# Estimating Loss Rates With TCP*

Mark Allman†
International Computer Science Institute
mallman@icir.org

Wesley M. Eddy, Shawn Ostermann
Ohio University
{weddy,ostermann}@eecs.ohiou.edu

## Abstract

Estimating loss rates along a network path is a problem that has received much attention within the research community. However, deriving accurate estimates of the loss rate from TCP transfers has been largely unaddressed. In this paper, we first show that using a simple count of the number of retransmissions yields inaccurate estimates of the loss rate in many cases. The mis-estimation stems from flaws in TCP's retransmission schemes that cause the protocol to spuriously retransmit data in a number of cases. Next, we develop techniques for refining the retransmission count to produce a better loss rate estimate for both Reno and SACK variants of TCP. Finally, we explore two SACK-based variants of TCP with an eye towards reducing spurious retransmits, the root cause of the mis-estimation of the loss rate. An additional benefit of reducing the number of needless retransmits is a reduction in the amount of shared network resources used to accomplish no useful work.

## 1 Introduction

Assessing network properties is a topic that has received a great deal of attention in the literature. Among the measurement techniques developed by the research community is a set of methods to derive information about the dynamics of a path from TCP [Pos81] connections. For instance, [Pax97] assesses the dynamics of a number of paths through the analysis of pairs of sender-side and receiver-side TCP traces, while [JD02] details techniques for assessing the round-trip time of a path by watching TCP segments from an arbitrary location in the network, and [BPS99] uses TCP transfers to explore the prevalence of packet reordering. These are but a sampling of a rich range of papers in the literature.

This paper adds to the body of measurement techniques by detailing and validating a method for estimating the loss rate experienced by a TCP connection by observing the connection's segments close to the data sender (or in the sender-side TCP implementation). Previous work in the literature has assessed TCP segment losses by comparing segment traces from the two endpoints of a TCP connection [Pax97] or by monitoring only the data segments of a connection at some point in the middle of the network [BV02]. Our goal is to monitor the connection at only the sender-side and to be as accurate as possible. Hence we leverage information from both the data and ACK streams.

There are several attractive applications and properties of TCP sender-side estimation of the loss rate, including:

- A proposal for Cumulative Explicit Transport Error Notification (CETEN) [KAPS02, EOA03] requires either that the network provide explicit and fine-grained information about the level of congestion or that TCP be able to estimate this based on the loss rate observed. [KAPS02] notes the problems with using a simple count of the number of retransmissions as an indication of the level of network congestion. We explore this problem empirically in § 3. While [KAPS02] uses explicit information from the network, a lighter weight scheme whereby the sender could accurately assess the loss rate of the network would be easier to deploy (as discussed in [EOA03]).

- Measuring the loss rate of networks using tools like *ping* (or the like) may provide an unrealistic estimate of the loss rate a TCP application will actually experience for several reasons. First, *ping* is generally rate-based and therefore does not share TCP's sending pattern, which inherently effects the loss probability of the segments. In addition, it is hard to determine some "right" rate for sending measurement probes into the network. If the rate is too low the measurement is necessarily gross and may not capture certain characteristics of the network. On the other hand, if the rate is too high, the measurement traffic will be disruptive and the measurement will end up being biased by its own traffic. These issues are explored further in [MA01]. While using some form of random sampling may mitigate these disadvantages somewhat, such a probing scheme still fails to capture TCP's burstiness or its dependence on the

*This paper appears in ACM Performance Evaluation Review, December 2003.

†Mark Allman was with BBN Technologies and supported by NASA's Glenn Research Center when this research was conducted.

feedback loop. Estimating the loss rate using sender-side TCP information (or traces) is attractive in that it derives a loss rate on timescales that matter to applications and the estimate is formed using an accepted network-friendly sending rate.

- Estimating loss rates based only on information available at the sending side of a TCP connection allows researchers to measure networks in which they only control one side of a TCP connection. This makes wide-scale measurement easier than the case when monitoring points on both ends of the connection are necessary (e.g., as used in [Pax97]).

- Deriving loss rates using TCP can aid the research community in verifying and refining our TCP models (e.g., [MSMO97, PFTK98]) using sender-side only traces.

- Comparing loss rates with TCP's retransmission rate offers insight into the effectiveness of TCP's retransmission strategies.

We present several techniques for determining the loss rate experienced by a TCP connection. The first is a simple count of the number of retransmissions. We then introduce $L$oss $E$stimation $A$lgorithm$S$ for $T$CP ($LEAST$) for TCP Reno and TCP with selective acknowledgments (SACK) ($LEAST_r$ and $LEAST_s$ respectively) and present validations of both algorithms. The measurements highlight the large difference between the actual number of losses and the number of retransmits TCP uses to repair those losses. Finally, we test a second SACK-based loss recovery algorithm with an eye towards reducing the number of spurious retransmissions sent (and, therefore, reducing the complexity of loss estimation techniques).

This paper is organized as follows. § 2 outlines our experimental environment, tools and methodology. § 3 discusses the accuracy of using a simple count of the number of retransmissions as an estimate of the loss rate. § 4 discusses our TCP Reno loss estimator ($LEAST_r$), while § 5 discusses our SACK-based version of the loss estimator ($LEAST_s$). § 6 discusses an implementation path for choosing which $LEAST$ variant to use for a given transfer. § 7 discusses a second SACK-based loss recovery algorithm that may aid $LEAST$ by using more accurate accounting practices during loss recovery. Finally, § 8 offers conclusions and suggests future work.

# 2 Methodology

To evaluate $LEAST$, we use transfers conducted across the NIMI measurement mesh [PMAM98, PAM00]. We use the bulk transfer capacity [MA01] tool *cap* [All01] to conduct the transfers. This section describes the TCP variants we tested, our experimental methodology, and provides a high level description of the measurements taken.

## 2.1 TCP Variants

We used a number of TCP variants in our investigation as follows:

- **Reno**. This version supports TCP's basic congestion control algorithms: slow start, congestion avoidance, fast retransmit and fast recovery [APS99].

- **SACK**. This version builds on TCP's standard congestion control algorithms by using the selective acknowledgment (SACK) option as specified in [MMFR96] and the loss recovery algorithm outlined in [FF96]. SACKs are used to enhance TCP's cumulative acknowledgment scheme by allowing the receiver to provide fine-grained feedback about exactly which segments have arrived.

- **SACK+DSACK**. This version builds on both the standard congestion control algorithms and the SACK enhancements by adding the use of the DSACK option [FMMP00]. DSACKs allow the receiver to inform the sender about segments that have arrived more than once.

Note: In our experiments we use only Reno and SACK+DSACK transfers. Since the DSACK option does not change any of TCP's on-the-network algorithmic dynamics, we can ignore the DSACK information in our analysis to study the SACK without DSACK case.

While real TCP implementations use byte-based sequence numbers for reliability (and ordering), *cap* is based on segment numbers for simplicity. In this paper, we will discuss our algorithms in terms of segment numbers. We believe the transformation to byte counts is fairly straightforward, but will require a bit of care in accounting for things like retransmits that do not include exactly the same sequence space as the original transmission and like problems.

Finally, TCP Reno is susceptible to a phenomenon called *successive fast retransmits* [Flo95]. In this situation, spurious retransmissions cause enough duplicate ACKs to trigger the fast retransmit algorithm during recovery which ($i$) reduces TCP's congestion window needlessly and ($ii$) often triggers additional spurious retransmits. [FH99] outlines a "bugfix" that prevents these successive fast retransmits from triggering. Our TCP Reno implementation does not use this bug fix for two reasons. First, we believe that estimating the loss rate without the bugfix is more difficult than when the bugfix is implemented so we are testing our estimation techniques in the

worst case environment. Without using the bugfix, spurious fast retransmits and the duplicate ACKs they trigger are common. Therefore, the loss recovery process is messier without bugfix [Flo95] and therefore we believe it provides a more rigorous test of our loss estimation techniques. The second reason for not using the bugfix is that we have no information on its implementation status in real world TCP implementations and therefore did not want to make an unrealistic assumption that would hinder the application of our techniques in real networks.

## 2.2 Platform

Our experiments involve a mesh of 14 NIMI hosts using the *cap* bulk transfer capacity tool to take measurements. The NIMI machines are hosted by research centers and universities. Of the 14 NIMIs used in our experiments, 8 are located in the United States, 4 in Europe, 1 in the Far East and 1 in South America. Both Reno and SACK+DSACK are implemented in *cap*. We scheduled a transfer between two randomly chosen hosts at intervals chosen by a Poisson process with a mean of 60 seconds. Each transfer consists of 5000 segments, using a packet size of 1500 bytes, an infinite advertised window (simulating automatic socket buffer tuning [SMM98]) and the TCP timestamp option [JBB92]. We collect packet traces from both the sender and receiver and compare them to obtain the actual loss rate for a given connection. We run $LEAST$ across only the sender-side packet trace to assess the algorithm's ability to estimate the loss rate along the path.

## 2.3 Measurements

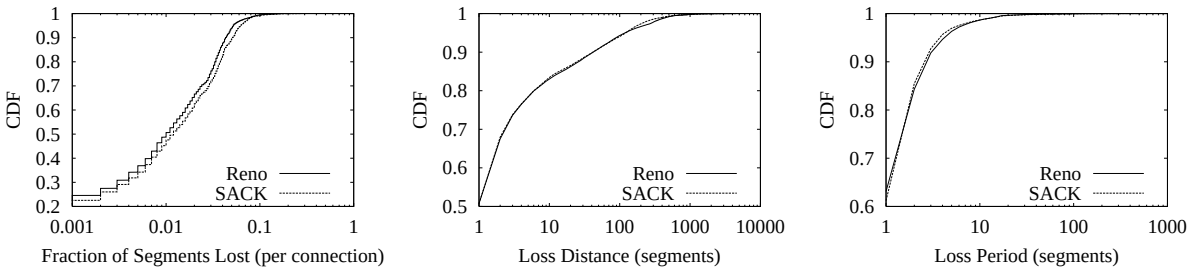
We scheduled 16320 transfers between February 24, 2003 and March 10, 2003. Of the scheduled transfers, we ended up with a dataset of 5123 transfers. The final dataset consists of 2546 valid Reno transfers and 2577 valid SACK+DSACK transfers. For the transfers not in the final dataset, the failures were caused by a myriad of problems in the network and the NIMI mesh. The largest problem was that we scheduled tests involving 6 additional NIMI hosts (above the 14 hosts described above) that turned out to be misconfigured and could not support our measurements (this accounts for roughly 30% of the scheduled experiments). In addition, time synchronization problems between machines caused the source and sink of the transfer to execute at different times and hence no transfer is conducted (even after introducing a 5 minute window to try to mitigate this). Another example failure is the route between two hosts being lost during the transfer. [PAM00] discusses the problems of taking measurements in the NIMI mesh. While the failure rate is high, we do not believe our results are biased since we need only a sample of transfers with a variety of loss rates and loss patterns to assess our loss estimation techniques.

Figure 1 illustrates several characteristics of the loss present in our dataset. The first plot shows that the transfers in the dataset experienced a variety of per connection loss rates. Over 20% of the transfers (both Reno and SACK) experienced no losses. This is explained by the quality of the connections between some of the NIMI hosts. Over a number of paths, the 5000 segment transfers used in our study did not load any links to the point of packet loss. As an example, one of the paths included in our dataset is between hosts at the International Computer Science Institute and the Lawrence Berkeley National Laboratory, both of which are in Berkeley, CA, USA. The hosts are separated by roughly 1.5 miles over a path with an RTT of roughly 5 ms and bandwidth of 100 Mbps. While at different institutions, these hosts are essentially connected via a LAN-type network, explaining why transfers between them are loss free. At the other end of the spectrum, from figure 1(a) we also note that a small percentage (0.6%) of the connections experienced a loss rate of more than 10%.

The second plot in figure 1 shows the *loss distance* [KR02]. The loss distance is calculated for each lost packet $P$ and is defined as the number of packets sent since the last packet loss. For example, if the $3^{rd}$ and $5^{th}$ packets are lost the loss distance for packet 5 will be 2. This metric provides information about both how clustered the losses are and how often losses are experienced. The plot shows that roughly 50% of the loss distances are 1, indicating that half of the losses belong to larger groups of losses (e.g., clumped losses that happen at the end of slow start). We also note a range of loss distances, with over 15% of the distances being greater than 16 segments.

The final plot in the figure shows the distribution of *loss periods* [KR02]. The length of a loss period is the number of losses that occur in a row. In this figure we again see a range in the loss patterns. Most of the loss periods are 1 segment in length (over 60%), however we did note one loss period of 886 segments!

All the plots show that the loss characteristics of Reno and SACK are largely similar. We believe the differences between the two versions of TCP correspond to SACK's aggressiveness in keeping the *cwnd* larger than Reno and therefore keeping more segments in the network, as well as TCP's bursty sending pattern in slow start-based loss recovery (which happens more frequently in Reno than SACK). Finally, we believe that figure 1 shows that our experiments cover a variety of loss characteristics (which is the key to evaluating a loss estimator).




(a) Loss rate. (b) Loss distance. (c) Loss period.

Figure 1: Per connection loss characteristics across NIMI mesh.

# 3 Retransmissions

Because TCP is a reliable transport, all segments lost in the network should be repaired with retransmissions from the data originator. Therefore, a natural first choice for estimating the loss rate experienced by a connection is to count the number of retransmissions. Figure 2 provides the distribution of the percent difference between the actual loss rate and the retransmission rate for the Reno and SACK transfers in the NIMI mesh. For TCP Reno transfers, we have found that retransmits exactly estimate the loss rate in roughly 26% of the transfers. However, in roughly two-thirds of the transfers, using retransmits as an estimate of the loss rate is off by more than 10%. Further, in approximately 16% of the transfers, the discrepancy between retransmissions and losses is over 100%. Finally, the median percent difference between the number of retransmits and the actual number of losses in the Reno transfers is roughly 33%.

This discrepancy between retransmits and losses in Reno TCP is explained by the use of slow start after the retransmission timer (RTO) fires. In this mode, and in the absence of SACK blocks informing the sender exactly which segments have arrived, Reno often retransmits packets that have not been clearly indicated as lost. For example, say segment $N$ is retransmitted after the expiration of the RTO timer. The reception of segment $N$ will cause an ACK covering segment $M$ to be sent (where $M \geq N$). As previous work has shown [Hoe96, FH99], this ACK indicates that segment $M + 1$ has likely been lost. Since the TCP sender is in slow start, segments $M+1$ and $M+2$ are retransmitted. However, the sender received no indication that $M + 2$ needs to be retransmitted – and, as figure 2 implies, in a large number of cases the sender is guessing wrong and needlessly retransmitting data. This whole process is further aggravated because, without selective acknowledgments, TCP is more prone to relying on the RTO timer to repair losses [FF96].

In the absence of SACK information, TCP may still be able to be more intelligent about which packets are transmitted, thus reducing the number of needless retransmissions. For instance, the use of NewReno [Hoe96, FH99] refines the retransmission algorithms in an effort to be more precise in retransmitting data.

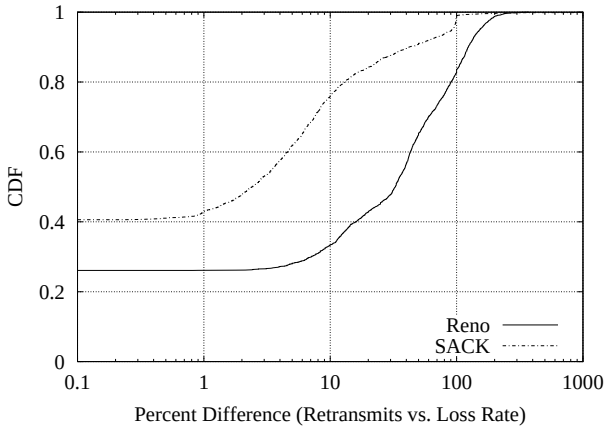



Figure 2: Accuracy of retransmissions as an estimate of the loss rate.

Figure 2 also shows the performance of retransmissions as a loss estimator using TCP SACK. The plot shows a significant improvement over Reno, with a median difference between the loss rate and the retransmission rate of roughly 2% and with roughly 75% of the estimates within 10% of the actual loss rate. However, the plot shows that there are still cases where TCP SACK retransmits unnecessarily and hence skews the loss estimate. We find that the cause of these needless retransmits is the pattern of SACK information that is sent from the receiver after a timeout. Upon a timeout, the sender clears its copy of the SACK scoreboard (per RFC 2018 [MMFR96]). In addition, the receiver always acknowledges the blocks corresponding to the most recently transmitted blocks.

Therefore, the receiver sometimes does not re-populate the sender's scoreboard appropriately and so the sender believes that some packets need to be retransmitted even though they have arrived at the receiver. This problem is discussed in greater detail in § 7.

Using the number of retransmissions as the basis for a loss rate estimate may work for some applications (especially for TCP with SACK). However, for applications that require a more accurate estimate of the loss rate, we explore leveraging information from the ACK stream to refine the retransmission-based estimate. Our algorithms are detailed in the next sections.

# 4 LEAST for TCP Reno

This section discusses the TCP Reno version of the loss estimator, $LEAST_r$.

## 4.1 Algorithm

Figure 3 shows the Python code for implementing the $LEAST_r$ algorithm. The principle behind $LEAST_r$ is that, after the RTO timer fires and TCP starts using slow start-based loss recovery, needless retransmissions will trigger the receiver to transmit duplicate acknowledgments. For instance, consider the case when a sequence of four packets, $n...n+3$, is transmitted and only $n+2$ arrives at the receiver. Figure 4 shows the sequence of events after the RTO timer fires, as follows:

1. The TCP sender times out and resends segment $n$, causing the receiver to send an ACK covering segment $n$ (i.e., expecting segment $n+1$).

2. Upon reception of the ACK covering segment $n$, the sender increases the congestion window (*cwnd*) to 2 segments. The incoming ACK points to the next missing segment, $n+1$, that the sender retransmits. However, since *cwnd* is now 2 segments, the sender also retransmits $n+2$ even though no specific knowledge about whether $n+2$ has been lost has arrived at the sender.

3. When segment $n+1$ arrives at the receiver an ACK for segment $n+3$ is generated (since $n+2$ is now the highest in-order data segment that has arrived).

4. When the spurious copy of segment $n+2$ arrives at the receiver a second ACK covering segment $n+3$ is transmitted.

$LEAST_r$ uses the receipt of this second (duplicate) ACK for segment $n+3$ as an indication that a spurious retransmission occurred. A counter tracks the number of unneeded retransmissions and the $LEAST_r$ estimate is calculated as:

$$LEAST_r = R_{total} - R_{spurious} \quad (1)$$

where $R_{total}$ is the total number of retransmissions and $R_{spurious}$ is the estimated number of unnecessary retransmissions. We count duplicate ACKs for accumulation in $R_{spurious}$ during the "RTO event" – which starts when the RTO timer fires and ends when the TCP sender receives an ACK for the highest segment outstanding when the event was initiated. We found a number of situations that cause errors in $LEAST_r$, which will be illustrated in the next subsection. While the principle of counting duplicate acknowledgments seems straightforward, the algorithm given in figure 3 contains several rules to cover special cases, as follows:

- Duplicate ACKs that do not cover the segment most recently transmitted via the RTO timer should not be taken as indicating that spurious segments arrived at the receiver. To understand why, assume segment $n$ is lost, fast retransmitted and lost again. Eventually, the RTO timer will fire and segment $n$ will be retransmitted for a second time. However, between the fast retransmit and the expiration of the RTO timer, fast recovery governs the sending of segments. Any segments sent will trigger duplicate ACKs for segment $n$ (i.e., the receiver is expecting segment $n$). These acknowledgments do not indicate spurious retransmissions. However, if the RTO timer fires while these ACKs are still streaming into the sender, they would skew the $LEAST_r$ estimate without this rule.

- When the RTO timer expires after the connection is already in slow start-based loss recovery, the current event must be extended to account for the most recent segment retransmitted via the RTO timer and the outstanding data at the time of the latest RTO timer expiration.

- The point at which the event is terminated needs to be extended when previously unsent data is transmitted during the event in order to catch the last few (possible) duplicate ACKs.

- When an ACK arrives that passes the recovery point, we add the number of spurious retransmits we have counted in the current event to the total $R_{spurious}$ count. Ideally, we are counting only duplicate ACKs for spurious retransmissions. However, as outlined below, situations arise that cause our count to be wrong. Therefore, as a double check, we actually add the minimum of the number of duplicate ACKs counted during the event (i.e., spurious retransmits) and the number of retransmits sent during the event (which is an upper bound on the number of spurious retransmits sent). This rule does not necessarily

```
seqno = ackno = highdata = highack = retransmits = dup_xmits = 0
in_rto_event = False

for pkt in snd_trace:
    if pkt.IsAck () and (pkt.AckNo () > highack):
        highack = pkt.AckNo ()

    if pkt.IsData ():
        if pkt.SeqNo () > highdata:
            highdata = pkt.SeqNo ()
        else:
            retransmits += 1
            ## an RTO that initiates slow start-based loss recovery
            if not in_rto_event and pkt.IsRTO ():
                in_rto_event = True
                recovered = recovered_orig = highdata
                rto_segment = pkt.SeqNo ()
                event_retrans = 1
                event_dupacks = 0
                continue

    ## in slow start-based loss recovery
    if in_rto_event:
        if pkt.IsData ():
            ## count retransmits in the event
            if pkt.IsRetrans () and (pkt.SeqNo () < recovered):
                event_retrans += 1

            ## an RTO within the RTO event; extend the event
            if pkt.IsRTO ():
                recovered = recovered_orig = highdata
                rto_segment = pkt.SeqNo ()

            ## track new packets sent during recovery -- we need to
            ## account for the last few duplicate ACKs
            if not pkt.IsRetrans () and (highack <= recovered_orig):
                recovered = pkt.SeqNo ()

        else:
            ## an ACK that terminates the RTO event
            if pkt.AckNo () > recovered:
                dup_xmits += min (event_dupacks, event_retrans)
                in_rto_event = False

            ## count duplicate ACKs received in the event -- but, not
            ## any associated with the RTO segment (which are not caused
            ## by needless retransmissions)
            elif (pkt.AckNo () == last_ackno) and (pkt.AckNo () >= rto_segment):
                event_dupacks += 1

    ## track the last ACK number
    if pkt.IsAck ():
        last_ackno = pkt.AckNo ()

least = retransmits - dup_xmits
```

Figure 3: TCP Reno LEAST algorithm.

make our estimate exactly right; however, it bounds the error.

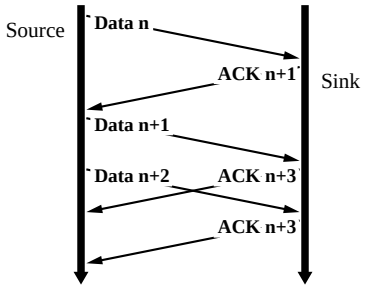



Figure 4: Sample TCP retransmit pattern after RTO timer fires.

## 4.2 Validation

As outlined in § 2, we obtained 2546 valid Reno transfers from the NIMI measurement mesh. We calculated the loss rate for each transfer by analyzing both sender and receiver traces and comparing the packets transmitted by the sender with those arriving at the receiver. We then use the sender-side trace to derive a $LEAST_r$ estimate of the loss rate. In the discussion below, we also use a *corrected* version of $LEAST_r$ that uses receiver-side information to confirm the sources of error in the estimate.

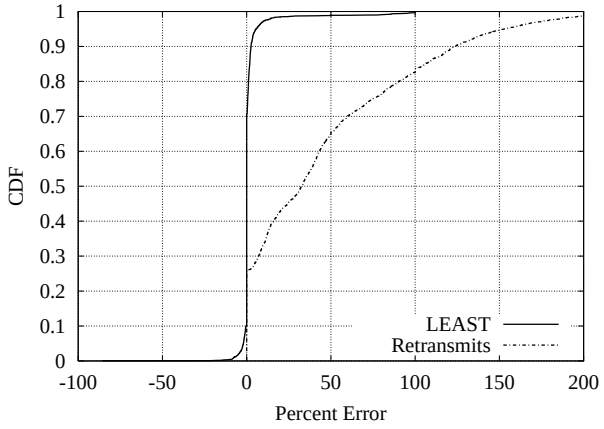



Figure 5: Accuracy of $LEAST_r$ compared to the actual loss rate (percent differences that are less than zero indicate underestimates while differences greater than zero indicate overestimates).

Figure 5 shows the distribution of the percent error between the actual loss rate for the connections and the loss rate derived from $LEAST_r$. In addition, for comparison, we plot the percent error in using retransmits as a loss estimate (discussed previously in § 3 in more detail). When using $LEAST_r$, roughly 56% of the transfer loss rate estimates are exactly right, with roughly 3% of the estimates differing from the (measured) loss rate by more than 10%. These results again highlight the grossness of Reno's retransmission behavior. Using the straightforward $LEAST_r$ estimator, we are able to predict the loss rate to within 10% in over 96% of the transfers in our dataset. We now look at several categories of errors found in the $LEAST_r$ estimate.

### 4.2.1 Sources of Error

As we will show below, the five sources of error explained in this section account for the majority of the error in the $LEAST_r$ estimate. Each error listed below is given a one-letter identifier for the purposes of the discussion of the results.

**Spurious Fast Retransmit ($\mathcal{A}$).** This error is not caused by the $LEAST_r$ algorithm itself, but rather from spurious fast retransmits caused by packet reordering in the network. Sufficient packet reordering can cause spurious fast retransmits [BPS99, BA02]. However, since $LEAST_r$ only works after the RTO timer fires, these spurious fast retransmits are counted in $R_{total}$ but since they are not detected as spurious they are not counted in $R_{spurious}$. Therefore, this causes an overestimate in $LEAST_r$.

**Lost Duplicate ACKs ($\mathcal{B}$).** When a duplicate ACK that indicates a spurious retransmission to the $LEAST_r$ algorithm is lost in the network, the information about a spurious retransmit is lost. For instance, in figure 4, if the second ACK for $n+3$ is lost, the sender will be unable to determine whether the retransmission of segment $n+2$ was required. Losing duplicate ACKs during the RTO event causes overestimation.

**Spurious Retransmit Triggers Partial ACK ($\mathcal{C}$).** Sometimes we have noticed spurious retransmits resulting in partial ACKs[1] arriving at the TCP sender. This problem works hand-in-hand with the lost duplicate ACK phenomena described above. Losing a partial ACK causes a subsequent duplicate ACK to be effectively turned into a partial ACK and therefore the information conveyed in the duplicate ACK will be lost. If the first ACK for $n+3$ in figure 4 is lost, the second (duplicate) ACK for $n+3$ will look like a partial ACK when arriving at the sender. Hence, there will be no second (duplicate) ACK following to indicate a spurious retransmit occurred. Losing partial ACKs during the RTO event causes overestimation.

**Lost Retransmissions ($\mathcal{D}$).** A lost retransmission during slow start-based loss recovery can trigger duplicate ACKs that do not indicate spurious retransmits and hence skew the estimate. Figure 6 shows the sequence of events after the RTO timer fires (for segment $n$) in the case where seg-

[1]A partial ACK covers previously unacknowledged data, but not enough previously unacknowledged data to terminate recovery.

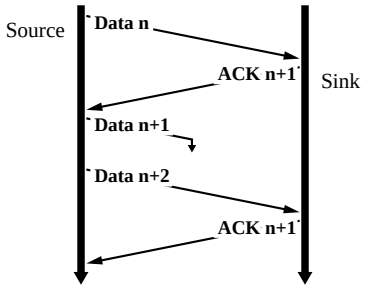



Figure 6: Example of sequence of events following a lost retransmission.

ments $n$, $n+1$ and $n+2$ are lost before the RTO timer expires. The diagram shows the events that occur when the retransmit of segment $n+1$ is lost (for a second time). In this case, the ACK returned for segment $n+2$ is a duplicate ACK, which $LEAST_r$ takes as indicating a spurious retransmit even though the retransmit of segment $n+2$ is required in this case. This series of events yields an underestimate in the $LEAST_r$ algorithm.

**Spurious Fast Retransmit Terminates Recovery ($\mathcal{E}$).** As outlined in [Flo95], TCP Reno is susceptible to successive fast retransmits within a window of data. If one of these successive fast retransmits happens at the end of slow start-based loss recovery, a duplicate ACK indicating that the retransmission was spurious will fall outside the "recovery event". Therefore, the $LEAST_r$ algorithm will not detect the duplicate ACK. While we may be able to extend the event to wait for the resulting duplicate ACK, it is not obvious how long to extend the event waiting for an ACK that may or may not arrive – or, may arrive delayed or out-of-order. $LEAST_r$ makes the assumption that the retransmit was necessary and thus overestimates when such a retransmit is sent needlessly. This overestimate is limited to the number of RTO events that happen within a transfer.

The above list of estimation errors is likely not complete. Other sources of error could be packet reordering, packet duplication or other situations yet unknown. However, as will be shown in the next section, we believe the above sources of error capture the major causes of estimation error found in our measurements.

#### 4.2.2 Quantifying Errors

In our NIMI dataset, $LEAST_r$ exactly matched the loss rate in roughly 56% of the transfers (or 1418 of the 2546 Reno transfers). In 57 of these exact matches, $LEAST_r$ mis-estimates in more than one of the above outlined ways, but the estimation problems exactly cancel out to yield a correct overall estimate.

Next we observe that in 10% (or 256) of the transfers in our NIMI dataset, $LEAST_r$ underestimates the loss rate. The only source of underestimation outlined in § 4.2.1 is duplicate ACKs returned for needed retransmits because a needed retransmission was lost ($\mathcal{D}$). We found that correcting our estimate based on the problems identified in § 4.2.1 yields an exact accounting of the errors in roughly 69% of the transfers yielding an underestimate. In another 20% of the transfers the corrected estimate is closer to the actual loss rate, while 5% of the transfers ended up being overestimates after the correction. Finally, in 5% of the transfers yielding an underestimate, the correction had either no effect or increased the underestimation. From this analysis we believe that while we have not found all the causes of underestimation in our data we have identified the major causes.

Next we turn our attention to the 34% (or 872) of the transfers in our NIMI dataset in which $LEAST_r$ overestimates the loss rate. Of those, 71% of the overestimates can be exactly corrected by taking into account the sources of error from § 4.2.1. In these transfers, duplicate ACK losses ($\mathcal{B}$) are the largest cause of error in the estimate with 45% of the error, followed by receiving partial ACKs in response to spurious retransmits ($\mathcal{C}$) with 36% of the error, reordering causing spurious fast retransmits that are not accounted for by $LEAST_r$ ($\mathcal{A}$) with 14% of the error and lost retransmits triggering duplicate ACKs ($\mathcal{D}$) with 3.7% of the error. In 29% of the overestimates in which the mis-estimate could not be exactly corrected for, we noted a variety of sources of error. However, we note that the median difference between the corrected estimate and the actual loss rate is roughly 0.6%, indicating that we have identified the majority of the errors that skew the estimate.

Finally, we note that the phenomenon whereby a spurious fast retransmit is sent at the end of the RTO event ($\mathcal{E}$) discussed in § 4.2.1 is not a large contributor to the error in $LEAST_r$. This case accounts for less than 1% of the error in $LEAST_r$ across all our TCP Reno NIMI transfers, indicating that the assumption outlined in 4.2.1 that the retransmit is needed does not greatly skew $LEAST_r$.

### 4.3 Summary

In this section we have shown that the $LEAST_r$ estimate is accurate within 10% of the actual loss rate in over 96% of the transfers. Furthermore, for roughly 56% of the transfers, $LEAST_r$ exactly matches the loss rate. In addition, when $LEAST_r$ does not match the loss rate, we have identified the vast majority of the errors in the estimate such that we believe that (given only the information available at the sender-side of a TCP connection), $LEAST_r$ is forming a near-optimal estimate of the loss rate.

```
highdata = retransmits = dup_xmits = 0

for pkt in snd_trace:
    if pkt.IsData ():
        if pkt.SeqNo () > highdata:
            highdata = pkt.SeqNo ()
        else:
            retransmits += 1

    if pkt.IsACK ():
        if using_DSACK and pkt.DSACK () and WasRexmted (pkt.DSACK ()):
            dup_xmits += 1
        elif not using_DSACK and IsSACKRedundant (pkt):
            dup_xmits += 1

least = retransmits - dup_xmits
```

Figure 7: TCP SACK LEAST algorithm.

# 5 LEAST for TCP SACK

In this section we explore the $LEAST_s$ variant for TCPs that support the selective acknowledgment option [MMFR96]. Selective acknowledgments allow a TCP receiver to inform the TCP sender about the sequence space that has actually arrived at the receiver in a more fine-grained way than simply using the standard cumulative acknowledgment mechanism. As shown in § 3, the use of SACK allows TCP to be more accurate in resending data and therefore the number of retransmits is a better estimate of the loss rate than when using TCP Reno. However, needless retransmissions are still sent by SACK-based algorithms and therefore we have developed a mechanism that uses clues in the returning ACKs to form a better loss estimate.

## 5.1 Algorithm

The $LEAST_s$ algorithm is given in figure 7. The first portion of the code counts all retransmits. The second half of the `for` loop in the code is used to estimate the number of spurious retransmits sent. The code is different depending on whether the receiver supports the DSACK option [FMMP00].

If the receiver ($i$) supports the DSACK option [FMMP00], ($ii$) the incoming acknowledgment contains DSACK information and ($iii$) the DSACK information reported is for a retransmitted segment then the TCP sender considers the retransmission to be spurious[2]. DSACK blocks are only sent on one acknowledgment packet. Therefore, if an ACK with a DSACK is lost in the network, the information conveyed in the DSACK will not be resent and the sender's estimate of the loss rate will be affected.

If the receiver does not support DSACK, the $LEAST_s$ algorithm looks for *redundant* SACKs. That is, returning ACKs that do not advance the cumulative ACK point and contain no previously unknown SACK information. Such an ACK is assumed to have been caused by a needlessly retransmitted data segment that does not update the state of the receiver's buffer. However, ACK reordering can also cause an ACK to be deemed redundant in the case where a later ACK passes an earlier ACK in the network and conveys the same information (and likely more) than the earlier ACK. When the ACK that was originally transmitted first (but, arrives second) is processed by the sender, all the information contained within the ACK is redundant, hence meeting our criteria for being counted as indicating a needless retransmission and fooling our algorithm.

As with $LEAST_r$, the SACK variant of the algorithm is also susceptible to packet duplication in the network path. However, when using DSACK, the sender has some protection by ensuring that a segment reported as arriving multiple times was actually retransmitted by the sender.

## 5.2 Validation

As outlined in § 2, we obtained 2577 valid SACK transfers from the NIMI measurement mesh. We calculated the loss rate for each transfer by comparing the sender and receiver traces. We then use the sender-side trace to derive

[2]Note: to work optimally, the receiver should not delay an ACK containing DSACK information. This advice agrees with [APS99]'s guidance that out-of-order arrivals should trigger immediate ACKs. If an ACK with a DSACK must be delayed, the DSACK information should be included in the delayed acknowledgment. In our experiments the receiver immediately transmits an ACK when a DSACK is required.

two $LEAST_s$ estimates of the loss rate (with and without DSACK information).

Figure 8 shows the distribution of the percent difference between the actual loss rate and our estimates. Using the number of retransmits in the connection (as discussed in § 3) is the worst of our estimators. The two remaining estimators use the $LEAST_s$ algorithm to attempt to correct for spurious retransmissions. As the plot shows, when using DSACK, nearly 96% of the estimates are exact, with less than 1% of the connections experiencing estimates that are more than 10% different from the actual loss rate. When DSACK is not used and a count of the redundant (S)ACKs is employed instead, we note approximately 60% of the estimates being exact, with roughly 9% of the connections showing estimates that are more than 10% different from the actual number of losses.

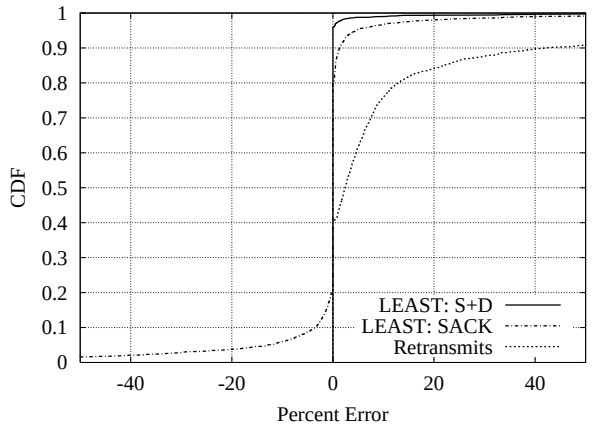



Figure 8: Accuracy of $LEAST_s$ compared to the actual loss rate (percent differences that are less than zero indicate underestimates while differences over zero indicate overestimates).

When using DSACK, the error in the $LEAST_s$ algorithm is always explained by lost spurious retransmits or lost ACKs that contain DSACK information. The source of error could be reduced by either ($i$) making DSACK more robust to ACK loss by sending it more than once (ala the rest of the SACK information) or ($ii$) by making TCP's retransmission machinery less likely to transmit spurious retransmits (see § 7). In addition, several non-DSACK schemes for detecting spurious retransmits have been outlined in the literature [LK00, SKR02] and may be generalizable to the problem of finding spurious retransmits for loss estimation.

Without DSACK, $LEAST_s$ is more prone to misestimation. To determine the source of error, we use the $LEAST_s$ algorithm to analyze the trace made by the receiver so that we see the exact stream of acknowledgments that are sent. Using the receiver-side acknowledgment stream yields exact $LEAST_s$ estimates in over 84% of the connections and estimates within 10% of the actual loss rate in over 99% of the connections. In addition, the underestimation shown in figure 8 is eliminated. This analysis illustrates that most of the error in the $LEAST_s$ (without DSACK) estimates is caused by the packet dynamics along the network path traversed by the ACKs (e.g., losses, reordering, etc.). Therefore, we believe that improving $LEAST_s$ further would require either more information (from more than just the sender's vantage point) or heuristics that try to infer additional information from the ACK stream.

# 6 Implementation Path

In the previous sections, we have described three variants of $LEAST$: $LEAST_r$ for TCP Reno connections and two variants of $LEAST_s$ for connections that support SACK (with and without the DSACK option). The natural question that arises in the face of three different algorithms is which to use for a given (arbitrary) TCP connection (or trace). The following is a sketch of a scheme that can be used to determine which variant of $LEAST$ to employ.

1. If either of the hosts involved in a connection fails to advertise support for selective acknowledgments in TCP's three-way handshake, then loss estimation should proceed using $LEAST_r$.

2. Assuming SACK is supported by both hosts in a connection, loss estimation can proceed using one of the SACK variants. Unfortunately, the DSACK option is not negotiated during the connection setup phase. Therefore, which variant of $LEAST_s$ to use is not immediately obvious. Two approaches to loss estimation are possible, as follows.

   In the case when LEAST is being computed by a TCP implementation or by some form of measurement tool based on TCP (ala *sting* [Sav99] or TBIT [PF01]), the first two data segments sent can explicitly overlap by 1 byte. If the receiver supports DSACK, this overlap will cause a DSACK to be reported in the returning ACK. Therefore, based on the returning ACK, the TCP can determine which variant of $LEAST_s$ to employ.

   On the other hand, in the case of estimating loss rates by passively monitoring TCP connections, the above active manipulation of the byte-stream is not possible. Therefore, the recommended loss estimation approach is to assume the receiver does not support DSACK until a DSACK notification arrives and then switch variants of the algorithm. To implement this approach, the code given in figure 7 is changed slightly, as follows. The `using_DSACK` boolean starts being set to "false". When ($a$) ACK

segments arrive, ($b$) `using_DSACK` is set to "false" and ($c$) the incoming ACK contains a DSACK then the `using_DSACK` flag is set to "true" and the count of duplicate transmits (`dup_xmits`) is reset to zero. This results in a fresh start at loss estimation using the DSACK variant of $LEAST_s$.

Using the above approach, $LEAST$ can be used in environments without a priori knowledge of the TCP variant being utilized (or, from traces containing a multitude of different TCP variants).

# 7 Reducing Spurious Retransmissions With SACK

As shown in previous sections, the fundamental problem with estimating the loss rate of a TCP connection is that TCP retransmissions are not an accurate reflection of the actual loss. Designing algorithms that make better choices about what to retransmit will simplify loss estimators (possibly obviating the need for anything over a retransmission count), as well as reduce the shared network resources expended on carrying traffic that accomplishes no useful work. In this section we show that more aggressive accounting of data during loss recovery with SACK can reduce the number of needless retransmits sent by TCP (and, hence, reduce the amount of estimation that an algorithm, such as $LEAST_s$, has to do to arrive at the actual loss rate). In this section we explore two different SACK-based loss recovery schemes in terms of the number of needless retransmits triggered by each algorithm.

## 7.1 Why SACK Needlessly Retransmits

The specification for TCP's selective acknowledgment (SACK) option [MMFR96] outlines the information a SACK receiver is to return to the sender when the receiver's socket buffer is non-contiguous, as follows:

- The first SACK block returned is to contain the received range of data that includes the arriving data segment.
- Any remaining option space is to be used to resend the most recent discontiguous SACK blocks transmitted.

In addition, [MMFR96] specifies that TCP senders clear any collected SACK information upon the expiration of the retransmission timer to allow for the possibility the receiver may reneg[3] on a previously sent SACK block.

[3] [MMFR96] allows a receiver to discard received data that it has not cumulatively acknowledged (to recover buffer space, for example). The receiver *renegs* by not keeping data that it *implied* to the sender (through a SACK) would not need to be retransmitted.

The above specification creates a situation where the TCP sender sometimes never obtains valuable information about data in the receiver's buffer after a timeout, which leads to the possibility that the sender will needlessly retransmit segments. As an example, table 1 shows a map of the receiver's socket buffer at the point when the TCP sender's RTO timer fires. Note that segments 1, 4, 7, 9, and 13 are missing. The latest three SACK blocks transmitted to the data sender are blocks: $B_6$, $B_8$ and $B_{10}$ (covering segment 8, segments 10-12, and segment 14 respectively). The following events occur after the RTO timer fires:

1. The sender retransmits segment 1.
2. When segment 1 arrives at the receiver, an acknowledgment is sent containing a cumulative ACK covering segment 3 and the SACK blocks $B_6$, $B_8$ and $B_{10}$ (the most recently transmitted SACK blocks).
3. When the ACK sent in step 2 arrives, the sender will increase the congestion window by 1 segment and will retransmit segments 4 and 5. While segment 4 requires retransmission at this point, segment 5 does not (as shown in table 1). However, the TCP sender has not been informed that segment 5 has arrived (since clearing the scoreboard) and therefore assumes it requires retransmission.
4. Segment 4 arrives at the receiver which generates a cumulative ACK covering segment 6 and again sends the SACK blocks $B_6$, $B_8$ and $B_{10}$. When this ACK arrives, the sender will have complete knowledge of the receiver's buffer and will not needlessly retransmit any more segments.

The above example shows a simple situation where the sender transmits one needless segment into the network. As the amount of outstanding data grows and becomes more fractured (requiring more SACK blocks to describe), the number of spurious retransmissions increases because the receiver only reports information about the far right-side of the window. This general problem is the cause for the vast majority of the needless retransmits observed in our SACK dataset described in § 5. Several possible solutions to this problem exist. For instance, the receiver could change the scheme it uses to choose which SACK blocks to include in an ACK to provide the sender with more timely information. Alternatively, the sender could be made more conservative – retransmitting segments that are in a "bounded hole" in the sequence space where the receiver has informed the sender about arrived segments on each side of the segment being retransmitted (e.g., retransmitting segment 13 before segment 5 in the above example). A third possible mitigation for this problem is to

| | Blocks of Received and Missing Segments | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | $B_1$ | $B_2$ | $B_3$ | $B_4$ | $B_5$ | $B_6$ | $B_7$ | $B_8$ | $B_9$ | $B_{10}$ |
| **Received Segment Range** | | 2-3 | | 5-6 | | 8-8 | | 10-12 | | 14-14 |

Table 1: Sample socket buffer map at receiver.

repair as much loss as possible before the RTO timer expires, thus allowing the receiver to describe the state of its buffer in fewer SACK blocks. This last solution does not attempt to fix the fundamental problem, but rather tries to avoid the problem. We experiment with a second SACK-based loss recovery algorithm to understand the degree to which it is able to repair more loss before the RTO timer expires and describe our results in the following subsections.

## 7.2 SACK Algorithm Descriptions

The SACK algorithm implemented in *cap* is based on [FF96] which, in turn, is codified in *ns2*'s *sack1* TCP variant. The algorithm keeps an estimate, $pipe$, of the number of segments in the network. When loss recovery is started, $pipe$ is initialized to the amount of outstanding data. For each duplicate ACK received during recovery, $pipe$ is decremented by 1 segment. For each segment sent (new or retransmit), $pipe$ is incremented by 1 segment. For each partial ACK received, $pipe$ is decremented by 2 segments (one for the original segment transmitted and one for the retransmit). When $pipe$ is less than *cwnd*, TCP can send (retransmitting if data for resend is available or sending new segments if not).

[BAFW03] outlines a second SACK-based loss recovery algorithm, which we will denote *sack2*, that is based on the principles of *sack1* but is more careful in estimating how much data is in the network[4]. The key difference between *sack1* and *sack2* is that *sack2* can declare a segment "lost" and therefore deduct it from the $pipe$ estimate. *Sack1* does not do this, but rather relies only on ACK arrivals to declare that data has left the network (missing the fundamental impossibility of a lost segment triggering an ACK). *Sack2*'s more aggressive estimation of $pipe$ provides (re)transmission opportunities sooner than when using *sack1*. Therefore, in the case of the RTO timer expiring (e.g., if a retransmit is, itself, lost) *sack2* has an easier job than *sack1* because *sack2* has likely repaired more loss before the RTO timer fires than *sack1*.

[4]Note: The authors of *sack1* note in [FF96] that one may be able to design a better algorithm by being more careful – but, that was beyond the scope of their initial study.

| Parameter | Range | Increment |
|---|---|---|
| $D_f$ | 25% – 75% | 5% |
| $D_d$ | 0% – 10% | 1% |
| $D_a$ | 0% – 10% | 1% |

Table 2: Ranges for simulation parameters.

## 7.3 Simulation Comparison

To explore the SACK algorithms detailed above, we wrote a small simulator in Python that models both the sender and receiver during the loss recovery phase of a TCP SACK connection. The simulator, *tcpsim*, consists of a sender and receiver separated by a link with a one-way delay of 0.25 seconds and a bandwidth of 10 Mbytes/second. The simulator's data originator uses either *sack1* or *sack2* for loss recovery. The simulator starts by transmitting a window of 250 data segments (assumed to be the last window sent in slow start, for instance). The simulation ends when loss recovery is finished – i.e., upon receipt of an ACK covering the highest segment outstanding when recovery started.

From the first window of data, the first segment, $S_1$, is always dropped (and subsequently fast retransmitted). In addition, segments are dropped from the first window of data randomly with probability $D_f$. After the first window of data transmission, *tcpsim* drops data segments with probability $D_d$ and drops acknowledgments with probability $D_a$. Table 2 outlines the parameter space used for the simulations presented in this paper. We use two different loss rates for data segments to approximate the situation at the end of TCP's slow start phase where TCP roughly doubles the congestion window every round-trip time. This causes a situation where one window of data often experiences drastically different loss characteristics than would be expected given the steady state loss rate of the network path. We conducted 30 random simulations with each permutation of the parameter space and report medians in this paper.

In addition to always dropping the first segment sent, $S_1$, *tcpsim* also always drops the first retransmit of $S_1$ to ensure that the retransmission timer (RTO) is required to recover some of the loss[5].

[5]Both *sack1* and *sack2* require the use of the RTO timer to recover from lost retransmits.

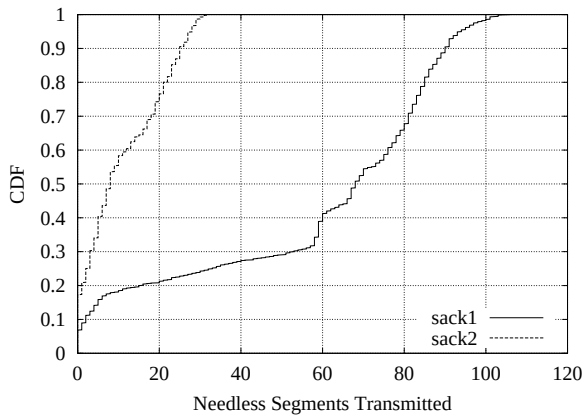



Figure 9: Distribution of needless retransmits across all *tcpsim* simulations.

Figure 9 shows the distribution of the number of needless retransmits sent by each SACK variant on each transfer. The plot shows the distribution of the median of the 30 random simulations of each loss scenario described above. As shown, the amount of needlessly retransmitted data sent by *sack1* is 3–17 times the amount spuriously sent by *sack2*. While *sack2* suffers fewer spurious retransmits, it also sends 7–39% more unique bytes during recovery than *sack1* and loss recovery takes approximately 20 seconds (or roughly 40 round-trip times) less than when using *sack1* (on median).

These results show that *sack2*'s more aggressive accounting during SACK-based loss recovery allows it to be more accurate in its overall retransmission behavior. *Sack2*'s use of a more aggressive recovery before the RTO timer fires largely avoids the problems caused by the receiver not re-populating the sender's SACK scoreboard after an RTO. In addition, we note that *sack2* uses its transmission opportunities more wisely since it sends more unique data than *sack1*. Finally, we note that *sack2*'s aggressiveness does not violate the spirit of TCP's congestion control principles [Flo00] in that multiplicative decrease is applied.

The results in this section suggest that the TCP sender's choice of which particular SACK-based loss recovery algorithm to utilize *can* have an impact on the performance of a loss estimator such as $LEAST$. By reducing the number of needless retransmits sent into the network, the TCP sender reduces the amount of *estimation* that needs to happen to accurately assess the loss rate and distills the problem to *counting* retransmissions. The loss estimation techniques outlined in this paper are still useful for assessing the loss rate on a wide variety of arbitrary traffic. However, the results of this section suggest that when using an active measurement strategy, researchers would be well served to choose a SACK-based loss recovery strategy carefully.

# 8 Conclusions and Future Work

The following are the major contributions of this paper:

- Through measurements from the NIMI mesh of measurement points, we have shown that using a count of the number of retransmissions sent by TCP provides a poor estimate of the number of packets actually lost.
- We have developed sender-side loss estimation techniques for TCP Reno, SACK and SACK with DSACK that estimate the loss rate of the network path within 10% of the actual loss rate in over 90% of the transfers we conducted over the NIMI measurement mesh.
- We have found the majority of the sources of error in the $LEAST$ estimate of the loss rate. The main causes of errors in the estimate come from network dynamics that cannot be mitigated from information only available on the sender side of the TCP connection (e.g., ACK loss).
- We found that, in some situations, TCP's SACK generation scheme does not provide the TCP sender with timely information about the state of the receiver's buffer. This triggers spurious retransmits from the TCP sender. We explored a second SACK-based loss recovery algorithm (outlined in [BAFW03]) and show that it is effective at reducing the number of needless retransmits (by roughly an order of magnitude in the cases we tested). In turn, this makes the job of accurately estimating the loss rate easier.

In addition, the results outlined in this paper bring up several areas for future research:

- In § 7 we outlined a general problem with SACK-based loss recovery after TCP's RTO timer fires (and the TCP sender purges its copy of the SACK scoreboard). The fundamental problem is that the receiver only informs the sender about the right side of the window and so the sender's retransmission of the data on the left side of the window is fairly gross. While we examined an alternate SACK algorithm that mitigates the adverse effect of the missing information, we did not fix the problem itself. Future work should include examining ways to send more timely SACK information after the RTO timer fires.
- Testing $LEAST$ against different variants of TCP (e.g., NewReno [FH99]) to assess how well the techniques apply would be useful.

- Testing the applicability of $LEAST$ to various tasks, such as modeling TCP performance or using $LEAST$ with CETEN techniques (which attempt to aid TCP performance by taking into account packets lost due to corruption when choosing a congestion response) would be useful. While the experiments outlined in this paper illustrate that the estimate of the loss rate is often "quite good", it is unclear what problems the estimate is "good enough" for and what problems need an even better estimate (which, arguably, would require multiple vantage points).

- A more complete comparison of *sack1* and *sack2* would be useful.

- We believe that merging the techniques presented in this paper with those given in [BV02] may allow for the leveraging of better loss estimation from arbitrary vantage points.

## Acknowledgments

This paper benefited from discussions with Ethan Blanton, Josh Blanton and Joseph Ishac. David Irimies helped with *tcpsim*. David Irimies and the anonymous reviewers provided valuable comments on a draft of this paper. Andy Adams and Vern Paxson provided a large amount of assistance with NIMI. Our thanks to all!

## References

[All01] Mark Allman. Measuring End-to-End Bulk Transfer Capacity. In *ACM SIGCOMM Internet Measurement Workshop*, November 2001.

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

[BA02] Ethan Blanton and Mark Allman. On Making TCP More Robust to Packet Reordering. *ACM Computer Communication Review*, 32(1):20–30, January 2002.

[BAFW03] Ethan Blanton, Mark Allman, Kevin Fall, and Lili Wang. A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP, April 2003. RFC 3517.

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

[BV02] Peter Benko and Andras Veres. A Passive Method for Estimating End-to-End TCP Packet Loss. In *Proceedings of IEEE Globecom*, 2002.

[EOA03] Wesley Eddy, Shawn Ostermann, and Mark Allman. New Techniques for Making Transport Protocols Robust to Corruption-Based Loss, July 2003. Under submission.

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

[FH99] Sally Floyd and Tom Henderson. The NewReno Modification to TCP's Fast Recovery Algorithm, April 1999. RFC 2582.

[Flo95] Sally Floyd. TCP and Successive Fast Retransmits. Technical report, Lawrence Berkeley Laboratory, May 1995.

[Flo00] Sally Floyd. Congestion Control Principles, September 2000. RFC 2914.

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

[Hoe96] Janey Hoe. Improving the Start-up Behavior of a Congestion Control Scheme for TCP. In *ACM SIGCOMM*, August 1996.

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

[JD02] Hao Jiang and Constantinos Dovrolis. Passive Estimation of TCP Round-Trip Times. *ACM Computer Communication Review*, 32(3), July 2002.

[KAPS02] Rajesh Krishnan, Mark Allman, Craig Partridge, and James P.G. Sterbenz. Explicit Transport Error Notification (ETEN) for Error-Prone Wireless and Satellite Networks. Technical Report TR-8333, BBN Technologies, March 2002.

[KR02] Rajeev Koodli and Rayadurgam Ravikanth. One-Way Loss Pattern Sample Metrics, August 2002. RFC 3357.

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

[MA01] Matt Mathis and Mark Allman. A Framework for Defining Empirical Bulk Transfer Capacity Metrics, July 2001. RFC 3148.

[MMFR96] Matt Mathis, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow. TCP Selective Acknowledgement Options, October 1996. RFC 2018.

[MSMO97] Matt Mathis, Jeff Semke, Jamshid Mahdavi, and Teunis Ott. The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm. *Computer Communication Review*, 27(3), July 1997.

[PAM00] Vern Paxson, Andrew Adams, and Matt Mathis. Experiences with NIMI. In *Proceedings of Passive and Active Measurement*, 2000.

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

[PF01] Jitendra Padhye and Sally Floyd. Identifying the TCP Behavior of Web Servers. In *ACM SIGCOMM*, August 2001.

[PFTK98] Jitendra Padhye, Victor Firoiu, Don Towsley, and Jim Kurose. Modeling TCP Throughput: A Simple Model and its Empirical Validation. In *ACM SIGCOMM*, September 1998.

[PMAM98] Vern Paxson, Jamshid Mahdavi, Andrew Adams, and Matt Mathis. An Architecture for Large-Scale Internet Measurement. *IEEE Communications*, 1998.

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

[Sav99] Stefan Savage. Sting: a TCP-based Network Measurement Tool. In *Proceedings of the 1999 USENIX Symposium on Internet Technologies and Systems*, October 1999.

[SKR02] Pasi Sarolahti, Markku Kojo, and Kimmo Raatikainen. F-RTO: A New Recovery Algorithm for TCP Retransmission Timeouts. Technical Report C-2002-07, University of Helsinki, February 2002.

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