On Making TCP More Robust to Packet Reordering

Ethan Blanton
Ohio University
eblanton@irg.cs.ohiou.edu

Mark Allman
BBN Technologies/NASA GRC
mallman@bbn.com

January 4, 2002

Abstract

Previous research indicates that packet reordering is not a rare event on some Internet paths. Reordering can cause performance problems for TCP's fast retransmission algorithm, which uses the arrival of duplicate acknowledgments to detect segment loss. Duplicate acknowledgments can be caused by the loss of a segment or by the reordering of segments by the network. In this paper we illustrate the impact of reordering on TCP performance. In addition, we show the performance of a conservative approach to "undo" the congestion control state changes made in conjunction with spurious retransmissions. Finally, we propose several alternatives to dynamically make the fast retransmission algorithm more tolerant of the reordering observed in the network and assess these algorithms.

1 Introduction

Previous research indicates that packet reordering is not a rare event over some Internet paths. A network path that persistently reorders segments will degrade the performance of traffic utilizing TCP, the Internet's most widely used transport protocol [MC00]. [BPS99] shows that segment reordering over the MAE-East exchange is not a rare event and eliminating reordering is a difficult problem. [Pax97] reports on the reordering observed in TCP transfers on a mesh of 35 measurement hosts. This study shows that 0.1%–2.0% of all segments (data and ACK) experience reordering in the network. In addition, the prevalence of reordering varied across different network paths (for instance 15% of the segments transmitted by one particular host were reordered).

TCP receivers generate *cumulative acknowledgments* that indicate the highest in-order piece of data that has arrived [Pos81]. For example, assume that three segments are transmitted into the network, S_1–S_3, and that the second two segments are reordered. When segment S_1 arrives, the receiver will transmit an acknowledgment (ACK) for S_1. The next segment to arrive is S_3, which is out of order. However, the TCP receiver ACKs only the last in-order segment received – S_1 in our example. When segment S_2 arrives the last in-order piece of data that has been received becomes S_3, and therefore the ACK transmitted will contain S_3.

TCP uses two basic mechanisms to detect segment loss. First, if an ACK for a given segment is not received in a certain amount of time the *retransmission timer* expires and the presumed lost data segment is retransmitted [Pos81]. Alternatively, TCP can use the fast retransmit algorithm [Jac88, APS99]. Fast retransmit uses duplicate ACKs (a cumulative acknowledgment for the same segment as the last ACK received) to infer that a particular data segment was dropped by the network. In an attempt to disambiguate duplicate ACKs caused by loss from those caused by reordering events, the fast retransmit algorithm calls for the TCP sender to wait until three duplicate ACKs have arrived before retransmitting a segment.

Reordering has a negative effect on TCP performance for several reasons:

- A network that reorders data segments such that 3 or more duplicate ACKs are triggered at the receiver will cause the TCP sender to use fast retransmit to resend a data segment that was not lost, hence wasting bandwidth.
- TCP assumes that loss is an indication of network congestion [Jac88], and so a sender perceiving reordering as loss will also incorrectly reduce the data transmission rate when sending a spurious retransmission. In this paper we address this problem by examining a number of algorithms that vary the number of duplicate ACKs required to trigger the fast retransmit algorithm (and hence adjust the congestion control state) based on the observed reordering.
- Segment reordering causes interruptions to TCP's ACK clock [Jac88], thereby causing its transmission to be more bursty. TCP's standard congestion control algorithms [APS99] do not allow TCP to send segments in response to duplicate ACKs before fast

retransmit is triggered. By not sending segments in response to these duplicate ACKs, TCP effectively stores permission to send new data. Therefore, if an ACK covering new data arrives before fast retransmit is triggered then the burst of data sent on this ACK will be larger than if reordering had not occurred. This problem may be mitigated by the use of the *limited transmit* algorithm [ABF01], which calls for the TCP sender to transmit new data segments upon the arrival of the first two duplicate ACKs. Another method that may reduce the size of these bursts is using a *max-burst* parameter, as outlined in [FF96]. This method places an upper bound on the number of segments a TCP sender can transmit in response to a single incoming acknowledgment.

To mitigate the problem of burstiness we extend the limited transmit algorithm [ABF01] to allow the TCP sender to transmit new data segments upon receipt of duplicate ACKs before determining that a retransmit is necessary. This extension is especially important after we introduce algorithms that increase the number of duplicate ACKs required to trigger fast retransmit. Further discussion of our extensions to the Limited Transmit algorithm is given in § 6.1.

Reordering of acknowledgments can also cause bursty TCP behavior. ACKs that convey no new information are discarded by the TCP sender and therefore can not be used to clock out new data segments. However, the next ACK that arrives and conveys new acknowledgment information will trigger a larger than desirable burst of data segments to be transmitted. (This phenomenon is discussed in more detail in § 6.1.)

- Segment reordering can also prohibit TCP from sampling the round-trip time (RTT) as frequently as in an ordered stream. The RTT is sampled and averaged to calculate the retransmission timeout (RTO) used by TCP to achieve reliable delivery, as outlined in [PA00]. RTT timing has traditionally taken the form of starting a timer before a given segment, S, is transmitted and then stopping the timer when an ACK covering segment S arrives. Reordering can falsely inflate the RTT estimate when no unnecessary retransmissions are sent, which can potentially hurt performance in that TCP would have to wait longer before sending a legitimate retransmission. In the case when a segment is retransmitted needlessly because of reordering, the corresponding RTT sample must be marked as invalid [KP87]. For example, if segment S is sent twice then the RTT sample is ambiguous in that the sender can never be sure whether the ACK is in response to the first or second transmission of segment S. The exception to this rule is the case when the TCP connection is using the timestamp option [JBB92] (however, the use of timestamps in the Internet appears to be limited [All00]).

  Traditional RTO timers have been based on course-grained clocks (e.g., 500 ms). In addition, [AP99] shows that large minimum RTOs are required to protect against spurious retransmissions. [PA00] specifies that the minimum RTO should be 1 second. We believe that with such a lower bound on the RTO the slight timing problems introduced by reordering will have only small effects on the RTO estimate. Therefore this problem is not further studied in this paper, although an investigation of this effect in real networks is an area for future work.

The investigation presented in this paper uses the information provided by the recently standardized *duplicate selective acknowledgment* (DSACK) option [FMMP00] to make TCP more robust in the face of reordering. We present simulation results using various techniques for changing the way TCP senders decide to retransmit data segments. This paper is intended to be a preliminary investigation on these algorithms and further testing over real networks is encouraged to determine the efficacy of the various mechanisms introduced in this paper in the face of more realistic reordering patterns.

The remainder of this paper is organized as follows. § 2 outlines the simulation environment we used in our investigation, including an outline of the changes we made to the *ns* simulator. § 3 discusses various methods that allow a TCP sender to detect spurious retransmissions. § 4 provides a brief illustration of the problem network reordering causes for TCP connections. § 5 examines a simple scheme to use the DSACK option to improve TCP performance in the face of packet reordering. § 6 discusses several methods for making TCP's retransmission decisions more adaptive, therefore making TCP more robust to network reordering. Our results are given in § 7. Finally, our conclusions and an outline of future work in this area is given in § 8.

2 Simulation Environment

We used *ns-2.1b7-snapshot-20000816* as the basis for our investigation. We added a number of new features to the simulator and patched several bugs. We used the *sack1* variant of TCP outlined in [FF96] with extensions to support DSACK for all experiments outlined in this paper. Appendix A outlines the changes we made to the *sack1* TCP variant and the scoreboard in order to make both work correctly in the face of reordered segments.

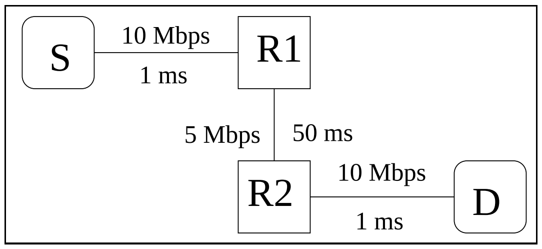


Figure 1: Simulated network topology.

2.1 Network Topology and Setup

Figure 1 illustrates the topology used in our experiments. The sending host, S, and destination host, D, are each connected to a router via 10 Mbps networks. The two routers, $R1$ and $R2$, are connected to each other over a 5 Mbps link with a one-way delay of 50 ms. All our simulations use 1500 byte segments, currently a common packet size on the Internet [All00]. The sender uses the the *sack1* variant of TCP with a maximum congestion window of 500 segments to simulate the use of autotuned socket buffers [SMM98]. The TCP sender never reaches the advertised window in our experiments indicating that the characteristics of the network path are dictating the performance of the transfer. The window size used also guarantees that a single TCP flow is able to congest the network such that packets are dropped by the bottleneck routers. Finally, the TCP sender uses a clock granularity of 500 ms for the retransmission timer. The TCP receiver uses the *sack1* TCP sink with delayed ACKs. The delayed ACK timer is implemented as a heartbeat timer with 200 ms granularity. The routers in our topology use maximum queue sizes of 65 segments and a drop-tail queueing strategy.

2.2 Reordering Router

The stock version of *ns* does not provide a good way to introduce segment reordering for experimentation. We implemented a command that can be used in simulation scripts to swap two segments in a router's queue at a given time. As outlined in the following sections, we varied the number of random queue swaps and the frequency that we forced reordering. The disadvantage of our method of introducing reordering is that a queue must have formed for the reordering to happen. However, this is consistent with the finding in [BPS99] that shows a relationship between reordering and congestion. Our topology has been engineered such that a persistent queue is formed by our TCP traffic, so that reordering happens as we expect.

We are not familiar with a good model of reordering as it happens in real networks, however we believe that future research should attempt to create such a model for use in future simulation studies. Our approach is designed to create reordering events in two dimensions. Informally, these two notions are: (i) how often reordering takes place, and (ii) how many packets are involved in the reordering event. We believe that this method of introducing reordering into our traffic is sufficient to explore TCP behavior when faced with a range of reordering behavior.

2.3 Traffic Pattern

All of the experiments in this paper are conducted using a single bulk TCP transfer. While not a particularly realistic traffic pattern, such a transfer allows us to gauge the ideal performance of the algorithms we investigate in a controlled manner. Clearly additional experiments involving these algorithms in real networks with more realistic traffic patterns and reordering patterns is necessary before widespread use of these algorithms is suggested. Additionally, the lack of competing traffic in our simulations causes all reordering to be applied to one connection. This is likely the worst-case scenario. Reordering events would not be as harmful to TCP senders if the reordered segments were all from different connections.

Note that we do not consider short transfers in this paper. While most of the connections on the Internet are short-lived flows [TMW97], we do not expect the algorithms discussed in this paper to be useful to short data transfers, as short transfers do not have time to increase their congestion window before terminating. Mechanisms that examine past reordering events in an attempt to make TCP more robust against future events are therefore not likely to have a large impact on short transfers. However, an area of future work may be to gauge the ability and the efficacy of sharing reordering information across TCP connections (similar to the sharing congestion control state, as proposed in the literature [Tou97, BRS99]).

3 Detecting Spurious Retransmits

The first key item required to mitigate the impact of reordering on TCP performance is the ability for the TCP sender to detect spurious retransmissions. Several methods for determining when TCP has sent a needless retransmission have been proposed, as follows:

1. The Eifel algorithm [LK00] uses the TCP timestamp option [JBB92] or two bits from the TCP reserved field to disambiguate an original transmission from a retransmission. Eifel is robust to up to a congestion window's worth of lost acknowledgments. When using the reserved bits, the algorithm requires negotiation of Eifel during the initial three-way handshake used to initiate every TCP connection.

2. The DSACK option [FMMP00] allows a TCP receiver to report to the sender any duplicate segments that arrive. Using DSACK information and a history of which segments have been retransmitted, the sender can determine when a retransmission is likely spurious. A single DSACK notification is sent in one acknowledgment for each duplicate data segment that arrives. Because DSACKs are only sent once, DSACK is not robust to ACK loss. That is, if an ACK containing DSACK information is dropped or corrupted by the network, the information about that particular segment is lost and the sender will never detect the spurious retransmission.

3. A new option could be designed that reported the arrival of duplicate data segments in a more robust fashion than DSACK. For instance, the receiver could report the information in several acknowledgments, much like SACK information is currently transmitted. Such an option would increase the probability of the TCP sender obtaining all available information about spurious retransmissions.

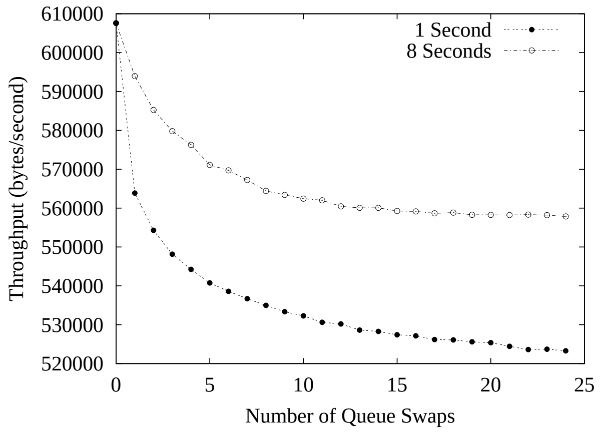
4. [AP99] proposes a method for timing the ACK of a retransmitted segment. If the ACK returns in less than $\frac{3}{4} \cdot RTT_{min}$, where RTT_{min} is the minimum RTT observed thus far in the connection, the retransmission is likely spurious. This method has been shown to be effective in determining whether a retransmission based on the RTO was required, but has not yet been evaluated on retransmissions triggered by the fast retransmit algorithm.

There are real-world tradeoffs in choosing a mechanism to detect needless retransmissions, however investigating these tradeoffs is outside the scope of this paper. The goal of this paper is to investigate appropriate behavior *after* a spurious retransmission has been detected, and any of the above mechanisms would have worked for the purpose of this investigation. We therefore chose to use the DSACK option because it is the only alternative that has been standardized at the time of this study.

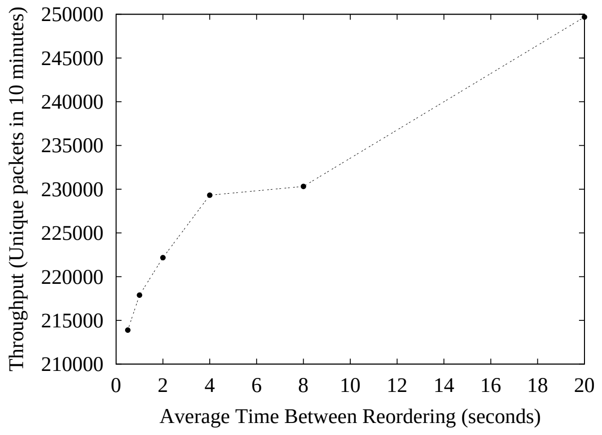
4 The Impact of Reordering

This section provides a simplistic evaluation of the impact of reordering on bulk TCP transfers. Our goal in this section is to illustrate that reordering does in fact hurt performance, and to provide a baseline for subsequent sections. Also note that the absolute results presented in this section are less important than the qualitative results. In different environments with a different (more realistic) traffic mix, we would expect different results. The goal of this section is to illustrate the pattern caused by various levels of reordering.

Figure 2 shows the average throughput of a 10 minute TCP connection with periodic reordering events. Figure 2(a) illustrates the throughput as a function of the number of random queue swaps performed roughly every 1 or 8 seconds (the actual interval was randomly determined using a Poisson process with a mean of 1 or 8 seconds). Figure 2(b) shows the throughput impact as a function of the average interval between reordering events (each consisting of 12 random queue swaps). Each point on both plots is the mean of 30 simulations.



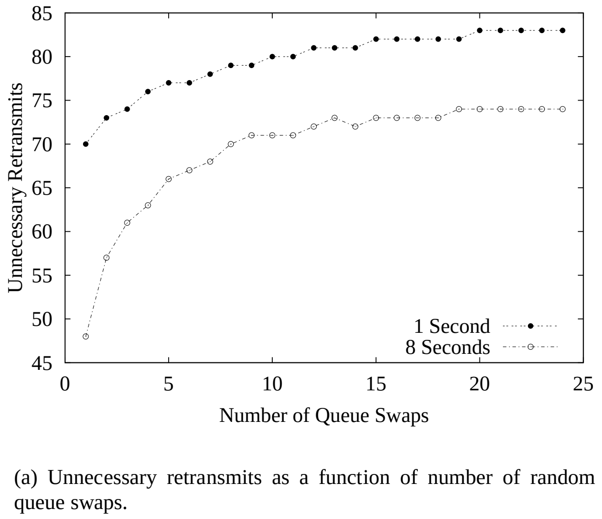
(a) Throughput as a function of the number of random queue swaps.



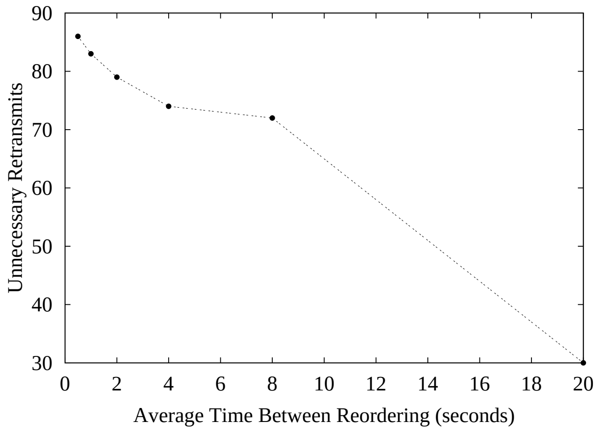
(b) Throughput as a function of the frequency of reordering.

Figure 2: Performance of standard SACK-based TCP when faced with packet reordering.

As shown in figure 2(a), the throughput is indirectly proportional to the number of packets swapped in the queue. Also, as the number of swaps grows the throughput stabilizes. This shows that after a certain point the reordering nearly always causes a needless fast retransmit (and halving of *cwnd*). Once a spurious fast retransmit is triggered, the sending rate is reduced in a uniform fashion. After this point, therefore, reordering has little *additional* effect. In addition, these plots indicate that throughput drops as the frequency of the reordering events increases.



(a) Unnecessary retransmits as a function of number of random queue swaps.



(b) Unnecessary retransmits as a function of the frequency of reordering.

Figure 3: Number of unnecessary retransmissions when faced with packet reordering.

Figure 3 shows the number of unnecessary retransmissions sent during our simulations. Figure 3(a) shows the number of spurious retransmits as a function of the number of random queue swaps. The queue swaps are performed roughly every 1 and 8 seconds for comparison. Meanwhile, Figure 3(b) shows the number of needlessly retransmitted segments as a function of the average time between reordering events (each of which consists of 12 random queue swaps). Again, each data point represents an average of 30 simulations. The plots show that the number of spurious retransmissions increase with the number of queue swaps per reordering event. This confirms the reason for the drop in throughput shown in Figure 2. Additionally, Figure 3 shows that as the interval between reordering events is reduced, the number of spurious retransmissions increases as expected.

5 "Undoing" Bad Congestion Control Decisions

As discussed in § 3, a number of possible methods for detecting spurious retransmissions exist. As mentioned, we chose to use a conservative algorithm based on the DSACK option because DSACK is currently a standards track mechanism while the other methods are still being researched. Further, we believe that the results shown in this paper are likely to be similar using alternate methods for detecting spurious retransmits.

Using the DSACK option [FMMP00], a TCP sender is informed about duplicate segments that arrive at the receiver. Duplicate segments can be caused by either a spurious retransmission sent by the TCP sender or by some quirk in the network that causes packet replication. [Pax97] shows that packet replication by the network is exceedingly rare. A DSACK therefore has a high probability of reporting a spurious retransmission. As an additional check, the TCP sender should ensure that the segment reported as arriving multiple times was actually retransmitted. Once a TCP sender determines that a spurious retransmission has occurred using the DSACK information, the effect the retransmit had on the congestion control state can be corrected.

We utilize the following changes to TCP's congestion control state processing (as generally outlined in [FMMP00]):

- Upon detection of a packet loss (i.e., congestion indication), we save the value of the congestion window (*cwnd*) as $cwnd_{prev}$ before reducing the congestion window.

- Upon the arrival of a DSACK and the determination that the duplicate was caused by a spurious retransmission, the sender notes this until the "loss" recovery event is finished.

- Once the "loss" recovery event is completed, we check to make sure that no *real* loss was detected. If all retransmits were found to be spurious, the slow start threshold (*ssthresh*) is set to $cwnd_{prev}$. This causes the TCP sender to use slow start to increase *cwnd* to its value prior to the spurious retransmission.[1]

We changed the *ns sack1* model to perform as described above when using DSACK. In addition, the SACK algorithm in *ns* made several assumptions that did not allow for the graceful handling of DSACK, which required several changes to the algorithm as outlined in Appendix A.

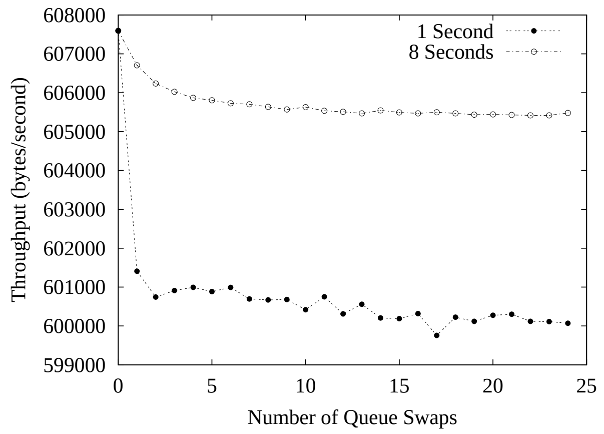


Figure 4: Throughput of TCP with DSACK as a function of the number of queue swaps.

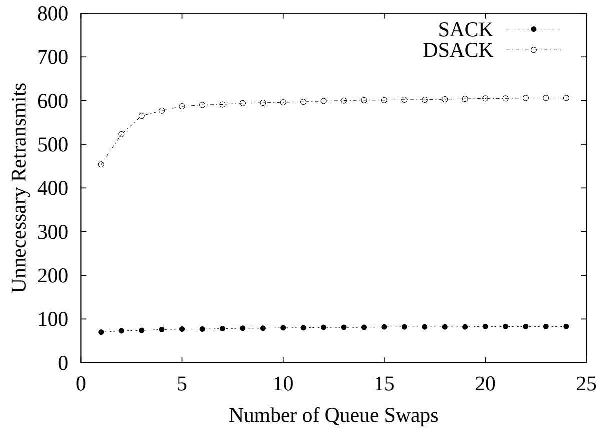


Figure 5: Number of spurious retransmits with and without DSACK as a function of the number of queue swaps.

The above modifications allow a TCP sender to mitigate the throughput problems caused by spurious retransmissions. Figure 4 shows the throughput obtained by a DSACK enabled TCP as a function of the number of queue swaps performed. The plot shows that even under persistent reordering (approximately one event per second) DSACK allows TCP to maintain throughput. The decrease in throughput is roughly 1% in the worst case shown on the plot.

[1]We have chosen to modify *ssthresh* instead of modifying *cwnd* directly to prevent the injection of a (potentially) large burst of segments into the network.

The cost of the additional performance is that TCP with DSACK is much more aggressive in comparison to TCP without DSACK. Figure 5 shows the number of spurious retransmissions performed with and without DSACK when reordering is introduced at intervals of approximately one second. As shown, using DSACK increases the number of bad retransmissions by roughly a factor of six. This is caused by DSACK TCP's ability to keep the congestion window large when compared to standard SACK TCP, therefore causing it to experience more reordering than a typical SACK TCP sender that is forced to slow down in response to each mistake. These spurious retransmissions waste network resources and could contribute to *congestion collapse* [FF99]. The next section explores a number of ways to mitigate this problem in DSACK while retaining the performance benefits.

6 Avoiding Mistakes

While the algorithm in § 5 enables TCP to effectively recover from unneeded congestion control adjustments, it is not sufficient to solve the entire reordering problem. By reopening *cwnd* without attempting to prevent further unnecessary retransmissions, we are effectively allowing TCP to inject potentially large amounts of useless data into the network (as shown in figure 5). These useless packets could lead toward congestion collapse. Without DSACK, TCP is required to reduce the sending rate and effectively "pays the price" for sending unnecessary retransmissions. If the disadvantage of sending spurious retransmission is eliminated, however, some new algorithm is required to attempt to prevent future retransmissions caused by reordering.

The ideal solution is obviously for the sending TCP to know precisely how much reordering is present in the network path at any given time so retransmissions can be appropriately triggered. However, given the characteristics of IP, this is difficult to determine in the general case. Therefore, TCP must empirically measure the amount of reordering perceived in the network path and adjust the retransmission algorithms accordingly.

The fast retransmit algorithm provides protection against reordering events that cause slight reordering such that the receiver does not generate 3 duplicate acknowledgments (e.g., two successive packets being swapped in the network). By waiting until the arrival of the third duplicate ACK to trigger a retransmission the TCP sender can successfully disambiguate loss from reordering in these cases. We experiment with compensating for reordering by making the threshold that triggers fast retransmit a variable, *dupthresh*, and adjusting the variable based on the amount of reordering measured in the network path.

This is a more delicate process than it first seems. If the adjusted value is not large enough, TCP will continue to send unnecessary retransmissions. On the other hand, if the threshold becomes too large, fast retransmit may not be triggered at all and loss will be recovered via the (often costly) RTO timer. The next several subsections outline various algorithms for adjusting *dupthresh*. The point of this paper is not necessarily to provide a compelling case for one of these algorithms over the others. The goal is to explore (via simulation) the advantages and disadvantages of each scheme. We believe that experimentation over the Internet is required before making a decision on which of these algorithms is “the best”.

6.1 Extending the Limited Transmit Algorithm

Before we consider changing the duplicate ACK threshold required to trigger fast retransmit, we must extend the Limited Transmit algorithm [ABF01] to ensure that the ACK clock is preserved during a reordering or loss event. Limited Transmit calls for sending a new (previously unsent) segment upon receipt of each of the first two duplicate ACKs in the hopes of ensuring that even when operating with a small *cwnd* we can generate enough duplicate ACKs to trigger fast retransmit and not rely on the retransmission timer. Our version of Limited Transmit adds transmission of new data on every second duplicate ACK that arrives after the first two. This keeps the ACK clock going while reducing the sending rate in case congestion has occurred (by sending only half as much data as the incoming ACKs are acknowledging). This scheme is similar to the rate-halving congestion control mechanism [MSML99].

In addition to keeping data and ACK packets in the network path so that feedback continues this extension reduces the potential burst caused by increasing the threshold to enter fast retransmit (as will be discussed below). For instance, assume it takes the receipt of 10 duplicate acknowledgments to trigger fast retransmit to properly disambiguate loss from reordering. Say the sender receives 9 duplicate ACKs and then a new cumulative ACK (i.e., there was a reordering event). If we were to use TCP *without* extending Limited Transmit we would burst on the order of 10 packets into the network when the non-duplicate ACK arrives. However, with our extension to limited transmit the TCP sender would burst roughly 5 segments into the network. The Limited Transmit extension does not completely prevent bursts, but ameliorates them to some degree.

We also note that depending on how well the algorithms presented in the following subsections actually disambiguate reordering and loss in real networks the community may wish to revisit this algorithm and send new data on *every* duplicate ACK before fast retransmit is triggered rather than on every second duplicate ACK (which we believe to be the conservative first approach). Transmitting on each duplicate ACK would likely eliminate the bursting problem a larger duplicate ACK threshold creates in the general case.

6.2 Constant Increase of the Duplicate ACK Threshold

The first mechanism we introduce is to simply increase *dupthresh* by some value K every time we detect a spurious fast retransmit. This algorithm has the advantage of being simple to implement. An associated disadvantage is that it may take a number of “mistakes” before TCP determines the appropriate value of *dupthresh* for the current network conditions. The actual performance of the algorithm depends on the amount of reordering happening in the network, the value of K and the value of *cwnd*. For the experiments presented in this paper, we used $K = 1$.

6.3 Increasing Threshold Based on Length of Reordering Event

The next algorithm attempts to use the length of the reordering event as the basis for increasing *dupthresh*. The TCP sender first determines the number of duplicate ACKs that would have disambiguated reordering from loss, C. The average of C and *dupthresh* is then used as the new value of *dupthresh*, with the additional guarantee that *dupthresh* is incremented by at least 1. The advantage of this scheme is that a TCP sender may converge to the optimal value of *dupthresh* after fewer mistakes than when simply increasing *dupthresh* by some fixed constant as proposed in the last subsection. The disadvantage is that a single, lengthy reordering event may inflate *dupthresh* unreasonably and thus cause a later timeout, while increasing by only a small constant on each mistake makes such pathological behavior less likely to cause the TCP connection to experience an RTO on the next real packet loss.

6.4 Using a Duplicate ACK Threshold and a Timer

Additionally, we tested a method first outlined in [Pax97] that calls for the use of 3 duplicate ACKs in addition to a small amount of time to trigger retransmits. If an acknowledgment for the segment believed to have been lost arrives before the timer fires, the pending retransmission is cancelled. We base the amount of time we wait on the amount of time that would have been required to obtain enough duplicate ACKs to disambiguate reordering from loss in previously experienced reordering events. At face value this method is essentially the same as the mechanism outlined in the previous subsections (gauging the number of duplicate ACKs we need to observe). However, using the passage of time rather than the arrival of duplicate ACKs may be more robust to ACK loss, as well as to the size of the reordering event. The disadvantage of this method is that it requires an additional timer for each TCP connection, which is more overhead than the previously discussed methods.

6.5 Using a Running Average of the Duplicate ACK Threshold

This algorithm keeps an exponentially weighted moving average (EWMA) of the length of perceived reordering events, and adjusts *dupthresh* accordingly. Each time a pure reordering event of length N duplicate ACKs is detected, the EWMA is updated, as follows:

$$avg = \begin{cases} \alpha \cdot N + (1-\alpha) \cdot avg & \text{if } N > avg \\ (\alpha \cdot x) \cdot N + (1 - \alpha \cdot x) \cdot avg & \text{otherwise} \end{cases} \tag{1}$$

Where the EWMA gain, α, and the multiplicative factor, x, varied in our simulations. We then update *dupthresh* based on the new *avg*, as follows:

$$dupthresh = \lfloor avg + 0.5 \rfloor \tag{2}$$

At initialization, *dupthresh* and *avg* are set to 3 duplicate ACKs.

6.6 Reducing the Duplicate ACK Threshold

In our simulations, when a TCP sender uses the RTO to trigger a retransmission we take this as an indication that the current estimate of the amount of reordering in the network is invalid and reset *dupthresh* to 3 duplicate ACKs. When the RTO fires either (i) TCP's current estimate of *dupthresh* is outdated such that enough duplicate ACKs did not arrive to trigger fast retransmit before the RTO expired, or (ii) the amount of ACK loss was sufficient to prevent the fast retransmit algorithm from detecting loss. In either case *dupthresh* requires adjustment so that TCP can continue to operate effectivly. The method we use (reduce *dupthresh* back to 3) is conservative in that it is no worse than current TCP implementations. However, we envision additional adjustment techniques may lead to better overall performance and should be studied in future work (e.g., reducing *dupthresh* by half when the RTO expires).

In addition, if a TCP stack varies its duplicate ACK threshold to compensate for reordering and subsequently experiences actual packet loss causing *cwnd* to be reduced below *dupthresh*, the sender may be unable to generate enough duplicate ACKs to trigger a fast retransmit. In order to avoid this situation, *dupthresh* must always be less than *cwnd*. In our simulations, we cap *dupthresh* at 90% of *cwnd*, with a maximum of $cwnd - 1$ segments. While we did not vary these constants in our simulations we did not notice any performance impact from their choice.

7 Results

Figure 6 shows the throughput (number of data bytes per second) as a function of the number of queue swaps performed approximately every 1 second for the various *dupthresh* compensation schemes outlined above. The TCP connection is 10 minutes long, as in the previous experiments. As shown on the plot, all the compensation schemes except tracking *dupthresh* with an EWMA ($\alpha = 1$; $x = \frac{1}{16}$) improve throughput slightly over the case when no *dupthresh* compensation is employed (denoted "DSACK–No mitigation" on the plot). Using an EWMA hurt performance slightly when the reordering was heavy. All lines shown on the plot are within approximately 1% of each other; therefore we conclude that the impact any particular compensation scheme has on performance is minimal and that the performance increase comes from the ability to revert to the previous congestion control state when a retransmission is determined to be spurious.

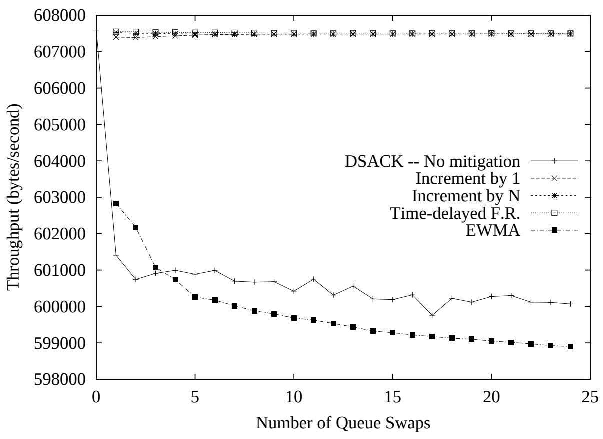


Figure 6: Throughput as a function of the number of queue swaps for various *dupthresh* compensation schemes.

Figure 7 (with the y-axis plotted on a log-scale) shows the effects of the algorithms described above on unnecessary retransmissions. Each of these algorithms reduces the number of unnecessary retransmissions when compared to reverting the congestion window without any at-

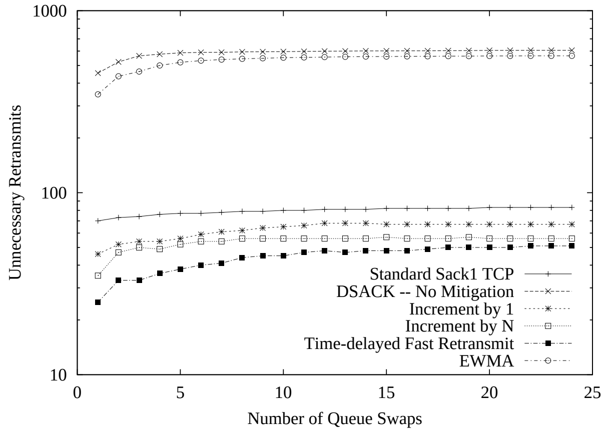


Figure 7: Median number of spurious retransmissions as a function of the number of queue swaps.

tempt to compensate for premature fast retransmits caused by the reordering. Furthermore all algorithms except the algorithm using an EWMA (again $\alpha = 1$; $x = \frac{1}{16}$) generate fewer unnecessary retransmissions than stock *ns sack1* TCP (without any capacity for correcting bogus changes to the congestion control state). Finally, all schemes for adjusting *dupthresh* reduce the number of unnecessary retransmissions when compared to using DSACK to revert the congestion control state without attempting to change *dupthresh*.

The plot shows that the best algorithm, in terms of avoiding needless retransmits, is the time-delayed fast retransmit algorithm. The time-delayed fast retransmit algorithm and the increase-by-N algorithm are similar in that they attempt to retransmit at the same point and only differ in the triggering event. The former algorithm uses the passage of a certain amount of time, while the latter uses the receipt of a certain number of duplicate ACKs. The simulation results show that using a timeout is more robust than using the receipt of a given number of duplicate ACKs. This is explained because ACKs can be lost or the algorithm may be slightly off in predicting the number of duplicate ACKs that should be used to trigger fast retransmit. However, using a timeout has the advantage that these events do not hinder the firing of the fast retransmission.

The figure also shows that the increase-by-N algorithm performs fewer needless retransmissions than the increase-by-1 algorithm. The difference shows that increasing *dupthresh* by 1 duplicate ACK per needless retransmit provides a slower convergence time than increasing by the desired amount at one time. Using an EWMA to track the appropriate *dupthresh* is not nearly as effective as the other schemes in these simulations. Below we consider the implications of choosing different values for the α and x on the performance of the EWMA.

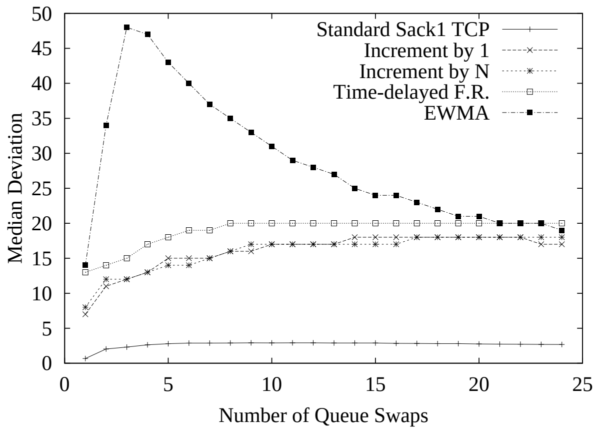


Figure 8: Median deviation of unnecessary retransmissions.

Figure 8 shows the median deviation of each scheme from its median performance. Under our model of reordering, these algorithms performed with varying degrees of consistency in their effectiveness at preventing unnecessary retransmissions. The algorithms that prevented more unnecessary fast retransmits on median also performed less consistently, with a median deviation approaching half of the median number of unnecessary retransmissions. In addition to allowing more needless retransmissions, the EWMA tracking of *dupthresh* also shows wider variation than the other schemes.

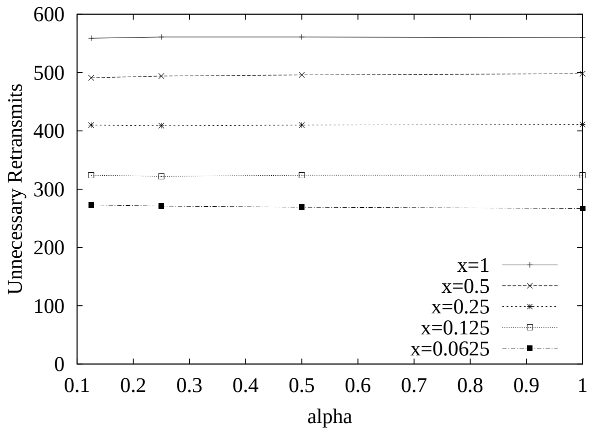


Figure 9: Unnecessary retransmissions as a function of α for various values of x.

Figure 9 illustrates the number of spurious retransmissions as a function of α for various values of x. As the figure shows, the different values of α have very little impact on the *dupthresh* estimate formed. The plot does show, however, that the difference between the gain for increasing and decreasing the EWMA makes a significant difference in the number of spurious retransmits triggered. When the rate of increase and decrease is the same ($x = 1$), the plot shows over 550 needless retransmissions over the course of the simulation. Meanwhile, when increasing sixteen times as fast as decreasing the number of unneeded retransmissions drops to roughly 275. However, even when $x = 1/16$ the number of spurious retransmits is significantly higher than any of the other methods we used to adjust *dupthresh*, which all triggered under 100 unnecessary retransmits in our simulations.

8 Conclusions and Future Work

The simulation results presented in this paper suggest methods that can be used to effectively mitigate the performance impact reordering imposes upon TCP. In many cases, a TCP capable of restoring its congestion control state upon discovery that a spurious retransmit has occurred performs nearly as well under heavy reordering conditions as a standard TCP without reordering. The approach of simply restoring congestion control state each time a spurious retransmit is detected, however, leads to an undesirable increase in the number of unnecessary retransmissions injected into the network. We determined that some method of preventing the spurious retransmits *a priori* is desirable and showed several methods that look promising for estimating the proper value for *dupthresh*.

Suppression of spurious retransmits was found to be effectively controlled by modifying the conditions under which the fast retransmit algorithm is initiated. Adjustment of the duplicate acknowledgment threshold used to trigger fast retransmit and the insertion of a small delay before transmitting the “lost” packet were found to be effective means of reducing the frequency of spurious retransmissions under the variety of reordering conditions studied in the investigation presented in this paper. Meanwhile, our simulations involving an EWMA estimate of the proper duplicate ACK threshold show that the method does not work nearly as well as the other methods.

Future work in this area includes:

- The simulated results presented in this paper need verification in the real network. Our model of reordering, as mentioned in Section 2, is simplistic and may not accurately represent the behavior of physical networks. While it is sufficient to validate the methods outlined here as proof-of-concept ideas, quantitative results regarding the absolute efficacy of these algorithms using this simulation model is not advised.

- Further, a realistic model of reordering based on empirical observations would improve the accuracy of future simulations. While the prevalence of reordering in the network has been documented, our present understanding of the qualities of reordering is incomplete.

- The possibility of proactively avoiding spurious retransmits should be researched. The algorithms in this paper are reactive, correcting a spurious retransmission that has already occurred and taking steps to prevent future mistakes. It would be desirable to avoid the unnecessary retransmission in the first place.

Acknowledgments

We would like to thank Sally Floyd for numerous helpful tips and discussions, as well as the initial work of adding DSACK notification to *ns*. We would also like to thank Joseph Ishac, Hans Kruse, Reiner Ludwig, Shawn Ostermann and Craig Partridge for their suggestions and discussions during the development of this work. Finally, we thank the anonymous CCR reviews for a number of suggestions that improved this paper.

References

[ABF01] Mark Allman, Hari Balakrishnan, and Sally Floyd. Enhancing TCP’s Loss Recovery Using Limited Transmit, January 2001. RFC 3042.

[All00] Mark Allman. A Web Server’s View of the Transport Layer. *Computer Communications Review*, 30(5):10–20, October 2000.

[AP99] Mark Allman and Vern Paxson. On Estimating End-to-End Network Path Properties. In *ACM SIGCOMM*, September 1999.

[APS99] Mark Allman, Vern Paxson, and W. Richard Stevens. TCP Congestion Control, April 1999. RFC 2581.

[BPS99] Jon Bennett, Craig Partridge, and Nicholas Shectman. Packet Reordering is Not Pathological Network Behavior. *IEEE/ACM Transactions on Networking*, December 1999.

[BRS99] Hari Balakrishnan, Hariharan Rahul, and Srinivasan Seshan. An Integrated Congestion Management Architecture for Internet Hosts. In *ACM SIGCOMM*, September 1999.

[FF96] Kevin Fall and Sally Floyd. Simulation-based Comparisons of Tahoe, Reno, and SACK TCP. *Computer Communications Review*, 26(3), July 1996.

[FF99] Sally Floyd and Kevin Fall. Promoting the Use of End-to-End Congestion Control in the Internet. *IEEE/ACM Transactions on Networking*, 7(6), August 1999.

[FMMP00] Sally Floyd, Jamshid Mahdavi, Matt Mathis, and Matt Podolsky. An Extension to the Selective Acknowledgement (SACK) Option for TCP, July 2000. RFC 2883.

[Jac88] Van Jacobson. Congestion Avoidance and Control. In *ACM SIGCOMM*, 1988.

[JBB92] Van Jacobson, Robert Braden, and David Borman. TCP Extensions for High Performance, May 1992. RFC 1323.

[KP87] Phil Karn and Craig Partridge. Improving Round-Trip Time Estimates in Reliable Transport Protocols. In *ACM SIGCOMM*, pages 2–7, August 1987.

[LK00] Reiner Ludwig and Randy Katz. The Eifel Algorithm: Making TCP Robust Against Spurious Retransmissions. *Computer Communication Review*, 30(1), January 2000.

[MC00] Sean McCreary and K. Claffy. Trends in Wide Area IP Traffic Patterns A View from Ames Internet Exchange. May 2000. http://www.caida.org/outreach/papers/AIX0005/.

[MSML99] Matt Mathis, Jeff Semke, Jamshid Mahdavi, and Kevin Lahey. The Rate-Halving Algorithm for TCP Congestion Control, August 1999. Internet-Draft draft-mathis-tcp-ratehalving-00.txt (work in progress).

[PA00] Vern Paxson and Mark Allman. Computing TCP's Retransmission Timer, November 2000. RFC 2988.

[Pax97] Vern Paxson. End-to-End Internet Packet Dynamics. In *ACM SIGCOMM*, September 1997.

[Pos81] Jon Postel. Transmission Control Protocol, September 1981. RFC 793.

[SMM98] Jeff Semke, Jamshid Mahdavi, and Matt Mathis. Automatic TCP Buffer Tuning. In *ACM SIGCOMM*, September 1998.

[TMW97] Kevin Thompson, Gregory Miller, and Rick Wilder. Wide-Area Internet Traffic Patterns and Characteristics. *IEEE Network*, 11(6):10–23, November/December 1997.

[Tou97] Joe Touch. TCP Control Block Interdependence, April 1997. RFC 2140.

A TCP SACK Changes

After introducing segment reordering into the traffic pattern used in our simulations, we found several problems with the *ns sack1* implementation. These changes do not have any impact on transfers that do not experience packet reordering, as the validation tests included with *ns* still succeed with our changes. The following are the changes we made:

- We only consider an ACK to be a "duplicate ACK" (for the purposes of triggering fast retransmit) when it contains new SACK information. This has the effect of preventing DSACKs from triggering fast retransmit.

- The initialization of "pipe" in *ns* made some assumptions about limited transmit that, while usually correct without our modifications, were insufficient for our purposes. This initialization was modified to correctly calculate the amount of outstanding data at the beginning of loss recovery in all cases.

- As outlined in [FF96], the SACK algorithm decrements the *pipe* variable (the sender's estimate of the number of segments currently in the network) by 2 segments whenever a partial ACK is received. Partial ACKs are assumed to indicate that the original transmission and the retransmission have both left the network. However, when reordering is present a partial ACK may not be associated with a retransmitted segment. Therefore, we limit the number of times the algorithm is allowed to decrement *pipe* by 2 segments to the number of segments retransmitted. Otherwise, *pipe* is decremented by 1 segment for each partial ACK that arrives.

- As outlined above, reordering may cause a number of "normal" looking ACKs during "loss" recovery. Per the above restriction, the sender will decrease *pipe* by 1 segment for most of these ACKs. However, when the receiver is using delayed ACKs the incoming ACKs may indicate that more than one segment has left the network. Therefore, we introduced a new rule whereby if the incoming ACK does not contain SACK information and the scoreboard is empty the ACK is taken at face value and *pipe* is decremented by the number of new segments cumulatively ACKed.

- The scoreboard data structure required numerous changes to cope with DSACK information.