A Performance Study of Loss Detection/Recovery in Real-world TCP Implementations

Sushant Rewaskar Jasleen Kaur F. Donelson Smith
University of North Carolina at Chapel Hill
{rewaskar, jasleen, smithfd}@cs.unc.edu

Abstract— **TCP is the dominant transport protocol used in the Internet and its performance fundamentally governs the performance of Internet applications. It is well-known that packet losses can adversely affect the connection duration of TCP connections—however, what is not fully understood is how well does the TCP design deal with losses. In this paper, we systematically evaluate the impact of design parameters associated with TCP's loss detection/recovery mechanisms on the performance of real-world TCP connections. For this, we rely on an analysis tool that partially emulates the sender-side TCP implementations of 5 prominent OSes for passively analyzing the traces of TCP connections. Our study conducts passive analysis of more than 2.8 million real Internet TCP connections. We find that the recommended as well as widely-implemented settings of TCP parameters are not optimal for a significant fraction of Internet connections.**

I. Introduction

TCP is the dominant transport protocol used by Internet applications—including the world-wide web, peer-to-peer file sharing, and media streaming [1], [2], [3], [4]. The reason for TCP's popularity is that it offers several useful service semantics—perhaps the most useful of these is that of *reliable* data transfer. TCP implements reliability by detecting loss of data segments and retransmitting lost segments—unfortunately, loss detection/recovery mechanisms can be time-consuming. While it is generally known that segment losses can adversely impact the duration[1] of TCP connections, the extent to which they do so in the Internet has never been quantified. In this paper, we address this issue by evaluating the impact of TCP loss detection/recovery mechanisms on the performance of real-world TCP connections.[2]

TCP detects and recovers from losses using two basic types of mechanisms: retransmission-timeouts (RTOs) and fast-retransmit/recovery (FR/R). Two performance-related goals guide the design of these detection mechanisms. First, TCP should *accurately* identify segment losses. In particular, if TCP erroneously inferred that a segment was lost, it would unnecessarily invoke loss recovery and increase the connection duration. Second, TCP should *quickly* identify segment losses. A longer detection period adversely impacts connection duration as well. Unfortunately, these two goals conflict with each other—a "quick" inference of segment loss would also be erroneous when segments (or their ACKs) are not lost but merely delayed or reordered in the network. To achieve high loss-estimation accuracy, therefore, TCP would necessarily have to wait longer for ACKs that may merely be delayed.

As detailed in Section II, this fundamental trade-off between accuracy and timeliness is controlled by several design parameters associated with RTO and FR/R based loss detection—these include the dupACK threshold, the minimum RTO, the RTT-smoothing factor, the weight of RTT variability in the RTO-estimator, and the RTO estimator algorithm itself. While the proposed standards for TCP recommend values for each of these design parameters, TCP implementations in prominent operating systems (OSes) differ (sometimes significantly) in the values used. Our objective in this paper is to systematically vary these parameters and: (i) study the accuracy and timeliness of TCP loss detection/recovery in real-world TCP connections originating from different sender OS stacks, and (ii) study the impact of loss detection/recovery on overall durations of these connections.

We rely on passive analysis of traces of more than 2.8 million real-world TCP connections originating from 5 prominent sender-side OSes—including Linux, Windows XP, MacOS, Solaris, and FreeBSD. Our study thus incorporates a large, diverse, and realistic mix of applications, user behavior, network paths, and traffic conditions. Our key findings are:

- Most of the current implementations of RTO estimators are conservative in incorporating variability in TCP RTT. Reducing the influence of RTT variability can help significantly reduce the connection durations of TCP connections.
- Timer granularity and the minimum RTO no longer significantly limit TCP performance.
- The Linux RTO estimator converges fast and is the most efficient. If properly configured, this estimator has the greatest potential for improving connection durations.

The rest of this paper is organized as follows. We formulate the problem of configuring TCP loss detection/recovery in Section II. We present our data sources and methodology in Sections III and IV, respectively. A detailed analysis of the loss detection mechanism is presented in Section V. We summarize related work in Section VI and our conclusions in Section VII.

II. Problem Formulation

TCP senders assign sequence numbers to all data bytes transmitted and receivers send *cumulative* acknowledgments (ACKs) to confirm receiving data. Senders detect segment

[1]Throughout this paper, we define the *connection duration* of a TCP connection as the total duration of the connection (the time taken to complete all data transfers). This includes service times and user think times for applications that use persistent TCP connections.

[2]This research was supported in part by NSF CAREER award CNS-0347814 and NSF RI grant EIA-0303590. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

losses using two types of mechanisms that rely on the returned ACKs: *retransmission timeouts* (RTOs) and *fast retransmit/recovery* (FR/R):

RTOs: TCP sets a timer to expire after an RTO-amount of time when a segment is transmitted; if an ACK confirming that segment is not received before the timer expires, the sender concludes that the segment was lost. The value of RTO is determined using the relation: $RTO = m * srtt + k * rttvar$, where: $srtt$ is a moving average of the connection round-trip time (RTT), computed as: $srtt = (1-b) * srtt + b * rtt$; $rttvar$ is a moving average of the variability in RTT, computed as: $rttvar = (1-a) * rttvar + a * |srtt - rtt|$; a, b, m, k, are positive constants and $a, b \in [0, 1]$. The value of RTO increases with m and k, whereas a and b determine the weight given to history when RTT is quite variable. The actual value of the RTO timer is set to a predetermined value, $minRTO$, if the value computed above is smaller than $minRTO$. The above formulation is intended to compute an RTO that is greater than the current RTT, in order to avoid inferring loss of segments for which the ACK is merely delayed. Since RTT variability can be high, the value of RTO can be high, especially with the recommended settings for the five parameters, $a, b, m, k, minRTO$ [5]—RTO-based loss detection can, therefore, be time-consuming by delaying the response to a loss.

FR/R: FR/R is a faster means of detecting losses—if a segment is lost, delivered segments with higher sequence numbers trigger duplicate (cumulative) ACKs for its preceding segment. Hence, when a sender receives duplicate ACKs for a segment, it can conclude that the next higher segment was lost. However, reordering of segments in the network can also trigger the generation of duplicate ACKs. In order to avoid erroneously inferring loss in such cases, TCP senders usually wait for $D > 1$ duplicate ACKs [6] before concluding a segment was lost.

TCP receivers may also use *selective acknowledgments* (SACKs) for informing the sender of missing segments—this helps quickly detect subsequent losses when multiple segments are lost.

When loss is detected, segments are immediately retransmitted. Loss recovery is also accompanied by a reduction in TCP sending rate as a means of congestion control [6]—the reduction is quite significant for RTO-based loss detection. The invoking of loss detection/recovery can thus be quite costly in terms of connection duration. The exact cost depends on the choice of values for each of the 6 parameters associated with loss detection: $D, a, b, m, k, minRTO$. Two performance-related goals guide the optimal setting of these parameters:

- *High accuracy of loss detection:* First, a TCP sender should be accurate when it identifies segment losses. If TCP erroneously infers that a segment was lost, it would unnecessarily invoke loss recovery and increase the connection duration. Accuracy of FR/R-based loss detection can be improved by *selecting a larger value of* D, the duplicate ACK threshold—a larger D would help avoid spurious retransmissions when duplicate ACKs are generated by segment reordering in the network.
  Accuracy of RTO-based loss detection can be improved by *selecting a larger value of RTO*, which is determined by the parameters $a, b, m, k, minRTO$—a larger RTO would help avoid spurious retransmissions when segments or their ACKs are not lost, but merely delayed in the network.
- *Timeliness of loss detection:* Second, a TCP sender should quickly identify segment losses. The longer a sender takes to detect a loss, the greater is the potential delay before sending new data[3]—the longer, thus, is the connection duration. This is especially true for RTOs, which have long detection times—these can be reduced by *selecting a smaller value of RTO*.
  The loss detection times for FR/R can also be reduced slightly by *selecting a smaller value for* D—in this case, the sender has to wait for a smaller number of duplicate ACKs before it can infer a loss. However, much more significantly, a smaller value of D enables more losses to be discovered using FR/R, rather than RTOs—this is especially true for small connections that do not transmit enough segments to generate D duplicate ACKs. Given that RTO-based loss recovery is more costly than FR/R-based recovery, this further helps improve connection durations.

Parameter	Linux	Windows	FreeBSD	Solaris
Timer granularity	10ms	100ms	10ms	10ms
Initial RTO (s)	3	3	3	3.375
$minRTO$ (ms)	200	200	1200	400
a	0.25	0.25	0.25	0.25
b	0.125	0.125	0.125	0.125
m	1	1	1	1.25
k	4	4	4	4
D	3	2	3	3
RTO	srtt + vartt	srtt + 4*rttvar	srtt+ 4*rttvar	1.25*srtt + 4*rttvar

TABLE I
VALUES OF KEY PARAMETERS IN DIFFERENT TCP STACKS

It is apparent from the above that the goals of accuracy and timeliness of loss detection impose conflicting requirements on the values of the design parameters—accuracy requires the RTO and D to be large, while timeliness requires these to be small.

The proposed standards for TCP recommend values for each of these parameters [5], [6]—however, these recommendations are based on empirical evidence collected more than a decade ago. Furthermore, real-world TCP implementations differ, sometimes significantly, in their default settings of these parameters (see Table I).[4] This naturally raises two important questions regarding the efficacy of TCP loss de-

[3]Typically, such delays occur when the sender can not advance the send window because the lost segment has not been ACKed.

[4]Linux adjusts D dynamically, depending on the rate of occurrence of spurious FR/R-based retransmissions.

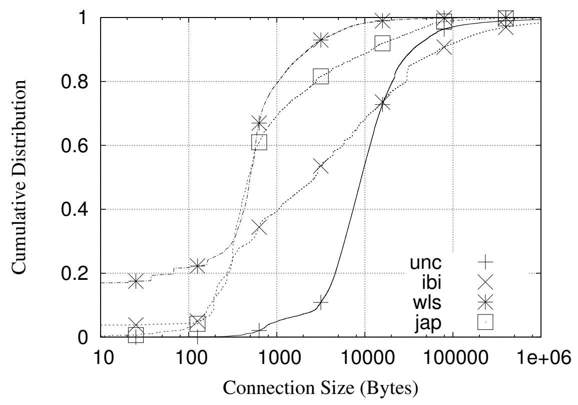


Fig. 1. Distribution of Bytes Transmitted in Each Connection

tection/recovery: *Are the parameter settings in different TCP implementations working well in reducing connection durations? Are the decade-old recommended settings in the TCP standards optimal for the current Internet?* These questions have been partially addressed in a couple of key studies [7], [8]—however, as described in detail in Section VI, most studies were conducted nearly a decade ago; consequently, these do not incorporate contemporary properties of real-world TCP implementations, Internet paths, and application behavior. More importantly, to the best of our knowledge, no previous study has modeled the impact of TCP parameters on the overall durations of TCP connections.

Our Contribution: In this paper, we consider each of the design parameters associated with TCP loss detection and: (i) quantify the impact of these on accuracy and timeliness of real-world TCP implementations, and (ii) model and quantify the impact of these on overall TCP connection durations in the current Internet. Our study incorporates the behavior of 5 prominent sender-side OSes and relies on passive analysis of more than 2.8 million real-world TCP connections. To the best of our knowledge, this is the largest and most comprehensive study of these design parameters.

In the following sections, we elaborate on our analysis methodology and results. We first describe the connection traces used for our study.

III. Data Sources

Table II describes the traces used in our analysis. These traces are bi-directional and are collected from links with transmission capacity ranging from 155 Mbps to OC-48. The *jap* trace [9] was collected off a trans-Pacific link connecting Japan to the US by the MAWI working group; the *unc* trace was collected at the campus-to-Internet link of the University of North Carolina at Chapel Hill (UNC); the *wls* trace captures wireless TCP connections from over 600 wireless access points within the UNC campus; and the *ibi* trace captures traffic served by a cluster of high-traffic web-servers (ibiblio.org). All traces except the one from the link to Japan were collected using Endace DAG cards [10]; the *jap* trace was collected using tcpdump [11]. These traces are fairly diverse in their geographic location, proximity to TCP senders, as well as types of users represented. The traces also vary significantly in the distribution of bytes transmitted per connection (see Fig 1).

IV. Methodology

Our objective is to study the impact of different design parameters on the performance of TCP loss detection/recovery mechanisms. Specifically, given a packet-header trace of a TCP connection, our passive analysis would need to: (i) infer the configuration of the 6 design parameters at the sender; (ii) identify all instances of loss detection/recovery attempts by the sender; (iii) determine the accuracy and timeliness of each loss detection; and (iv) vary the 6 design parameters and estimate the impact on the overall connection duration. We address each of these steps as described below.

A. Identifying Loss Detection Attempts and Parameters:

We rely on our recently-developed passive analysis tool [12], [13] that does two things relevant to our analysis. First, it identifies all segment retransmissions for each TCP connection in a packet trace, and classifies these based on the corresponding loss-detection mechanism—including RTO, FR/R, or SACK. Second, it identifies if the retransmission was necessary or spurious (depending on whether the original segment was actually lost or not). In order to do this analysis for traces of real-world connections, the tool partially emulates an enhanced version of the sender-side TCP state-machine for 5 prominent OSes—including, Linux 2.4.x, Windows XP/2000, Solaris 10, FreeBSD 5.2, and MacOS.[5] It is capable of identifying the sender-side OS for each connection—more relevantly, it identifies the setting of the 6 design parameters of interest to us.

We run this tool against all TCP connections traced and select those for which the tool can unambiguously identify the sender-side OS. We validated the OS-identification using *p0f*, a passive fingerprinting tool [14], and find the OS-identification accuracy to be more than 99.9%. Since our objective is to study loss detection/recovery, we consider traces of only those connections that experience at least one segment loss.[6] Table III summarizes the impact of applying these filters to our traces—a total of more than 2.8 million connections—as well as the distribution of connections across the 4 OSes. Our traces provided a large set of Windows and Linux connections, but relatively few Solaris or FreeBSD connections.

B. Studying Accuracy and Timeliness of Loss Detection:

Note that the above tool helps compute the accuracy of loss detection by identifying which retransmissions are spurious. In order to compute the timeliness of loss detection/recovery, we augment the tool as follows. For each loss detection event, we determine the time spent in loss detection—defined as the time difference between the original transmission and the retransmission of a segment—as well as the the time spent in recovery—defined as the time difference between the loss detection and receiving of a ACK for the highest segment transmitted before the detection (indicating that all losses have been recovered).

[5]MacOS and FreeBSD have the same TCP implementation—henceforth, we refer to connections from either of these OSes as “FreeBSD” connections.

[6]The tool can also analyze connections with no losses to measure the false-positives rate of TCP loss detection (when TCP erroneously infers losses) in these—due to space constraints we omit this analysis from this paper.

Trace	Duration	Avg TCP Load	# Connections	# Bytes	# Packets
japan-155Mbps-2004 (jap)	4h	1.93 Mbps	0.3 M	3.5 G	3.7 M
UNC-1Gbps-2005 (unc)	4h	74 Mbps	14.5 M	133.3 G	151.0 M
ibiblio-1Gbps-2005 (ibi)	4h	90.64 Mbps	0.9 M	163.2 G	158.9 M
wireless-2006 (wls)	178h	0.61 Mbps	9.7 M	48.5 G	68.9 M

TABLE II

GENERAL CHARACTERISTICS OF PACKET TRACES

	Lossy Connections			Lossy Explained Connections			Distribution across OSes			
Trace	% Conn	% Bytes	% Packets	% Conn	% Bytes	% Packets	% Windows	% Linux	% Solaris	% FreeBSD
jap	51.34	84.0	85.42	39.83	26.0	31.25	23.70	28.40	2.23	45.67
unc	15.83	77.28	67.73	14.91	39.41	39.18	84.36	2.22	10.69	2.73
ibi	10.67	83.15	81.60	8.13	26.10	33.44	0	100	0	0
wls	4.23	77.50	72.36	4.19	21.32	27.34	99.63	0	0	0.37

TABLE III

CONNECTIONS USED IN OUR ANALYSIS

C. Studying Impact of Design Parameters:

For this, we create several different instances of the emulated sender-side TCP state-machine—one instance for each of several different configurations of the 6 design parameters. We then re-process our traces using these modified state-machines and estimate for each sender configuration: (i) whether a segment loss would be detected by either FR/R or RTO, (ii) whether a spurious retransmission would be avoided, and (iii) whether a segment would be spuriously retransmitted due to a premature RTO or spurious dupACKs. We use this data, to compute the accuracy and timeliness of the corresponding sender configuration.

It is important to note that this passive evaluation methodology does not let us incorporate the interaction between the modified state machines, TCP congestion control, and subsequent network feedback (RTTs and losses)—only active experimentations with modified kernels could let us do that. Instead, we assume that RTTs and losses are independent of TCP loss detection/recovery behavior, and estimate how efficient each parameter configuration is.

D. Studying Impact on Connection Durations:

Finally, we quantify the impact of changes in the accuracy and timeliness of loss-detection, on the overall connection durations. Note that a change in parameter settings can result in one or more of the following events—for each such event, we augment the tool to re-process the trace of each connection and compute the reduction in connection duration:

- *A spurious FR/R-based loss detection is avoided.* In this case, the sender would not unnecessarily retransmit a segment. More significantly, the sender would not reduce its sending rate after "recovering" from the perceived loss. If the TCP flight size[7] was $2x$ before the segment was retransmitted, it would take the sender $x + 1$ RTTs to recover the same flight size after exiting FR/R. However, the sender also achieves some goodput in this duration. For each such avoided spurious FR/R-based loss recovery, we derive in [15] an estimate of the overall reduction in connection duration (in units of RTT) to be:

$$F(x) \quad = \quad x + 1 - \frac{3x^2 - x}{5x + 1} \qquad (1)$$

[7]Flight size is the number of segments transmitted but not ACKed.

- *A spurious RTO-based loss detection is avoided.* When a sender avoids a spurious RTO-based retransmission, it saves time spent on recovering the flight size. Assume that the flight size was $2x$ before the RTO expired; on RTO-expiry, the flight size is reduced to 1. It would take the sender $log(x) + x - 1$ RTTs to recover the flight size. However, the sender also achieves some goodput in this duration. For each such avoided spurious RTO-based loss recovery, we derived an estimate of the overall reduction in connection duration (in units of RTT) to be [15]:

$$R(x) \quad = \quad x + \log x - 1 - \frac{3x^2 + x - 4}{5x + \log x - 1} \qquad (2)$$

- *A loss is detected by FR/R, instead of an RTO.* Assume that the flight size was $2x$ when a loss occurred. If the loss is detected by FR/R instead of an RTO, there are two ways in which the connection duration reduces. The first is in the time it takes to detect the loss, and is given by the difference between the RTO and time at which the D^{th} dupACK is received (usually around 1 RTT). The second reduction is due to the fact that the TCP sending rate after exiting from FR/R (flight size is reduced to x) is usually higher than after an RTO expiry (flight size is reduced to 1). It would take the receiver $log(x) - 1$ RTTs to gain a flight size of x after an RTO. However, the sender would also achieve some goodput in this duration. For each loss that is detected by FR/R, instead of an RTO, we derived an estimate of the overall reduction in connection duration in the post-loss-recovery period (in units of RTT) to be [15]:

$$RF(x) \quad = \quad \log x - 1 - \frac{2x - 4}{2x + \log x - 1} \qquad (3)$$

- *An RTO-detected loss is detected after a different RTO.* For non-spurious RTO-based retransmissions, we compute the change (increase or decrease) in loss-detection time (difference between the original value of RTO and the new estimated value)—this is also equal to the change in connection duration for each such event.

Based on the above methodology, we next present our analysis results for existing TCP implementations, as well as for variants created by varying the 6 design parameters. For space reasons, we present only the most significant results—a detailed tabulation of all results can be found in [15].

OS	Total Retransmits	Non-spurious		Spurious	
		RTO	FR/R	RTO	FR/R
Windows	1074097	638969 (59.5%)	117040 (10.9%)	279358 (26%)	38730 (3.6%)
Linux	310418	175922 (56.7%)	115295 (37.1%)	10759 (3.5%)	8442 (2.7%)
Solaris	27105	19170 (70.7%)	5399 (19.9%)	1322 (4.9%)	1214 (4.5%)
FreeBSD	2312	1308 (56.6%)	166 (7.2%)	733 (31.7%)	105 (4.5%)

TABLE IV
CLASSIFICATION OF TCP RETRANSMISSIONS

V. ANALYSIS OF TCP LOSS DETECTION

A. Baseline Performance of Real-World TCP Implementations

Before assessing whether the performance of TCP loss detection can be improved by reconfiguring its parameters, we first evaluate if it is even worthwhile to do so by asking: *how much scope do we really have for improving TCP loss-detection performance in current TCP implementations?*

In order to answer this, we ask three specific questions for each connection in our data-set: (i) how often are segments retransmitted spuriously (the original transmission had reached the receiver)? (ii) how much time is spent in detecting and recovering from losses (both actual and perceived)? and (iii) by how much (upper bound) can the connection duration be improved by doing loss detection/recovery in a more accurate and timely manner? We also study whether the answer to any of the above depends on the sender-side OS of a connection. We address each of these questions below.

1) ***Accuracy:*** Table IV summarizes the total number of spurious retransmissions that were triggered by RTOs as well as FR/Rs, across connection traces originating from each of the 4 sender-side OSes. We observe that:

- A significant number of TCP retransmissions are spurious. In all of these cases, TCP inaccurately inferred that a segment was lost and retransmitted it. Most of the spurious retransmissions are triggered due to the expiry of an RTO (as against due to FR/R).
- The frequency of spurious RTOs varies significantly across OSes. This is somewhat to be expected, since the implementations and parameters of RTO estimators differ across current OSes—the Linux RTO estimator differs most significantly from the rest. We find that among Windows connections, nearly 26% of all retransmissions are due to spurious RTOs, while for Linux, less than 4% of retransmissions are due to spurious RTOs.
- The fraction of all retransmissions that are caused by spurious FR/R events is much smaller (3 - 5%), and does not differ much across OSes. It is important to note that the spurious FR/Rs occur only when network reordering events result in the generation of D or more duplicate ACKs—the occurrence of such events is independent of the sender-side OS. All OSes (other than Windows that uses a value of 2) use 3 as the value of D—see Table I. We find that this value does not result in a large number of inaccurate loss inferences.

2) ***Timeliness:***

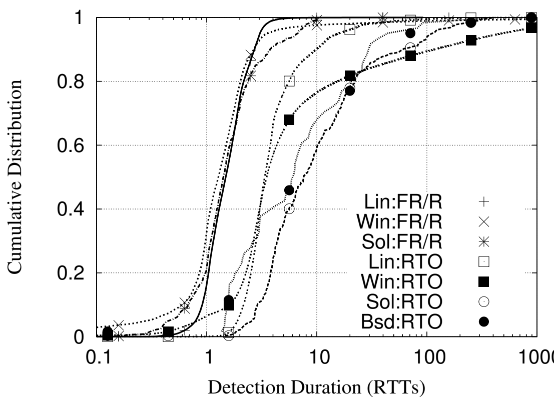


Fig. 2. Distribution of Detection duration for FR/R and RTO (Normalized with RTT)

a) Detection Durations: RTO-based loss detection has, in general, a longer duration than that based on FR/R—Figure 2 plots the distribution of loss detection durations in units of the moving average of RTT,[8] for all FR/R- and RTO-based retransmissions.[9] We find that:

- Most FR/R-based loss detection takes about 1-2 RTTs for all OSes.[10] For Solaris TCP connections, however, around 7% of FR/R detections take more than 5 RTTs.
- RTOs take much longer than FR/Rs to detect losses. Also unlike FR/Rs, the RTO-based detection durations differ significantly across OSes. While the median RTO detection duration for Windows and Linux is around 4 RTTs, it is almost 10 RTTs for Solaris and FreeBSD. The Solaris RTO-estimator uses a minimum RTO of 400 ms and an *srtt* multiplier of 1.25; FreeBSD uses a minimum RTO of 1200 ms—these values are significantly higher than for Windows or Linux. As a result, for connections with relatively small and stable RTTs, the RTOs computed by Solaris and FreeBSD tend to be higher—TCP connections on these OSes, therefore, take longer to detect a loss using RTOs.
- The tail of the distribution of RTO detection duration for Windows differs significantly from that for Linux. For instance, while only around 10% of RTOs are larger than 10 RTTs for Linux, nearly 25% of Windows RTOs are larger than this amount. On close inspection of our traces, we find that several of these larger RTOs in Windows result from losses at the beginning of the respective TCP connections, when the RTO is primarily governed by the initial RTO and has not converged to a value representative of the network path. Linux updates its estimates of RTT and RTO at a much higher frequency (once per segment) than Windows (once per flight) and

[8]The exponential-weighted moving average is computed using a weight of 1/4 for the current RTT sample.

[9]The detection time is below 1 RTT in some cases since the values are normalized due to the *moving average* of RTT, which can be larger than the current RTT sample.

[10]We do not plot data for FR/R events in FreeBSD since our traces yield a fairly small set of data points for this OS (only 271 FR/R events)—any conclusion may not be statistically significant. The number of RTO events in FreeBSD connections, however, exceeds 2000; hence, the distribution of RTO-based detection times is statistically more significant and is plotted.

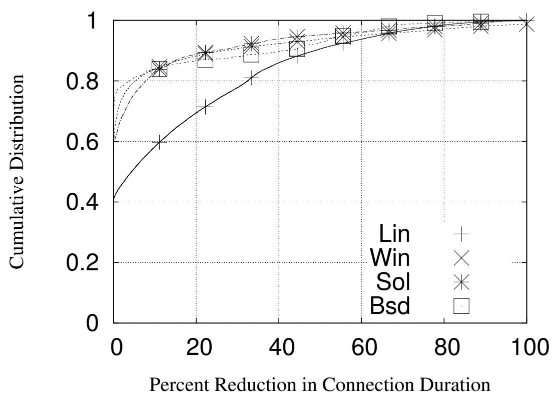


Fig. 3. Distribution of Best-Case Reduction in Connection Duration

converges faster.

Table IV summarizes the relative frequency of occurrence of RTO-based vs. FR/R-based loss detection. We find that 60-88% of TCP retransmissions are triggered by RTOs. Our observation above shows that all such RTO durations can be quite long. The prime reason for the high frequency of RTOs rather than FR/Rs is that there are often not enough segments in flight to trigger duplicate ACKs for a connection that experiences a loss. Thus, reducing the value of D should increase the likelihood of FR/R-based detection when losses occur. Reducing the value of RTO should reduce the loss-detection times when RTOs are unavoidable.

b) Recovery Durations: The time spent by a TCP connection in recovering from segment losses is independent of the loss-detection mechanism, and depends primarily on the number of segments lost within a flight. We analyze our traces to study recovery durations and find that these are also relatively independent of the sender-side OS. We also find that that use of *selective acknowledgments* (SACKs) helps reduce recovery times, but only when 3 or more segments are lost in a TCP flight. Due to space considerations, we emit the details here and refer the reader to [15]. Most relevantly, note that none of the six design parameters considered in this paper impact loss recovery; TCP immediately retransmits segments on inferring a loss—a TCP-like protocol can not do better than that. Thus, in this paper, what we are really studying is the design of loss detection mechanisms.

*3) **Scope for Improving Connection Durations:*** The observations made above on accuracy and timeliness suggest that real-world TCP implementations can deal more effectively with segment loss detection. A natural question to ask, though, is: *by how much does TCP's design really impact connection performance?* Or more importantly, *what is the maximum amount by which one can hope to improve connection durations by new settings for parameters related to TCP loss detection?*

In order to address these questions, we attempt to characterize the *Best-Case* reduction (an upper-bound) in connection durations that can be achieved by an ideal set of loss detection mechanisms. Our analysis is optimistic and assumes that in an *ideal* setting, (i) *all* spurious retransmissions (based on either FR/R or RTOs) are avoided, and (ii) *all* RTO-based loss detections take no more than the maximum RTT of the corresponding connection.[11] Specifically, for each connection in our traces: (i) we identify all instances of spurious FR/R-based loss detection and use Equation (1) to compute the savings in connection duration if the inaccurate loss inference is avoided; (ii) we identify all instance of spurious RTO-based loss detection and use Equation (2) to compute the savings in connection duration if the inaccurate loss inference is avoided; and (iii) we identify all needed RTO-based loss detection and compute the savings in connection duration if the RTO was instead equal to the maximum RTT of the connection (as described in Section IV-D). For each connection, we compute the total savings in connection duration as the sum of each of the above.

Figure 3 plots the distribution of the Best-Case reduction, as a percent of the original connection duration, for connections belonging to each of the 4 OSes. We observe that:

- 45-75% connections have little potential (less than 1%) for improving their connection duration by improving the configuration of loss detection.
  However, a significant fraction (15-40%) of connections can see greater than 10% reduction in their connection durations by improving loss detection.
- The potential for improving connection durations in Linux differs from that in other OSes. While more than 40% of Linux connections can see greater than 10% improvement in connection durations, less than 15% of Windows, FreeBSD, and Solaris connections have a similar opportunity.

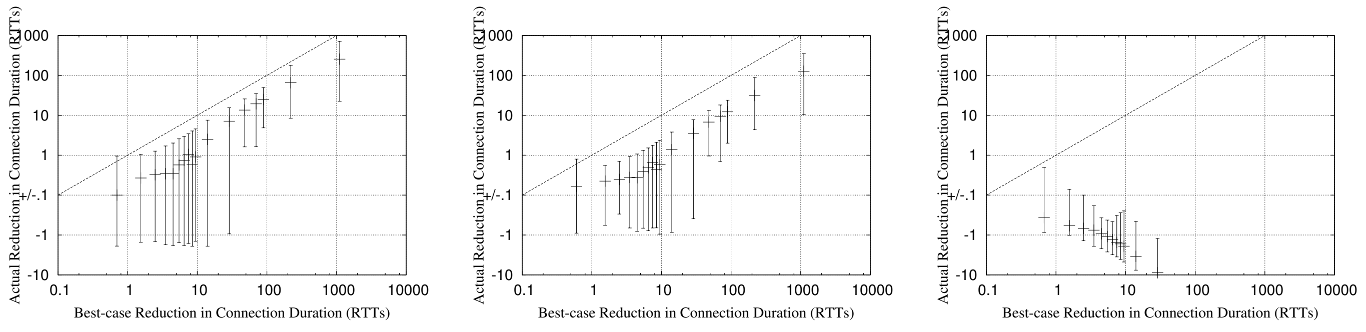
Based on the above, we expect to be able to significantly reduce connection durations by changing the configuration of loss detection parameters. In the rest of this section, we carefully compare the impact of each of these parameters to the upper-bound computed above.

B. Impact of The RTO Estimator

The value of RTO is controlled by each of the 4 parameters: $k, minRTO, a, b$.[12] If the RTO value is large, the number of spurious RTO-based retransmissions is reduced and helps improve connection durations. Furthermore, there is an increased likelihood of detecting losses by FR/R (rather than RTO). On the other hand, if the value of RTO is small, the time spent on detecting the loss (given by the RTO) is reduced and helps improve the connection duration. We vary the above 4 parameters—k in {2, 4, 6, 8}, minRTO in {100ms, 200ms, 400ms, 800ms, 1000ms}, a in {1, 1/2, 1/4, 1/8, 1/16, 1/32}, and b in {1, 1/2, 1/4, 1/8, 1/16, 1/32}—and for each combination of their values, estimate the RTOs that would be computed within each connection. We then estimate what

[11]Our analysis assumes that an oracle informs the configuration of both FR/R and RTOs. Specifically, the oracle helps the sender achieve 100% accuracy in FR/R-based loss detection by informing it when dupACKs are generated by events other than segment loss. The oracle also helps achieve ideal accuracy and timeliness of RTO-based loss detection by informing the sender of the maximum RTT that can be witnessed by the connection—the sender can then use this value as the RTO and: (i) avoid spurious RTO-based retransmissions, and (ii) quickly invoke non-spurious retransmissions.

[12]The current default setting for parameter $m = 1$ was observed to be optimal in our evaluations; we do not include evaluation results for varying m in this paper.



(a) k = 2 (b) k = 3 (c) k = 6

Fig. 4. Linux: Impact of k on Connection Durations

segments would be retransmitted spuriously due to premature RTOs, which RTO-detected losses would instead be detected by FR/R, and which RTO-detected losses would incur different detection durations.

We then study how each of these different phenomena interplay to impact the overall duration of each connection, by asking three questions: (i) *What is the reduction in connection duration when a spurious RTO is avoided?* (ii) *What is the reduction in connection duration when a loss is detected by FR/R (instead of an RTO)?* (iii) *What is the reduction in connection duration when the value of RTO is small?* All of these questions have already been addressed in Section IV-D. For each combination of RTO-related parameters, and for each connection, we then list all instances of any of the above phenomena, and use the analysis presented in Section IV-D to compute the total reduction in connection duration.[13]

Recall from Figure 3 that the scope for reduction in connection duration can differ by several orders of magnitude across different connections. To put our observations in perspective, therefore, throughout this section we plot the total reduction in duration for each connection (y-axis) as a function of the Best-Case reduction for that connection (x-axis), both computed in units of the average RTT for the connection. For improved readability of these plots, we first divide the x-axis (Best-Case reduction in connection duration) into logarithmically-sized bins. We then consider all connections that fall within each bin, and compute the average and the 95-percentile values of the actual reduction in duration for these connections. We then plot these per-bin average and 95-percentile values (plotted as error-bars) against the average Best-Case reduction in connection duration for that bin. Achieving the upper-bound of Best-Case reduction is represented by the dashed line with slope of 1.

We present our results for each of the 4 parameters below. For space reasons, we include graphs only for the Linux OS; graphs for other OSes are plotted only when the trend is different from that of Linux.

1) ***Impact of*** ***k:*** Figures 4(a), 4(b), and 4(c) plot the reductions in durations of connections as a function of their upper-bound, for k equal to 2, 3, and 6, respectively. We find that:

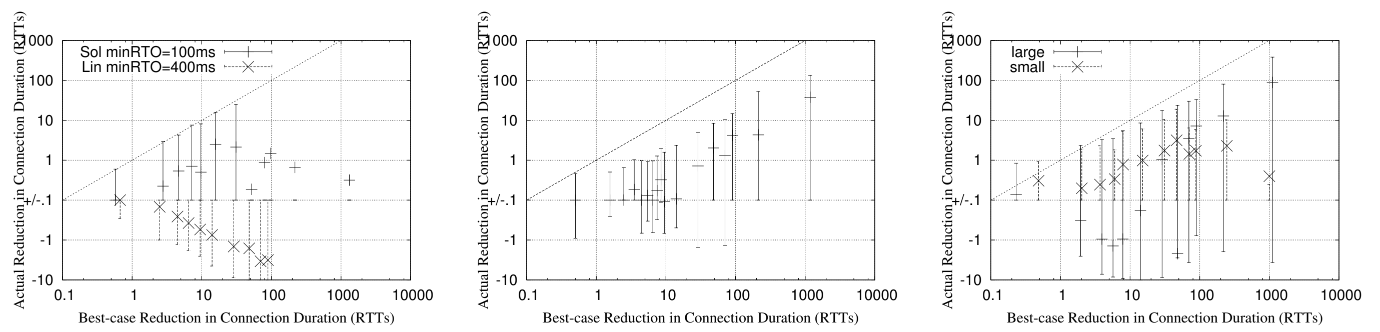
- The value of k significantly impacts the connection durations. We find that a small value of k (2 or 3) can help significantly reduce the connection durations of most connections. A value of 2, in fact, achieves an average reduction in connection duration that is within a factor of 5 of the Best-Case reduction. This suggests that perhaps k is the single-most influential parameter related to TCP loss detection and that setting it to a small value of 2 (rather than the in-use and recommended value of 4) can help significantly reduce connection durations. k = 6 consistently increases the connection durations.
A small fraction of connections do seem to experience a slight increase (1 RTT) in their connection durations even with a small value of k—this is especially true for connections with a small potential for Best-Case improvement ($<$ 10 RTTs).
- The impact of k on other OSes is similar in trend, but not as pronounced as for Linux. It is important to note that Linux tracks RTT on a finer time-scale and hence its RTO estimate is robust even with a small value of k.

[13]A negative value of the reduction implies an increase in connection duration.

2) ***Impact of*** ***$minRTO$:*** Figure 5(a) plots the reductions in connection durations for Solaris with $minRTO$ equal to 100ms and Linux with $minRTO$ equal to 400ms, respectively. We find that a larger value of $minRTO$ adversely impacts the connection durations of almost all connections. This suggests that the overall performance of TCP loss detection is significantly adversely impacted by a large value of $minRTO$. This observation was also made in [7], although the $minRTO$ evaluated was a very large value of 1 second. Fortunately, both Linux and Windows use a minRTO of $200ms$; FreeBSD and Solaris, however, use larger values. We find that reducing the Solaris $minRTO$ from 400ms to 100ms improves the connection durations of the Solaris connections—the improvement, however, is not as significant as that observed using small values of k. Reducing the Linux $minRTO$ to 100ms had negligible impact on the connection durations of most connections.

3) ***Impact of Smoothing Factors,*** ***a and b:*** The smoothing factors a and b have a less deterministic impact on the connection durations of TCP connections of all OSes other than Linux. Recall that the default values of smoothing factors implemented in all OSes are: $a = 1/4$, and $b = 1/8$. In general, a larger value of a or b seems to help reduce the connection durations of a larger fraction of connections; however a significant fraction of connections also witness an increase in connection durations.

Figure 5(b) plots the average and 95-percentile reduction in



(a) $minRTO$ = 100ms, 400ms (b) $a = 1/2$ (c) D = 2

Fig. 5. Linux: Impact of $minRTO$, a, and D

connection durations of the Linux connections as a function of their Best-Case reductions for $a = 1/2$. We find that a larger value of a helps reduce the durations for most Linux connections, especially those that have a large Best-Case potential for reduction. This correlates well with the fact that the Linux RTO estimator updates at a higher frequency (once per segment) than most other OSes (once per flight) and hence is less sensitive to large fluctuations in the measured RTT. This implies that in computing the RTT variation, the most recent sample of $rttvar$ should be given a weight of at least 1/4. $a = 1/32$ (or anything smaller than $1/4$) increases the connection durations of most Linux connections. Changing the value of b (to a larger or smaller value than $1/8$) has negligible impact—less than 1 RTT–on the connection durations of most Linux connections, independent of their Best-Case reductions.

C. *Impact of The dupACK Threshold*

Recall that the value of D can impact the connection durations in opposing ways. Table V—that lists the changes in total number of RTOs, FR/Rs, and spurious FR/R-based retransmissions, for $D = 2, 3, 4$—highlights this fact. For instance, we find that increasing D to 4 avoids 1.2% of Linux retransmissions due to spurious FR/R events, but at the same time causes 7.2% of retransmissions due to FR/R events to become RTO events. In order to see which of these factors has a more pronounced effect on connection durations, we use Equations (1) and (3) to evaluate all such events in each connection. In this way, for each connection, we compute the total reduction in connection duration when the dupACK threshold is varied from 4 to 2.

Note that the likely impact of D on a connection depends on its average flight size. A larger flight is likely to benefit from a large value of D that helps avoid spurious retransmissions due to dupACKs caused by network reordering. When the flight is small, however, a large D does avoid spurious FR/R retransmissions, but also implies that a genuine segment loss can not be detected by the faster FR/R mechanism and has to suffer delay from an RTO-based detection. To put this observation in perspective, Figure 5(c) plots separately for small and large Linux connections, the total connection duration reduction as a function of the Best-Case reduction, when D is changed from 3 to 2—we refer to connections that transmit 15KB ($\sim$ 10 MSS-sized segments) or less as "small" connections; such connections do not achieve a flight size larger than 4. We find that:

OS	# FR/R	# RTO	D=2 (D=3 for Win)		D=4	
			RTO to FR/R	Spurious Caused	FR/R to RTO	Spurious Avoided
Win	155770	918327	-35417 (-3.3%)	-30622 (-2.9%)	54016 (5.0%)	37561 (3.5%)
Lin	123735	186680	19115 (6.2%)	37673 (12.1%)	22280 (7.2%)	3709 (1.2%)
Sol	6613	20491	911 (3.4%)	1122 (4.1%)	2533 (9.4%)	992 (3.7%)
BSD	271	2041	42 (1.8%)	21 (0.9%)	25 (1.1%)	85 (3.7%)

TABLE V
IMPACT OF THE DUPACK THRESHOLD

- Reducing D from 3 to 2 reduces the connection duration of most (including the 95-percentile) of the small connections. Generally speaking, as the Best-Case reduction increases, so does the average reduction in connection durations—however, the average reduction is always more than an order-of-magnitude smaller than the Best-Case. Thus, D comes only after k in its ability to help reduce connection durations.
  Reducing D to 2 helps reduce the connection duration of only those large connections that have large values of Best-Case potential reduction—other large connections experience an increase in connection durations using $D = 2$.
  The impact of using $D = 2$ is similar for connections emanating from other sender-side OSes.
- Increasing the dupACK threshold to 4 consistently worsens the performance of most connections (including the 95-percentile performance), irrespective of the connection size, the potential Best-Case reduction, or the sender-side OS.

The observations made above suggest that D should be adaptable—it should take a small value when flight sizes are small and a larger value of 3 otherwise.

D. *Impact of The Smart Configuration*

In this section, we adopt the best-performing settings (the *Smart-Config*) for each of the 5 parameters and quantify the total improvement in connection durations. Specifically, we set $k = 2$, $minRTO = 100ms$, $a = 1/4$, and $b = 1/8$. The dupACK threshold is set according to the rule: $D = \max\{1, \min\{3, F-2\}\}$, where F is the current flight size of a connection.

Fig 6 plots the percentage change in per-connection con-

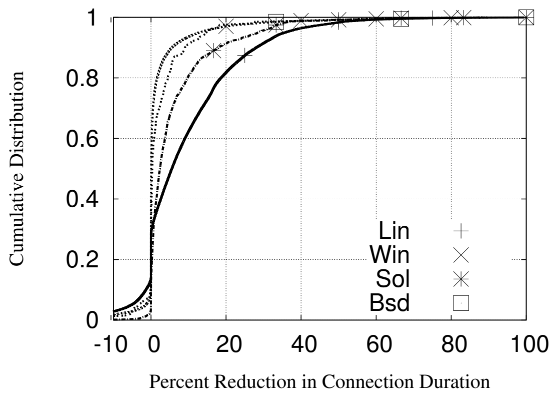


Fig. 6. Smart Config: Relative Change in Connection Durations

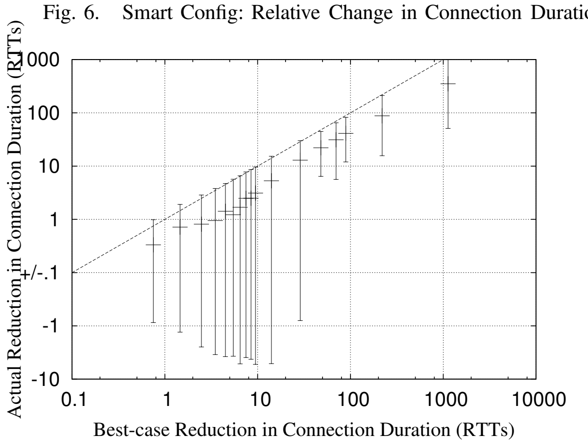


Fig. 7. Smart Config: Actual Connection Duration vs. Upper Bound in Reduction

nection duration with this configuration. We find that with the Smart-Config, the connection durations are reduced by more than 10% for nearly 40% of Linux connections. A smaller percentage of connections—7% for Windows, 20% for Solaris, and 10% for BSD—experience a similar improvement in the other OSes. The greater improvement for Linux connections is mainly because the Linux RTO estimator is less adversely affected by a reduction in the key parameter K—while reducing K to 2 causes only 2-6% additional spurious RTO events in Linux connections, it causes 12-34% additional spurious RTO events in Windows. The Smart-Config also results in an *increase* in connection durations for some connections. However, less than 3% of connections in each OS suffer an increase of more than 10%.

Fig 7 plots the connection duration change (in units of RTT), as a function of the best-achievable savings. We find that the *average* observed reduction in connection duration closely matches (within a factor of 2) the theoretical upper bound computed in our estimate of the Best-Case reduction in connection duration. There is a significant variability in the actual reduction in connection duration among connections with a similar estimate of the Best-Case upper bound. The variability, however, is smaller for larger values of the Best-Case estimate.

VI. Related Work

The evaluation of TCP detection/recovery mechanisms has been the focus of several key publications over the past two decades [7], [8], [16]. We briefly describe some of this work below.

In [8], Paxson investigated the effect of changing dupACK threshold on the number of spurious retransmissions. The data was collected by actively establishing approximately 20,000 connections that transferred 100KB of data between multiple machines, and by capturing the corresponding packet flow using tcpdump [11]. Paxson found that increasing dupACK threshold to 4 improves the ratio of needed FR/R to spurious FR/R by a factor of 2.5. However, it also reduces the chance of detecting a loss using FR/R by 30%. Reducing the dupack threshold to 2 increases the number of FR/R by 65-70% but the ratio of non-spurious to spurious FR/R reduces by a factor of 3.

While the above study presented novel insights into the trade-off in selecting the dupACK threshold, it is not very useful in configuring current TCP implementations for two reasons: (i) [8] does not evaluate how this trade-off impacts the overall performance of a TCP connection. For instance, it is not clear whether $D = 4$ is a better choice than $D = 3$, without first analyzing the impact on connection duration of a reduction in spurious FR/Rs as well as the impact of increase in frequency of RTOs. In Section IV-D of this paper, we develop models to estimate this impact and use these to make design decisions regarding the configuring of D and other TCP parameters. (ii) The TCP connections analyzed in [8] were generated by instantiating fixed-size bulk transfers between a set of 35 experimental sites. The resultant connection-set is limited in its sampling of diversity in network properties (RTTs, loss rates, bandwidth), connection properties (size of connection), and application properties (data generation behavior and user think times). It is not clear if the conclusions drawn using such a data-set are representative of the connections in the general Internet. For instance, most of current TCP connections are small (see Section III), and our analysis shows that small connections are more likely to benefit from a small value of D. If, on the other hand, a study analyzes mostly large connections, the results are biased against such observations. Also the Internet has grown tremendously over the past decade—it is not clear if contemporary networks and TCP implementations interact in a manner similar to a decade ago. In this paper, we passively analyzed recent traces of millions of real-world TCP connections.

More recently [16] suggested changes similar to those proposed in [8]. [16] recommends waiting for a certain time τ before reacting to duplicate ACKs. The recommended value of τ is 1 RTT. However, there is no substantial evaluation of the impact of this recommendation on TCP performance.

In [7], the authors investigate the impact of changing several parameters related to RTO-estimation on the accuracy and timeliness of RTO detection. The data used for this study is same as that used for [8] and hence suffers from the same lack of diversity/representativeness. Furthermore, at the time this study was conducted, the most popular $minRTO$ value was 1 second and the timer granularity was 500ms. The study found that these two parameters have a significant impact on the accuracy and timeliness of RTO, while the other parameters—including, k, a, and b—have negligible impact.

Eight years after that study, we find that timer granularity no longer impacts the performance of TCP loss detection (indeed, several current OSes use a 10ms timer). While the $minRTO$ does limit performance in some OSes (Solaris and BSD), it is not a dominant factor for most connections. Instead, we find that the multiplicative factor, k, is quite significant and a low value of k can help many connections achieve close to their Best-Case reduction in connection durations. Finally, we explicitly model and evaluate the impact of timeliness and accuracy of RTO-based loss detection on overall connection durations.

The design and evaluation of new detection/recovery mechanisms for TCP has also received considerable attention over the past few years. Several techniques have been designed to (i) detect and rectify the adverse impact of spurious retransmissions in an ongoing TCP transfer [17], [18], [19], [20], [21], and (ii) detect losses using alternate mechanisms [22], [23], [24], [25], [26], [27]. Unfortunately, due to deployment hurdles, most of these techniques have not been widely deployed in TCP implementations. Furthermore, the impact of these techniques on the overall connection duration of TCP connections has not been studied—we hope to study this impact in a manner similar to this paper as part of future work.

VII. Concluding Remarks

In this paper, we evaluate the impact of the configuration of TCP loss detection parameters on the performance of TCP connections. Our study relies on passive analysis of traces of more than 2.8 million real-world TCP connections. We analyze the impact of parameters on the trade-off between accuracy and timeliness of loss detection. We also explicitly model and evaluate the impact of this trade-off on the connection duration of TCP connections—to the best of our knowledge, this has not been done before.

We find that current RTO estimators are typically too conservative in incorporating RTT variability—we find that when the weight given to RTT variability is reduced by a factor of 2, TCP connections can achieve close to the best-achievable efficiency in loss detection. Also, unlike observations made in past work, the $minRTO$ and timer granularity are no longer the most influential parameters. Our study also reveals that the Linux RTO estimator is considerably more efficient than the proposed standard for an RTO estimator (which is also adopted by FreeBSD, Solaris, and Windows).

Our analysis suggests that by re-tuning the configuration of TCP parameters, up to 40% of Linux connections can witness a significant reduction (more than 10%) in their connection durations. For a majority of connections, this is close to the best-achievable reduction. Our findings should help guide the default configuration of TCP loss-detection parameters in prominent OS stacks.

References

[1] K. Thompson, G. Miller, and R. Wilder, "Wide-area internet traffic patterns and characteristics," *Network, IEEE*, vol. 11, no. 6, pp. 10–23, November 1997.

[2] K. Claffy, G. Miller, and K. Thompson, "The nature of the beast: Recent traffic measurements from an internet backbone," in *INET'98*, July 1998.

[3] S. Saroiu, P. K. Gummadi, and S. D. Gribble, "A Measurement study of peer-to-peer file sharing systems," in *Proceedings of Multimedia Computing and Networking 2002*, vol. 4673, Jan. 2002, pp. 156–170.

[4] K. Sripanidkulchai, B. Maggs, and H. Zhang, "An analysis of live streaming workloads on the Internet," in *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*. New York, NY, USA: ACM Press, 2004, pp. 41–54.

[5] V. Paxson and M. Allman, "RFC 2988: Computing TCP's retransmission timer," November 2000.

[6] W. Stevens, "RFC 2001: TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms," January, 1997.

[7] M. Allman and V. Paxson, "On estimating end-to-end network path properties," in *Proceedings of ACM SIGCOMM*, New York, NY, 1999, pp. 263–274.

[8] V. Paxson, "End-to-end Internet packet dynamics," in *SIGCOMM '97: Proceedings of the ACM SIGCOMM '97 conference on Applications, technologies, architectures, and protocols for computer communication*, vol. 27, no. 4. New York, NY, USA: ACM Press, October 1997, pp. 139–152.

[9] "URL http://tracer.csl.sony.co.jp/mawi/."

[10] "The dag project,univ. of waikato, URL http://dag.cs.waikato.ac.nz/."

[11] V. Jacobson, C. leres, and S. McCanne, "tcpdump: URL http://www.tcpdump.org."

[12] S. Rewaskar, J. Kaur, and F. Smith, "A passive state-machine approach for accurate analysis of TCP out-of-sequence segments," *ACM Computer Communication Review*, vol. 36, no. 3, pp. 51–64, July 2006.

[13] URL http://www.cs.unc.edu/∼jasleen/research/.

[14] M. Zalewski, "Passive OS fingerprinting tool: URL http://lcamtuf.coredump.cx/p0f.shtml." 2006.

[15] S. Rewaskar, J. Kaur, and F. Smith, "A performance study of loss detection/recovery in real-world TCP implementations," *Technical Report, Department of Computer Science, University of North Carolina at Chapel Hill*, April 2007.

[16] S. Bhandarkar, A. L. N. Reddy, M. Allman, and E. Blanton, "RFC 4653: Improving the robustness of TCP to non-congestion events," August 2006.

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

[18] E. Blanton and M. Allman, "On making TCP more robust to packet reordering," *SIGCOMM Comput. Commun. Rev.*, vol. 32, no. 1, pp. 20–30, 2002.

[19] M. Zhang, B. Karp, S. Floyd, and L. Peterson, "RR-TCP: A reordering-robust TCP with DSACK," in *ICNP '03: Proceedings of the 11th IEEE International Conference on Network Protocols*. Washington, DC, USA: IEEE Computer Society, 2003, p. 95.

[20] R. Ludwig and R. Katz, "The Eifel algorithm: making TCP robust against spurious retransmissions," *SIGCOMM Comput. Commun. Rev.*, vol. 30, no. 1, pp. 30–36, 2000.

[21] P. Sarolahti, M. Kojo, and K. Raatikainen, "F-RTO: An enhanced recovery algorithm for TCP retransmission timeouts," *SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 2, pp. 51–63, 2003.

[22] N. Fonseca and M. Crovella, "Bayesian packet loss detection for TCP," in *Proceedings of Infocom 2005*, Mar. 2005.

[23] S. Bohacek, J. P. Hespanha, J. Lee, C. Lim, and K. Obraczka, "TCP-PR: TCP for persistent packet reordering," in *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*. Washington, DC, USA: IEEE Computer Society, 2003, p. 222.

[24] A. Kesselman and Y. Mansour, "Optimizing TCP retransmission timeout," in *ICN ' 05: Proceedings of The 4th International Conference on Networking*, 2005, pp. 133–140.

[25] R. Jain, "A delay-based approach for congestion avoidance in interconnected heterogeneous computer networks," *SIGCOMM Comput. Commun. Rev.*, vol. 19, no. 5, pp. 56–71, October 1989.

[26] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson, "TCP vegas: New techniques for congestion detection and avoidance," *SIGCOMM Comput. Commun. Rev.*, vol. 24, no. 4, pp. 24–35, October 1994.

[27] D. X. Wei, C. Jin, S. H. Low, and S. Hegde, "FAST TCP: Motivation, architecture, algorithms, performance," *IEEE/ACM Trans. Netw.*, vol. 14, no. 6, pp. 1246–1259, 2006.