# Empirical Analysis of TCP Losses and Its Detection/Recovery Mechanisms*

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

*Technical report No. TR05-017*

January 21, 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 this paper, we address the questions: *to what extent, and in what manner, do real-world TCP connections experience packet losses? And how effectively do TCP loss detection and recovery mechanisms deal with the packet losses?* We answer these questions by conducting an extensive passive analysis of TCP packet traces. Our analysis attempts to (i) characterize the TCP loss process, (ii) evaluate the accuracy and timeliness of loss detection in TCP, and (iii) evaluate the timeliness and efficacy in avoiding further losses of the loss recovery mechanisms in TCP. To facilitate this analysis, we first 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 packet losses. Using these state machines we analyze traces of more than 29 *million* TCP connections, collected from 6 different locations around the world. Our analysis sheds several insights on TCP loss detection and recovery. 40% of TCP loss detection is triggered due to timeouts. 90% of detection using duplicate acks take place within 2 RTTs. TCP experiences no more than 2 consecutive losses 91% of the time; however, 20-30% of loss-recovery periods last more than 2 RTTs. Our results, thus, confirm the efficiency of some TCP mechanisms and highlight the deficiency of others.

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

# 1 Introduction

TCP is the dominant transport protocol used by Internet applications. It provides several useful service abstractions, including that of a byte-stream transmission, flow control, congestion control, and reliable delivery. Of these, the service semantics of *reliable delivery* is perhaps one of the prime reasons for TCP's popularity. TCP implements reliability by using two kinds of mechanisms: one for *detecting* packet losses and one for *recovering* lost segments. It is well-known that the timeliness performance of a TCP connection degrades significantly whenever it experiences packet losses and invokes these two mechanisms. Given the popularity of TCP, it is, therefore, important to understand how often this set of two mechanisms is invoked in practice and how well it works. In particular, three key questions arise in this context. First, *in what manner are real-world TCP connections subject to packet losses?* Second, *with what accuracy does TCP detect packet losses?* And third, *how quickly does it detect and recover from losses?* It is our objective to help answer these questions by studying real-world TCP connections.

Our approach is to passively analyze traces of real-world TCP connections in order to infer the occurrence of three kinds of events: (i) real packet losses, (ii) invocation of a TCP loss-detection mechanism, and (iii) successful retransmission following each loss-detection phase. Following this, we analyze these events for several properties, including their occurrence and duration.

In the rest of this paper, we discuss our trace analysis methodology in Section 2. Our mechanism evaluation methodology and results are described in Sections 3

and 3.4, respectively. We summarize related work in Section 4 and conclude in Section 5.

# 2 Passive Loss Inference Methodology

It is well-known that packet losses can significantly impair the ability of a TCP connection to transfer data in a timely manner. TCP uses a combination of detection and recovery mechanisms to deal with packet losses. In this section, we first briefly summarize the specifics of these TCP mechanisms and then elaborate on our methodology for inferring loss characteristics from passively collected TCP traces.

## 2.1 TCP Loss Detection and Recovery

TCP identifies each application data byte transferred within a connection using a *sequence number*. The mechanism of *cumulative acknowledgments* is used to confirm delivery of bytes at the receiver—on receiving a fresh segment, the receiver sends back an acknowledgment to the sender with the largest sequence number, such that it has received correctly all bytes till that sequence number.

TCP variants differ in how they detect and recover from packet losses. Most of current operating system stacks implement a variant of the NewReno [9] version of TCP (and several stacks support the SACK [11, 8] feature as well) [12, 13]. Our discussion in this paper is, therefore, focused primarily on NewReno and SACK.

A NewReno sender detects and recovers lost packets in the following ways:

*Retransmission Timeouts (RTO):* The TCP sender uses a timer to estimate when segments can be assumed to be lost and should be retransmitted. This RTO timer is reset whenever ACK for new data is received (and in some cases, when new data is transmitted). If the RTO timer expires before the next ACK for new data arrives, the segment with the smallest unacknowledged byte is assumed to be lost and is retransmitted. Loss of the retransmissions is also detected using the timer—but the timer interval is exponentially increased for detecting loss of successive retransmissions of the segment.

*Fast Retransmit/Recovery (FR/R)*: When a TCP receiver receives out-of-order segments (those with sequence numbers higher than the next expected), it sends a *duplicate* ACK for the highest in-order segment received. When the sender receives three duplicate ACKs (TDA) contiguously, it infers a segment loss and retransmits the smallest unacknowledged segment.

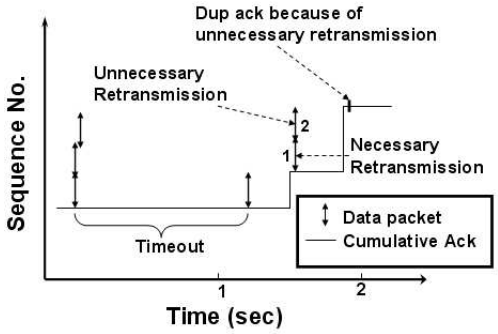


Figure 1: Implicit TCP Retransmission

The NewReno sender then enters the *fast recovery* phase, which ends only after *all* of the data sent before the first segment loss detection, is acknowledged. During fast recovery, if the sender receives a *partial* ACK (PA) for only some of this data, it infers that the next segment expected by the receiver has been lost and retransmits it. If the RTO timer expires on any segment, the sender exits FR/R and adopts the timer-based recovery mechanisms described before.

*Selective ACKs (SACKs)*: SACKs are used in loss recovery and help the sender determine which segments are lost in case of multiple losses. The TCP SACK option is used along with duplicate ACKs and contains up to 4 SACK blocks, which specify contiguous blocks of the most recently received out-of-order data. Because SACK is tied to TDA detections, it is used only in FR/R. The exact manner in which a sender responds to the receipt of SACK blocks has not been standardized.

## 2.2 Passive Inference of TCP Losses and Detection/Recovery Mechanisms

**Why not consider all retransmissions?** Our first objective is that of reliably inferring packet losses from passively-collected TCP traces. Since TCP *retransmits* segments on detecting packet losses, the simplest ap-

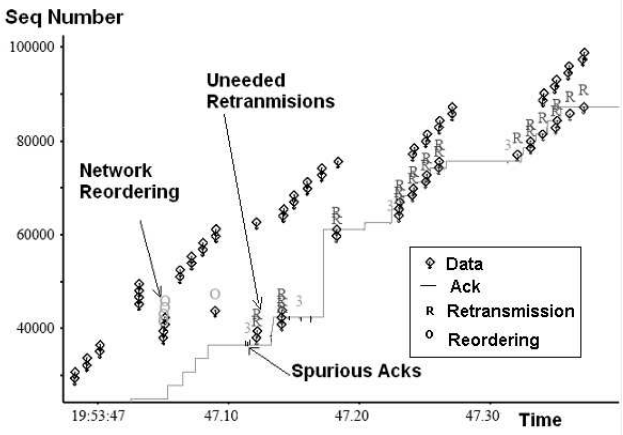


Figure 2: Unneeded Retranmission

proach is to simply look for the reappearance of some segments in the TCP link trace and assume that the original transmission was lost (and subsequently detected and retransmitted by TCP). However, this approach can lead to over-estimation of losses as illustrated in Figure 1, which depicts part of a TCP connection selected from the UNC-2004 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 and recovery mechanisms.*

**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 is not sufficient for reliably inferring packet losses. This is because of two main reasons related to TCP's inability to accurately infer packet losses:

*Some packet losses do not trigger TCP's loss detection/recovery phases.* For implementation efficiency, TCP senders maintain only a limited history/memory 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 envocation of the loss detection/recovery mechanisms described above. Figure 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.

*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, Figure 2, which again depicts part of a TCP connection selected from the UNC-2004 trace (and visualized using the *tcptrace* utility [5]), plots the example of a connection in which a *single* packet reordering event resulted in the triggering of 64 subsequent phases of fast retransmit/recovery, which lasted for more than 5 seconds!

**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 infer real packet losses with better accuracy than TCP. Using this basic approach, we can track three quantities related to TCP losses within each connection: **(i)** all retransmissions, **(ii)** actual packet losses, and **(iii)** segments that are explicitly retransmitted by TCP (henceforth, referred to as *explicit retransmissions*) as a result of triggering of its detection/recovery mechanisms. In Section 3.2, we compare the distribution of these three quantities across the connections we analyze.

## 2.3 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.

### 2.3.1 Diverse and Non-documented TCP Stacks

**The Challenge** TCP implementations written by different operating system (OS) programmers 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: 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.

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 [13]. Specifically, we have installed four different OSes—namely, Windows XP, Linux 2.4.2, FreeBSD 4.X, 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, timeouts, partial ACKs, etc). 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 [16]. 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 are not used in the reported results.

#### 2.3.2 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, 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 simple 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.

Note that the RTT measured at the monitor (monitor-receiver-monitor) is less than that measured at the sender (sender-receiver-sender). For passive loss analysis, 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—this is because 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.

#### 2.3.3 Non-availability of SACK Options

**The Challenge** A large number of traces do not capture the TCP option field. Sack blocks are transmitted in these 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.

**Our Approach** To overcome this problem, we develop heuristic to identify whether a packet could have been triggered by incoming sack information. Since sack information is used in FR/R we apply our heuristic only in this state.

In FR/R a retransmission can be triggered by a timeout, a partial ack, or a sack block. Using this information, we develop our heuristic as follows. We mark a packet retransmission as sack triggered if (i) the connection is in FR/R, (ii) the packet is not explained by either timeout or a partial ack, and (iii) the sequence number of the retransmitted packet is less than the highest sequence number that was in flight when the connection entered FR/R. We validated this heuristic using a one hour segment from the unc traces. We first ran the state machine with the sack blocks available and noted the retransmissions which were sack triggered. Then we removed the sack blocks from this trace and ran the state machine with the heuristic to identify sack triggered retransmissions. The state machine with the complete information about the sack blocks identified 54160 sack events. The state machine using the heuristics 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.

### 2.4 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 2.3.1.
2. We then replicate the loss detection/recovery related mechanisms 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 of the observed retransmissions. In case more than one machine matches this criteria, we select one randomly (both give same results).

We have implemented the above machines in the C programming language. Our implementations can run all four state machines on 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 [16].

Previously, Allman [6] and Jaiswal[10] have presented methods for passive estimations of losses. Allman's method of passively detecting TCP losses relies on duplicate acks transmitted in response to unnecessary retransmissions after a timeout. While this method works relatively well it does not identify exactly which of the retransmitted segments was unnecessarily retransmitted. Also for the traces we analyzed we found that the Allman method sometimes identifies only 25% of the unneeded retransmissions. Jaiswal et. al. used a state machine similar to our FreeBSD state machine for passive loss estimation. It does not take into consideration the variation that exits among the various implementation stack. They also do not try to explicitly identify the cause of a retransmission in the recovery region. We tested this state machine on the *ibi* trace and the *unc* trace. Jaiswal's state machine identified only 75% of the timeouts events for the ibiblio trace. This can be explained by the fact that the *ibi* trace comes from predominantly Linux machines and Linux uses a minimum timeout interval of $200ms$, which is not modeled by Jaiswal's state machine. For *unc*

trace which predominantly consists of Windows machine, Jaiswal's code identified only 60% of the retransmission due to duplicate acks. This happens because they use a fast retransmit threshold of 3 duplicate acks while the windows machines use a default threshold of 2 duplicate acks. Thus, we can see that by modeling the implementation details of different stacks we are able to identify more retransmission correctly.

# 3 Analysis of Lossy TCP Connections

The analysis methodology described in Section 2 gives us the ability to: (i) identify actual packet losses among the retransmissions within a connection, and (ii) identify the triggering of different loss detection/recovery mechanisms. We next apply our methodology to analyze TCP connection traces collected from 6 different global locations. In this section, we first describe these data sources and then present our characterization of the TCP loss detection/recovery phases observed in each trace.

## 3.1 Data Sources

Table 1 describes the traces used in our analysis. These traces are collected from links with transmission capacity ranging from 10 Mbps to OC-48. The *abi* traces [3] are collected from a backbone link of the Internet-2 network (Abilene); the *jap* trace [4] 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 hosted at UNC (ibiblio.org); the *bel* [2] trace collects traffic at the Internet link of a laboratory at Bell Labs. All traces except the one from the link to Japan were collected using Endace DAG cards. The Abilene, Leipzig, and Bell Labs traces are from the NLANR repository. Most traces, other than *bel*, represent a fairly diverse and large population. The *abi*, *lei*, and *bel* traces do not include SACK options.

For our analysis in the rest of this paper, we use only those connections that transmit at least 10 segments. Furthermore, since our objective is to study TCP losses, we select only those connections in which at least one segment retransmission is observed. Table 2 shows the impact of applying the latter filter. While less than 50% of connections that transmit at least 10 segments also retransmit some segments, these connections carry most of the bytes in this class. This is further confirmed by Figure 3, that plots the distribution of bytes transmitted within lossy TCP connections as well as connections with no losses, for the *abi*, *ibi*, and *jap* traces. We find that within each trace, the average number of bytes transmitted within lossy connections was significantly higher than among connections with no loss. Furthermore, the traces vary significantly in the distribution of bytes transmitted per connection—this adds to the diversity of our results.

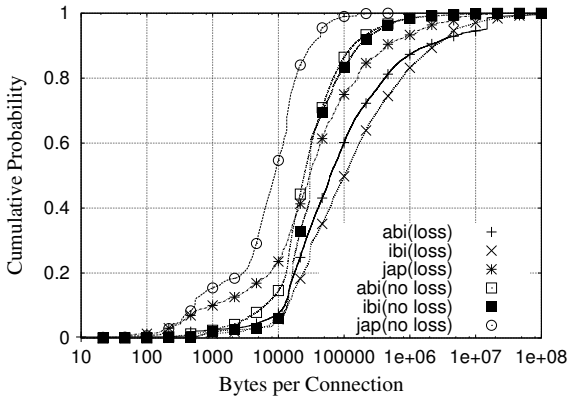


Figure 3: Distribution of Bytes-per-connection

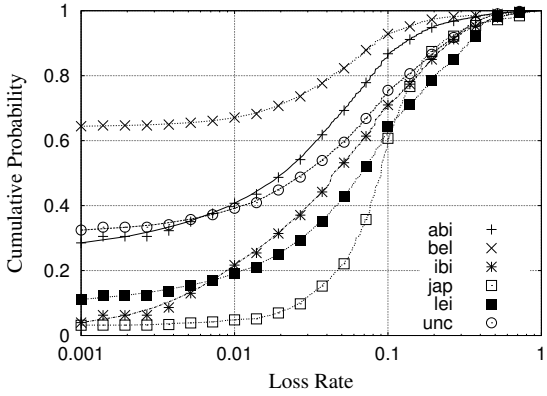


Figure 4: Loss Rate Distribution Among Lossy Connections

| Trace | Duration | Average 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 |
| BellLabs-10Mbps-2002 (bel) | 144h | 0.82 Mbps | 4.0 M | 53.0 G | 55.6 M |

Table 1: General Characteristics of Packet Traces

| Trace | Aggregate Loss Rate | All Connections | | | Lossy Connections | | |
|---|---|---|---|---|---|---|---|
| | | # Connections | # Bytes | # Packets | % Connections | % Bytes | % Packets |
| abi | 0.7 % | 388.9 K | 180.1 G | 148.5 M | 17.60 % | 68.11 % | 68.85 % |
| lei | 2.2 % | 75.4 K | 10.5 G | 12.6 M | 18.82 % | 74.88 % | 77.24 % |
| jap | 6.7 % | 18.5 K | 3.3 G | 3.1 M | 48.65 % | 96.08 % | 93.44 % |
| unc | 1.8 % | 774.8 K | 121.3 G | 129.6 M | 21.82 % | 78.45 % | 77.83 % |
| ibi | 1.2 % | 287.5 K | 161.8 G | 157.2 M | 27.31 % | 83.30 % | 82.37 % |
| bel | 0.7 % | 368.4 K | 50.2 G | 51.6 M | 42.09 % | 80.67 % | 83.58 % |

Table 2: Characteristics of Connections That Transmit More Than 10 Segments

Figure 4 plots the distribution of loss rates observed among the lossy connections (with losses estimated using the methodology described in Section 2). We find that connection loss rates observed in our traces vary from less than $0.1\%$ to more then $80\%$ with most in the range of 1-10%. Furthermore, the distribution of loss rates among lossy connections varies significantly across the traces. This illustrates the diversity of the data sets we are using for our analysis.

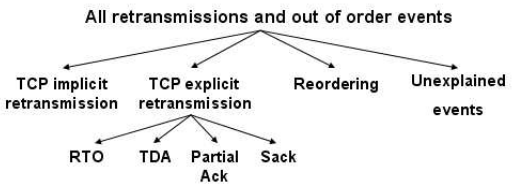


Figure 5: Taxonomy of Retranmissions

## 3.2 Accuracy of TCP Loss Detection

Note that our analysis methodology of Section 2 lets us classify a retransmission into one of four categories (also see Figure 5):

- *Explicit Retransmissions*: As described in Section 2.2, these are retransmissions that occur as a result of an explicit triggering of TCP's loss detection/recovery mechanisms. These mechanisms could be one of 4—RTO-based, TDA-based, PA-based, and SACK-based.
- *Implicit Retransmissions*: These are the extra segments that get retransmitted after a timeout, but not as a direct result of an explicit loss detection attempt by TCP (examples given in Section 2.2).
- *Network Reordering*: Retransmission of segments that are lost between the sender and the monitor appears mostly in the form of *out-of-order* (OOO) segments. These segments may be classified as either explicit or implicit retransmissions by our analysis programs. Some OOO segments, however, are not due to retransmissions, but simply packet reordering

| Trace | Total Retransmission | Actual % Loss | Explicit TCP retransmissions | | | | | Network Reordering |
|---|---|---|---|---|---|---|---|---|
| | | | % Timeout | % Dupack | % Partial Ack | % Sack | % Unexp | |
| abi | 1456.8.5 K | 67.9 % | 31.0 % | 16.7 % | 3.0 % | 0.0 % | 8.68 % | 14.9 % |
| lei | 338.9 K | 80.6 % | 41.0 % | 11.0 % | 5.2 % | 7.0 % | 0.2 % | 15.1 % |
| jap | 258.7 K | 79.3 % | 33.8 % | 19.15 % | 5.1 % | 6.5 % | 5.1 % | 6.4 % |
| unc | 2929.1 K | 78.1 % | 35.1 % | 13.6 % | 5.7 % | 6.7 % | 11.6 % | 6.6 % |
| ibi | 2502.0 K | 71.6 % | 31.0 % | 16.3 % | 5.9 % | 0.3 % | 0.5 % | 4.0 % |
| bel | 1935.1 K | 18.4 % | 14.0 % | 3.29 % | 0.8 % | 0.5 % | 43.3 % | 32.1 % |

Table 3: Classification of retransmissions seen

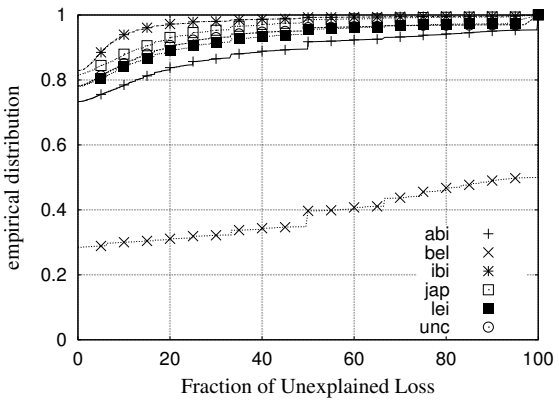


Figure 6: Fraction of Unexplained Events

on the path between the sender and the monitor.

- *Unexplained Retransmissions*: Finally, it is not feasible for us to incorporate all variants of sender-side TCP implementations in our analysis. Consequently, we may not be able to explain some retransmissions in some connections.

Tables 3 and 4 classify all retransmissions observed in lossy connections, into one of the above four categories. We observe that:

- Our state machines are able to classify 47-99.8% of all retransmissions that appear in the traces. We believe this high rate of success has been achieved due to the world-wide dominance of the Windows XP and Linux OSes, which are incorporated in our analysis.

  Figure 6 plots the distributions of the fractions of retransmission events within a lossy connection, that are unexplained by our analysis programs. We find that most of the unexplained events come from a small fraction of connections.

- Nearly 1-10% of TCP retransmissions are implicit, and occur without any explicitly loss detection/recovery attempt by TCP.

- Around 4-32% of OOO events occur as a result of network packet reordering between the sender and the monitor.

- TCP detects losses by RTOs 14-40% of the time, and by TDAs 4-19% of the time. The large use of expensive RTOs is indicative of the inability of TCP to use the faster FR/R under current traffic conditions. 1-6% and 0-7% of subsequent losses are detected in FR/R by PAs and SACKs, respectively. Both of these mechanisms help reduce the time spent by TCP in detecting and recovering from multiple losses.

We see a large number of unexplained results in the *bel* trace. We manually inspected these traces and found that a large number of connections do not originate from any of the TCP implementations that we have modeled.

**TCP's Loss Detection Accuracy** For any retransmission, our analysis also attempts to find out if the retransmission was unnecessary (i.e., the original transmission had successfully reached the receiver). We do this by observing if the segment gets acknowledged before the retransmission time plus the minimum RTT of

| Trace | Total Retran | Actual % Loss | % Unneeded | Implicit % Retran |
|---|---|---|---|---|
| abi | 1456.8.5 K | 67.9 % | 1.2 % | 9.8 % |
| lei | 338.9 K | 80.6 % | 1.9 % | 7.0 % |
| jap | 258.7 K | 79.3 % | 1.9 % | 2.4 % |
| unc | 2929.1 K | 78.1 % | 7.7 % | 9.9 % |
| ibi | 2502.0 K | 71.6 % | 0.6 % | 1.1 % |
| bel | 1935.1 K | 18.4 % | 0.04 % | 2.8 % |

Table 4: distribution of retransmissions

the connection—if so, the retransmission is unnecessary. We identify unnecessary retransmissions in a post-timeout phase by analyzing the pattern of ACKs generated in response to the retransmission. For example, the retransmission of segment 2 is unnecessary in Figure 1, as is indicated by the generation of duplicate ACKs for it.

Table 4 lists the fraction of retransmissions that are unnecessary in each of our traces. We find that this fraction can range from 0-8%.

The table also lists the fraction of actual packet losses that do not explicitly trigger TCP's loss detection mechanisms (and are recovered due to implicit retransmissions). We find that this fraction ranges from 1-10%, indicating that TCP is oblivious to this fraction of packet losses.

### 3.3 Characterization of TCP Loss Detection/Recovery

Our state-machine analysis is able to determine which of three high-level states a TCP connection is in during its entire lifetime: normal, detection, and recovery. We define the normal state for a TCP connection to begin with the transmission time of the first data segment of the connection or at the end of a recovery interval (defined below). The normal state ends and the detection state begins at the transmission time of the first segment that is ultimately lost. The detection state ends and the recovery state begins with the transmission time for the retransmission of the lost segment that began the detection state. The recovery state ends and the normal state begins with the receipt of an ACK that includes the highest sequence number in flight at the end of the detection period. Except for the case discussed next, a TCP connection that experiences losses cycles through the state sequence: (normal, detection, recovery). The common exception to this sequence is that a detection interval (state) can be logically embedded in the recovery state when multiple segments in flight are lost and the loss is detected, for example by a timeout. If a detection interval occurs during the recovery state, the termination condition for the recovery state is simply extended so that it ends with the receipt of an ACK for the highest sequence number in flight at the end of the embedded detection interval. We use the phrase "normal, detection, recovery interval" to mean the duration that the connection spends in the indicated state.

We can further refine the TCP detection and recovery states by noting how a loss was detected – timeout or 3 duplicate ACKs. Thus we can report results for 5 TCP states:normal, detection-RTO, detection-TDA, recovery-RTO, recovery-TDA. Note that for recovery intervals that have embedded detection intervals, the initial loss-detection condition is used to classify the TCP state, e.g., if a recovery state is initiated because of loss detection by 3 duplicate ACKs, we report the results for the entire recovery interval for recovery-TDA even though timeout loss detections may occur during the recovery interval (and, therefore, extend its duration).

In this section we provide a detailed examination of the TCP mechanisms by characterizing loss processes with respect to these states. Our motivation for doing this is to better understand the dynamics of TCP loss detection and recovery mechanisms and how they relate to the loss processes experienced by a TCP connection. By relating loss processes to TCP connection states we can examine such issues as:

- How is the number of segments in flight related to the

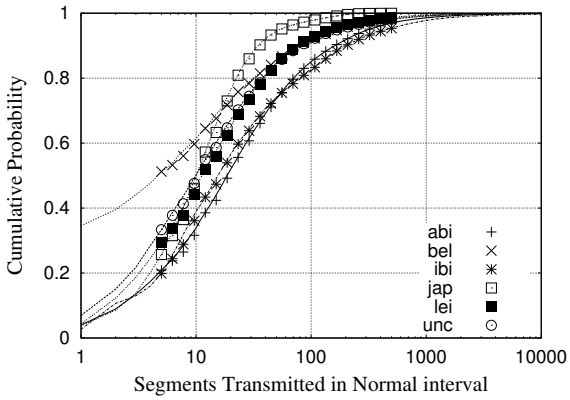


Figure 7: No.of segment in normal loss free runs

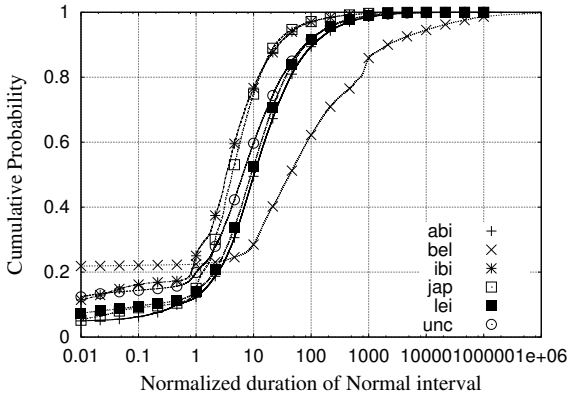


Figure 8: Duration in RTT of normal loss free run

number of lost segments from that flight?

- How does the number of segments lost at the beginning of (and during) a recovery interval relate to the duration of the recovery interval?

- How does the loss process during a recovery interval (both retransmitted segments and new segments sent in fast recovery) compare with the loss process elsewhere?

We believe the results reported in this section are the first detailed examination of losses experienced by TCP connections considering the current state of the TCP connection when loss occurs.

**Normal State** When TCP is operating in the normal state, we are primarily interested in the number of successful segment transmissions and the duration of the interval before a loss (recall that the transmission of a segment that is ultimately lost marks the end of a normal interval and the beginning of a detection interval). Figures 7-8 give the distributions of the number of segments transmitted in normal intervals and the normal interval duration normalized to the current estimated RTT. The RTT used for this normalization is computed as the current weighted average at the interval's termination using the TCP weights: RTT = (7/8)*RTT + (1/8)*sampleRTT where sampleRTT is measured for every segment yielding a valid sample value according to Karn's algorithm instead of once per window as is done in most TCP implementations. The median number of segments transmitted in normal intervals is in the 10-20 range across traces. For some traces only about 20% of normal intervals contain more than 20 segments while in others 20% of normal intervals contain 100 or more segments. About 15% of normal interval last less than 1 RTT, the median duration is 5-10 RTTs, and only around 10% last longer than 100 RTTs. If most of these intervals represent times when TCP is in congestion avoidance, the congestion window would have limited opportunity to increase. Note that these results are somewhat different from those in Figures 25-26 discussed in section 3.4 that show similar distributions for loss-free runs, including those that occur during detection or recovery intervals.

**Detection State** When TCP is operating in the detection state, we are primarily interested in (a) the number of segments in flight at the time the first loss is detected, (b) the number of those in-flight segments that are also lost (the number of outstanding lost segments at the beginning of the recovery period), and (c) the duration, normalized to estimated RTT, of the detection interval (the time between the transmission of the lost segment and the time its loss is detected). The latter is an indication of how effective real TCP connections are at detecting loss so they can initiate recovery actions. The distributions of these metrics are given in Figures 9-12 for detection-RTO states and in Figures 13-16 for detection-TDA states.

In the vast majority of RTO cases (90%) there are five or fewer segments in flight when a loss is detected. Combined with our finding that a large percentage of loss detections are by RTO this indicates that TCP windows are

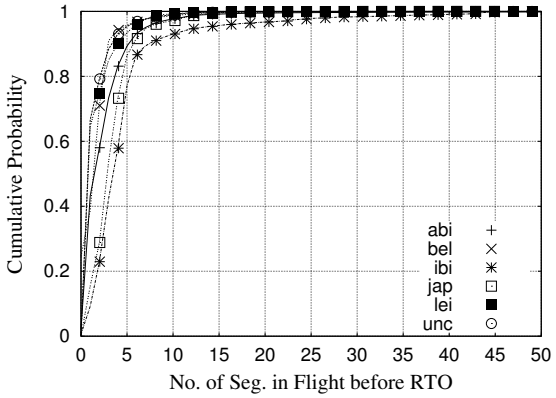


Figure 9: No. of packets in flight at detection-TO

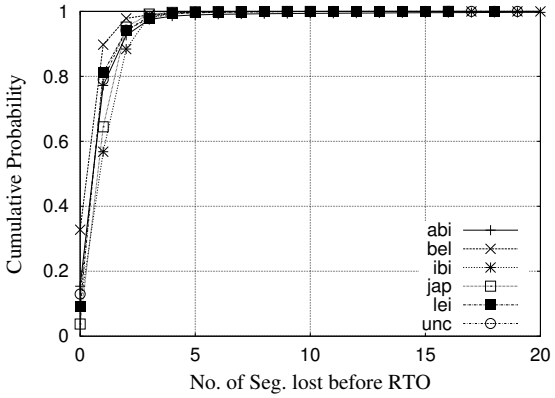


Figure 11: No. of Segments lost at detection-TO

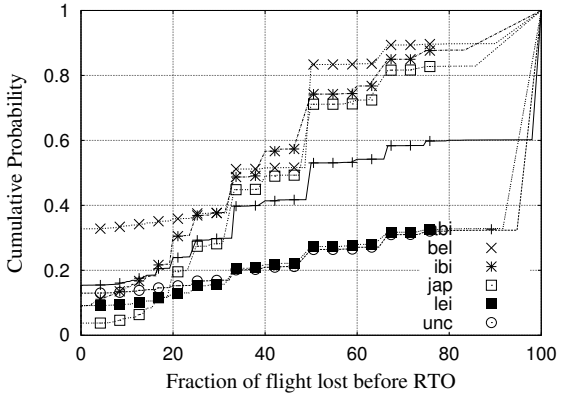


Figure 10: Fraction of packets in flight that where lost at detection-TO

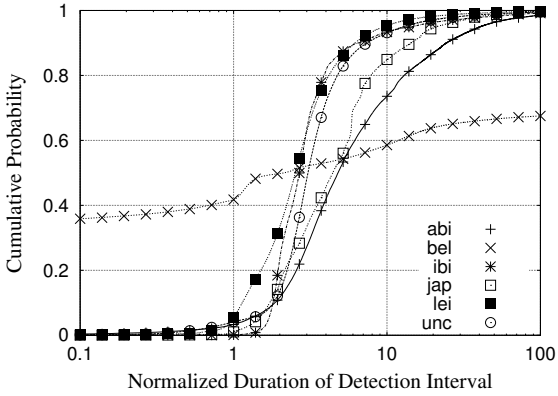


Figure 12: Duration in RTT for detection-TO intervals

often too small to allow for sending enough segments to trigger detection of losses by three duplicate ACKs. As expected from the small flight sizes, the number of segments lost in the flight is typically only 1 in about 70% of detection-TO intervals and 2 in about 10%. Note that in these traces we observe zero real losses in 5-30% of TO intervals. These represent cases in which the ensuing retransmission was unnecessary, primarily because one or more ACKs were lost. We found that a non-trivial fraction (about 20%) of timeout detections occurred in less than 2 RTTs. As expected, the majority (roughly 60%) of timeout detections happened in 3-5 RTTs while a substantial number required more than 10 RTTs. These longer intervals probably reflect the exponential growth in RTO caused by losses of retransmissions. Analysis for the *bel* show that for 50% of the connections, the minimum timeout is itself greater than 100 RTTs. This result in the very long detection durations for these connections.

As expected, we find almost double the segments in flight when loss is detected by three duplicate ACKs when compared with RTO detection. However, most (70%) of the flight sizes fall in a relatively narrow range between 4 and 10 segments. Because these flights are larger the number of lost in-flight segments also increases. In 50% of flights more than 25% of in-flight segments have been lost when a loss has been detected by three duplicate ACKs. From a timeliness standpoint, loss detection by three duplicate ACKs is clearly effective. About 90% of detection intervals require 2 or fewer RTTs. Unfortunately this advantage is offset by our finding that RTO detection is often required, probably because of small window sizes.

**Recovery State** When TCP is operating in the recovery state, we are primarily interested in (a) the number and fraction of additional segments lost during the recovery interval, considering both retransmitted segments and new segments sent in fast recovery, and (b) the duration normalized to estimated RTT of the recovery interval (the time between the detection of the initial lost segment and the full recovery of all outstanding lost segments including those that may be lost during the recovery interval). The distributions of these metrics are given in Figures 17-19 for recovery-TO states and in Figures 20-22 for recovery-TDA states.

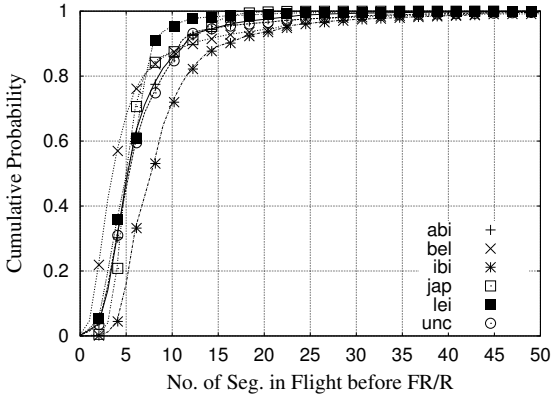


Figure 13: No. of packets in flight at detection-TDA

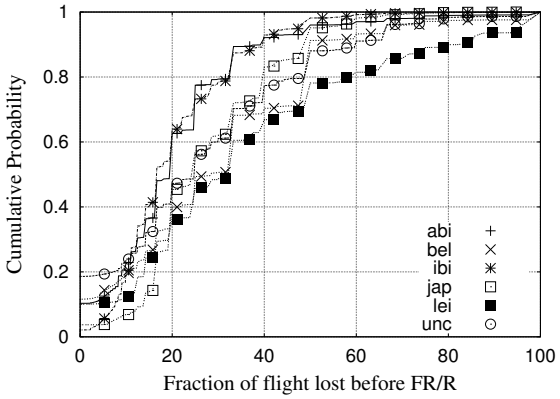


Figure 14: Fraction of packet in flight that were lost at detection-TDA

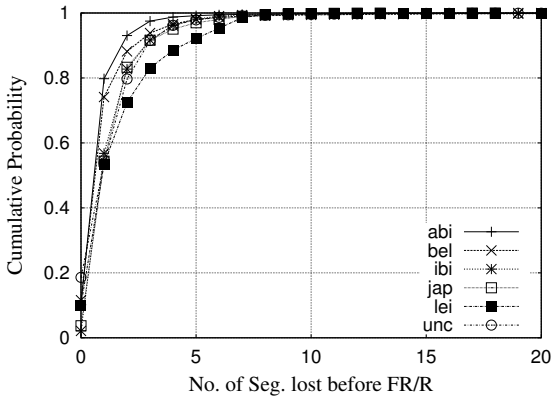


Figure 15: No. of segments lost during detection-TDA

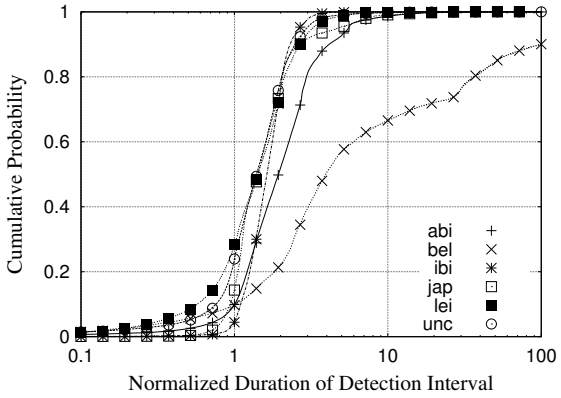


Figure 16: Time in RTT for detection-TDA interval

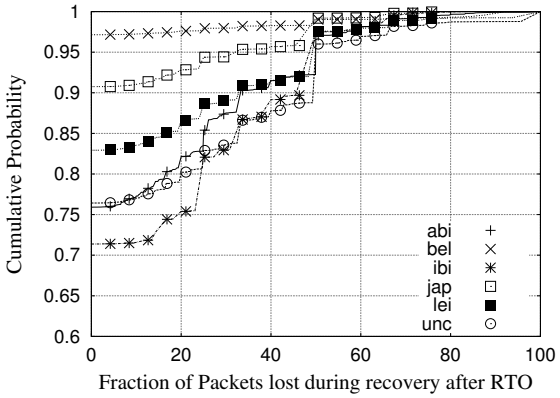


Figure 17: Fraction of packet lost during a recovery-TO interval

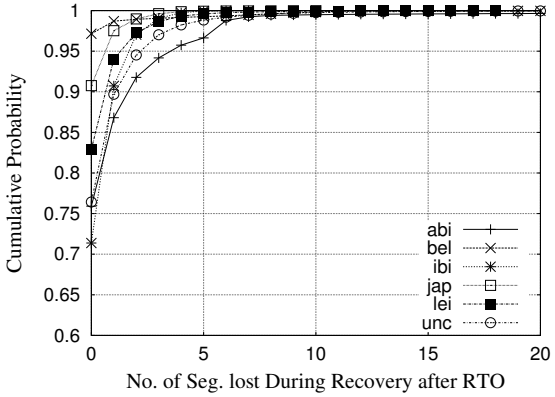


Figure 18: No. of segments lost during a recovery-TO interval

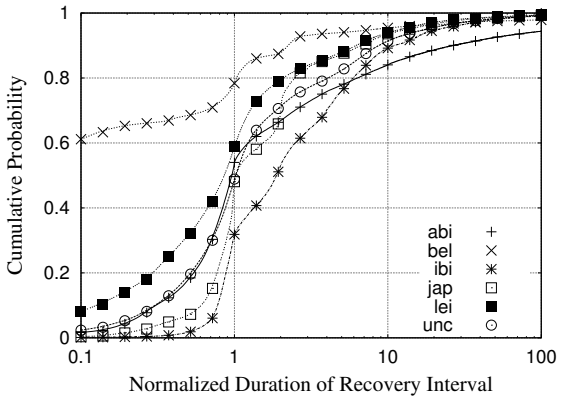


Figure 19: Time in RTT of the duration of recovery-TO interval

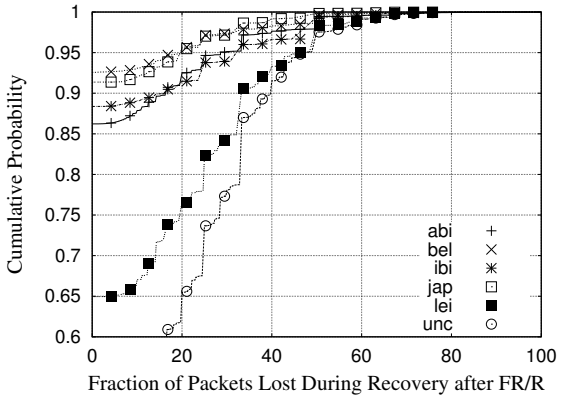


Figure 20: Fraction of packet lost during a recovery-TDA interval

It is relatively rare for additional segments to be lost during recovery after an RTO. In 75-95% of such recovery intervals zero additional segments are also lost. When losses do occur, in only a small percentage of intervals are more than 2 or 3 additional segments lost. About half of the RTO recovery intervals take around 1 RTT but a significant fraction (20%) take more than 2-3 RTT. In four of the traces, we found no significant differences in these metrics between recovery after RTO and recovery after 3 duplicate ACKs (TCP's fast recovery algorithm). In two of the traces (Leipzig and UNC) there were noticeably more segments lost in fast recovery.

Figure 24 shows the distribution of new segments transmitted during recovery intervals initiated by RTO. Figure 23 shows the distribution of new segments transmitted during recovery intervals initiated by 3 duplicate ACKs (TCP's fast recovery algorithm). Comparing these two figures gives a indication that increasing the congestion window for additional duplicate ACKS during fast recovery is effective in keeping more segments in flight (between 10 and 35% of recovery intervals transmit 2 or more new segments with fast recovery compared with 5% of interval that transmit 2 or more new segments during recovery after RTO). Combined with the finding that very few transmissions during recovery are themselves lost, this indicates that fast recovery can achieve the desired effect of helping keep new segments entering the network as old segments depart.

## 3.4 General Loss characteristics

We next report the results from analyzing segment losses in TCP connections without considering the state of the connection with respect to loss detection and recovery. One of our objectives in reporting the results in this section is to see if measurements taken from more recent traces and analyzed with a more comprehensive state machine approach confirm prior reported results. In considering the results we report in this section it is important to avoid interpreting them as indicative of the inherent packet-loss processes along the paths traversed by the connections. Because TCP fundamentally operates in a "closed loop" process where losses cause TCP to adapt its sending rate downward, the losses observed by the connection are likely to be reduced as a result. Further, other TCP connections that share congested path segments with

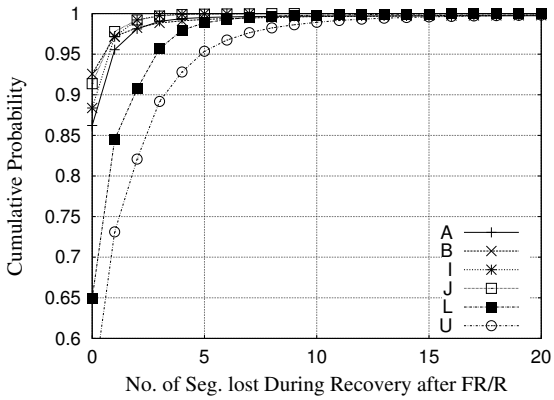


Figure 21: No. of segments lost during a recovery-TDA interval

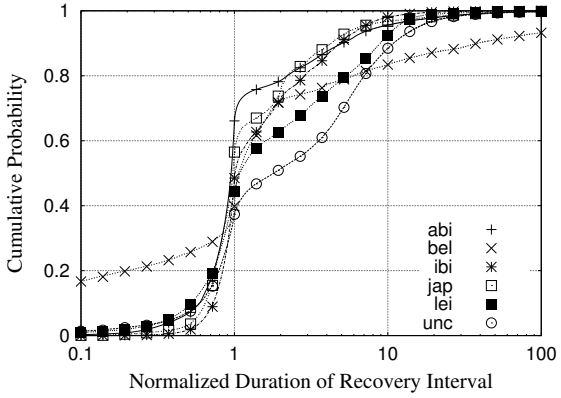


Figure 22: Time in RTT of the duration of a recovery-TDA interval

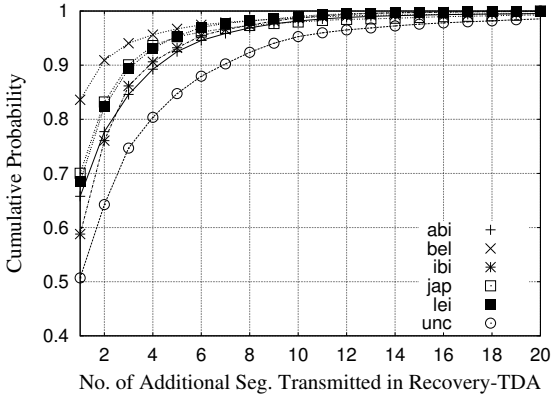


Figure 23: Number of Segments transmitted in recovery-TDA

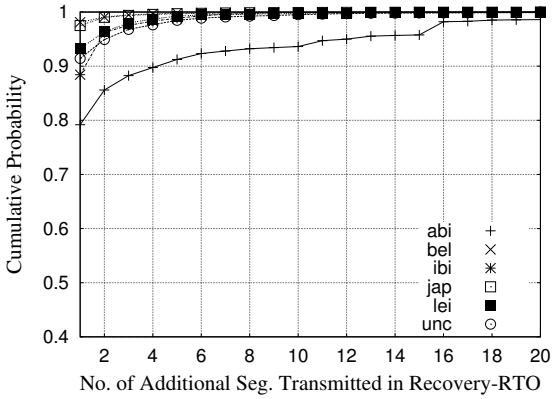


Figure 24: Number of Segments transmitted in recovery-RTO

the TCP connection being analyzed are likely to observe losses also and adapt their sending rates to achieve fewer losses. As discussed in [14] this points out the problem with trying to infer Internet loss rates using data collected with a mechanism (TCP) whose goal is to reduce what we are trying to measure. To determine overall Internet packet loss rates, sampling approaches that don't use TCP to send packets are preferred (see for example [18]). Our results should be interpreted as characterizing the loss processes actually experienced by TCP connections as influenced by both inherent network conditions and the TCP adaptive mechanisms.

In our analysis of loss we follow the notations used in [18]. We encode the sequence of segment transmissions within each TCP connection as a binary time series (with necessarily unequal time spacing) where a successfully received segment is represented by 0 and a lost segment is represented by 1. We then apply simple counting functions to the time series for each connection to count the number of segments represented by sequences of consecutive 0 values ("loss-free runs") and consecutive 1 values ("loss runs"). A run, either loss-free or of losses, at the end of the connection is not included to avoid a bias toward underestimating run lengths. For each TCP connection, we also record a second time series consisting of the time at which the segment was transmitted (for both successful and lost transmissions). Using these time stamps along with the counting functions on the binary time series we can determine the duration of loss-free and loss runs. As we did in reporting results in earlier sections, we consider only those connections that transmit 10 or more segments and that have 1 or more of those segments that are retransmitted.

Figure 25 shows the distribution of the number of segments in loss-free runs over all the connections analyzed. The median loss-free run is approximately 10 segments and loss-free runs of more than 100 segments are rare. These results indicate significantly longer loss-free runs than reported by Allman, et al [6]. For example, we find that 40-50% of loss-free runs are more than 10 segments while they found about 15% longer than 10 segments. The distribution of the durations of loss-free runs is shown in Figure 26 for durations normalized to the current estimated RTT of the connection. Note that as in previous sections, the RTT used for this normalization is computed as the current weighted average at the run's termination

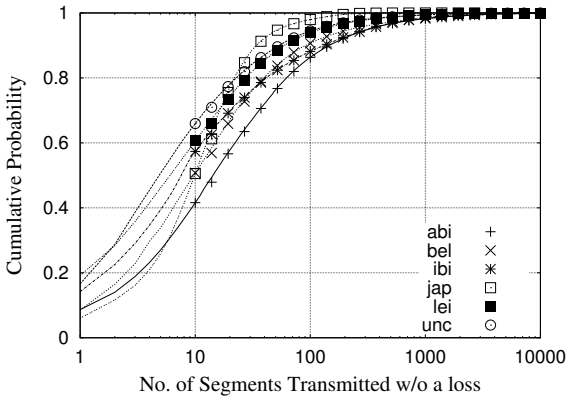


Figure 25: No. of segment in loss free runs

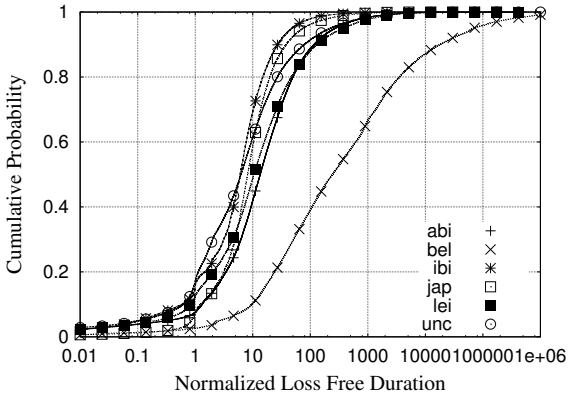


Figure 26: Duration in RTT of loss free runs

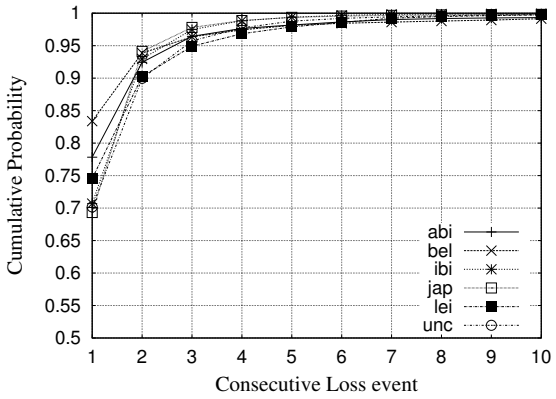


Figure 27: No. of consecutive segments lost

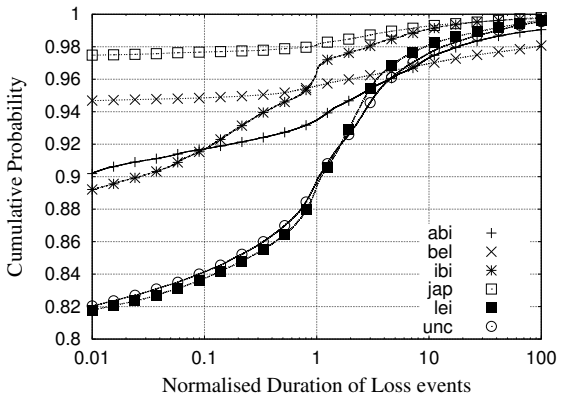


Figure 28: Duration in RTT of loss run

using the TCP weights: RTT = (7/8)*RTT + (1/8)*sampleRTT where sampleRTT is measured for every segment yielding a valid sample value according to Karn's algorithm instead of once per window as is done in most TCP implementations. The median loss-free duration is about 10 RTT while those lasting more than 100 RTT are infrequent.

Figure 27 shows the distribution of the number of segments in loss runs over all the connections analyzed. Around 75% of loss runs are a single segment and approximately 90% are at most 2 segments. These results are quite similar to those reported in [6] and [17]. The distribution of the durations of loss runs is shown in Figure 28 for durations normalized to the current estimated

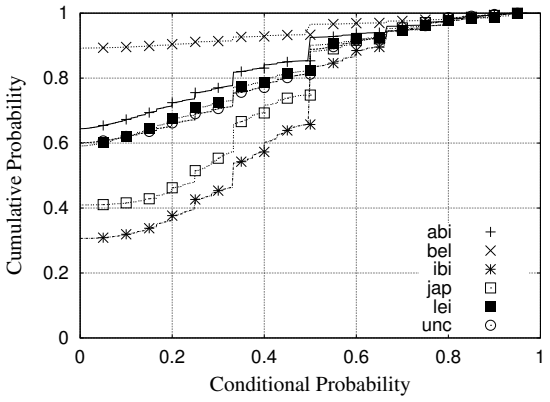


Figure 29: Conditional probability of loss

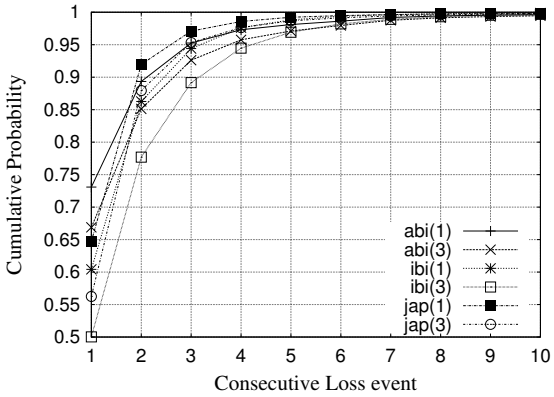


Figure 30: No. of continuous losses when one or up to 3 successful retransmission are neglected

RTT of the connection. Since loss runs are at most a few segments, most durations last only a fraction of an RTT. We note however, that approximately 10% of loss runs occur over durations longer than 1 RTT with some lasting more than 100 RTTs. The most likely explanation for this observation is that several consecutive retransmission are lost, each being sent after longer intervals as the RTO timer backs off exponentially. Another issue to consider is to what extent do the losses observed by TCP often appear as "bursts" of consecutive (correlated in time) losses as analyzed in [15]. One direct indicator that losses are not independent can be obtained by comparing the overall unconditional loss probability with the conditional loss probability conditioned on the event that the previous packet was lost [15]. For each TCP connection we computed its unconditional loss rate and its overall conditional loss probability considering all the lost segments. Figure 29 shows the distribution of conditional loss probabilities per connection for all the traces (refer to Figure 4 for the unconditional loss rates). The conditional loss probabilities per connection vary substantially from trace to trace indicating that network conditions leading to correlated losses (e.g., persistent full queues) may be quite different in parts of the Internet. Interestingly, a substantial number of connections had a conditional loss probability of zero (ranging from 30% of connections in one trace to 90% in another with most in the 40%-60% range). The number of connections with conditional loss probability of 0.50 or greater ranges from 40% to 10%. Overall, this seems to indicate that in many connections all losses experienced are independent events but in others losses are strongly correlated. This could imply that a single loss model (independent vs correlated) is not appropriate for all TCP connections.

For the results shown in Figures 27-28, loss runs are considered to be terminated by a single successful segment transmission. Because one successful transmission may occur even during a loss episode in which almost all segments are lost, treating a loss run as ending at a single successful transmission may tend to underestimate the impact of bursts of losses. We examined this issue by relaxing the termination condition for counting the number of segment in a loss run. We tried requiring both 2 and 4 successful transmissions to terminate the count of segments in a loss run. The distribution of the number of segments in loss runs over all the connections in three of the traces is shown in Figure 30 using these new termination conditions (results for the other three traces are similar). While the change tends to produce somewhat more longer runs, we conclude that the differences are not substantial.

# 4 Related Work

There is a considerable body of work on passive analysis of TCP connections and its loss characteristics. Tcptrace [5] is one of the many tools available for passive analysis. However, these tools do not maintain enough state to accurately infer the loss characteristics TCP. In [10], the authors has proposed a state machine based approach for analysis of TCP connections. They find that 9-19% of retransmission seen in the backbone where unneeded while 7-25% of out of sequence events were because of packet reordering. In [6], the authors have presented a method to passively estimate unneeded retransmission occurring after a timeout. They found that on an average 33% retransmission in Reno implementation and 2% of retransmission for Sack TCP implementation were unneeded. 60% of loss events were identified as singleton events.

There has also been considerable work on loss characteristics of a path using active measurements. [7] and [14] study the correlation between packet losses. Both of them found that if an earlier packet is lost the next packet is much more likely to be lost. [15] and [18] study the loss periods using active measurements. 95% of the loss periods seen by [18] were smaller than $220ms$.In [15], the author observed that the outage duration spanned an order of magnitude. While 10% of the outages lasted less than 33ms there was 10% of outages which lasted more than 3.2 seconds.

Finally, we look at some work on identifying TCP behavior. In [13], the authors have propsed an active methods of estimating the TCP protocols used and their implementation details.Alberto [12] used the tbit tool and identified the behavior of a large number of servers over a difference of 3 years. They found that currently almost 68% of servers use sack information. This highlights the importance of using sack information for TCP analysis.

# 5 Concluding Remarks

We have developed a state machine based approach to passively identify loss events for TCP connections. To account for the diverse and non-documented TCP stacks, we have developed several versions of our state machines for many important variants. We demonstrate the effectiveness of our method in classifying retransmissions and associating the losses with TCP's detection and recovery mechanism.

Using the state machines, we conducted a large scale study of diverse set of traces. Few of the interesting observations we made were as follows. 40% of loss detection occurs due to timeouts. This happens mainly because of TCP's inability to reach large window sizes in the presence of losses. Average run of normal phase is only 5-10 RTTs. 5-30% of detection due to timeout occurs because the ack is lost and not the data packet. As expected, 30% of timeouts occur in 3-5 RTTs while 90% of detection due to duplicate acks take place in less than 2 RTTs. TCP experiences no more than 2 consecutive losses 91% of the time; however, 20-30% of loss-recovery takes more than 2 RTTs. Finally, we find that while there are a few connections for which the loss events are independent, there also exist connections where the loss events are highly correlated. This has significant implication from modeling perspective because an assumption about loss independence cannot be made in either direction.

In our opinion, this is the first attempt at studying the loss events as they occur in the TCP detection and recovery interval. The implementation sensitive state machine approach can also be used to address a wide range of research questions like the growth of congestion windows in a connection, validation of assumption made in models, etc. Currently we are working on improving the performance of the state machine in terms of identifying more TCP variants.

# References


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

[2] http://pma.nlanr.net/Traces/long/bell1.html.

[3] http://pma.nlanr.net/Traces/long/ipls1.html.

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

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

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

[7] J. Bolot. End-to-end packet delay and loss behavior in the internet. In *SIGCOMM '93: Conference proceedings on Communications architectures, protocols and applications*, 1993.

[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. Floyd, T. Henderson, and A. Gurtov. The NewReno Modification to TCP's Fast Recovery Algorithm, 2004.

[10] 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.

[11] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgement Options, 1996.

[12] A. Medina, M. Allman, , and S. Floyd. Measuring the Evolution of Transport Protocols in. In *ACM Computer Communications Review*, April 2005.

[13] J. Padhye and S. Floyd. On Inferring TCP Behavior. In *Proceedings of ACM SIGCOMM*, 2001.

[14] V. Paxson. End-to-End Internet Packet Dynamics. In *Proceedings of ACM SIGCOMM*, September 1997.

[15] V. Paxson. *Measurements and Analysis of End-to-End Internet Dynamics*. PhD dissertation, University of California, April 1997.

[16] S. Rewaskar, J. Kaur, and D. Smith. Using TCP State Machines for Passive Trace Analysis. *Under submission,* (also available as *Technical Report, Department of Computer Science, University of North Carolina at Chapel Hill*), February 2005.

[17] M. Yajnik, S. Moon, J. Kurose, and D. Towsley. Measurement and Modeling of the Temporal Dependence in Packet Loss. In *INFOCOM*, 1999.

[18] Y. Zhang and N. Duffield. On the constancy of internet path properties. In *IMW '01: Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, 2001.