A Passive State-Machine Based Approach for Reliable Estimation of TCP Losses[*]

Sushant Rewaskar **Jasleen Kaur** **Don Smith**
Department of Computer Science
University of North Carolina at Chapel Hill

Technical report No. TR06-002

January 20, 2006

Abstract - While it is well-known that TCP performance degrades significantly on experiencing packet losses, not much is known about the way in which TCP losses occur in the real-world. In order to understand this issue, in this paper, we develop a passive analysis methodology for reliably inferring the loss processes that real-world TCP connections are subject to. We instantiate our methodology in analysis tools that implement detailed sender-side state machines for several prominent TCP stacks (Windows, Linux, BSD, and Solaris) and augment these with extra logic to correctly track TCP sender state as well as actual segment losses. Using these state machines we analyze traces of more than 25 *million* TCP connections, collected from 5 different locations around the world and report our findings.

1 Introduction

TCP is the dominant transport protocol used by Internet applications. Perhaps one of the most useful service semantics provided by TCP is that of *reliable delivery*. TCP implements reliability by *detecting* packet losses and *retransmitting* lost segments. It is well-known that the timeliness performance of a TCP connection degrades significantly whenever it experiences packet losses and invokes this pair of mechanisms. Given the popularity of TCP, it is, therefore, important to understand how are real-world TCP connections subject to packet losses and how well do its detection-recovery mechanisms deal with them. It is the goal of our research to do so.

As a first crucial step of this endeavor, in this paper, we address the issue of: *how to reliably derive the loss process that real-world TCP connections are subject to?* One of the most powerful approaches for studying TCP connections that represent real-world users and applications, is to passively analyze packet traces of ongoing TCP connections in the Internet. In this paper, we argue that existing approaches for passive inference of TCP losses do not adequately deal with either the inefficiencies of TCP's loss detection-recovery mechanisms, or the diversity in TCP implementations. In an effort involving several person-months, our approach is to recreate TCP sender state related to loss detection-recovery in a set of passive analysis tools, and augment these with extra logic for reliably inferring TCP losses, We validate our tools against several controlled experiments with diverse TCP senders. We then run these tools against packet traces of 25 million Internet TCP connections collected from 5 different locations, and report our findings.

In the rest of this paper, we discuss past work and our passive analysis methodology in Section 2. We present tool validation in Section 3 and the results of analysis of Internet connections in Section 4. We outline the implications of our analysis in Section 5.

[*]This research was supported in part by NSF CAREER grant CNS-0347814, a UNC Junior Faculty Development Award, and NSF RI grant EIA-0303590.

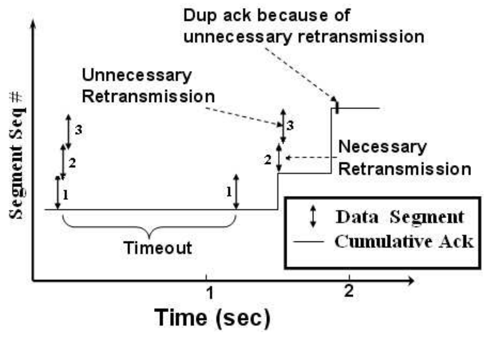

Figure 1: Implicit TCP Retransmission

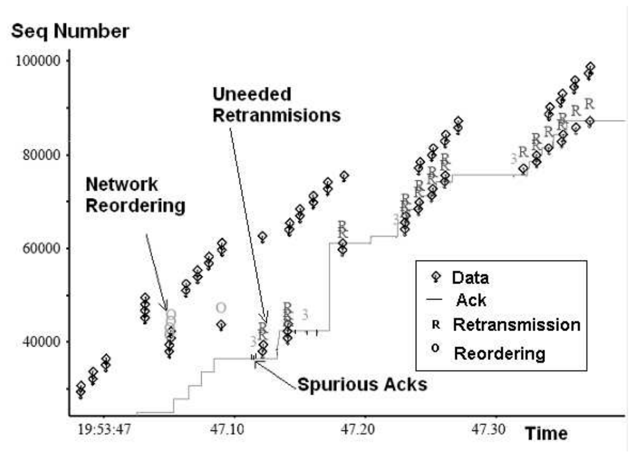

Figure 2: Unneeded Retransmission

2 Passive Loss Inference Methodology

TCP uses a well-known combination of detection and recovery mechanisms to deal with packet losses—we refer the reader to [5, 10, 8, 14] for details of *retransmission timeouts* (RTOs), *fast retransmit/recovery* (FR/R), *triple duplicate acks* (TDA), *partial acks* (PAs), and *selective acks* (SACKs). Each of these mechanisms is used to *retransmit* segments that are perceived to be lost. Below we consider several approaches for reliably inferring packet losses from the packet trace of a TCP connection.

2.1 Passive Inference of TCP Losses

Why not consider all retransmissions? Since TCP *retransmits* segments on detecting packet losses, the simplest (and common) approach for inferring segment loss is to simply look for the reappearance of some segments in the TCP packet trace and assume that the original transmission was lost. However, this approach can lead to over-estimation of losses as illustrated in Fig 1, which depicts part of a TCP connection selected from the *unc* trace. Segment 2 is retransmitted during a post-timeout period, although the original transmission was successful (as is confirmed by the subsequent ACK sequence). Note that while the segment was retransmitted, this was not the result of any *explicit* loss detection/recovery attempt by the TCP protocol. This example, thus, illustrates that in order to reliably infer packet losses, it is important to *track the explicit triggering of TCP's loss detection mechanisms—namely, RTO, TDA, PA, and SACK.*

Why not simply replicate TCP sender state? It turns out that even simply tracking the triggering of loss detection/recovery mechanisms in a TCP connection—as is done in [9]—is not sufficient for reliably inferring packet losses. This is because of two reasons related to TCP's inability to accurately infer packet losses:
Some losses do not trigger TCP's loss detection phases. For implementation efficiency, TCP senders maintain only a limited history about unsuccessful transmissions. In particular, if multiple packet losses are followed by a timeout, the sender explicitly discovers and recovers only from the first of those losses. As a result, the remaining packet losses may not get discovered by simply tracking the invocation of TCP's four loss detection mechanisms described above (RTO, TDA, PA, SACK). Fig 1 illustrates this for segment 1, which was unsuccessfully transmitted the first time. The segment gets retransmitted in the post-timeout period, but without explicitly triggering TCP's loss detection/recovery mechanisms. It is, thus, important to *track implicit retransmissions that are needed for recovering from packet losses*.
A TCP sender may incorrectly infer packet losses. TCP may retransmit a packet too early if its RTO computation is not conservative. Furthermore, some packet re-ordering events may result in the receipt of TDAs, triggering a loss detection/recovery phase in TCP. In fact, Fig 2, which

again depicts part of a TCP connection selected from the *unc* trace (and visualized using the *tcptrace* utility [4]), plots a connection in which a *single* packet reordering event resulted in the triggering of 64 subsequent phases of fast retransmit/recovery, that lasted for more than 5 seconds! It is, thus, important to *track explicit retransmissions that are not needed for recovering from packet losses*.

Basic Approach Based on the above discussion, our basic approach for passive inference of TCP losses is to: **(i)** implement partial state-machine for a TCP sender that uses the ACK stream to track the triggering of loss detection/recovery mechanisms, and **(ii)** augment the state machine with extra state and logic about the transmission order and timing of *all* previously-transmitted packets, in order to classify retransmissions as needed or not. Using this basic approach, we can classify segment retransmissions as triggered by: (i) RTOs, (ii) TDAs, (iii) PAs, (iv) SACKs, and (v) implicit. Furthermore, each retransmission is further classified as *needed* or *unneeded*. Fig 3 depicts this classification taxonomy.

2.2 Practical Challenges in Loss Inference

Three kinds of practical concerns complicate the implementation of the above approach. We describe these concerns and how we address them below.

Diverse and Non-documented TCP Stacks

The Challenge: TCP implementations written by different operating system (OS) vendors may differ (sometimes significantly) in either their interpretations or their conformance to TCP specification/standards. Furthermore, a few aspects of TCP—such as how a sender responds to SACK blocks—are not standardized. As a result, the sender-side state machines are specific to the OS they run on. This results in two main challenges in implementing our basic approach. First, the difference in implementations on different OSes necessitates that we implement different programs to analyze connections originating from different sender-side OSes. More significantly, given the trace of a TCP connection, it is non-trivial to identify the corresponding sender-side OS and decide which OS-specific analysis program to use for analyzing the connection. Second, most OSes either have proprietary code or have insufficient documentation on their TCP implementations. Without detailed knowledge of the loss detection/recovery implementations, it is not possible to replicate these mechanisms in our OS-specific analysis programs.

Our Approach: We extract sufficient details about the implementation of loss detection/recovery in several prominent OS stacks by using an approach similar to the *t-bit* approach described in [11]. Specifically, we install four different OSes—namely, Windows XP, Linux 2.4.2, FreeBSD 4.10, and Solaris—on experimental lab machines and run the Apache web-server on each machine. We expect this range of OSes to cover a majority of the connections we analyze. We then implement an application-level TCP receiver (by borrowing from the t-bit code base) that initiates TCP connections to each of the server machines and requests HTTP objects. Once the server machines start sending the objects, the receiver artificially generates different sequences in the ACK stream to trigger loss detection/recovery mechanisms on the sender-side stacks (including TDAs, RTOs, PAs, SACKs). We then use the manner in which the server responds to the ACK stream for inferring several characteristics of the sender-side TCP implementation, including the computation of RTO, the number of duplicate ACKs that trigger FR/R, and the response to SACK blocks. Details of the extracted characteristics can be found in [13]. We use these details in our implementation of four OS-specific trace analysis programs.

For each TCP connection to be analyzed, we run its packet trace against all four analysis programs. We then select the program that is able to explain each retransmission event. Events that cannot be explained by any program are counted as unexplained and connections with a large fraction of unexplained events are discarded.

Delays and Losses Between Monitor and Sender

The Challenge: Packet traces used in passive analysis are typically collected at links that aggregate traffic from a large and diverse population. As a result, there may be several network links on the path between a TCP sender and the trace monitoring point. Thus, the data packets transmitted by the sender may experience delays,[1] losses, or reordering before the monitor observes them; the same is true for ACK packets that traverse between the monitor and the sender. Consequently, the data and ACK streams observed at the monitor may differ from those seen at the TCP sender. In particular, if some of the TDAs observed at the monitor fail to reach the sender, the analysis programs may incorrectly conclude that the sender has entered FR/R. Similarly, if a data packet gets lost before it reaches the monitor, and subsequently gets retransmitted, the analysis programs may fail to infer that the packet has been *re*-transmitted. Thus, the programs may not be able to accurately track the sender-side state machine.

Our Approach: In order to deal with this complication, we use a general approach in which loss indications in the ACK stream trigger only *tentative* state changes in the monitor state machine, which are *confirmed* only by subsequent retransmission behavior by the sender. In addition, we consider all *out-of-sequence* (OOS) segments (and not just retransmitted segments) as possible indicators of packet loss. Furthermore, we infer network reordering by (i) detecting reordering in the *IP-id* field of packets seen from a given TCP source, and (ii) detecting if an out-of-sequence segment appears within a fraction of the connection's minimum RTT after the segment with the next higher sequence number.

Non-availability of SACK Options

The Challenge: A large number of traces do not capture the TCP option field. SACK blocks are transmitted as TCP options and hence are not available for passive analysis of these traces. The sender may have used the SACK block information to retransmit certain packets. In absence of these blocks, the monitor will fail to accurately identify the cause of these retransmissions and hence may not mark them correctly.

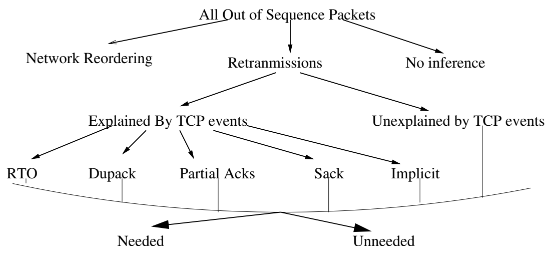

Figure 3: Classification Taxonomy

Our Approach: To overcome this problem, we develop the following heuristic to identify whether a packet could have been triggered by incoming SACK information. We classify a segment retransmission as SACK-triggered if: (i) the connection is in FR/R, (ii) the retransmission is not explained by either RTO or a PA, and (iii) the sequence number of the retransmitted segment is less than the highest sequence number that was in flight when the connection entered FR/R. We validate this heuristic using a one hour segment from the *unc* trace. We first run our analysis tools with the SACK blocks available and note the retransmissions that were SACK-triggered. Then we remove the SACK blocks from this trace and run the tools with the above heuristic. The analysis that relied on availability of SACK blocks identified 54160 SACK events. The heuristic-based analysis identified all of these events, but also marked 4333 of the unexplained events as SACK triggered. Apart from the low number of unexplained events incorrectly classified the heuristics works very well.

2.3 Summary of Our Methodology

Our methodology for reliably inferring TCP losses and the triggering of TCP loss detection/recovery mechanisms can be summarized as follows.

1. We first extract the implementation details of four prominent TCP stacks (Windows XP, Linux 2.4.2, FreeBSD, Solaris) using the approach described in Section 2.2.

2. We then replicate the loss detection/recovery mecha-

[1]The RTT measured at the monitor (monitor-receiver-monitor) is less than that measured at the sender (sender-receiver-sender). This implies that the RTO computed at the monitor may be smaller than that used by the sender. Fortunately, this discrepancy does not negatively impact our analysis—the RTO is used as a *minimum* threshold for the gap between the original transmission and retransmission of a lost segment, in order to identify retransmissions that occur due to timeouts. Therefore, a smaller-than-actual value of RTO would simply lower the threshold and still be able to correctly infer such retransmissions.

nisms in four OS-specific analysis state machines—these state machines use the data and ACK streams as input. Loss indications in the ACK stream are used to only tentatively trigger state transitions, which are confirmed only by subsequent segment retransmission behavior.

3. We then augment these machines with extra logic and state about all previously-transmitted packets, in order to infer packet losses with accuracy greater than TCP.

4. We then run each connection trace against all four machines and use the results from the one that can explain most ($> 20\%$) of the observed OOS segments. In case more than one machine matches this criteria, we select one randomly.

Our methodology classifies all OOS segments that appear within the packet trace of a TCP connection, according to the taxonomy depicted in Fig 3.

We have implemented the above machines in the C programming language. All four implementations can analyze more than a million connections in a few minutes. Several details of our methodology and implementation have not been included in this section due to space constraints. These details can be found in [13].

Previously, Allman [6] and Jaiswal[9] have presented methods for passive estimations of losses. Allman assumes that all explicit retransmissions are necessary; he counts the *number* of implicit retransmissions that were unneeded by counting the number of duplicate ACKs generated in a post-timeout recovery period. While this approach works relatively well, it does not identify exactly *which* of the implicit retransmissions were unneeded. Jaiswal et. al. use a state-machine based approach derived from the specifications in RFC 2581 [5]—this is similar to our FreeBSD state machine. This approach does not take into consideration variations in other TCP stacks. Furthermore, since the purpose of [9] is not to study packet losses in detail, their analysis tool is limited in the granularity with which OOS segments are classified. In the next two sections, we compare our methodology with both of the above.

3 Validation

We validate our analysis tools against TCP connections for which the "ground truth" about the classification of each OOS segment is known. To do this, we modified the TCP Behavior Inference Tool (*tbit*) [11]. *tbit* emulates a TCP protocol stack for the receiver side of a unidirectional data transfer where the sender is a normal application (in our case a Web server) running over a real TCP implementation in a specific operating system. We modify *tbit* to simulate different packet loss scenarios that would trigger sender responses by withholding ACKs, sending duplicate ACKs, and providing SACK blocks reflecting various gaps in received sequence numbers. Because the state machine analysis also depends on inferring the TCP sender's RTO to identify retransmissions triggered by timeouts, we also use *tbit* to delay ACKs thus simulating variable round-trip delays. For some of the validation scenarios described below we also use dummynet on the *tbit* machine to create additional latency between the sender and receiver.

For each validation scenario we used two machines, one running *tbit* and the other running a web server, connected over a switched 100/1000 Mbps Ethernet that is shared by users in the Computer Science department. *tbit* established a TCP connection to the web server and sent a valid HTTP request for a large file. *tbit* then implemented the desired validation scenario with a specific generated ACK stream. Unless stated otherwise, each validation scenario was repeated 100 times because not all sources of variation in timing—ranging from 1ms to 10ms—could be controlled (e.g. OS scheduling, Ethernet switch delays, etc.). Separate estimates of these uncontrolled delays concluded that the majority were less than 1 millisecond and nearly all were less than 10 milliseconds.

The entire suite of validation scenarios was run with *tbit* connecting to four different TCP implementations on the sender machine – Windows XP, Solaris, Linux 2.4.2, and FreeBSD 4.10. Bidirectional tcpdumps of all packets were taken on these sender machines and the traces were then used as input to our validation procedures. The procedures we used have two parts – (1) to verify that each TCP implementation responds in real operation as we expected, and (2) to verify that the state machine analysis programs correctly emulate each implementation's responses. For the first part we processed the tcpdump traces with *tcptrace* [4] and other tools to verify the implementations' responses. For the second part, we used the tcpdumps as input to the state machine analysis programs and recorded their outputs. By comparing the results from the state machine analysis with the known implementation responses, we could determine how correct its inferences about conditions at the sender were. We also used the tcpdumps as input to the analysis program, *tcpflows*, presented in [9] but report the results from this only when they differ substantially from ours.

RTO classification: The first group of validation scenarios deal with how well the state machine analysis can infer the sender's estimate of RTT and RTO which are critical in identifying retransmissions triggered by timeouts. In this group of validation scenarios all retransmissions are known to be triggered by timeouts. The state machine for each implementation requires correct values for parameters defining the initial and minimum RTO, the timer granularity, and the equations used in computing RTO. These parameters are verified as part of the validation results. Table 1 gives the values used for these parameters in the state machine for each TCP implementation.[2]

RTT estimation: Dummynet was used to set a constant minimum RTT—of 50, 100, 150, 200, 400, 1000, and 2000 ms—between the two machines. All RTTs estimated segment/ACK pairs by our state machines were within +10 milliseconds of the value set by dummynet.

Initial RTO setting: The initial RTO parameter helps classify retransmissions of SYN or SYN+ACK segments at connection establishment. *tbit* initiated a connection (sent SYN) but did not respond to the SYN+ACK sent by the server. This resulted in a retransmission of the SYN+ACK after the initial RTO interval. Our state machines for each implementation correctly identified the SYN+ACK retransmission as being triggered by RTO; further, the measured RTO was equal to the value expected by our state machines +/- the timer granularity (also shown in Fig 1).

[2]Details about the RTO computation (srtt and rttvar) are taken from RFC 2581 [5]. Linux, however, uses a significantly different computation for the variance in RTT—we extract this from the Linux source code. The details can be found in [13].

Minimum RTO setting: No delays were added to the actual RTT (typically 1 millisecond) over the switched Ethernet. Thus any retransmission triggered by an RTO should occur after an interval approximately equal to the minimum RTO. *tbit* received and ACKed a significant number of segments (typically 50 or more) so the sender's RTO calculation stabilized before withholding all ACKs to trigger an RTO retransmission. The tcpdump showed that the retransmissions occurred after the expected time intervals +/- the timer granularity. The state machine for each implementation correctly identified these retransmissions as triggered by RTO using these minimum values and timer granularities.

RTO estimation - constant RTT: Dummynet was used to set a constant minimum RTT—ranging from 50ms to 2s—between the two machines with variations only caused by switch delays. *tbit* received and ACKed a significant number of segments (typically 80 or more) so the sender's RTO calculation stabilized before withholding ACKs to trigger RTO retransmissions. The actual time differences found in the tcpdump between initial and subsequent transmissions of the same segment are summarized in Table 2 showing the mean, min, and max time intervals. The values shown are in good agreement with the expected values given the fixed dummynet delays, the minimum RTT, and the timer granularity for each implementation.

RTO estimation - variable RTT: This is the same scenario as above, except that ACKs were delayed randomly by *tbit* by up to 50% of the dummynet imposed minimum RTT. The results are also summarized in Table 2. The values shown correspond to the expected values given the fixed dummynet delays, the added delay variability, the minimum RTT, and the timer granularity for each implementation.

Comparison to *tcpflows*: Table 2 shows the observed min, max and mean time interval for the RTO validation experiments. It shows that the various implementations differ widely in their RTO computations. Solaris, for instance, initializes RTO to a high value and hence, over a short interval, ends up with a higher RTO value then FreeBSD (even though its min RTO is much lower

Parameter	Linux	Windows	FreeBSD	Solaris
Timer granularity	10ms	500ms	10ms	10ms
Initial RTO (s)	3	3	3.375	3
Min RTO (ms)	200	1000	1200	400
RTO	srtt + vartt	srtt + 4*rttvar	srtt+ 4*rttvar	1.25*srtt + 4*rttvar
Dup-ACK threshold	3	2	3	3

Table 1: TCP variants

than that of FreeBSD). Linux converges to the correct value much faster and has the lowest RTO value. In all cases, we were able to correctly classify all retransmissions triggered by RTO. The *tcpflows* tool [9] failed to correctly handle RTO estimation for many of the constant and variable RTT scenarios—this is primarily because it does not incorporate the diversity across different implementations.

FR/R classification: The second group of validation scenarios deals with how well the state machine analysis can infer the sender's response to duplicate ACKs, partial ACKs in Fast Recovery, and SACK blocks.

Number of duplicate ACKs to trigger retransmission: *tbit* received and ACKed the first 15 segments then sent duplicate ACKs (without delays) for the 15th in response to subsequent segments (thus simulating loss of the 16th segment). The number of duplicate ACKs was varied from 1 to 4. Each of our state machines classified the corresponding retransmissions as triggered by duplicate ACKs. The *tcpflows* tool failed to recognize retransmissions triggered by 2 duplicate ACKs in Windows (and instead classified them as RTO retransmissions)—this is because it assumes 3 duplicate ACKs are needed.

Response to Partial ACKs in Fast Recovery: *tbit* triggered a retransmission by duplicate ACKs (as described above) and then sent partial ACKs for other segments transmitted between the original and re-transmission. Our results verified the expected retransmission events in the tcpdumps and the state machine for each implementation correctly identified the event triggering the retransmission. Note that Windows TCP does not retransmit on receiving a partial ACK during FR/R but instead retransmissions are triggered by RTO (does not implement newReno but does use SACK if present).

Response to SACK blocks *tbit* triggered a retransmission by duplicate ACKs (as described above) for the 15th segment and generated several different cases of SACK block contents indicating gaps in the received segments beyond the 15th. In all cases the correct missing segments were identified and retransmitted without incurring a timeout—and our state machines correctly classified such retransmissions. The *tcpflows* tool, which does not use SACK blocks, classified the above as simply retransmissions in recovery after duplicate acks or as being triggered by RTO (for Windows connections).

Unneeded and Needed Retransmissions: *tbit* simulates the implicit retransmission scenario of Fig 1. In a second scenario, it sends spurious duplicate ACKs to trigger an unneeded retransmission. Our state machines correctly classified the corresponding retransmissions as *needed* or *unneeded*. Allman [6] correctly identified the unneeded retransmission in the first scenario but failed to identify it in the second case.

4 Analysis of TCP Connections

We next apply our state machines to analyze TCP connection traces collected from 5 different global locations. Below, we first describe these data sources and then present our analysis results.

Min	Constant RTT				Variable RTT			
RTT	Linux	Windows	FreeBSD	Solaris	Linux	Windows	FreeBSD	Solaris
50	256 (244,260)	751 (611,856)	1199 (1199,1200)	2945 (2932,2.962)	255 (250,260)	580 (509,787)	1199 (1199,1200)	3216 (3023,3.466)
100	315 (300,337)	858 (748,965)	1199 (1199,1200)	2596 (2589,2605)	314 (300 ,330)	597 (500,692)	1199 (1199,1200)	3150 (2961,3.374)
150	385 (374,390)	953 (855,978)	1199 (1199,1200)	2246 (2240,2265)	385 (380,390)	740 (613,827)	1199 (1199,1200)	3150 (2905,3349)
200	455 (441,467)	1081 (961,1276)	1199 (1199,1200)	1896 (1890,1915)	435 (430,440)	877 (805,1087)	1199 (1199,1200)	3072 (2871,3308)
400	737 (712,750)	1780 (1764,1868)	1199 (1199,1200)	915 (0910,0920)	671 (661,680)	1281 (1205,1488)	1199 (1199,1200)	2889 (2655,3108)
1000	1585 (1570,1600)	3237 (3015,3400)	1577 (1573,1581)	1655 (1650,1661)	2562 (2542,2570)	2567 (2400,2702)	1398 (1367,1434)	2522 (2102,2855)
2000	3014 (3000,3020)	5741 (5633,5888)	2934 (2930,2940)	2895 (2881,2899)	2627 (2561,2722)	4948 (4619,7311)	2613 (2515,2697)	3948 (3611,4286)

Table 2: RTO estimation results for constant and variable RTT (s). This table gives the mean RTO and the min and max RTO (in parenthesis) from 100 repetitions of the validation scenarios.

Trace	Duration	Avg TCP Load	# Connections	# Bytes	# Packets
Abilene-OC48-2002 (abi)	2h	211.41 Mbps	7.1 M	190.3 G	160.1 M
Liepzig-1Gbps-2003 (lei)	2h 45m	9.53 Mbps	2.4 M	11.8 G	17.3 M
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

Table 3: General Characteristics of Packet Traces

Data Sources: Table 3 describes the traces used in our analysis. These traces are collected from links with transmission capacity ranging from 155 Mbps to OC-48. The *abi* traces [2] are collected from a backbone link of the Internet-2 network (Abilene); the *jap* trace [3] is collected off a trans-Pacific link connecting Japan to the US; the *unc* and *lei* [1] traces are collected at the campus-to-Internet links of the University of North Carolina and University of Leipzig, respectively; the *ibi* trace captures traffic served by a cluster of high-traffic web-servers (ibiblio.org) hosted at UNC; All traces except the one from the link to Japan were collected using Endace DAG cards.

For our analysis, we use only those connections that transmit at least 10 segments. Furthermore, since our objective is to study TCP retransmissions, we select only those connections in which at least one OOS segment is observed ("OOS" connections). Fig 4 shows the impact of applying the latter filter. While less than 50% of connections that transmit at least 10 segments also have some OOS segments, these connections carry most of the bytes in this class. Furthermore, the traces vary significantly in the distribution of bytes transmitted per connection—this adds to the diversity of our results.

Efficiency of TCP Loss Detection Table 5 shows our classifications for OOS segments in the five traces, according to the top-levels of the taxonomy of Fig 3. We find that:

- Our state machines are able to infer a cause for more than 90% of all OOS segments in the traces and explain the triggering mechanism for 88-98% of all re-

Trace	All Connections			OOS Connections		
	# Conn	# Bytes	# Packets	% Conn	% Bytes	% Packets
abi	388.9 K	180.1 G	148.5 M	17.60 %	68.11 %	68.85 %
lei	75.4 K	10.5 G	12.6 M	18.82 %	74.88 %	77.24 %
jap	18.5 K	3.3 G	3.1 M	48.65 %	96.08 %	93.44 %
unc	774.8 K	121.3 G	129.6 M	21.82 %	78.45 %	77.83 %
ibi	287.5 K	161.8 G	157.2 M	27.31 %	83.30 %	82.37 %

Table 4: Connections That Transmit More Than 10 Segments

Trace	Total OOS	% Network Reorder	% No Inference	Retransmissions						
				# Total	% RTO	% Dupack	% PA	% SACK	% Implicit	% Unexp
abi	409.9 K	18	9.6	296.2 K	32.5 (45)	11.5 (15.9)	2.0 (2.78)	4.5 (6.21)	18.3 (25.3)	3.5 (4.8)
lei	51.1 K	0.47	1.7	49.9 K	46.3 (47.4)	5.9 (6.0)	1.9 (1.9)	4.9 (5.0)	27.7 (28.3)	11.1 (11.4)
jap	51.6 K	2.9	0.4	49.9 K	47.5 (49.1)	11.9 (12.4)	2.6 (2.7)	1.7 (1.8)	31.4 (32.4)	1.6 (1.6)
unc	697.7 K	29	6.69	445.6 K	34.2 (53.5)	6.1 (9.6)	2.8 (4.4)	1.5 (2.3)	13.2 (20.6)	6.0 (9.5)
ibi	504.2K	0.2	0.7	499.3 K	33.1 (33.4)	14.0 (14.0)	4.8 (4.9)	0.0 (0.0)	44.1 (44.5)	3.0 (3.1)

Table 5: Classification of OOS segments (numbers in parenthesis are normalized w.r. to total retransmissions)

transmissions. We believe this high rate of success has been achieved due to the world-wide deployment of Windows XP and Linux OS, which are explicitly incorporated in our analysis.

- For the *abi* and *unc* traces,[3] nearly 20-30% of OOS events are classified as due to network packet reordering between the sender and the monitor—these numbers appear unusually high. To investigate these events further, in Fig 4, we plot the time gap (referred to as the *resequencing delay*) between each such OOS segment and the segment with the next higher sequence number. We find that most of the resequencing delays are within 5 ms—this indeed corresponds to timescales of network reordering and is much smaller than typical RTTs. The small fraction of OOS segments with large resequencing delays occur in connections with large minimum RTTs as well.

- In 4 out of the 5 traces, almost 50% of retransmissions were triggered by timeouts (33% in the 5th trace). Only 10-15% of retransmissions are actually triggered by duplicate ACKs which causes TCP to enter the more efficient FR/R recovery phase.

- Approximately 20-45% of TCP retransmissions are *implicit* and occur as a result of the TCP approximation to a Go-Back-N recovery mechanism after a timeout.

We also used the *tcpflows* tool to process three of the traces (the other two could not be processed by *tcpflows* because of incompatible trace formats). The results are shown in Fig 6. We find that due to the classification inaccuracy of *tcpflows* for non-BSD TCP implementations

[3] A known contributor of excessive reordering in the UNC trace is the presence of intrusion detection appliances that divert selected IP packets from the fast data-path for deeper inspection.

Trace	Network Reorder	Retransmissions			
		RTO	Dupack	RTO-recovery	FR/R recovery
lei	0.4%	42.8%	17.0%	35.3%	4.5%
unc	35.8%	29.3%	9.9%	17.1%	7.9%
ibi	0.28%	23.5%	23.4%	27.1%	25.8%

Table 6: Results from *tcpflows*

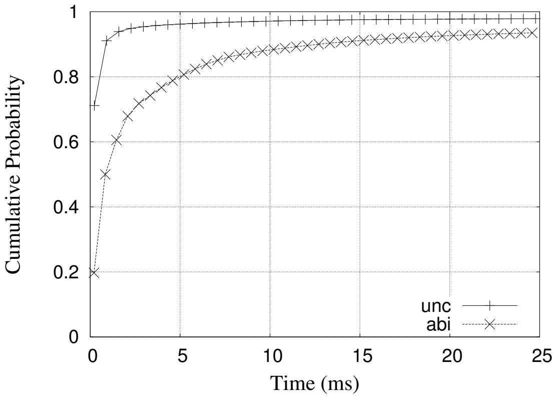

Figure 4: Resequencing delays for reordered segments

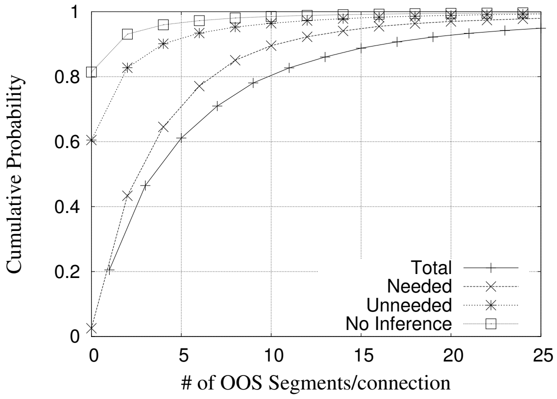

Figure 5: Distribution of Unneeded Retransmissions

(as demonstrated in Section 3), the classification results do not match those of ours.

Accuracy of TCP Loss Detection For any retransmission, our analysis also attempts to find out if the retransmission was unneeded (i.e., the original transmission had successfully reached the receiver). Table 7 lists the fraction of retransmissions that are classified as unneeded in each of our traces. We find that this fraction can range from 18-38%. The results are quite consistent with those reported by Allman [6] for three of the traces but are somewhat higher for two. There are two reasons for this. First, as demonstrated in Sections 2 and 3, [6] does not identify explicit retransmissions that are unneeded. Second, [6] classifies *all* implicit retransmissions as needed, unless proved otherwise. Our analysis, on the other hand, does not infer that an implicit retransmission is needed, unless proved so.

We find that a significant number of retransmissions could not be proved to be needed/unneeded, and suspect that these are classified as needed by [6].

Fig 5 plots the distribution of the number of needed and unneeded packets within each connection for the *ibi* trace. We find that while a majority of connections (80%) send less than 2 unneeded retransmissions, a non-negligible fraction (7%) send more than 5 unneeded retransmissions. Table 7 indicates that while most unneeded retransmissions are *implicit*, nearly 3 to 15.6 % can be those corresponding to RTOs or duplicate ACKs—such retransmissions result in an unnecessary and significant reduction in TCP sending rate, especially in the case of RTOs.

5 Implications of Our Analysis

Our analysis of real-world TCP connections suggest important implications for TCP performance (and especially for development of analytic models of TCP throughput as a function of loss rates [12, 7]). Considering the five traces in aggregate, we find that about 1% of all segments

Trace	Total # Retran	Our Approach									Allman [6]	
		Needed	Unneeded							No Inference	Needed	Unneeded
			%Total	Implicit	RTO	TDA	PartialAck	Sack	Unexp			
abi	296.2 K	70.0%	20.1%	4.4%	5.8%	1.1%	0.2%	1.0%	7.6%	9.9%	87.4%	12.6%
lei	49.9 K	46.1%	26.8%	7.1%	7.3%	0.7%	0.3%	0.4%	10.9%	27.1%	88.8%	11.2%
jap	49.9 K	70.7%	18.2%	13.3%	2.9%	0.3%	0.2%	0.0%	1.7%	11.1%	84.3%	15.7%
unc	445.6 K	38.5%	38.5%	3.3%	13.3%	2.3%	1.5%	0.1%	17.2%	23.0%	64.3%	35.7%
ibi	499.3 K	67.5%	21.9%	15.3%	2.0%	1.0%	1.2%	0.0%	2.5%	10.6%	77.8%	22.2%

Table 7: Needed and Unneeded Retransmissions

in TCP connections that transmit 10 or more segments are OOS. Interpreting all these as retransmissions of *lost* segments would significantly overstate loss rates. Even considering only those OOS segments that are retransmissions and not network reorderings, the implied loss rate would be 0.73%. This still overstates the actual loss rate because the necessary retransmissions are only 0.42% of segments.

For modeling TCP throughput, the most significant factors are the RTO and duplicate-ACK triggered retransmissions that mark the beginning of loss episodes in response to which the congestion window is reduced; the partial-ACK, SACK block, and implicit retransmissions all occur during recovery phases following the beginning of such loss episodes. Our results show that 0.32% of segments transmitted result in an RTO timeout and 0.09% result in duplicate ACK retransmissions. From this perspective, our results indicate that roughly 80% of all loss episodes are detected using timeouts (as against FR/R). This finding has significant implications for TCP performance because TCP enters slow-start following a timeout. Further, TCP practically stalls segment transmission during the timeout interval because its window is not advancing.

Finally, we find that a significant fraction of unneeded retransmissions occur due to RTOs (2-13.3%) and FR/R (0.3-2.3%) phases. Each of these phases cause a reduction in TCP congestion window and stall the connection throughput during recovery.

References

[1] http://pma.nlanr.net/special/leip1.html.

[2] http://pma.nlanr.net/traces/long/ipls1.html.

[3] http://tracer.csl.sony.co.jp/mawi/.

[4] tcptrace. http://jarok.cs.ohiou.edu/software/tcptrace/tcptrace.html.

[5] M. Allman, V. Paxson, and W. Stevens. Tcp congestion control, 1999.

[6] Mark Allman, Wesley M. Eddy, and Shawn Ostermann. Estimating loss rates with tcp. *SIGMETRICS Perform. Eval. Rev.*, 31(3), 2003.

[7] Neal Cardwell, Stefan Savage, and Thomas Anderson. Modeling TCP latency. In *INFOCOM (3)*, pages 1742–1751, 2000.

[8] K. Fall and S. Floyd. Simulation-based comparisons of tahoe, reno, and sack tcp. *ACM Computer Communication Review*, 26(3), July 1996.

[9] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, and D. Towsley. Measurement and classification of out-of-sequence packets in a tier-1 ip backbone. In *IMW '02: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, 2002.

[10] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. Tcp selective acknowledgement options, 1996.

[11] J. Padhye and S. Floyd. On inferring tcp behavior. In *Proceedings of ACM SIGCOMM*, 2001.

[12] Jitendra Padhye, Victor Firoiu, Don Towsley, and Jim Kurose. Modeling tcp throughput: a simple model and its empirical validation. In *Proceedings of ACM SIGCOMM*, pages 303–314. ACM Press, 1998.

[13] S. Rewaskar, J. Kaur, and D. Smith. Passive inference of tcp losses using a state-machine based approach. *Technical Report, Department of Computer Science, University of North Carolina at Chapel Hill*, October 2005.

[14] W. Richard Stevens. *TCP/IP illustrated (vol. 1): the protocols*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.