation proportional to the number of holes in the sequence number space, *i.e.*, the extent of packet reordering. Thus, reordering measurement only costs computation when reordering occurs, exactly when our scheme offers a throughput benefit. Upon detecting a reordered packet, there is a single histogram insertion requiring constant time. When a reordered packet expires from the histogram, its removal costs constant time. The oldest histogram members are checked for expiration on each insertion. A coarse-grained timer expires them when no insertions occur for a long period. Thus, the computational costs of measuring reordering length are negligibly greater than those of standard SACK.

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.

4.3.1 Cost Function: RTOs

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 RTO 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 RTO.
- **True timeouts,** for which no DSACK returns, occur when loss causes an RTO.

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 beyond the halved window. 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 DSACK information 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.[2]

4.3.2 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 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}.

4.3.3 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 RTO 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 RTO 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 $C(\text{limited transmit}) = \frac{I}{R}W - d$.

[2] 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.

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 $C(\text{limited transmit}) > C(\text{false fast retransmit})$.

4.3.4 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 detected, 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. 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.

4.4 Choosing an Accurate, Conservative RTO

Karn's Algorithm dictates that retransmitted packets *not* be included in RTT sampling because of the retransmission ambiguity, described in Section 3. On paths where packets are delayed, however, it is the delayed packets that are most likely to provoke retransmissions. Thus, in the presence of reordering, Karn's Algorithm introduces a sampling bias against including long RTT samples in the RTO estimator. In cases where packets are delayed for significant periods relative to the RTT, this bias may cause TCP to adopt an RTO that is insufficiently conservative, and causes false timeouts.

Algorithm Name	Description
SACK	Standard SACK
DSACK-R	DSACK + FFR recovery
DSACK-FA	DSACK-R + fixed FA ratio
DSACK-FAES	DSACK-FA + enhanced RTT sampling
DSACK-TA	DSACK-FA + timeout avoidance
DSACK-TAES	DSACK-TA + enhanced RTT sampling
DSACK-HG	DSACK-TA + histogram RTO estimator

Table 1: Algorithms compared in simulation. FFR recovery includes detection of False Fast Retransmits with DSACKs and restoration of the old, larger window size.

For all falsely retransmitted packets, either by fast retransmit or timeout, two ACKs return, the second of which is a DSACK. If both these ACKs arrive at the sender, the sender may approximate the RTTs experienced by the packets by pairing the first ACK to return with the first transmission, and the second ACK with the second transmission; computing the time elapsed for each; and taking the mean of these two values as a single RTT sample for the RTO estimator. As before, the scoreboard holds the time information associated with the packet's transmission and retransmission. Note that this mean value doesn't change, regardless of whether the packets and ACKs are paired properly. If one packet was delayed, or both were delayed, the mean of their RTTs will be include a portion of the delay in an RTT sample that otherwise would have been ignored by Karn's Algorithm. The sample cannot exceed the true RTT for the longer of the two delayed packets. In this sense, DSACK enables RTT sampling that is more inclusive for delayed packets, yet still conservative. If no DSACK returns, either the original or retransmitted packet was lost, and we don't sample that packet's RTT. In this sense, we comply with Karn's Algorithm for retransmissions caused by packet drops, but include additional RTT samples for retransmissions caused by packet delays. We term this RTT sampling extension DSACK-ES, for **E**nhanced RTT **S**ampling.

5 Experimental Evaluation

This section presents simulation results to demonstrate the improvement DSACK-based sender-side algorithms make to TCP's performance over paths that reorder or delay packets. We compare the performance of several variants of DSACK, both those we propose in Section 4 and those proposed by others in prior work. Table 1 summarizes the algorithms we compare in simulation.

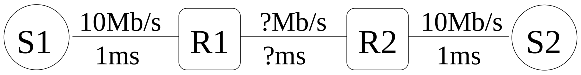


Figure 3: Network topology used in simulations.

We simulate these algorithms in the *ns-2* network simulator [17], version 2.1b8, which includes support for generation of DSACK information at the receiver in its `sack1` implementation. To introduce reordering, we extended *ns-2* to delay a configurable percentage of packets that traverse a link. An undelayed packet is forwarded on the link immediately; a delayed one is scheduled to traverse the link later, and a delay later than subsequent packet arrivals introduces reordering. The delay distribution is configurable, such that simulation with delays distributed according to the random processes supported in *ns-2* (normal, exponential, hyperexponential, Pareto, Pareto II, and uniform) is possible. Independently of the delay, we also control the drop rate associated with a link.

We simulate a wide variety of delay (reordering) distributions to demonstrate the value of our algorithms in improving TCP's performance under reordering. 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 other the other distributions we simulate; in Section 5.5 we demonstrate this fact. We stress that the reordering sources we seek to address result in reordering behaviors that are as tractable for our algorithms as those we simulate. Multipath routing will produce modal delays; successive packets sent on paths with different RTTs will be reordered proportionally to the paths' RTT differences; we consider such a modal distribution in Section 5.6. Parallelism in router forwarding software or hardware will result in relatively short reordering lengths. Satellite links present long propagation delays, and delay packets retransmitted at the link layer by multiples of the RTT. Our simulations are relevant for all these cases.

Our simulations consist of a single, long-lived TCP flow traversing the network topology shown in Figure 3. The flow lasts 1000 seconds. All data points in simulation results plots are the 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 parameters used in our simulations in Table 2. Note that we bound the number of packets TCP may send with limited transmit to be one current congestion window value. This limit ensures that the sender delays window reduction after a loss by fast retransmit no longer than one RTT.

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. The packet delay process is 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.

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× *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.

5.1.1 Varying Packet Delay Rate

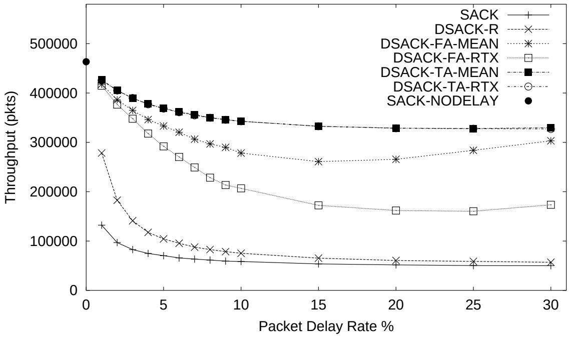


Figure 4: Throughput *vs.* fraction of delayed packets. 50 ms propagation delay, normally distributed pkt delay, mean 25 ms, stdev 8 ms.

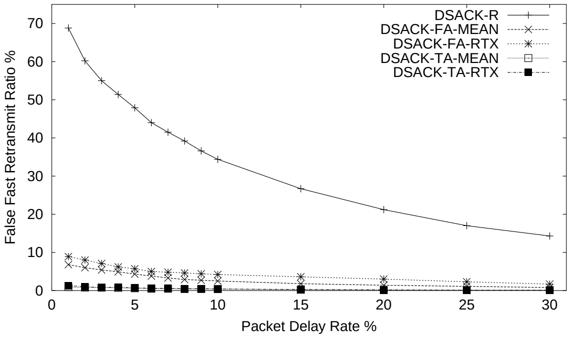


Figure 5: False fast retransmit ratio *vs.* fraction of delayed packets. 50 ms propagation delay, normally distributed pkt delay, mean 25 ms, stdev 8 ms.

First, we vary the percentage of delayed packets from 1% to 30%, without introducing packet drops. As shown in Figure 4[3], 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 5 reveals that the fraction of packets resent with fast retransmit for which retransmission is false under

[3] 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.

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.

In cases with virtually no RTOs, 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 measuring the reordering lengths of retransmitted packets. Recall the ambiguity in matching ACKs with retransmitted packets. The -RTX variant uses the RTX bit [15] (a reserved bit in the TCP header) to mark retransmitted packets and their ACKs differently, thereby resolving the ambiguity. The -MEAN variant uses the technique described in Section 4.1.1, where no RTX bit is used, but instead the two packets are paired with ACKs in the order the ACKs return, and the mean of these two reordering lengths is used as the sample for the reordering length histogram. 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, while DSACK-FA-RTX uses the reordering length of the original transmission only. 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.

5.1.2 Varying Packet Drop Rate

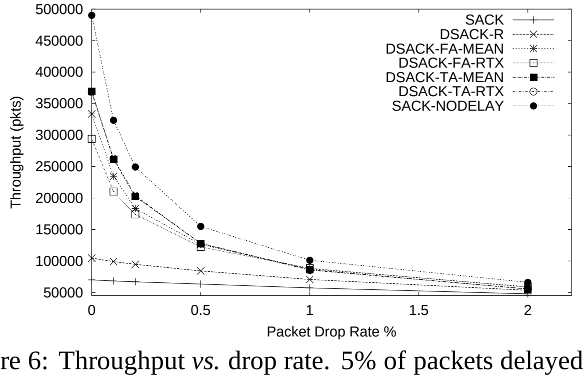


Figure 6: Throughput *vs.* drop rate. 5% of packets delayed. 50 ms propagation delay, normally distributed pkt delay, mean 25 ms, stdev 8 ms.

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 6, 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. As before, SACK-NODELAY shows the throughput SACK TCP achieves under these loss rates when no packets are reordered. In the case of the reordering-robust 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

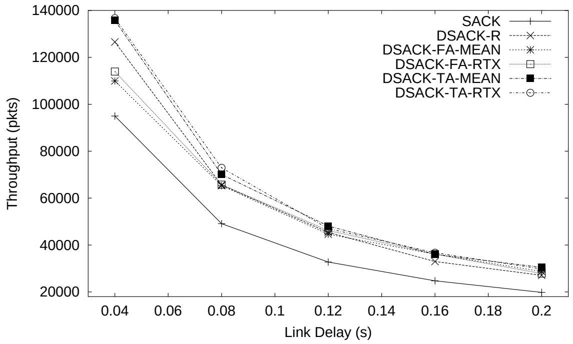
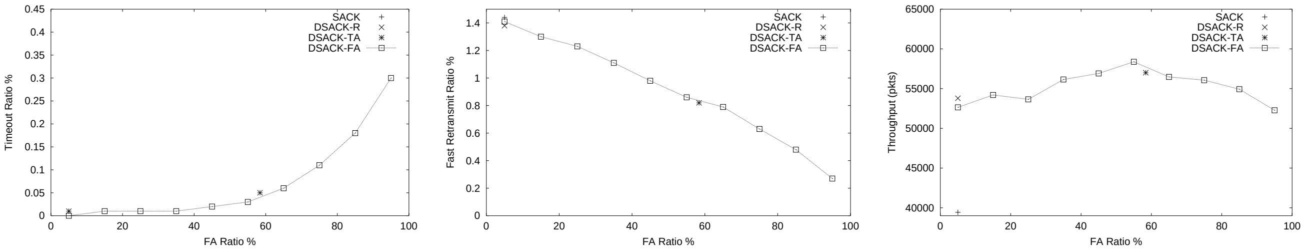


Figure 7: Throughput *vs.* link propagation delay. 1.4% of packets delayed; 0.6% of packets dropped; uniform packet delay in $[0, 4P]$.

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 7, 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 8 shows (a) the RTO behavior, (b) the fast retransmit behavior, and (c) the throughput behavior of DSACK-FA under these conditions. In Figure 8a, 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 8b 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 8c 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 8c that DSACK-TA adapts *dupthresh* such that



(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 8: Timeout avoidance: comparing DSACK-FA and DSACK-TA.

it achieves approximately the maximum throughput available among all possible fixed FA ratios.

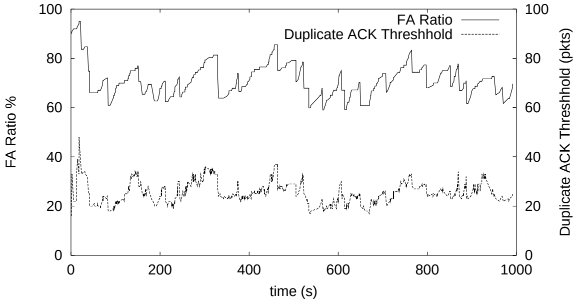


Figure 9: FA ratio and *dupthresh vs.* simulated time for a single simulation.

Figure 9 shows the dynamic behavior of DSACK-TA as it adapts the FA ratio and *dupthresh* ; note that the ratio oscillates about the 70% point, shown in Figure 8c to offer roughly the maximum throughput among all FA ratios. *dupthresh* oscillates around 25.

Figure 7 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. Thus, the performance of DSACK-FA approaches that of DSACK-TA as the link delay increases.

5.3 More Conservative RTO Estimator

We next study the effect of the RTO estimator on the performance of DSACK and SACK. As explained previously, a surfeit of retransmissions provoked by delayed packets may cause Karn's Algorithm to choose RTO values that are biased to be too short, leading to an overly aggressive RTO. In the following, we use a propagation delay P of 200 ms for (R_1, R_2). We delay 4% of packets, and drop none. Packets are delayed according to a normally distributed process with mean kP and standard deviation $\frac{k}{3}P$, so that most delays last between 0 and $2kP$ seconds (mean $\frac{k}{2}RTTs$). These parameters, again, represent cases in the upper range of Internet link delays, and the extreme stress of severe packet delays. We observe similar phenomena in cases where the packet delay can exceed the one-second-minimum RTO; satellite links that use link-layer retransmission fall into this category.

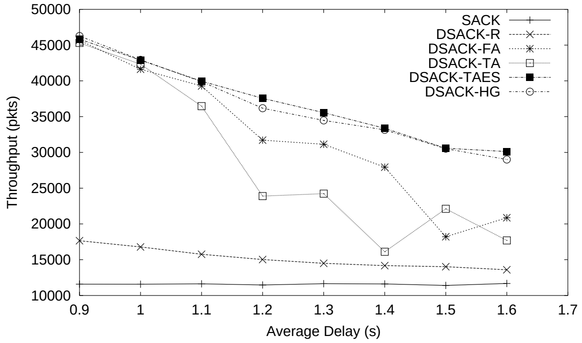


Figure 10: Throughput *vs.* average delay time. 4% of packets delayed; no packet drops; $P = 200$ ms propagation delay; normally distributed pkt delay; mean kP; stdev $\frac{k}{3}P$.

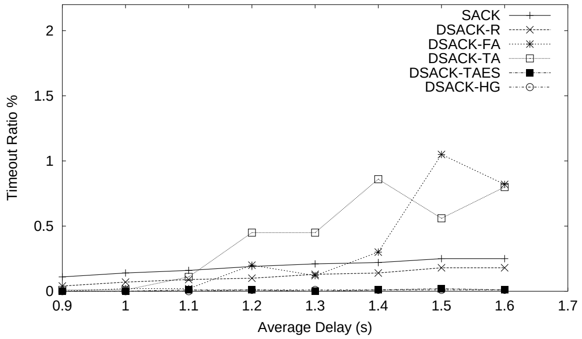


Figure 11: Fraction of packets that encounter timeout *vs.* average delay time. 4% of packets delayed; no packet drops; $P = 200$ ms propagation delay; normally distributed pkt delay; mean kP; stdev $\frac{k}{3}P$.

In Figures 10 and 11, we gradually increase k from 4.5 to 8.0. As k grows, the RTT of a delayed packet may exceed the mininum RTO of 1 second, and may thus trigger a false timeout. In Figure 10, as packet delay increases, the throughput decreases, as expected. But when k goes beyond 5.5, which corresponds to a mean delay of 1.1 seconds in Figure 10, the throughput of DSACK-TA drops sharply and becomes even worse than that of DSACK-FA. It may seem that the timeout avoidance mechanism of DSACK-TA doesn't work well here, but actually, the cause is the RTO estimator rather than timeout avoidance. Figure 11 shows that the fraction of packets that encounter timeouts under DSACK-TA increases rapidly, and exceeds that for DSACK-FA after the mean delay surpasses 1.1 seconds. DSACK-TA is designed to avoid timeouts by increasing the number of fast retransmits. But Karn's Algorithm causes false fast retransmits to skew the sampling of RTTs by excluding RTT samples for delayed packets. DSACK-TAES, which implements the enhanced RTT sampling strategy described in Section 4.4, maintains a low incidence of timeouts even when packet delays are severe by including conservative mean RTT samples from packets that experience false retransmits. The more conservative RTO chosen by DSACK-TAES leads to fewer timeouts and increased throughput.

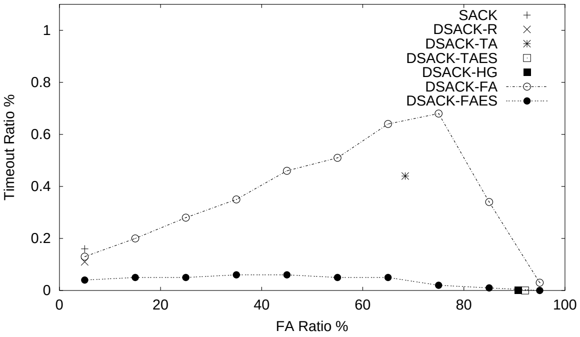


Figure 12: Fraction of packets that encounter timeout *vs.* FA ratio. 4% of packets delayed; no packet drops; 200 ms propagation delay; normally distributed pkt delay; mean 1200 ms, stdev 400 ms.

SACK	1.0	DSACK-R	1.0
DSACK-FA	3.1	DSACK-TA	1.6
DSACK-TAES	3.1	DSACK-HG	2.3

Table 3: Mean RTO (s) across all RTOs.

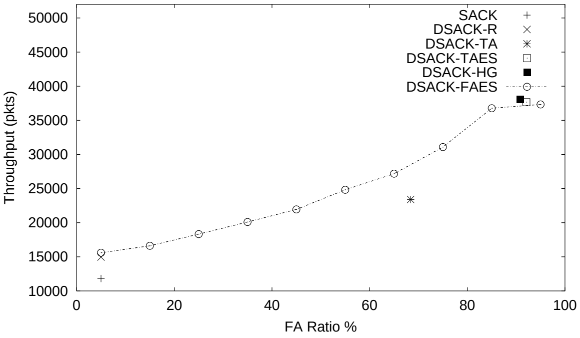


Figure 13: Throughput *vs.* FA ratio. 4% of packets delayed; no packet drops; 200 ms propagation delay; normally distributed pkt delay; mean 1200 ms, stdev 400 ms.

To look at the interaction between timeout avoidance and the RTO estimator more closely, we fix *k* at 6 and vary the FA ratio from 95% down to 5% in Figures 12 and 13. As shown in Figure 12, the fraction of timeouts under DSACK-FA when the FA ratio is 95% is far less than that under DSACK-TA, which causes more than 0.4% of packets to encounter timeouts. Note further that under DSACK-FA, many more packets encounter timeouts as the FA ratio decreases below 95%, unlike in prior results, where the incidence of timeouts steadily decreases as the FA ratio decreases. Now consider DSACK-FAES, which samples RTTs for packets falsely retransmitted. The incidence of timeouts under DSACK-FAES always remains low. Table 3 reveals the effect of sampling additional RTTs during false retransmits on the mean RTO value during a simulation. Without additional samples, the average RTO used by DSACK-TA is much shorter than that used by DSACK-FA with an FA ratio of 95%. By adding samples for falsely retransmitted packets, DSACK-TAES eliminates the bias against long RTT samples, and restores the RTO to a value comparable with that used by DSACK-FA. For comparison, we include a curve for DSACK-HG, which uses a detailed histogram of RTT samples for each packet, rather than an exponentially weighted moving average, to compute the RTO. DSACK-HG targets the 99.8% point in the cumulative RTT distribution in formulating its RTO estimate. Note that both DSACK-FA and DSACK-TAES are more conservative than DSACK-HG. Thus, the standard EWMA RTO estimator is inherently very conservative if not skewed by false fast retransmits, as compared with the estimate given by a histogram with nearly perfect information concerning the RTT distribution.

Figure 13 shows that timeout avoidance continues to perform well under severe packet delays, once a sufficiently conservative RTO estimator is added to it in DSACK-TAES. DSACK-TAES matches the best throughputs achieved by DSACK-FAES across all FA ratio values, and even DSACK-HG. Even across the range of delays in Figures 10 and 11, DSACK-TAES offers high throughput and low timeout incidence, comparable to those of DSACK-HG.

5.4 Robustness to ACK Reordering

The mechanisms we propose for making the TCP sender robust against reordering address the effects of reordered data packets on the sender's window size. Reordering may also occur on the reverse path, such that ACKs arrive out-of-order at the sender. Because the algorithms we present 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. 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.

Figure 14 shows the throughput attained by the previously described DSACK-aware sender variants as the fraction of ACK packets reordered increases. Note that 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.

5.5 Independence of Reordering Distribution

Figure 15 shows that across all reordering length distributions simulated, DSACK-TA achieves similar throughput gains over SACK.

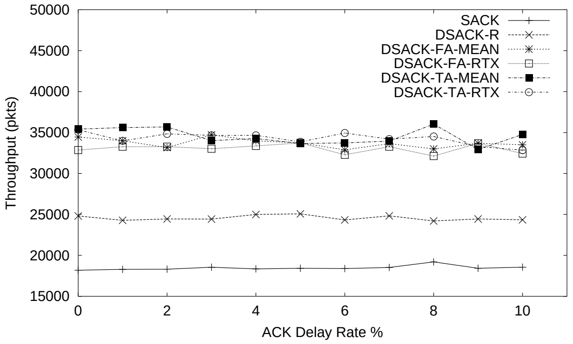


Figure 14: Throughput *vs.* fraction of reordered ACKs. 200 ms propagation delay, 0.5% of data packets dropped, 3.5% of data packets delayed, normally distributed data and ACK pkt delay, mean 100 ms, stdev 33 ms.

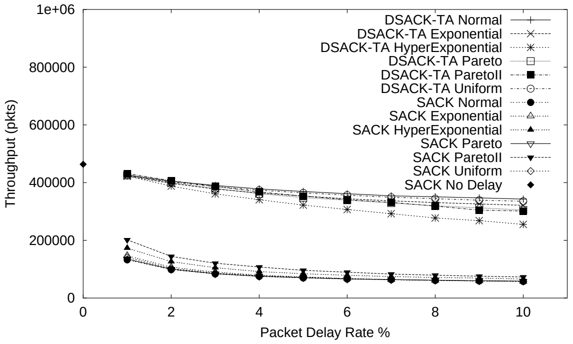


Figure 15: Throughput *vs.* fraction of delayed packets. 50 ms propagation delay, mean pkt delay 25 ms.

5.6 Robustness for Multi-Path Routing

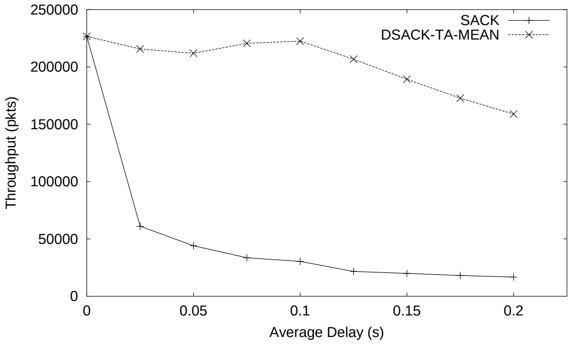


Figure 16: Throughput *vs.* pkt delay. 50 ms propagation delay, no loss.

In Figure 16, 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 SACK. Note that the performance advantage of DSACK-TA-MEAN over SACK begins to diminish at delays longer than 100 ms; at this point, packets are being 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.

5.7 Comparison with Prior Work

Blanton and Allman propose several techniques for adapting *dupthresh* in response to reordering [7]. 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. An advantage of their proposals is that they require little state at the sender; our false fast retransmit and timeout avoidance algorithms maintain a reordering histogram. This section compares the behavior of Blanton and Allman's algorithms with our own, in the interest of exploring the space of alternatives with different costs in sender-side state.

We begin by characterizing 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 during the simulation. This great *dupthresh* value may increase end-to-end delay for dropped packets in cases where the reordering length has a heavy-tailed distribution. When a network path reorders less severely than before, their algorithms without a *dupthresh* decrease strategy must rely on a timeout to reset *dupthresh* to 3. In comparison, in exchange for the extra state associated with the reordering histogram, our timeout avoidance algorithm avoids most false fast retransmits, while ignoring rare and extremely long reorderings. Should reordering lengths change in distribution over time, the histogram reflects any such change, and causes *dupthresh* to change accordingly.

Resetting *dupthresh* to 3 discards history learned about the network's reordering behavior, and re-accumulating history thereafter takes time. Again, in exchange for increased state, use of a reordering length histogram preserves knowledge of the path's characteristics across timeouts.

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 $0.9 \times \text{cwnd}$ can't prevent false fast retransmits in cases where reordering lengths are longer than $0.9 \times \text{cwnd}$, 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*,[4] but

[4] 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

with our delay and loss models, used previously in this paper. Our observations after simulating these algorithms on identical networks follow.

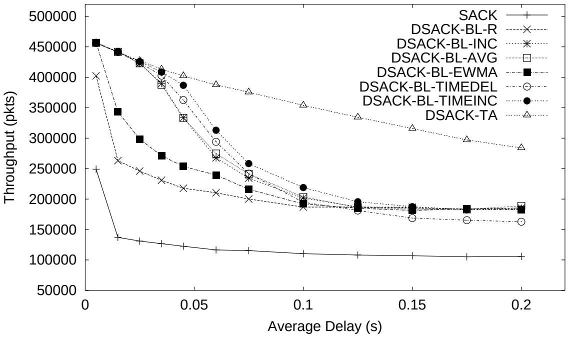


Figure 17: Throughput *vs.* average delay time. Link delay P = 50 ms; 1% of packets delayed; none dropped. Packet delay normally distributed, mean kP, stdev $\frac{k}{3}P$.

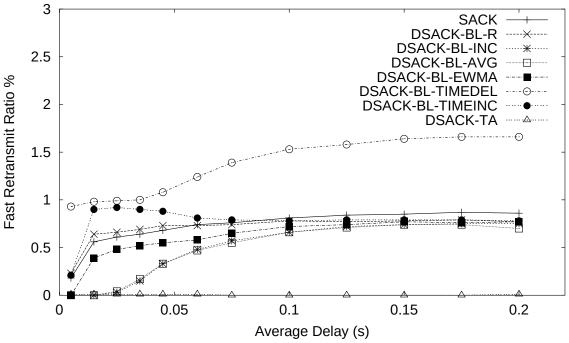


Figure 18: Incidence of fast retransmits *vs.* average delay time. Link delay P = 50 ms; 1% of packets delayed; none dropped. Packet delay normally distributed, mean kP, stdev $\frac{k}{3}P$.

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 $\frac{k}{3}$, such that most packet delays lie between 0 and $2kP$. As shown in Figures 17 and 18, we gradually increase k from 0.1 to 4.0, and thus vary the packet delay between 5 ms and 200 ms.

The DSACK-BL-XXX curves represent results for Blanton and Allman's algorithms. As shown in Figure 17, 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 we simulated. Figure 18 shows that the fraction of fast retransmits suffered by DSACK-TA hovers around 0%, whereas the other schemes suffer increasingly from fast retransmits as the 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.

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

Table 4: Throughput and fast retransmit ratios, with and without bursty packet loss.

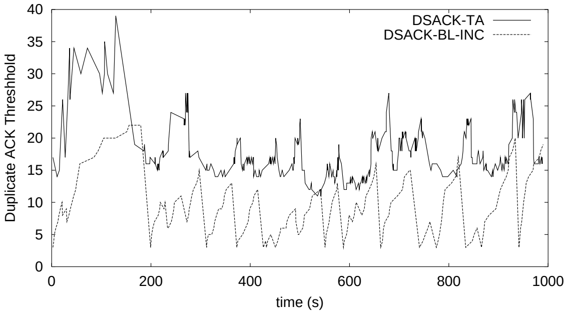


Figure 19: *dupthresh* values of DSACK-TA and DSACK-BL-INC *vs.* time for a single simulation.

We now explore the behavior of Blanton and Allman's algorithms under bursty packet loss, when multiple packets are dropped within a window. This dropping pattern can occur under low statistical multiplexing, when one or more of the few connections is in slow start, and exponential window increase causes a drop-tail router to drop a burst of packets.

Here, link (R_1, R_2) has $P = 200$ ms propagation delay, and packets are delayed according to a normal distribution with mean 100 ms and standard deviation 33 ms, such that most packet delays lie between 0 and 200 ms. 2% of the packets are delayed. 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 will trigger timeouts even with a $0.9 \times$ cwnd bound on *dupthresh*. In Table 4, we see that the throughput of DSACK-BL-INC is comparable to that of DSACK-TA when there aren't bursty packet drops. After we introduce a small fraction of bursty packet drops, however, the throughput of DSACK-BL-INC suffers more than that of DSACK-TA, because DSACK-BL-INC experiences more false fast retransmits. Figure 19 shows the variation in *dupthresh* for DSACK-TA and DSACK-BL-INC. Each timeout resulting from a drop burst will cause 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 keeps *dupthresh* around the optimal value because it maintains the histogram of reordering events. 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* (normal, exponential, hyperexponential, Pareto, Pareto II, and uniform). As expected, DSACK-TAES has the best overall performance because it combines the benefits of false fast retransmit avoidance, timeout avoidance, and enhanced RTT sampling. Under a few

2.1b8 when running their code in 2.1b7.

cases, other schemes, especially DSACK-FA or DSACK-BL-TIMEINC, offer slightly better throughput than DSACK-TAES. In these cases, the throughput differences are mostly $< 5\%$, and all $< 10\%$. We believe the factors at work in these cases include:

- Our cost functions use an EWMA to estimate some of the parameters, such as steady state window size and duration of false fast retransmits. The averages thus computed are not instantaneously accurate.

- The control loop for adapting the FA ratio takes time to converge to its optimum from its initial value. After it converges, it oscillates about this value.

6 Future Work

While we've exhaustively evaluated RR-TCP in simulation, we look forward to implementing it for a widely distributed host operating system, such as Linux or FreeBSD. Completing this implementation will allow us to gain deployment experience with the enhanced protocol, as well as learn how to minimize the processing overhead it incurs.

In this paper, we've pursued only sender-side designs for reordering robustness. These designs require the sender to store extra state for each connection in the SACK scoreboard, and in the reordering histogram. For a busy server with many thousands of open connections, these additional state requirements may be burdensome. We believe RR-TCP can be built in a receiver-side fashion, whereby the receiver measures reordering and keeps the relevant histogram, applies the *dupthresh* adjustment algorithms, and dynamically informs the sender of the *dupthresh* value it should use, perhaps in a TCP option. This design devolves the reordering-related state requirements from the server to each client, at the cost of requiring a modification to the over-the-wire protocol.

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 unaddressed. 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.

TCP's congestion control algorithms act at a granularity of a single window. Choosing the number of packets that limited transmit may send affects the maximum reordering length for which RR-TCP can avoid false fast retransmits. When limited transmit permits multiple windows of packets to be sent, the result is reordering robustness at a granularity greater than a single window. While ACK-clocking ensures limited transmit does not cause TCP to be unresponsive to congestion in the long term, the choice of the maximum allowed extent of limited transmit bears further investigation.

This paper has considered only long-lived flows in the interest of simplifying the evaluation of 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.

The cost functions we use to adapt the FA ratio make decisions based on individual false fast retransmit, timeout, and limited transmit idling events. We plan to investigate whether further history, in the form of event counts for some past period for each of these event types, can be used to adapt the FA ratio in a way that causes it to converge more quickly to the optimal value, or to change more smoothly over time, and how either of these outcomes affects the throughput RR-TCP achieves.

Finally, in some network environments, even the modified, conservative RTO estimator described in this work may not be conservative enough. In multi-hop ad hoc networks, RTTs are extremely variable because of link-layer ARQ. IEEE 802.11 [12], for example, backs off exponentially between link-layer retransmissions, and delays all subsequent packets during such backoff periods. A cascaded chain of many such hops offers formidable RTT variability. TCP estimates the RTO by $\text{RTO} = \text{SRTT} + k\text{RTTVAR}$. For the standard TCP RTO estimator [21], $k = 4$. A control loop to increase k dynamically in response to spurious timeouts, and decay k back to a minimum of 4 during periods without spurious timeouts, may aid in making TCP's RTO estimator sufficiently conservative in environments where RTTs are severely variable.

7 Conclusion

We have presented extensions to TCP that allow the sender to distinguish between reordering and loss, in the interest of 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. Our simulations on networks over a wide range of link delays, packet delays, and loss patterns show that our DSACK-TAES variant of 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.

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. The control loop converges to an FA ratio that offers throughput very nearly as great as the maximum attainable under any fixed FA ratio. Also, using a reordering histogram at the sender allows the sender to profit from knowledge of a connection's reordering history, even across timeouts. It is noteworthy that just as limited transmit aids in avoiding timeouts when *dupthresh* is fixed at three, it aids in avoiding timeouts under RR-TCP; the limited transmit cost function allows RR-TCP to reduce the FA ratio, and hence *dupthresh*, without having to incur a costly timeout in many circumstances. Finally, sampling RTTs for delayed packets in RR-TCP significantly improves TCP's throughput by ensuring that the RTO estimator is sufficiently conservative when packets are severely delayed.

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.

Despite its improved throughput on reordering paths, RR-TCP does not steal bandwidth from other, non-enhanced SACK TCPs; it is reordering that limits the throughput of non-enhanced SACK, not the transmission behavior of RR-TCP. In the face of persistent network congestion, RR-TCP is not significantly more aggressive than SACK TCP, even when it uses limited transmit, because it obeys ACK clocking. RR-TCP outperforms SACK TCP and previously published DSACK variants for limited transmit regimes that send anywhere between one half and two congestion windows' worth of data. Thus, the choice of the extent to which RR-TCP allows limited transmit can be made, within reason, based on the worst-case end-to-end delay tolerable to an application, and by the length of reordering to which the sender would like to be able to respond properly.

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] H. Balakrishnan, T. Henderson, and V. Padmanabhan. BSD/OS 3.0 SACK TCP Code. http://daedalus.cs.berkeley.edu/software/pub/tcpsack/bsdi-3.0/, Sept. 1998.

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

[5] 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.

[6] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differentiated services. *RFC 2475*, Dec. 1998.

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

[8] 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.

[9] S. Floyd. Re: TCP and out-of-order delivery. *Email to end-to-end interest mailing list*, Feb. 1999.

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

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

[12] IEEE Computer Society LAN MAN Standards Committee. Wireless LAN medium access control (MAC) and physical layer (PHY) specifications. *IEEE Standard 802.11-1997*, 1997.

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

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

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

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

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

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

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

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

[21] V. Paxson and M. Allman. Computing TCP's retransmission timer. *RFC 2988*, Nov. 2000.

[22] 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.