On Estimating End-to-End Network Path Properties*

Mark Allman
NASA Glenn Research Center
and
GTE Internetworking
21000 Brookpark Rd. MS 54-2
Cleveland, OH 44135
mallman@grc.nasa.gov

Vern Paxson
AT&T Center for Internet Research at ICSI
and
Lawrence Berkeley National Labratory
1947 Center Street, Suite 600
Berkeley, CA 94704-1198
vern@aciri.org

Abstract

The more information about current network conditions available to a transport protocol, the more efficiently it can use the network to transfer its data. In networks such as the Internet, the transport protocol must often form its own estimates of network properties based on measurements performed by the connection endpoints. We consider two basic transport estimation problems: determining the setting of the retransmission timer (RTO) for a reliable protocol, and estimating the bandwidth available to a connection as it begins. We look at both of these problems in the context of TCP, using a large TCP measurement set [Pax97b] for trace-driven simulations. For RTO estimation, we evaluate a number of different algorithms, finding that the performance of the estimators is dominated by their minimum values, and to a lesser extent, the timer granularity, while being virtually unaffected by how often round-trip time measurements are made or the settings of the parameters in the exponentially-weighted moving average estimators commonly used. For bandwidth estimation, we explore techniques previously sketched in the literature [Hoe96, AD98] and find that in practice they perform less well than anticipated. We then develop a receiver-side algorithm that performs significantly better.

1 Introduction

When operating in a heterogeneous environment, the more information about current network conditions available to a transport protocol, the more efficiently it can use the network to transfer its data. Acquiring such information is particularly important for operation in wide-area networks, where a strong tension exists between needing to keep a large amount of data in flight in order to fill the bandwidth-delay product "pipe," versus having to wait lengthy periods of time to attain feedback regarding changing network conditions, especially the onset of congestion.

In a wide-area network, such as the Internet, that does not provide any explicit information about the network path, it is up to the transport protocol to form its own estimates of current network conditions, and then to use them to adapt as efficiently as possible. A classic example of such estimation and adaptation is how TCP infers the presence of congestion along an Internet path by observing packet losses, and either cuts its sending rate in the presence of congestion, or increases it in the absence [Jac88].

In this paper we examine two other basic transport estimation problems: determining the setting of the retransmission timer (RTO), and estimating the bandwidth available to a connection as it begins. We look at both problems in the context of TCP, using trace-based analysis of a large collection of TCP packet traces. The appeal of analyzing TCP in particular is that it is the dominant protocol in use in the Internet today [TMW97]. However, analyzing the behavior of actual TCP implementations also introduces complications, because there are a variety of different TCP implementations that behave in a variety of different ways [Pax97a]. Consequently, in our analysis we endeavor to distinguish between findings that are specific to how different TCPs are implemented today, versus those that apply to general TCP properties, versus those that apply to general reliable transport protocols.

Our analysis is based on the $\mathcal{N}_2$ subset of TCP trace data collected in 1995 [Pax97b]. This data set consists of sender-side and receiver-side packet traces of 18,490 TCP connections among 31 geographically-diverse Internet hosts. The hosts were interconnected with paths ranging from 64 kbps up to Ethernet speeds, and each connection transferred 100 KB of data, recorded using *tcpdump*. We modified *tcpanaly* [Pax97a] to perform our analysis.

The rest of the paper is organized as follows. In § 2 we look at the problem of estimating RTO, beginning with discussions of the basic algorithm and our evaluation methodology. We analyze the impact of varying a number of estimator parameters, finding that the one with the greatest effect is the lower bound placed on RTO, followed by the clock granularity, while other parameters have little effect. We then present evidence that argues for the intrinsic difficulty of finding optimal parameters, and finish with a discussion of the cost of retransmitting unnecessarily and ways to detect when it has occurred. In § 3 we look at the problem of estimating the bandwidth available to a connection as it starts up. We discuss our evaluation methodology, which partitions estimates into different regions reflecting their expected impact, ranging from no impact, to preventing loss, attaining steady state, optimally utilizing the path, or reducing performance. We then assess a number of estimators, finding that sender-side estimation such as previously proposed in the literature is fraught with difficulty, while receiver-side estimation can work considerably better. § 4 summarizes the analysis and possible future work.

2 Estimating RTO

For an acknowledgment-based reliable transport protocol, such as TCP, a fundamental question is how long, in the absence of receiving an acknowledgment (ACK), should a sender wait until retransmitting? This problem is similar to that of estimating the largest possible round-trip time (RTT) along an end-to-end network path. However, it differs from RTT estimation in three ways. First, the goal is not to accurately estimate the truly maximal possible RTT, but rather a good compromise that balances avoiding unnecessary retransmission timeouts due to not waiting long enough for an ACK to arrive, versus being slow to detect that a retransmission is necessary. Second, the sender really needs to estimate the *feedback* time, which is the round-trip time from the sender to the receiver *plus* the amount of

*This paper appears in ACM SIGCOMM '99.

time required for the receiver to generate an ACK for newly received data. For example, a receiver employing the delayed acknowledgment algorithm [Bra89] may wait up to 500 msec before transmitting an ACK. Thus, estimating a good value for the retransmission timer not only involves estimating a property of the network path, but also a property of the remote connection peer. Third, if loss is due to congestion, it may behoove the sender to wait *longer* than the maximum feedback time, in order to give congestion more time to drain from the network—if the sender retransmits as soon as the feedback time elapses, the retransmission may also be lost, whereas sending it later would be successful.

It has long been recognized that the setting of the retransmission timer cannot be fixed but needs to reflect the network path in use, and generally requires dynamic adaptation because of how greatly RTTs can vary over the course of a connection [Nag84, DDK$^+$90]. The early TCP specification included a notion of dynamically estimating RTO, based on maintaining an exponentially-weighted moving average (EWMA) of the current RTT and a static variation term [Pos81]. This estimator was studied by Mills in [Mil83], which characterizes measured Internet RTTs as resembling a Poisson distribution overall, but with occasional spikes of much higher RTTs, and suggests changing the estimator so that it more rapidly adapts to increasing RTTs and more slowly to decreasing RTTs. (To our knowledge, this modified estimator has not been further evaluated in the literature.) [Mil83] also noted that the balance between responding rapidly in the face of true loss versus avoiding unnecessary retransmissions appears to be a fundamental tradeoff, with no obvious optimal solution.

Zhang [Zha86] discusses a number of deficiencies with the standard TCP RTO estimator: ambiguities in measuring RTTs associated with retransmitted packets; the conservative RTO policy of retransmitting only one lost packet per round-trip; the difficulty of choosing an initial estimate; and the failure to track rapidly increasing RTTs during times of congestion. Karn and Partridge [KP87] addressed the first of these, eliminating ambiguities in measuring RTTs. The introduction of "selective acknowledgments" (SACKs) [MMFR96] addressed the second issue of retransmitting lost packets too slowly. Jacobson [Jac88] further refined TCP RTO estimation by introducing an EWMA estimate of RTT variation, too, and then defining:

$$\mathrm{RTO} = \mathit{SRTT} + k \cdot \mathit{RTTVAR} \tag{1}$$

where *SRTT* is a smoothed estimate of RTT (as before) and *RTTVAR* is a smoothed estimate of the variation of RTT. In [Jac88], $k = 2$, but this was emended in a revised version of the paper to $k = 4$ [JK92].

While this estimator is in widespread use today, to our knowledge the only systematic evaluation of it against measured TCP connections is our previous study [Pax97b], which found that, other than for over-aggressive misimplementations, the estimator appears sufficiently conservative in the sense that it only rarely results in an unnecessary timeout.

The widely-used BSD RTO implementation [WS95] has several possible limitations: (1) the adaptive RTT and RTT variation estimators are updated with new measurements only once per round-trip, so they adapt fairly slowly to changes in network conditions; (2) the measurements are made using a clock with a 500 msec granularity, which necessarily yields coarse estimates (though [Jac88] introduces some subtle tricks for squeezing more precision out of these estimates); and (3) the resulting RTO estimate has a large minimum value of 1 second, which may make it inherently conservative.

With the advent of higher precision clocks and the TCP "timestamp" option [JBB92], all three of these limitations might be removed. It remains an open question, however, how to best reengineer the RTO estimator given these new capabilities: we know the current estimator is sufficiently conservative, but is it *too* conservative? If so, then how might we improve it, given a relaxation of the above limitations? These are the questions we attempt to answer.

2.1 The Basic RTO Estimation Algorithm

In Jacobson's algorithm, two state variables *SRTT* and *RTTVAR* estimate the current RTT and a notion of its variation. These values are used in Eqn 1 with $k = 4$ to attain the RTO. Both variables are updated every time an RTT measurement $\mathrm{RTT_{meas}}$ is taken. Since only one segment and the corresponding ACK is timed at any given time, updates occur only once per RTT (also referred to as once "per flight"). *SRTT* is updated using an EWMA with a gain of α_1:

$$\mathit{SRTT} \leftarrow (1 - \alpha_1)\mathit{SRTT} + \alpha_1 RTT_{\mathrm{meas}} \tag{2}$$

and Jacobson [Jac88] recommends $\alpha_1 = \frac{1}{8}$, which leads to efficient implementation using fixed-point arithmetic and bit shifting. Similarly, *RTTVAR* is updated based on the deviation $|\mathit{SRTT} - \mathrm{RTT_{meas}}|$ using $\alpha_2 = \frac{1}{4}$.

Any time a packet retransmitted due to the RTO expiring is itself lost, the TCP sender doubles the current value of the RTO. Doing so both diminishes the sending rate in the presence of sustained congestion, and ameliorates the possible adverse effects of underestimating the RTO and retransmitting needlessly and repeatedly.

SRTT and *RTTVAR* are initialized by the first $\mathrm{RTT_{meas}}$ measurement using $\mathit{SRTT} \leftarrow \mathrm{RTT_{meas}}$ and $\mathit{RTTVAR} \leftarrow \frac{1}{2}\mathrm{RTT_{meas}}$. Prior to the first measurement, $\mathrm{RTO} = 3$ sec.

Two important additional considerations are that all measurement is done using a clock *granularity* of G seconds, i.e., the clock advances in increments of G,[1] and the RTO is *bounded* by $\mathrm{RTO_{min}}$ and $\mathrm{RTO_{max}}$. In the common BSD implementation of TCP, $G = 0.5$ sec, $\mathrm{RTO_{min}} = 2G = 1$ sec, and $\mathrm{RTO_{max}} = 64$ sec. As will be shown, the value of $\mathrm{RTO_{min}}$ is quite significant. Also, since the granularity is coarse, the code for updating *RTTVAR* sets a minimum bound on *RTTVAR* of G, rather than the value of 0 sec that can often naturally arise.

Three oft-proposed variations for implementing the RTO estimator are to time every segment's RTT, rather than only one per flight; use smaller values of G; and lower $\mathrm{RTO_{min}}$ in order to spend less time waiting for timeouts. RFC 1323 [JBB92] explicitly supports the first two of these, and our original motivation behind this part of our study was to evaluate whether these changes are worth pursuing.

2.2 Assessing Different RTO Estimators

There are two fundamental properties of an RTO estimator that we investigate: (1) how long does it wait before retransmitting a lost packet? and (2) how often does it expire mistakenly and unnecessarily trigger a retransmit? A very conservative RTO estimator might simply hardwire $\mathrm{RTO} = 60$ sec and never make a mistake, satisfying the second property, but doing extremely poorly with regards to the first, leading to unacceptable delays; while a very aggressive estimator could hardwire $\mathrm{RTO} = 1$ msec and reverse this relationship, flooding the network with unnecessary retransmissions.

Our basic approach to assess these two properties is to use trace-driven simulation to evaluate different estimators, using the following methodology, which mirrors the RTO estimator implementation in [WS95]:

1. For each data packet sent, if the RTO timer is not currently active, it is started. The timer is also restarted when the data packet is the beginning of a retransmission sequence.

2. For each data packet retransmitted in the TCP trace due to a timeout, we assess whether the timeout was *unavoidable*, meaning that either the segment being retransmitted was lost, or all

[1] The BSD timer implementation also uses a "heartbeat" timer that expires every G seconds with a phase independent of when the timer is actually set. We included this behavior in our simulations.

ACKs sent after the segment's arrival at the receiver (up until the arrival of the retransmission) were lost. This check is necessary because some of the TCPs in the $\mathcal{N}_2$ dataset used aggressive RTO estimators that often fired prematurely in the face of high RTTs [Pax97a], so these retransmissions are not treated as normal timeout events.

3. If the timeout was unavoidable, then the retransmission is classified as a "first" timeout if this is the first time the segment is retransmitted, or as a "repeated" timeout otherwise. The estimator is charged the current RTO setting as reflecting the amount of time that passed prior to retransmitting (consideration (1) above), with separate bookkeeping for "first" and "repeated" timeouts (for reasons explained below). The RTO timer is also backed off by doubling it.

4. If the timeout was avoidable, then it reflects a problem with the actual TCP in the trace, and this deficiency is not charged against the estimator we are evaluating.

5. For each arrival of an ACK for new data in the trace, the ACK arrival time is compared with the RTO, as computed by the given estimator. If the ACK arrived after the RTO would have fired we consider the expiration a "bad" timeout, reflecting that the feedback time of the network path at that moment exceeded the RTO.

   If the ACK covers all outstanding data the RTO timer is turned off.

   If the ACK also yielded an RTT measurement (because it acknowledged the segment currently being timed, or because every segment is being timed), *SRTT* and *RTTVAR* are updated based on the measurement and the RTO is recomputed.

   Finally, the RTO timer is restarted.

6. The sending or receiving of TCP SYN or FIN packets is not assessed, as these packets have their own retransmission timers, and if interpreted as simple ACK packets can lead to erroneous measurements of RTT.

Note this approach contains a subtle but significant difficulty. Suppose that in the trace packet P is lost and 3 seconds later the TCP's real-life RTO expires and P is retransmitted. We treat this as a "first timeout," and charge the estimator with the RTO, R, it computed for P. Suppose $R = 100$ msec. From examining the trace it is impossible to determine whether retransmitting P after waiting only 100 msec would have been successful. It could be that waiting any amount of time less than 3 seconds was in fact too short an interval for the congestion leading to P's original loss to have drained from the network. Conversely, suppose P is lost after being retransmitted 3 seconds later. It could be that the first loss and the second are in fact uncorrelated, in which case retransmitting after waiting only R seconds would yield a successful transmission.

The only way to assess this effect would be to conduct live experiments, rather than trace-driven simulation, which we leave for future work. Therefore, we assess *not* whether a given retransmission was *effective*, meaning that the retransmitted packet safely arrived at the receiver, but only whether the *decision* to retransmit was *correct*, meaning that the packet was indeed lost, or all feedback from the receiver was lost. Related to this consideration, only the effectiveness of an RTO estimator at predicting timely "first" timeouts is assessed. For repeated timeouts it is difficult to gauge exactly how many of the potential repeated retransmissions would have been necessary.

Given these considerations, for a given estimator and a trace i let T_i be the total time required by the estimator to wait for unavoidable first timeouts. Let g_i be the number of "good" (necessary) first timeouts, and b_i the total number of "bad" timeouts, including multiple bad timeouts due to backing off the timer (since we can soundly assess that all of these repeated retransmissions were indeed unnecessary). If $b_i + g_i > 0$, that is, trace i included some sort of timeout, then define $\rho_i = \frac{b_i}{b_i+g_i}$, the normalized number of bad timeouts in the trace; otherwise define $\rho_i = 0$. Note that ρ_i may not be a particularly good metric when considering transfers of varying length. However, this study focuses only on transfers of 100 KB.

Minimum RTO	W	$\widetilde{W}$	B
1,000 msec	144,564	8.4	0.63%
750 msec	121,566	6.5	0.76%
500 msec	102,264	4.8	1.02%
250 msec	92,866	3.5	2.27%
0 msec	92,077	3.1	4.71%
RTO = 2,000 msec	229,564	15.6	2.66%
RTO = 1,000 msec	136,514	8.2	6.14%
RTO = 500 msec	85,878	4.5	12.17%

Table 1: Effect of varying RTO_{min}, $G = 1$ msec

For the jth good timeout, let RTO_i^j be the RTO setting of the expiring timer, and RTT_i^j be the most recently observed RTT (even if it was not an RTT that would have been measured for purposes of updating the *SRTT* and *RTTVAR* state variables). Let $\xi_i^j = \text{RTO}_i^j/\text{RTT}_i^j$, so ξ_i^j reflects the cost of the timeout in units of RTTs. We can then define an average, normalized timeout cost of $\psi_i = E_j[\xi_i^j]$, or 0 if trace i does not include any good timeouts.

For a collection of traces, we then define $W = \sum_i T_i$ as the total time spent waiting for (good) first timeouts; $\widetilde{W} = E_{i:g_i>0}[\psi_i]$ as the mean normalized timeout cost per connection that experienced at least one good timeout; and $B = E_i[\rho_i]$ as the mean proportion of timeouts that are *bad*, per connection, including connections that did not include any timeouts (because we want to reward estimators that, for a particular trace, don't generate any bad timeouts).

W can be dominated by a few traces with a large number of timeout retransmissions, for which the total time waiting for first timeouts can become very high, so it is biased towards highlighting how bad things can get. $\widetilde{W}$ is impartial to the number of timeouts in a trace, and so better reflects the overall performance of an estimator. B likewise better reflects how well an estimator avoids bad timeouts overall. For some estimators, there may be a few particular traces on which they retransmit unnecessarily a large number of times, as noted below.

Finally, of the 18,490 pairs of traces in $\mathcal{N}_2$, 4,057 pairs were eliminated from our analysis due to packet filter errors in recording the traces, the inability to pair packets across the two traces (this can occur due to packet filter drops or IP ID fields changed in flight by header compression glitches [Pax97c]), or *tcpanaly*'s inability to determine which retransmissions were due to timeouts. This leaves us with 14,433 traces to analyze, with a total of 67,073 timeout retransmissions. Of those, 53,110 are "first" timeouts, and 34% of the traces have no timeout retransmissions.

2.3 Varying the Minimum RTO

It turns out that the setting of RTO_{min}, the lower bound on RTO, can have a major effect on how well the RTO estimator performs, so we begin by analyzing this effect. We first note that the usual setting for RTO_{min} is two clock "ticks" (i.e., $\text{RTO}_{\text{min}} = 2G$), because, given a "heartbeat" timer, a single tick translates into a time anywhere between 0 and G sec. Accordingly, for the usual coarse-grained estimator of $G = 0.5$ sec, RTO_{min} is 1 sec, which we will see is conservative (since a real BSD implementation would use a timeout between 0.5 sec and 1 sec). But for $G = 1$ msec, the two-tick minimum is only 2 msec, and so setting RTO_{min} to larger values can have a major effect.

Granularity	W	$\widetilde{W}$	B
500 msec	272,885	19.2	0.36%
[WS95] (500 msec)	245,668	15.4	0.23%
250 msec	167,360	10.2	0.67%
100 msec	142,940	8.4	0.95%
50 msec	143,156	8.4	0.84%
20 msec	143,832	8.4	0.70%
10 msec	144,175	8.4	0.67%
1 msec	144,564	8.4	0.63%

Table 2: Effect of varying granularity G, $\text{RTO}_{\text{min}} = 1$ sec

Table 1 shows W, $\widetilde{W}$ and B for different values of RTO_{min}, for $G = 1$ msec. We see that W runs from 144,564 seconds for a minimum of 1 sec to about 64% as much when using no minimum. The column for $\widetilde{W}$ shows that the 1 sec minimum means that a typical RTO costs a bit more than 8 RTTs, but much of this expense disappears as we decrease the minimum. B, on the other hand, shows that for a 1 sec minimum, on average only about 1 in 150 timeouts is bad, while for no minimum, nearly 1 in 20 is (these bad timeouts are not clustered among a particular small subset of the traces). Clearly, adjusting the minimum RTO provides a "knob" for directly trading off timely response with premature timeouts, with no obvious "sweet spot" yielding an optimal balance between the two.

As noted above, "delayed" acknowledgments in TCP can result in elevating RTTs by up to 500 msec, and in a number of common implementations, frequently elevate RTTs by up to 200 msec. Accordingly, it is not clear that a minimum RTO of two ticks for $G = 1$ msec is sound. However, for the bulk of our subsequent analysis, we consider estimators with no minimum bound, both to highlight the contribution to estimator efficiency of factors other than the quite-dominant minimum RTO, and to keep in mind that transport protocols different from TCP might not introduce such a minimum.

For comparison, we include three static timers that use a constant setting for RTO (except they double the RTO on repeated timeouts). The table highlights the heavy cost of not using an adaptive timer. The constant estimators generate about 10 times as many bad timeouts as the adaptive estimators with similar relative performance figures ($\widetilde{W}$). The values of B don't tell the whole story for the static timers, however, because their bad timeouts are clustered among relatively few traces. For example, RTO = 2,000 msec results in a bad timeout in 538 traces, while for $\text{RTO}_{\text{min}} = 250$ msec, which has a similar value of B, spreads its bad timeouts over more than twice as many traces.

2.4 Varying Measurement Granularity

With the above caution regarding the considerable importance of RTO_{min} in mind, we now look at the effect of varying G. In Table 2, G ranges from 500 msec down to 1 msec. In order to compare the different granularities on an even footing, we hold $\text{RTO}_{\text{min}} = 1$ sec constant, rather than having the relative differences between the granularities overwhelmed by using $\text{RTO}_{\text{min}} = 2G$. We include one additional row, "[WS95]," which is the estimator as implemented in [WS95]. This implementation includes fixed-point arithmetic and bit-shifting in order to estimate *SRTT* at an effective granularity of 62.5 msec and *RTTVAR* at a granularity of 125 msec, though RTO itself is computed with a granularity of 500 msec.

We first note that for $G \leq 100$ msec, the performance for good timeouts, both absolute (W) and relative ($\widetilde{W}$) is essentially identical, regardless of how fine the granularity becomes. But we steadily gain in avoiding bad timeouts (minimizing B) as the granularity becomes finer. The reason for the gain is that the more coarse granularities will often take no action in the face of a minor change in RTT, while the finer granularity estimator will adapt to reflect the change, and this gives it a slight edge.

Above $G = 100$ msec, however, we start trading off reduced performance for avoiding bad timeouts. We can cut the average rate of bad timeouts by nearly a factor of two by using $G = 500$ msec, but at a cost of more than a factor of two in performance. We also note that the [WS95] estimator clearly performs better than $G = 500$ msec, with both $\widetilde{W}$ and B lower. It gains by performing better on some very-large-RTT traces, because it is able to better reflect relatively small RTT changes due to its finer effective granularities for *SRTT* and *RTTVAR*.

Parameters	W	$\widetilde{W}$	B
[WS95]	245,668	15.4	0.23%
[WS95]-*every*	241,100	14.7	0.25%
take-first ($\alpha_1, \alpha_2 = 0, \text{RTO}_{\text{min}} = 1$ *s*)	158,199	8.5	0.74%
take-first ($\alpha_1, \alpha_2 = 0$)	131,180	4.4	2.93%
very-slow ($\alpha_1 = \frac{1}{80}, \alpha_2 = \frac{1}{40}$)	113,903	3.9	3.97%
slow-every ($\alpha_1 = \frac{1}{32}, \alpha_2 = \frac{1}{16}$)	102,544	3.4	4.28%
slow ($\alpha_1 = \frac{1}{16}, \alpha_2 = \frac{1}{8}$)	96,740	3.4	3.84%
std ($\alpha_1 = \frac{1}{8}, \alpha_2 = \frac{1}{4}$)	92,077	3.1	4.71%
std-every ($\alpha_1 = \frac{1}{8}, \alpha_2 = \frac{1}{4}$)	94,081	3.1	5.09%
fast ($\alpha_1 = \frac{1}{2}, \alpha_2 = \frac{1}{4}$)	90,212	3.0	7.27%
take-last ($\alpha_1, \alpha_2 = 1$)	93,490	3.3	19.57%
take-last-every ($\alpha_1, \alpha_2 = 1$)	97,098	3.5	20.20%
take-last ($\alpha_1, \alpha_2 = 1, \text{RTO}_{\text{min}} = 1$ *s*)	145,571	8.5	1.30%

Table 3: Effect of varying EWMA parameters α_1, α_2

2.5 Varying the EWMA Parameters

Table 3 shows the estimator's performance when varying α_1 (per Eqn 2) and α_2, holding $G = 1$ msec and $\text{RTO}_{\text{min}} = 0$ msec fixed, except where noted. The first two rows are the [WS95] implementation, which uses $G = 500$ msec, with the second row reflecting a variant that derives an RTT measurement from every ACK arriving at the sender. We see that the more frequent *SRTT* and *RTTVAR* updates have little effect on the estimator's performance, only making it slightly more aggressive.

The remaining estimators all use $G = 1$ msec. The *take-first* extreme of $\alpha_1 = \alpha_2 = 0$ simply uses the first RTT measurement to initialize both *SRTT* $\leftarrow$ RTT and *RTTVAR* $\leftarrow \frac{1}{2}$RTT, yielding RTO $\leftarrow$ 3RTT. It never changes *SRTT*, *RTTVAR*, or RTO again (other than to back off RTO in the face of repeated retransmissions, and undo the backing off when the retransmission epoch ends). The first variant of it reflects using $\text{RTO}_{\text{min}} = 1$ sec, the second, $\text{RTO}_{\text{min}} = 0$ sec. At the other extreme, we have *take-last*, which always sets *SRTT* $\leftarrow$ RTT and *RTTVAR* $\leftarrow |\mathit{SRTT}_{\text{prev}} - \text{RTT}|$. The *take-last-every* variant is the same except every packet is timed rather than just one packet per round trip, and the final variant raises the minimum RTT to 1 sec.

In between these extremes we run the gamut from *very-slow*, which uses one-tenth the usual parameters (which are given for the *std* estimator), to *fast*, which uses twice the parameters, with some time-every-packet variants.

From the table we see that the settings of the EWMA parameters make little difference in how well the estimator performs. Indeed, if our goal is to minimize the rate of bad timeouts and still remain aggressive, we might pick the exceedingly simple *take-first* estimator, which only barely adapts to the network path conditions;[2] or we might pick *slow*, which on average incurs 25% less normalized delay per timeout, and occupies a sweet spot that locally minimizes

[2]Even though *take-first* and *take-last* show overall decent performance compared to the other RTO estimators, these RTO estimators could perform extremely poorly over network paths that exhibit large, sudden changes in RTT.

RTTVAR factor	W	$\widetilde{W}$	B
$k = 16$	168,002	7.0	0.59%
$k = 12$	144,053	5.7	0.81%
$k = 8$	118,858	4.4	1.52%
$k = 6$	105,681	3.8	2.43%
adapt	94,220	3.2	4.44%
$k = 4$	92,077	3.1	4.71%
$k = 3$	85,264	2.8	7.68%
$k = 2$	78,565	2.5	13.64%
$RTO_{min} = 750$ msec, $k = 6$	128,266	6.7	0.50%
$RTO_{min} = 750$ msec	121,566	6.5	0.76%
take-first$_{250\text{msec}}$, $k = 6$	163,799	6.4	0.70%
$RTO_{min} = 500$ msec, $k = 6$	112,514	5.1	0.69%
$RTO_{min} = 500$ msec	102,264	4.8	1.02%
$RTO_{min} = 250$ msec, $k = 6$	106,139	4.0	1.29%
$RTO_{min} = 250$ msec	92,866	3.5	2.27%

Table 4: Effect of varying *RTTVAR* factor, k

B. As we found for [WS95], timing every packet makes little difference over timing only one packet per RTT, even though by timing every packet we run many more measurements through the EWMAs per unit time. This in turn causes the EWMAs to adapt *SRTT* and *RTTVAR* more quickly to current network conditions, and to more rapidly lose memory of conditions further in the past, similar in effect to using larger values for α_1 and α_2.

We note that as the timer more quickly adapts, B steadily increases, with *take-last-every* generating on average one bad timeout in every five, indicating correlations in RTT variations that span multiple round-trips. We can greatly diminish this problem by raising RTT_{min} to 1 sec, but only by losing a great deal of the estimator's timely response, and we are better off instead using the corresponding *take-first* variant.

We also evaluated varying the EWMA parameters for $RTO_{min} =$ 500 msec. We find that $\widetilde{W}$ increases by roughly 50%, with the variation among the estimators further diminishing, while B falls by a factor of 4–8, further illustrating the dominant effect of the RTO minimum.

Finally, a number of the paths in $\mathcal{N}_2$ contain slow, well-buffered links, which lead to steady, large increases in the RTT (up to many seconds). We might expect *take-first* to do quite poorly for these connections, since the first measured RTT has little to do with subsequent RTTs, but in fact *take-first* does quite well. The key is the last part of step 5 in § 2.2 above: the RTO timer is restarted with each arriving ACK for new data. Consequently, when data is flowing, the RTO has an implicit extra RTT term [Lud99], and for *take-first* this suffices to avoid bad timeouts even for RTTs that grow by two orders of magnitude. Indeed, *take-first* does *better* for such connections than estimators that track the changing RTT! It does so because more adaptive estimators wind up waiting much longer after the last arriving ACK before RTO expires, while *take-first* retransmits with appropriate briskness in this case. But this advantage is particular to the highly-regularized feedback of such connections. It does, however, suggest the notion of a "feedback timeout," discussed briefly in § 4.

2.6 Varying the *RTTVAR* Factor

The last RTO estimation parameter we consider is k, the multiplier of *RTTVAR* when computing RTO, per Eqn 1. For the standard implementation, $k = 4$. Table 4 shows the effects of varying k from 2–16, for $G = 1$ msec and $RTO_{min} = 0$ sec. The *adapt* estimator starts with $k = 4$ but doubles it every time it incurs a bad timeout.

k clearly provides a knob for trading off waiting time for unnecessary timeouts, with no obvious sweet spot. This balance changes a bit, however, when we increase RTO_{min}, as shown in the second half of the table. For example, we find that $RTO_{min} = 250$ msec, $k = 4$ performs strictly better than the no-minimum $k = 6$ variant, and $RTO_{min} = 250$ msec, $k = 6$ performs better than the $k = 8$ variant. Even the extremely simple *take-first* estimator, if using $k = 6$ and $RTO_{min} = 250$ msec, performs a bit better than the regular $RTO_{min} = 750$ msec estimator.

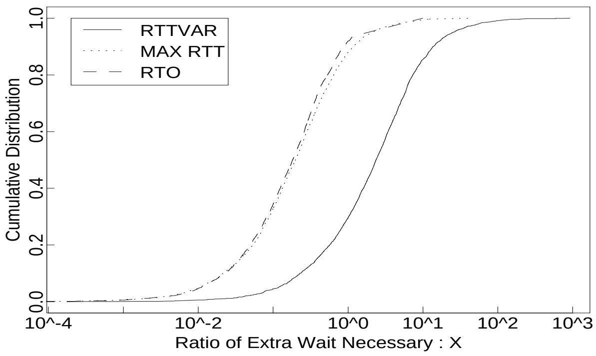

Figure 1: Extra waiting time necessary to avoid bad RTO

2.7 Can We Estimate RTO Better?

Having evaluated the effects of different estimator parameters and, for the most part, only found tradeoffs and little in the way of compelling "sweet spots," we now turn to the question of whether there are indeed opportunities to devise still better estimators. A key consideration for answering this question is: when we underestimate, by how much is it? If, for example, underestimates tend to be off by less than RTT, then that would suggest a modification to Eqn 1 in which *SRTT* has a factor of 2 applied to it.

Let A denote the amount of additional waiting time needed to avoid a bad RTO. Figure 1 plots the cumulative distribution of the ratio of A to *RTTVAR* (solid), the maximum RTT seen so far (dotted), and RTO (dashed), for the usual $G = 1$ msec estimator. The ratio of A to *RTTVAR* ranges across several orders of magnitude, indicating that finding a particular value of k in Eqn 1 that efficiently takes care of most of the remaining bad timeouts is unlikely.

Also shown is that A is generally less than the current RTO and also the maximum RTT seen so far; this suggests adding one of those values to RTO to make it sufficiently conservative to avoid bad timeouts. However, doing so has much the same effect as other estimator variants that wait longer based on other factors (e.g., the value of k). For example, changing the standard $k = 4$ estimator shown in Table 4 to use twice the computed RTO (i.e., add in an additional RTO term) lowers B from 4.71% to 0.57%, but increases $\widetilde{W}$ from 3.1 to 5.7—a bit better than just using $k = 12$, but not compellingly better.

For $RTO_{min} = 0.5$ sec, the plot is very similar, with slightly more separation between the RTO and MAX RTT lines. Thus, Figure 1 suggests a fundamental tradeoff between aggressiveness and suffering bad timeouts.

A related question is: if a packet is unnecessarily retransmitted, does it reflect a momentary increase in RTT, or a sustained increase? We find that about 62% of the bad timeouts were followed by RTTs less than the current RTO, so the bad timeout reflected a transient RTT increase. Another 24% were followed by exactly one more elevated RTT, though a bit more than 2% were followed by 10 or more elevated RTTs. Thus, most of the time a significant RTT increase is quite transient—but there is non-negligible tail-weight for sustained RTT increases.

2.8 Impact of Bad Timeouts

We finish our study of RTO estimators with brief comments concerning the impact of bad timeouts.

Any time a TCP times out unnecessarily, it suffers not only a loss of useful throughput, but, often more seriously, unnecessarily cuts *ssthresh* to half the current, sustainable window, and begins a new slow start. In addition, because the TCP is now sending retransmitted packets, unless it uses the TCP timestamp option, it cannot safely measure RTTs for those packets (per Karn's algorithm [KP87]), and thus it will take a long time before the TCP can adapt its RTT estimate in order to improve its broken RTO estimate. (See [Pax97a] for an illustration of this effect.)

Bad timeouts can therefore have a major negative impact on a TCP connection's performance. However, they do *not* have much of an adverse impact on the *network*'s performance, because by definition they occur at a time when the network is not congested to the point of dropping the connection's packets. This in turn leads to the observation that if we could undo the deleterious effects upon the TCP connection of cutting *ssthresh* and entering slow start, then a more aggressive RTO estimator would be more attractive, as TCP would be able to sustain bad timeouts without unduly impairing performance or endangering network stability.

When TCP uses the timestamp option, it can unambiguously determine that it retransmitted unnecessarily by observing a later ACK that echoes a timestamp from a packet sent prior to the retransmission. (A TCP could in principle also do so using the SACK option.) Such a TCP could remember the value of *ssthresh* and *cwnd* prior to the last retransmission timeout, and restore them if it discovers the timeout was unnecessary.

Even without timestamps or SACK, the following heuristic might be considered: whenever a TCP retransmits due to RTO, it measures ΔT, the time from the retransmission until the next ACK arrives. If ΔT is less than the minimum RTT measured so far, then arguably the ACK was already in transit when the retransmission occurred, and the timeout was bad. If the ACK only comes later than the minimum RTT, then likely the timeout was necessary.

We can assess the performance of this heuristic fairly simply. For our usual $G = 1$ msec estimator, a total of 8,799 good and bad timeouts were followed by an ACK arriving with ΔT less than the minimum measured RTT. Of these, fully 75% correspond to *good* timeouts, indicating that, surprisingly, the heuristic generally fails. The failure indicates that sometimes the smallest RTT seen so far occurs right after a timeout, which we find is in fact the case, perhaps because the lull of the timeout interval gives the network path a chance to drain its load and empty its queues.

However, if the threshold is instead $f = \frac{3}{4}$ of the minimum RTT, then only 20% of the corresponding timeouts are *good* (these comprise only 1% of all the *good* timeouts). For $f = \frac{1}{2}$, the proportion falls to only 2.5%. With these reduced thresholds the chance of detecting a bad timeout falls from 74% to 68% or 59%, respectively.

We evaluated the modified heuristic and found it works well: for $f = \frac{1}{2}$, B drops from 4.71% to 2.39%, a reduction of nearly a factor of two, and enough to qualify the estimator as a "sweet spot."

3 Estimating Bandwidth

We now turn to the second estimation problem, determining the amount of bandwidth available to a new connection. Clearly, if a transport protocol sender knows the available bandwidth, it would like to immediately begin sending data at that rate. But in the absence of knowing the bandwidth, it must form an estimate. For TCP, this estimate is currently made by exponentially increasing the sending rate until experiencing packet loss. The loss is taken as an implicit signal that the rate had grown too large, so the rate is effectively halved and the connection continues in a more conservative fashion.

In the context of TCP, the goal in this section is to determine the efficacy of different algorithms a TCP connection might use during its start-up to determine the appropriate sending rate without pushing on the network as hard as does the current mechanism. In a more general context, the goal is to explore the degree to which the timing structure of flights of packets can be exploited in order to estimate how fast a connection can safely transmit.

We assume familiarity with the standard TCP congestion control algorithms [Jac88, Ste97, APS99]: the state variable *cwnd* bounds the amount of unacknowledged data the sender can currently inject into the network, and the state variable *ssthresh* marks the *cwnd* size at which a connection transitions from the exponential increase of "slow start" to the linear increase of "congestion avoidance." Ideally, *ssthresh* gives an accurate estimate of the bandwidth available to the connection, and congestion avoidance is used to probe for additional bandwidth that might appear in a conservative, linear fashion.

A new connection begins slow start by setting *cwnd* to 1 segment,[3] and then increasing *cwnd* by 1 segment for each ACK received. If the receiver acknowledges every k segments, and if none of the ACKs are lost, then *cwnd* will increase by about a factor of $\gamma = 1 + \frac{1}{k}$ every RTT. Most TCP receivers currently use a "delayed acknowledgment" policy for generating ACKs [Bra89] in which $k = 2$ and hence $\gamma = \frac{3}{2}$, which is the value we assume subsequently.

Note that if during one round-trip a connection has N segments in flight, then during slow start it is possible, during the next RTT, to overflow a drop-tail queue along the path such that $(\gamma - 1)N = N/k$ segments are lost in a group, if the queue was completely full carrying the N segments during the first round-trip. Such loss will in general significantly impede performance, because when multiple segments are dropped from a window of data, most current TCP implementations will require at least one retransmission timeout to resend all dropped segments [FF96, Hoe96]. However, during congestion avoidance, which can be thought of as a connection's steady-state, TCP increases *cwnd* by at most one segment per RTT, which ensures that *cwnd* will overflow a queue by at most one segment. TCP's fast retransmit and fast recovery algorithms [Jac90, Ste97, APS99] provide an efficient method for recovering from a single dropped segment without relying on the retransmission timer [FF96].

Hoe [Hoe96] describes a method for estimating *ssthresh* by multiplying the measured RTT with an estimate of the bottleneck bandwidth (based on the packet-pair algorithm outlined in [Kes91]) at the beginning of a transfer. [Hoe96] showed that correctly estimating *ssthresh* would eliminate the large loss event that often ends slow start (as discussed above). Given that Hoe's results were based on simulation, an important follow-on question is to explore the degree to which these results are applicable to actual, measured TCP connections.

There are several other mechanisms which mitigate the problems caused by TCP's slow start phase, and therefore lessen the need to estimate *ssthresh*. First, routers implementing Random Early Detection (RED) [FJ93, BCC$^+$98] begin randomly dropping segments at a low rate as their average queue size increases. These drops implicitly signal the connection to reduce its sending rate before the queue overflows. Currently, RED is not widely deployed. RED also does not guarantee avoiding multiple losses within a window of data, especially in the presence of heavy congestion. However, RED also has the highly appealing property of not requiring the deployment of any changes to current TCP implementations.

Alternate loss recovery techniques that do not rely on TCP's re-

[3] Strictly speaking, *cwnd* is usually managed in terms of bytes and not segments (full-sized data packets), but conventionally it is discussed in terms of segments for convenience. The distinction is rarely important. Also, [APS99] allows an initial slow start to begin with *cwnd* set to 2 segments, and an experimental extension to the TCP standard allows an initial slow start to begin with *cwnd* set to 3 or possibly 4 segments [AFP98]. We comment briefly on the implications of this change below.

transmission timer have been developed to diminish the impact of multiple losses in a flight of data. SACK-based TCPs [MM96, MMFR96, FF96] provide the sender with more complete information about which segments have been dropped by the network than non-SACK TCP implementations provide. This allows algorithms to quickly recover from multiple dropped segments (generally within one RTT following loss detection). One shortcoming of SACK-based approaches, however, is that they require implementation changes at both the sender and the receiver. Another class of algorithms, referred to as "NewReno" [Hoe96, FF96, FH99], does not require SACKs, but can be used to effectively recover from multiple losses without requiring a timeout (though not as quickly as when using SACK-based algorithms). In addition, NewReno only requires implementation changes at the sender. The estimation algorithms studied in this paper all require changes to the sender's TCP implementation. So, we assume that the sender TCP implementation will have some form of the NewReno loss recovery mechanism.

3.1 Methodology

In this section we discuss a number of algorithms for estimating *ssthresh* and our methodology for assessing their effectiveness. We begin by noting a distinction between *available bandwidth* and *bottleneck bandwidth*. In [Pax97b] we define the first as the maximum rate at which a TCP connection exercising correct congestion control can transmit along a given network path, and the second as the upper bound on how fast *any* connection can transmit along the path due to the data rate of the slowest forwarding element along the path.

Our ideal goal is to estimate *available bandwidth* in terms of the correct setting of *ssthresh* such that we fully utilize the bandwidth available to a given connection, but do not exceed it (more precisely: only exceed it using the linear increase of congestion avoidance). Much of our analysis, though, is in terms of bottleneck bandwidth, as this is both an upper bound on a good *ssthresh* estimate, and a quantity that is more easily identifiable from the timing structure of a flight of packets, since for any two data packets sent back-to-back along an uncongested path, their interarrival time at the receiver directly reflects the bottleneck bandwidth along the path.[4]

Note that in most TCP implementations *ssthresh* is initialized to an essentially unbounded value, while here we concentrate on lowering this value in an attempt to improve performance by avoiding loss or excessive queueing. Thus, all of the algorithms considered in this section are *conservative*, yet they also (ideally) do not impair a TCP's performance relative to TCPs not implementing the algorithm. However, if an estimator yields too small a value of *ssthresh*, then the TCP will indeed perform poorly compared to other, unmodified TCPs.

As noted above, one bottleneck bandwidth estimator is "packet pair" [Kes91]. In [Pax97b] we showed that a packet pair algorithm implemented using strictly sender-side measurements performs poorly at estimating the bottleneck bandwidth using real traffic. We then developed a more robust method, Packet Bunch Mode (PBM), which is based on looking for modalities in the timing structure of groups of back-to-back packets [Pax97b, Pax97c]. PBM's effectiveness was assessed by running it over the NPD datasets (including the $\mathcal{N}_2$ dataset referred to earlier), arguing that the algorithm was accurate because on those datasets it often produced estimates that correspond with known link rates such as 64 kbps, T1, E1, or Ethernet.

PBM analyzes an entire connection trace before generating any bottleneck bandwidth estimates. It was developed for assessing network path properties and is not practical for current TCP implementations to perform on the fly, as it requires information from both the sender and receiver (and is also quite complicated). However, for our purposes what we need is an accurate assessment of a given network path's bottleneck bandwidth, which we *assume* that PBM provides. Thus, we use PBM to calibrate the efficacy of the other *ssthresh* estimators we evaluate.

Of the 18,490 traces available in $\mathcal{N}_2$, we removed 7,447 (40%) from our analysis for the following reasons:

- Traces marred by packet filter errors [Pax97a] or major clock problems [Pax98]: 15%. Since these problems most likely do not reflect network conditions along the path between the two hosts in the trace, removing these traces arguably does not introduce any bias in our subsequent analysis.

- Traces in which the first retransmission in the trace was "avoidable," meaning had the TCP sender merely waited longer, an ACK for the retransmitted segment would have arrived: 20%. Such retransmissions are usually due to TCPs with an initial RTO that is too short [Pax97a, PAD$^+$99]. We eliminate these traces because the retransmission results in *ssthresh* being set to a value that has little to do with actual network conditions, so we are unable to soundly assess how well a larger *ssthresh* would have worked. Removing these traces introduces a bias against connections with particularly high RTTs, as these are the connections most likely to engender avoidable retransmissions.

- Traces for which the PBM algorithm failed to produce a single, unambiguous estimate: 4%. We need to remove these traces because our analysis uses the PBM estimate to calibrate the different estimation algorithms we assess, as noted above. Removing these traces introduces a bias against network conditions that make PBM itself fail to produce a single estimate: multi-channel paths, changes in bottleneck bandwidth over the course of a connection, or severe timing noise.

After removing the above traces, we are left with 11,043 connections for further analysis. We use trace-driven simulation to assess how well each of the bandwidth estimation algorithms perform. We base our evaluation on classifying the algorithm's estimate for each trace into one of several *regions*, representing different levels of impact on performance.

For each trace, we define three variables, B, L and E. B is the bottleneck bandwidth estimate made using the PBM algorithm. L is the *loss point*, meaning the transmission rate in effect when the first lost packet was sent (so, if the first lost segment was sent with *cwnd* corresponding to W bytes, then $L = W/RTT$ bytes/second). If the connection does not experience loss, L' is the bandwidth attained based on the largest *cwnd* observed during the connection.[5] When $L > B$ or $L' > B$, the network path is essentially free of competing traffic, and the loss is presumed caused by the connection itself overflowing a queue in the network path. Conversely, if L or L' is less than B, the path is presumed congested. Finally, E is the bandwidth estimate made by the *ssthresh* estimation algorithm being assessed.

In addition, define $\text{seg}(x) = (x \cdot \text{RTT})/\textit{segment size}$ representing the size of the congestion window, in segments, needed to achieve a bandwidth of x bytes/second, for a given TCP segment size and RTT. (Note that as defined, $\text{seg}(x)$ is continuous and not discrete.)

3.1.1 Connections With Loss

Given the above definitions, and a connection which contains loss, we assess an estimator's performance by determining which of the following six regions it falls into. Note that we analyze the regions in the order given, so an estimate will not be considered for any regions subsequent to the first one it matches.

[4] Providing the path isn't "multi-channel" or subject to routing changes [Pax97b].

[5] Strictly speaking, it's the largest flight observed during the connection, which might be smaller than *cwnd* due to the connection running out of data to send, or exhausting the (32-64KB) receiver window.

No Estimate Made. The estimator failed to produce an *ssthresh* estimate before the first segment loss occurred in the trace.

No Impact. The estimate satisfies $E \geq \gamma L$. This means that E is a sufficiently large overestimate that the connection will behave no differently using that estimate than it would if no estimate were made.

Some Loss Prevention. When $L \leq E < \gamma L$ holds, the given *ssthresh* estimate prevents some, but not all, loss of data packets. While the estimate is greater than the loss point, it reduces the size of the last slow start flight by $N_s = \mathrm{seg}(\gamma L - E)$ segments. Therefore, up to N_s segment drops may be prevented.

Steady-State. When $\frac{L}{2} \leq E < L$ holds, we classify the *ssthresh* estimate as "steady-state." During congestion avoidance, which defines TCP's steady-state behavior [Jac88, MSMO97], *cwnd* decreases by half upon loss detection and then increases linearly until another loss occurs. So, given the loss point of L, *cwnd* can be expected to oscillate between $\frac{L}{2}$ and L after the connection's second loss event.[6] By making an estimate between $\frac{L}{2}$ and L, the estimator has found the range about which the connection will naturally oscillate, assuming the loss point is stationary.

Optimal. When the analysis reaches this point, we know that $E < \frac{L}{2}$ since none of the above conditions hold. If $\mathrm{seg}(E) \geq \mathrm{seg}(B) - 1$ also holds, then the *ssthresh* estimate reduces the queueing requirement, as follows. Since E is very close to or larger than the bottleneck bandwidth, yet less than $\frac{L}{2}$, we know that the loss point is greater than the bottleneck bandwidth, yet the *ssthresh* estimate is no less than the bottleneck bandwidth or one segment less than the bottleneck bandwidth. (We consider one segment less than the bottleneck bandwidth to be within the range because both slow start and congestion avoidance will take a single RTT to increase *cwnd* to correspond with B—and we prefer to reach that point via congestion avoidance rather than slow start, so we don't overshoot it.)

Thus, assuming the connection lasts long enough, the queue will still be filled to L. However, we will fill the queue more slowly and smoothly than with slow start. Furthermore, when we exceed the queue during congestion avoidance, it is only by one segment, whereas during slow start we will exceed the capacity of the queue by as much as γ times the capacity.[7] When a connection falls into this region, the queue length is initially reduced by $N_q = (L - E) \cdot \mathrm{RTT}$ bytes. Since this region reduces queueing, prevents loss, yet fully utilizes the network path, we deem it "optimal."

Reduce Performance. Finally, if none of the above conditions hold then $E < \frac{L}{2}$ and $E < B$ (these bounds are not tight). We therefore set *ssthresh* too low and force *cwnd* growth to continue linearly, rather than exponentially. When an estimator underestimates $\min(\frac{L}{2}, B)$ by more than half in 50+% of the connections in which performance would be reduced, we consider this to be an especially bad estimate. In this case, the reported percentage of connections experiencing reduced performance is marked with a "*".

[6]The size of *cwnd* when detecting the first loss event is roughly γL. Therefore, the first halving of *cwnd* causes it to be approximately $\frac{\gamma}{2}L$. Each subsequent loss event should only overflow the queue slightly and therefore *cwnd* will be reduced to $\frac{L}{2}$.

[7]Some implementations of congestion avoidance add a constant of $\frac{1}{8}$ times the segment size to *cwnd* for every ACK received during congestion avoidance. This non-standard behavior has been shown to lead to sometimes overflowing the queue by more than a single segment every time *cwnd* approaches L [PAD+99].

Algorithm	**No Est.**	**No Imp.**	**Prv. Loss**	**Stdy. State**	**Opt.**	**Tot.**	**Red. Perf.**
PBM$'$	23%	46%	9%	10%	11%	31%	0%
TSSF	42%	1%	1%	3%	0%	4%	52%*
$\mathrm{CSA}_{n=3}^{\nu=0.1}$	62%	20%	6%	9%	2%	17%	2%
$\mathrm{CSA}_{n=2}^{\nu=0.05}$	53%	37%	5%	4%	0%	9%	1%*
$\mathrm{CSA}_{n=2}^{\nu=0.1}$	45%	32%	8%	10%	2%	19%	4%*
$\mathrm{CSA}_{n=2}^{\nu=0.2}$	38%	24%	9%	13%	3%	25%	13%
TCSA	62%	14%	6%	11%	1%	19%	5%
TCSA$'$	70%	10%	6%	9%	2%	17%	2%
$\mathrm{Recv}_{\mathrm{min}}$	11%	32%	6%	13%	4%	23%	34%*
$\mathrm{Recv}_{\mathrm{avg}}$	11%	52%	10%	14%	9%	34%	3%
$\mathrm{Recv}_{\mathrm{med}}$	11%	48%	10%	14%	10%	34%	7%*
$\mathrm{Recv}_{\mathrm{max}}$	11%	65%	7%	8%	8%	23%	0%*

Table 5: Connections with Loss (8,257 traces)

3.1.2 Connections Without Loss

The following regions use L' to assess the impact of *ssthresh* estimation on connections in the dataset that do not experience loss. Each trace is placed into one of the following four regions. (Again, note that we analyze the regions in the order given, so an estimate will not be considered for any regions subsequent to the first one it matches.)

No Estimate Made. The estimator failed to produce an *ssthresh* estimate.

Unknown Effect. When $E \geq L'$ holds, the estimate does not limit TCP's ability to open *cwnd*, as it is above the maximum *cwnd* used by the connection. Since we do not have a good measure of the limit of the network path, nothing more can be assessed about the performance of the estimator.

Optimal. When $\mathrm{seg}(E) \geq \mathrm{seg}(B) - 1$ holds, the estimate is greater than the bottleneck bandwidth and therefore does not limit performance. However, we also know that $E < L'$ due to the above region. Therefore, the estimate reduces the initial queueing requirement similar to the "optimal" region in § 3.1.1.

Reduce Performance. At this point, $E < \min(L', B - \mathrm{seg}^{-1}(1))$ holds, indicating that the estimate failed to provide exponential window growth to L', which is a known safe sending rate. Furthermore, our failure to reach L' is not excused by providing exponential *cwnd* growth long enough to fill the pipe (B bytes/second). We again mark with a "*" those connections for which the reduction is often particularly large.

3.2 Benchmark Algorithm

As noted above, we use PBM as our benchmark in terms of accurately estimating the bottleneck bandwidth. For *ssthresh* estimation, we use a revised version of the algorithm, PBM$'$, to provide some sort of *upper bound* on how well we might expect any algorithm to perform. (It is not a strong upper bound, since it may be that other algorithms estimate the *available* bandwidth considerably better than does PBM$'$, but it is the best we currently have available.) The difference between PBM$'$ and PBM is that PBM$'$ analyzes the trace only up to the point of the first loss, while PBM analyzes the trace in its entirety. Thus, PBM$'$ represents applying a detailed, heavyweight, but accurate algorithm on as much of the trace as we are allowed to inspect before perforce having to make an *ssthresh* decision.

As shown in Tables 5 and 6, the PBM$'$ estimate yields *ssthresh* values that rarely hurt performance, regardless of whether the connection experiences loss. Each column lists the percentage of traces which, for the given estimator, fell into each of the regions discussed in § 3.1.1. The **Tot.** column gives the percentage of traces for which the estimator improved matters by attaining either the **prevent loss**,

Algorithm	No Est.	Unk. Imp.	Opt.	Red. Perf.
PBM′	0%	56%	44%	0%
TSSF	13%	2%	2%	82%*
$\text{CSA}^{\nu=0.1}_{n=3}$	24%	42%	13%	22%
$\text{CSA}^{\nu=0.05}_{n=2}$	19%	59%	11%	10%
$\text{CSA}^{\nu=0.1}_{n=2}$	14%	48%	11%	27%
$\text{CSA}^{\nu=0.2}_{n=2}$	13%	34%	11%	43%*
TCSA	24%	25%	8%	44%
TCSA′	27%	33%	11%	28%
Recv_{min}	1%	15%	2%	83%*
Recv_{avg}	1%	46%	23%	31%*
Recv_{med}	1%	45%	28%	26%
Recv_{max}	1%	71%	27%	1%

Table 6: Connections without Loss (2,786 traces)

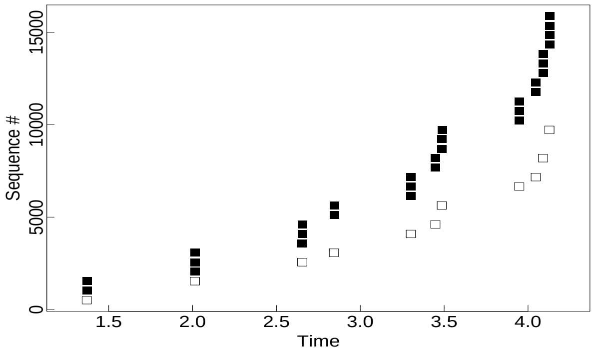

Figure 2: Delayed ACK leading to timing "lull"

steady-state, or **optimal** regions. This column can be directly compared with the last column (**reduce performance**) to assess how a given estimator trades off improvement in some cases with damage in others.

We see that PBM' provides some benefit (steady state, prevention of loss, or optimal) to 31% of the connections that experience loss, and, when no loss occurs, the estimate falls in the optimal region for 44% of the connections. The remaining estimates are overestimates, in the case when the connection experiences loss, or have an unknown impact (but, do not harm performance) in the connections that do not have dropped segments. This indicates that much of the time the *available* bandwidth is less than the raw bottleneck bandwidth that PBM measures, which accords with the finding given in [Pax97b].

3.3 Sender-Side Estimation Algorithms

The following is a description of the sender-side bandwidth estimation algorithms, and the corresponding *ssthresh* estimates, investigated in this paper. TCP's congestion control algorithms work on the principle of "self-clocking" [Jac88]. That is, data segments are injected into the network and arrive at the receiver at the rate of the bottleneck link, and consequently ACKs are generated by the receiver with spacing that reflects the rate of the bottleneck link. Therefore, sender-side estimation techniques measure the rate of the returning ACKs to make a bandwidth estimate. These algorithms assume that the spacing injected into the data stream by the network will arrive intact at the receiver and will be preserved in the returning ACK flow, which may not be true due to fluctuations on the return channel altering the ACK spacing (e.g., ACK compression [ZSC91, Mog92]). These algorithms have the advantage of being able to directly adjust the sending rate. In the case of TCP, they can directly set the *ssthresh* variable as soon as the estimate is made. However, a disadvantage of these algorithms is their reliance on the ACK stream accurately reflecting the arrival spacing of the data stream.

3.3.1 Tracking Slow Start Flights

The first technique we investigate is a TCP-specific algorithm that tracks each slow start "flight." The ACKs for a given flight are used to obtain an estimate of *ssthresh*. While this algorithm is TCP specific, the general idea of measuring the spacing introduced by the network in all segments transmitted in one RTT should be applicable to other transport protocols. We parameterize the algorithm by n, the number of ACKs used to estimate the bottleneck bandwidth. For our analysis, we used $n = 3$. Let F be the current flight size, in segments. The Tracking Slow Start Flights (TSSF) algorithm is then:

- Initialize the current segment S to the first data segment sent, and F to the initial value of *cwnd* in segments.
- For the current S and F, check whether S's ACK and the $n-1$ subsequent arriving ACKs are all within the sequence range of the flight. If so, then we use this flight to make an estimate. Otherwise, we continue to the next flight. However, if any of the ACKs arrive reordered or are duplicates, the algorithm terminates. When looking forward for the $n-1$ subsequent ACKs, the algorithm ignores any ACKs for a single segment, as they were presumably delayed.
- To find the next flight, advance S by F segments. If N_a is the number of ACKs for new data that arrive between the old value of S and its new value, then the size of the next flight is $F + N_a$ (the slow start increase).
- When we find a suitable flight, we estimate the bandwidth as the amount of data ACKed between the first and the nth ACK, divided by the time between the arrivals of these ACKs.

As the second rows of Tables 5 and 6 show, the performance of the TSSF algorithm is quite poor. The overwhelming problem with this estimator is underestimating the bandwidth, which would cause a reduction in performance.

The underestimation is caused in part by TCP's delayed acknowledgment algorithm. RFC 1122 [Bra89] encourages TCP receivers to refrain from ACKing every incoming segment, and to instead acknowledge every second incoming segment, though it also requires that the receiver wait no longer than 500 msec for a second segment to arrive before sending an ACK. Many TCP implementations use a 200 msec "heartbeat" timer for generating delayed ACKs. When the timer goes off, which could be any time between 0 and 200 msec after the last segment arrived, if the receiver is still waiting for a second segment it will generate an ACK for the single segment that has arrived. Using this mechanism can fail to preserve in the returning ACK stream the spacing imposed on the data stream by the bottleneck link. The time the receiver spends waiting on a second segment to arrive increases the time between ACKs, which is assumed by the sender to indicate the segments were further spaced out by the network, which leads to an underestimate of the bandwidth.

Furthermore, once a delayed ACK timer effect is injected into the ACK stream, the flight is effectively partitioned into two mini-flights for the duration of slow start, since data segments are sent in response to incoming ACKs. The sequence-time plot in Figure 2 illustrates this effect. In the plot, which is recorded from the sender's perspective, outgoing data segments are indicated with solid squares drawn at the upper sequence number of the segment, while incoming ACKs are drawn with hollow squares at the sequence number they acknowledge.

The first flight shown, which consists of two segments, elicits a single ACK that arrives at time $T = 2.0$. But the flight of three segments that this ACK triggers elicits two ACKs, one for two segments arriving at $T = 2.6$, but another for just one segment at time

$T = 2.8$. The latter reflects a delayed ACK. The next flight of five packets then has a lull of about 200 msec in the middle of it. This lull is duly reflected in the ACKs for that flight, plus an additional delayed ACK occurs from the first sub-flight of three segments (times $T = 3.3$ through $T = 3.5$). The resulting next flight of 8 segments is further fractured, reflecting not only the lull introduced by the new delayed ACK, but also that from the original delayed ACK, and the general pattern repeats again with the next flight of 12 segments. None of the ACK flights give a good bandwidth estimate, nor is there much hope that a later flight might.

This mundane-but-very-real effect significantly complicates any TCP sender-side bandwidth estimation. While for other transport protocols the effect might be avoidable (if ACKs are not delayed), the more general observation is that sender-side estimation will significantly benefit from information regarding just when the packets it sent arrived at the receiver, rather than trying to infer this timing by assuming that the receiver sends its feedback promptly enough to generate an "echo" of the arrivals.

3.3.2 Closely-Spaced ACKs

The *ssthresh* estimation algorithms in [Hoe96] and [AD98] are based on the notion of measuring the time between "closely spaced ACKs" (CSAs). By measuring CSAs, these algorithms attempt to consider ACKs that are sent in response to closely spaced data segments, whose interarrival timing at the receiver then presumably reflects the rate at which they passed through the bottleneck link. However, neither paper defines exactly what constitutes a set of closely-space ACKs.

We explore a range of CSA definitions by varying two parameters. The first, ν, is the fraction of the RTT within which the consecutive ACKs of the closely-spaced group must arrive in order to be considered "close." We examined ν values of 0.0125, 0.025, 0.05, 0.1 and 0.2. The second parameter, n, is the number of ACKs that must be close in order to make an estimate. We examined $n = 2, 3, 4, 5$. The bandwidth estimate is made the first time n ACKs arrive (save the first) within $\nu \cdot \mathrm{RTT}$ sec of their predecessors. This algorithm has the advantage of being easy to implement. Also, it does not depend on any of the details of TCP's congestion control algorithms, which makes the algorithm easy to use for other transport protocols. A disadvantage of the algorithm is that it is potentially highly dependent on the above two constants.

Our goal was to find a "sweet spot" in the parameter space that works well over a diverse set of network paths. Rows 3–6 of Tables 5 and 6 show the effectiveness of several of the points in the parameter space. Values of ν and n outside this range performed appreciably worse than those shown.

We chose $n = 3, \nu = 0.1$ as the sweet spot in the parameter space. However, the choice was not clear cut, as both $n = 2, \nu = 0.05$ and $n = 2, \nu = 0.1$ provide similar effectiveness. All of the parameter values shown, including the chosen sweet spot, reduce performance for a large number of connections that do not experience loss and yield no performance benefit in over 60% of the connections that did experience loss (due to an inability to form an estimate or overestimating).

3.3.3 Tracking Closely-Spaced ACKs

The *ssthresh* estimation algorithm in [AD98] assumes that the arrivals of closely-spaced ACKs are used to form tentative *ssthresh* estimates, with a final estimate being picked when these settle down into a form of consistency. We used a CSA estimator with $n = 3$ and $\nu = 0.1$ (the sweet spot above) to assess the effectiveness of their proposed approach. For their scheme, we take multiple samples and use the minimum observed sample to set *ssthresh*. We continue estimating until the point of loss, or we observe a sample within 10% of the minimum sample observed so far (in which case we are presumed to have converged). We show the effectiveness of using the "tracking closely-spaced ACKs" (TCSA) algorithm in Tables 5 and 6. As with the CSA method described above, the TCSA algorithm does not have a performance impact on the connection in over 75% of the connections with loss. Furthermore, the number of connections for which the performance would be reduced is increased by roughly a factor of 2 for both connections that experienced loss and those that did not when comparing TCSA with CSA.

Since TCSA shows an increase in the number of connections whose performance would be reduced, it clearly often estimates too low, so we devised a variant, TCSA$'$, that does not depend on the minimum observation (which is likely to be an underestimate). We compare each CSA estimate, E_i, with estimate E_{i-1} (for $i > 1$). If these two samples are within 10% of each other, then we use the average of the two bandwidth estimates to set *ssthresh*. Tables 5 and 6 show that TCSA$'$ is comparable to TCSA in most ways. The exception is that the number of underestimates that would reduce performance is decreased when using TCSA$'$, so it would be the preferred algorithm.

3.4 Receiver-Side Estimation Algorithm

The problems with sender-side estimation outlined above led to the evaluation of the following receiver-side algorithm for estimating the bandwidth. Estimating the bandwidth at the receiver removes the problems that can be introduced in the ACK spacing by delay fluctuations along the return path or due to the delayed ACK timer.

A disadvantage of this algorithm is that the receiver cannot properly control the sender's transmission rate.[8] However, the receiver could inform the sender of the bandwidth estimate using a TCP option (or some other mechanism, for a transport protocol other than TCP). For our purposes, we assume that this problem is solved, and note that alternate uses for the estimate by the receiver is an area for future work.

The receiver-side algorithm outlined below is TCP-specific. Its key requirement is that the receiver can predict which new segments will be transmitted back-to-back in response to the ACKs it sends, and thus it can know to use the arrivals of those segments as good candidates for reflecting the bottleneck bandwidth. Any transport protocol whose receiver can make such a prediction can use a related estimation technique. In particular, by using a timestamp inserted by the sender, the receiver could determine which segments were sent closely-spaced without knowledge of the specific algorithm used by the sender. This is an area for near-term future work.

For convenience, we describe the algorithm assuming that sequence numbers are in terms of segments rather than bytes. Let A_i denote the segment acknowledged by the ith ACK sent by the receiver. Let D_i denote the highest sequence number the sender can transmit after receiving the ith ACK. If we number the ACK of the initial SYN packet as 0, then $A_0 = 0$. Assuming that the initial congestion window after the arrival of ACK 0 is one segment, we have $D_0 = 1$. To accommodate initial congestion windows larger than one segment [AFP98], we increase D_0 accordingly.

The basic insight to how the algorithm works is that the receiver knows exactly which new segments the arrival of one of its ACKs at the sender will allow. These segments are presumably sent back to back, so the receiver can then form a bandwidth estimate based on their timing when they arrive at the receiver.

[8] The TCP receiver could attempt to do so by adjusting the advertised window to limit the sender to the estimated *ssthresh* value, even also increasing it linearly to reflect congestion avoidance. But when doing so, it diminishes the efficacy of the "fast recovery" algorithm [Ste97, APS99], because it will need to increase the artificially limited window, and, according to the algorithm, an ACK that does so will be ignored from the perspective of sending new data in response to receiving it.

Any time the receiver sends the $j+1$st ACK, it knows that upon receipt of the ACK by the sender, the flow control window will slide $A_{j+1} - A_j$ segments, and the congestion window will increase by 1 segment, so the total number of packets that the sender can now transmit will be $A_{j+1} - A_j + 1$. Furthermore, their sequence numbers will be $D_j + 1$ through D_{j+1}, so it can precisely identify their particular future arrivals in order to form a sound measurement. Finally, we take the first K such measurements (or continue until a data segment was lost), and from them form our bandwidth estimate. For our assessment below, we used $K = 50$.

(We note that the algorithm may form poor estimates in the face of ACK loss, because it will then lose track of which data packets are sent back-to-back. We tested an oracular version of the algorithm that accounts for lost ACKs, to serve as an upper bound on the effectiveness of the algorithm. We found that the extra knowledge only slightly increases the effectiveness of the algorithm.)

This algorithm provides estimates for more connections than any of the other algorithms studied in this paper, because every ACK yields an estimate. Tables 5 and 6 show the receiver-based algorithm using four different methods for combining the K bandwidth estimates. The first "Recv" row of each table shows the effectiveness of using the minimum of the K measurements as the estimate. This yields an underestimate in a large number of the connections, decreasing performance (34% of the time when the connection experiences loss and 83% of the time when no loss is present). The next row shows that averaging the samples improves the effectiveness over using the minimum: the number of connections with reduced performance is drastically reduced when the connection experiences loss, and halved in the case when no loss occurs. However, the flip side is the number of cases when we overestimate the bandwidth increases when loss is present in the connection. Taking the median of the K samples provides similar benefits to using the average, except the number of connections experiencing reduced performance increases by a factor of 2 over averaging when loss occurs. Finally, using the maximum of the K estimates further increases the number of overestimates for connections experiencing loss. However, using the maximum also reduces the number of underestimates to nearly none, regardless of whether the connection experiences loss. Of the methods investigated here, using the maximum appears to provide the most effective *ssthresh* estimate. However, we note that alternate algorithms for combining the K estimates is an area for near-term future work.

Finally, we varied the number of bandwidth samples, K, used to obtain the average and maximum estimates reported above to determine how quickly the algorithms converge. We find that when averaging the estimates, the effectiveness increases slowly but steadily as we increase K to 50 samples. However, when taking the maximum sample as the estimate, little benefit is derived from observing more than the first 5–10 samples.

4 Conclusions and Future Work

Our assessment of different RTO estimators yielded several basic findings. The minimum value for the timer has a major impact on how well the timer performs, in terms of trading off timely response to genuine lost packets against minimizing incorrect retransmissions. For a minimum RTO of 1 sec, we also realize a considerable gain in performance when using a timer granularity of 100 msec or less, while still keeping bad timeouts below 1%. On the other hand, varying the EWMA constants has little effect on estimator performance. Also, an estimator that simply takes the first RTT measurement and computes a fixed RTO from it often does nearly as well as more adaptive estimators. Related to this finding, it makes little difference whether the estimator measures only one RTT per flight or measures an RTT for every packet. This last finding calls into question some of the assumptions in RFC 1323 [JBB92], which presumes that there is benefit in timing every packet. Given that such benefit is elusive, the other goals of [JBB92] currently accomplished using timestamp options should be revisited, to consider using a larger sequence number space instead. We finished our RTO assessment by noting that timestamps, SACKs, or even a simple timing heuristic can be used to reverse the effects of bad timeouts, making aggressive RTO algorithms more viable.

Our assessment of various bandwidth estimation schemes found that using a sender-side estimation algorithm is problematic, due to the failure of the ACK stream to preserve the spacing imposed on data segments by the network path, and we developed a receiver-side algorithm that performs considerably better. A lingering question is whether the complexity of estimating the bandwidth is worth the performance improvement, given that only about a quarter of the connections studied would benefit. However, in the context of other uses or other transports, estimating the bandwidth using the receiver-side algorithm may prove compelling.

Our study was based on data from 1995, and would benefit considerably from verification using new data and live experiments. For RTO estimation, a natural next step is to more fully explore whether combinations of the different algorithm parameters might yield a significantly better "sweet spot." Another avenue for future work is to consider a bimodal timer, with one mode based on estimating RTT for when we lack feedback from the network, and the other based on estimating the variation in the feedback interarrival process, so we can more quickly detect that the receiver feedback stream has stalled. For bandwidth estimation, an interesting next step would be to assess algorithms for using the estimates to ramp up new connections to the available bandwidth more quickly than TCP's slow start. Finally, both these estimation problems merit further study in scenarios where routers use RED queueing rather than drop-tail, as RED deployment should lead to smaller RTT variations and a source of implicit feedback for bandwidth estimation.

5 Acknowledgments

This paper significantly benefited from discussions with Sally Floyd and Reiner Ludwig. We would also like to thank the SIGCOMM reviewers, Sally Floyd, Paul Mallasch and Craig Partridge for helpful comments on the paper. Finally, the key insight that the receiver can determine which sender packets are sent back to back (§ 3.4) is due to Venkat Rangan.

References

[AD98] Mohit Aron and Peter Druschel. TCP: Improving Startup Dynamics by Adaptive Timers and Congestion Control. Technical Report TR98-318, Rice University Computer Science, 1998.

[AFP98] Mark Allman, Sally Floyd, and Craig Partridge. Increasing TCP's Initial Window, September 1998. RFC 2414.

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

[BCC+98] Robert Braden, David Clark, Jon Crowcroft, Bruce Davie, Steve Deering, Deborah Estrin, Sally Floyd, Van Jacobson, Greg Minshall, Craig Partridge, Larry Peterson, K. Ramakrishnan, S. Shenker, J. Wroclawski, and Lixia Zhang. Recommendations on Queue Management and Congestion Avoidance in the Internet, April 1998. RFC 2309.

[Bra89] Robert Braden. Requirements for Internet Hosts – Communication Layers, October 1989. RFC 1122.

[DDK+90] Willibald Doeringer, Doug Dykeman, Matthias Kaiserswerth, Bernd Werner Meister, Harry Rudin, and Robin Williamson. A Survey of Light-Weight Transport Protocols for High-Speed Networks. *IEEE Transactions on Communications*, 38(11):2025–2039, November 1990.

[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.

[FJ93] Sally Floyd and Van Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

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

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

[Jac90] Van Jacobson. Modified TCP Congestion Avoidance Algorithm, April 1990. Email to the end2end-interest mailing list. URL: ftp://ftp.ee.lbl.gov/email/vanj.90apr30.txt.

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

[JK92] Van Jacobson and Michael Karels. Congestion Avoidance and Control, 1992. ftp://ftp.ee.lbl.gov/papers/congavoid.ps.Z.

[Kes91] Srinivasan Keshav. A Control Theoretic Approach to Flow Control. In *ACM SIGCOMM*, pages 3–15, September 1991.

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

[Lud99] Reiner Ludwig. A Case for Flow-Adaptive Wireless Links. Technical report, Ericsson Research, February 1999.

[Mil83] David Mills. Internet Delay Experiments, December 1983. RFC 889.

[MM96] Matt Mathis and Jamshid Mahdavi. Forward Acknowledgment: Refining TCP Congestion Control. In *ACM SIGCOMM*, August 1996.

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

[Mog92] Jeffrey C. Mogul. Observing TCP Dynamics in Real Networks. In *ACM SIGCOMM*, pages 305–317, 1992.

[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.

[Nag84] John Nagle. Congestion Control in IP/TCP Internetworks, January 1984. RFC 896.

[PAD+99] Vern Paxson, Mark Allman, Scott Dawson, William Fenner, Jim Griner, Ian Heavens, Kevin Lahey, Jeff Semke, and Bernie Volz. Known TCP Implementation Problems, March 1999. RFC 2525.

[Pax97a] Vern Paxson. Automated Packet Trace Analysis of TCP Implementations. In *ACM SIGCOMM*, September 1997.

[Pax97b] Vern Paxson. End-to-End Internet Packet Dynamics. In *ACM SIGCOMM*, September 1997.

[Pax97c] Vern Paxson. *Measurements and Analysis of End-to-End Internet Dynamics*. Ph.D. thesis, University of California Berkeley, 1997.

[Pax98] Vern Paxson. On Calibrating Measurements of Packet Transit Times. In *ACM SIGMETRICS*, June 1998.

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

[Ste97] W. Richard Stevens. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms, January 1997. RFC 2001.

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

[WS95] Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated Volume II: The Implementation*. Addison-Wesley, 1995.

[Zha86] Lixia Zhang. Why TCP Timers Don't Work Well. In *ACM SIGCOMM*, pages 397–405, August 1986.

[ZSC91] Lixia Zhang, Scott Shenker, and David Clark. Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two- Way Traffic. In *ACM SIGCOMM*, September 1991.