Real World Evaluation of Techniques for Mitigating the Impact of Packet Losses on TCP Performance

Sushant Rewaskar

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2008

Approved by:

Jasleen Kaur, Advisor
Kevin Jeffay, Reader
Ketan Mayer-Patel, Reader
F. Donelson Smith, Reader
Don Towsley, Reader

ABSTRACT

SUSHANT REWASKAR: Real World Evaluation of Techniques for Mitigating the Impact of Packet Losses on TCP Performance.
(Under the direction of Jasleen Kaur)

The real-world impact of network losses on the performance of Transmission Control Protocol (TCP), the dominant transport protocol used for Internet data transfer, is not well understood. A detailed understanding of this impact and the efficiency of TCP in dealing with losses would prove useful for optimizing TCP design. Past work in this area is limited in its accuracy, depth of analysis, and scale. In this dissertation, we make three main contributions to address these issues: (i) design a methodology for in-depth and accurate passive analysis of TCP traces, (ii) systematically evaluate the impact of design parameters associated with TCP loss detection/recovery mechanisms on its performance, and (iii) systematically evaluate the ability of Delay Based Congestion Estimators (DBCEs) to predict losses and help avoid them.

We develop a passive analysis tool, *TCPdebug*, that accurately tracks TCP sender state for many prominent OSes (Windows, Linux, Solaris, and FreeBSD/MacOS) and accurately classifies segments that appear out-of-sequence in a TCP trace. This tool has been extensively validated using controlled lab experiments as well as against real Internet connections. Its accuracy exceeds 99%, which is double the accuracy of current loss classification tools.

Using *TCPdebug*, we analyze traces of more than 2.8 million Internet connections to study the efficiency of current TCP loss detection/recovery mechanisms. Using models to capture the impact of configuration of these mechanisms on the durations of TCP connections, we find that the recommended as well as widely implemented configurations for these mechanisms are fairly sub-optimal. Our analysis suggests that the durations of up to 40% of Internet connections can be reduced by more than 10% by reconfiguring prominent TCP stacks.

Finally, we investigate the ability of several popular Delay Based Connection Estimators (DBCEs) to predict (and help avoid) losses using estimates of network queuing delay. We find that aggressive predictors work much better than conservative predictors. We also study the impact of connection characteristics—such as packet loss rate, flight size, and throughput—on the performance of a DBCE.

We find that high-throughput connections benefit the most from any DBCE. This indicates that DBCEs hold significant promise for future high-speed networks.

ACKNOWLEDGMENTS

First, I want to thank my advisor, Jasleen Kaur, for her constant support and guidance throughout my stay at UNC. Her immence patience with me and her attention to details taught me a lot about conducting research. I would also like to specially thank Don Smith for spending extraordinary amount of time with me dicussing my research and quiding me. Both of them made working of my dissertation a very rewarding and fun experience for me. I would like to thank Kevin jeffay, who has always been available to to provide input and insight in my work, as well as Ketan Mayer-Patel for his interesting and sometimes revolutionary ideas. I would also like to talk Don Towsley for taking time from this busy schedule to serve on my dissertation committee and providing good feedback which definetly improved the quality of my dissertation.

My friends, both from UNC and outside, always stood by me and helped me in every way possible. I am truely grateful for their quidance but more so for their support. During my stay at UNC, I had a lot of fun working on several class projects or discussing my random thoughts with the faculty here. I would like to thank the faculty and the Department of Computer Science in general for this valuable experience.

I cannot even think of the right words to thank my family. Without their love and support, I could not have been able to persue my dreams and get my PhD. My mother and father have always provided me with everything I ever needed even before I knew I needed it. My sister has been my role model for my whole life and it was her my desire to match her academic achievement which got me first interested in my studies to begin with. Finally, I would like to say to my wife, who always gave me encouragement, support and love — I could not have done this without you. Thanks a lot.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ACK	Positive acknowledgment TCP segment
BFA	Buffer Fill Avoidance
CARD	Congestion Avoidance using Round-trip Delay
CIM	Congestion Indication Metric
CDF	Cumulative Distribution Function
DAG	Data Acquisition and Generation
DAIMD	Delay-based Additive Increase Multiplicative Decrease
DBCA	Delay Based Congestion Avoidance
DBCC	Delay Based Congestion Control
DBCE	Delay Based Congestion Estimators
DECA	Delay-based End-to-end Congestion Avoidance
FAST	FAST AQM Scalable TCP
FIN	TCP control flag indicating "no more data from sender"
FR/R	Fast Retransmit and Recovery
FSM	Finite State Machine
GB	Gigabyte
IP	Internet Protocol
ISP	Internet Service Provider
KB	Kilobyte
kbps	Kilobit per second
MB	Megabyte
MSS	Maximum Segment Size
Mbps	Megabit per second
OOO	Out of Order
OS	Operating System
PA	Partial Acknowledgment
Pkts	Packets
RFC	Request For Comments
RST	TCP control flag indicating "connection reset"
RTO	Retransmission TimeOut
RTT	Round-Trip Time

SACK	Selective ACKnowledgment
SYN	Synchronize TCP control segment
SYN-ACK	Positive acknowledgement of SYN segment
TCP	Transport Control Protocol
Tri-S	Slow Start and Search
UDP	User Datagram Protocol
UNC	University of North Carolina at Chapel Hill

CHAPTER 1

Introduction

The important thing in science is not so much to obtain new facts as to discover new ways of thinking about them.

— Sir William Bragg (1862–1942)

It requires a very unusual mind to undertake the analysis of the obvious.

— Alfred North Whitehead (1861–1947)

1.1 Thesis Goals

The end-to-end principle of network design argues for the implementation of most features required only by an end-to-end protocol at end-hosts and not in routers [SRC84]. Based on this argument, the Internet was designed with a network core primarily concerned with a best-effort transfer of packets between end-systems. Any application that requires additional features such as reliable delivery of packets needs to rely on an end-to-end protocol. Such a protocol has to deal with the unreliable nature of the underlying Internet infrastructure by detecting packet losses and recovering from these. Transport Control Protocol (TCP) is the most popular of such protocols. It accounts for more than 80% of the traffic on the Internet [tcpb, TMW97, CMT98]. Given its wide-spread usage, TCP performance arguably governs the performance of Internet transfers.

It is generally accepted that network packet losses can adversely affect the performance of a TCP connection. Packet losses influence two important aspects of TCP - reliable delivery and congestion control. To understand how, lets consider how TCP implements these. A TCP sender assigns a sequence number to every packet it sends to a receiver and expects a positive acknowledgment (ack) for it. In the absence of an ack from the receiver, the sender assumes that the packet is lost and retransmits it — this is termed as loss recovery. TCP is a sliding-window based protocol in which the size of

the window determines the maximum number of outstanding unacknowledged packets in the network. While TCP is waiting for an ack for a lost packet, it is unable to send out new data (because of the window limit), thereby *stalling* the data transfer. Also, when TCP detects a loss, it assumes that the loss is due to congestion on the network; in order to reduce the load on the network, it reduces its window size and further witnesses degradation in its throughput. A degradation in throughput can also be characterized as an *increase* in the time taken to transfer all data, referred to as the *response time*. The above description argues that packet losses have the potential to significantly increase response time.

While the above argument is well established and it is generally believed that losses do impact TCP performance, the extent to which they do so in the real world is not well understood. The impact of losses on TCP's performance can be understood by studying two different interactions between TCP and losses – (i) TCP's efficiency in detecting packet losses when they occur and (ii) TCP's efficiency in predicting and hence avoiding losses when possible. We look at each of these below.

- *Ability of TCP to detect losses:* Two performance-related goals guide the design of TCP's detection mechanisms. First, TCP should *accurately* identify segment losses. In particular, if TCP erroneously infers that a segment was lost, it would unnecessarily invoke loss recovery which increases the connection duration. Second, TCP should *quickly* identify segment losses. A longer detection period would result in longer stalling periods for the connection while it waits for an ack. This would adversely impact the connection duration as well. Unfortunately, these two goals conflict with each other. A "quick" inference of segment loss would also be erroneous when segments (or their acks) are not lost but merely delayed or reordered in the network. To achieve high loss-estimation accuracy, therefore, TCP would necessarily have to wait longer for acks that may merely be delayed.

  This fundamental tradeoff between accuracy and timeliness is controlled by several design parameters associated with TCP's loss detection mechanisms. While the proposed standards for TCP recommend values for each of these parameters, TCP implementations in prominent Operating Systems (OS) stacks differ (sometimes significantly) in the values used. This naturally raises two questions regarding the state of loss detection in real world TCP connections: (i) how does the configuration of TCP loss detection impact the performance of Internet TCP connections, and (ii) Are the recommended values for these parameters optimal in reducing the connection duration?

  In this thesis, we address these questions. Specifically, our goal is to:

  - *Study the accuracy and timeliness of TCP loss detection/recovery in real world TCP connections originating from prominent sender-side OSes, and*
  - *Study the impact of changing parameter configurations on the overall durations of real world connections in order to identify the best parameter settings.*

- *Ability of TCP to predict losses:* In order to avoid the heavy penalty associated with packet losses, several studies have looked at alternate network signals, such as packet delays, to detect congestion [Jai89, WC91, WC92, BOP94, AR98, MNR03, WJLH06, YQC04, LSM+07]. These schemes rely on Delay Based Congestion Estimators (DBCEs) that assume that during periods of congestion, the packets of a connection would experience higher queuing delays than normal at the congested link – this should translate to an increase in packet round-trip times (RTTs). So by sampling per-packet RTTs, and comparing them to a base RTT (measured in the absence of congestion), a DBCE infers the onset as well as alleviation of congestion. More relevantly, a DBCE expects to avoid most packet losses by doing so. The success of the DBCEs, however, depends on the basic premise that the DBCEs can accurately predict the occurrence of losses. Unfortunately, the validity or invalidity of this premise has not been demonstrated convincingly. Specifically, most DBCE evaluation assume that delay is "always" an indicator of congestion. However, this assumption may be too strong given that the end-to-end delay signal may be too noisy due to several network and connection characteristics such as queuing at multiple routers, ack compression [ZSC91], and insufficient sampling of network delay information. The extent of the reliability of the delay signal and its relationship with network and connection characteristics in the real Internet needs to be understood to correctly interpret this signal.

  Our objective is to

  - *Systematically evaluate the ability of different Delay Based Congestion Estimators (DBCEs) to accurately predict losses.*
  - *Study the impact of connection characteristics on the performance of DBCEs*

Note that our prime objective is to address the issues raised above in the context of "real-world" TCP transfers. One promising approach for doing this is to passively analyze traces of real world connections collected from production Internet links. The main challenge in using this approach is that the packet traces are simply a sequence of packets on the network and they do not directly carry information about the configuration or state of end-host TCP mechanisms. This is particularly challenging because

end-hosts differ in the configuration and variant of TCP used and a packet trace originating from a host does not directly indicate which one is being used.

Hence, it is our objective in this dissertation to first develop passive analysis techniques that can result in an in-depth analysis of TCP reliability mechanisms by studying Internet packet trace

1.1.1 Summary of Dissertation Objectives

- Study the accuracy and timeliness of TCP loss detection/recovery in real world TCP connections originating from prominent sender-side OSes, and
- Study the impact of changing parameter configurations on the overall durations of real world connections in order to identify the best parameter settings.
- Systematically evaluate the ability of different Delay Based Congestion Estimators (DBCEs) to accurately predict losses.
- Study the impact of connection characteristics on the performance of DBCEs
- Develop a passive analysis tool for accurate and in-depth analysis of real world TCP connections.

In the rest of the chapter, I will briefly describe our approach for developing a passive analysis tool. I will then formulate the problem of analyzing and improving TCP loss detection and prediction. I will conclude this chapter with my thesis statement, a summary of major contributions and an overview of the rest of the dissertation.

1.2 Passive Analysis of TCP Traces

1.2.1 Benefits of Passive Analysis

The main approach of this dissertation is to conduct passive analysis of Internet TCP connections to understand their behavior in the real world. Passive analysis offers several key advantages. A single observation point provides access to a large number of connections that traverse different paths and experience widely different network conditions. Passive monitoring offers simpler logistics and lower overhead than conducting active network measurements; this is because it neither relies on a distributed measurement infrastructure nor does it inject additional probe traffic on the paths to be studied.

1.2.2 Challenges of Passive Analysis

Passive analysis present two significant challenges. First, with ever-increasing traffic volumes carried by Internet links, the collection, storage, and analysis of packet traces presents logistical as well as computational constraints. Passive analysis techniques, therefore, need to be designed to be able to operate with such constraints. The second challenge is that passive analysis does not allow any direct control over either the end-system mechanisms or the network and traffic conditions experienced by the traffic; one can simply observe what occurs. This makes it difficult to answer "what if" type of questions.

To see how these challenges impact our work, recall that the main goal of this dissertation is to study TCP's efficacy in recovering from and in predicting packet losses. This goal requires that our passive analysis should be able to: (i) identify all occurrences of packet losses in a given TCP connection trace, (ii) study the efficiency of the corresponding TCP sender's loss detection/recovery mechanisms in dealing with these losses, and (iii) study the ability of TCP to predict such losses. All three of these steps must address the first issue by making sure that the analysis is completed in a single-pass through a trace and does not require excessive memory or intermediate storage. The second and third steps also require that we be able to, respectively, modify loss detection/recovery mechanisms and introduce loss prediction mechanisms at the TCP sender corresponding to each trace, and then study the impact on connection performance. With passive analysis, we can do this only by relying on a "semi-experimental" approach, wherein we develop models to capture the above interaction.

We develop a passive analysis tool called TCPdebug that achieves the first of the above steps. Below we describe our methodology for developing TCPdebug.

1.2.3 Developing TCPdebug

Since TCP itself uses several detection/recovery mechanisms like – retransmission timeouts (RTO) [APS99] , Fast retransmit/recovery (FR/R) [Ste97], Partial Acks (PA), Selective Acks (SACK) – to detect 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 packet was indeed lost in the network [KSE+04]. However, this approach can lead to over-estimation of losses. TCP maintains limited state information and hence is not able to always correctly determine if a packet is actually lost in the network or not. This is especially true after a timeout as a TCP sender resets its state after

a timeout. In order to reliably infer packet losses from all segments retransmissions, it is important to *track the explicit triggering of TCP's loss detection mechanisms—namely, RTO, FR/R, PA, and SACK.*

It turns out that even simply tracking the triggering of loss detection/recovery mechanisms in a TCP sender—as is done in [JID+04]—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***: If multiple packets are lost together, the first packet is detecting using TCP's detection mechanism while the subsequent packet retransmissions may be triggered in some versions of TCP using implicit signals like Partial ack (PA) [FHG04]. It is, thus, important to *identify* ***implicit*** *retransmissions that are* ***needed*** *for recovering from packet losses.* It should be noted that if the history about all previously transmitted data packets is maintained, the ACK stream can help to identify such retransmissions.
- ***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 duplicate acks, triggering a loss detection/recovery phase in TCP. It is, therefore, important to *identify* ***explicit*** *retransmissions that are* ***not needed*** *for recovering from packet losses.*

The final significant issue is due to the diversity among real-world operating systems (OSes) in the implementation and configuration of TCP loss detection/recovery mechanisms. This diversity poses two types of challenges. First, the details of most of these implementations are either not well-documented or are hidden behind proprietary code. Second, given a TCP connection trace, it is not clear which sender-side implementation it corresponds to. Hence, it is important to *extract details of TCP loss detection/recovery from prominent TCP implementations and to develop techniques for identifying the sender-side OS for a given packet trace.*

Basic Approach: Our approach for achieving the first three requirements derived above is based on the belief that if sufficient state about previously transmitted packets are maintained, then subsequent patterns in the data and ACK streams of a TCP connection can help with accurate analysis. Second, we also believe that details of sender-side loss detection/recovery implementations in prominent OSes can be extracted by subjecting each to controlled patterns of loss and delay and by analyzing the observed

behavior. The extracted behavioral signatures can also help in identifying the sender-side OS for an arbitrary Internet trace. Thus, our basic approach for passive inference of TCP losses is as follows: (i) Use reverse-engineering to extract the implementation details of loss detection/recovery in four prominent TCP stacks (Windows XP, Linux 2.4.2, FreeBSD 4.10 (MacOS), and Solaris) by experimenting with these in a controlled lab setting, (ii) Replicate the loss detection/recovery mechanisms in four OS-specific analysis state machines—these state machines use the data and ACK streams, (iii)Augment these machines with extra logic and state about all previously-transmitted packets in order to classify retransmissions as needed or unneeded and to infer packet losses with accuracy greater than TCP, and (iv) Run each connection trace against all four machines and use the analysis results from the one that can explain and classify all of the observed out-of-sequence (OOS) segments.

1.3 TCP Loss Detection

Using *TCPdebug* , our first objective is to analyze the efficiency of TCP loss detection mechanisms. Below, we review these mechanisms and formulate the problem of analyzing these.

Loss detection mechanisms: TCP senders assign sequence numbers to all data bytes transmitted and receivers use *cumulative* acknowledgments (ACKs) to confirm receiving these. Senders detect segment losses using two types of mechanisms that rely on the ACK stream: *retransmission timeouts* (RTOs) [APS99] and *fast retransmit/recovery* (FR/R) [Ste97]:

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

$a, b, m, k, minRTO$ [PA00]—RTO-based loss detection can, therefore, be time-consuming.

FR/R: FR/Rs are a faster means of detecting losses—if a segment is lost, segments with higher sequence numbers trigger duplicate (cumulative) ACKs for its preceding segment. Hence, when a sender receives duplicate ACKs for a segment, it can conclude that the next higher segment was lost. However, reordering of segments by the network can also trigger the generation of duplicate ACKs. In order to avoid erroneously inferring loss in such cases, TCP senders usually wait for $D > 1$ duplicate ACKs [Ste97] before concluding a segment loss.

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

When loss is detected, segments are immediately retransmitted. Loss recovery is also accompanied by a reduction in TCP sending rate—the reduction is quite significant for RTO-based loss detection. The invoking of loss detection/recovery can thus be quite costly in terms of connection duration[1].

Factors affecting the loss detection efficacy: The exact cost of loss recovery depends on the choice of values for each of the 6 parameters associated with loss detection: $D, a, b, m, k, minRTO$. Two performance-impacting goals guide the optimal setting of these parameters:

- *High accuracy of loss detection:* First, a TCP sender should be accurate when it identifies segment losses. If TCP erroneously infers that a segment was lost, it would unnecessarily invoke loss recovery, reduce its sending rate and hence increase the connection's duration.

  Accuracy of FR/R-based loss detection can be improved by *selecting a larger value of* D, the duplicate ACK threshold—a larger D would help avoid spurious retransmissions when duplicate ACKs are generated not by segment losses, but by mere segment reordering in the network.

  Accuracy of RTO-based loss detection can be improved by *selecting a larger value of RTO*, which is determined by the parameters $a, b, m, k, andminRTO$—a larger RTO would help avoid spurious retransmissions when segments or their ACKs are not lost, but merely delayed in the network.

- *Timeliness of loss detection:* Second, a TCP sender should quickly identify segment losses. The longer a sender takes to detect a loss, the more is the opportunity lost for sending new data and

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

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

Table 1.1: Values of key parameters in different TCP Stacks. Timer granularity is the granularity of clock used in the OS to measure RTT and RTO. Initial RTO is the initial value of RTO used. $minRTO$ is the minimum value of the RTO permitted by the OS. a, b, m, k are the parameters of RTO equation used by the OS. D is the dupack threshold used by the OS. RTO equation is the outline of the equation used by the OS.

reducing the connection duration. This is especially true for RTOs, which have long detection times—these can be reduced by *selecting a smaller value of RTO*.

The loss detection times for FR/R can also be reduced slightly by *selecting a smaller value for D*—in this case, the sender has to wait for a smaller number of duplicate ACKs before it can infer a loss. However, much more significantly, a smaller value of D enables more losses to be discovered using FR/R, rather than RTOs—this is especially true for small connections that do not transmit enough segments to generate the required number of duplicate ACKs. Given that RTO-based loss recovery is more costly than FR/R-based recovery, this further helps improve connection durations.

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

The proposed standards for TCP recommend values for each of these parameters [PA00, Ste97]—however, these recommendations are based on empirical evidence collected more than a decade ago. Furthermore, real world TCP implementations differ, sometimes significantly, in their default settings of these parameters (see Table 1.1[2]). This naturally raises two important questions regarding the efficacy of TCP loss detection/recovery: *Are the parameter settings in different TCP implementations working well in reducing connection durations? Are the decade-old recommended settings in the TCP standards optimal for the current Internet?* These questions have been partially addressed in a couple

[2]For Linux, the threshold D is actually adaptive based on the amount of reordering seen in the network

of key studies described below – however, most studies were conducted nearly a decade ago; consequently, these do not incorporate diversity in properties of real world TCP implementations, Internet paths, and application behavior. More importantly, no previous study has modeled the impact of TCP configuration on the overall connection durations of TCP connections.

In this dissertation we answer these questions using a two pronged approach. First, we study the current state of the TCP packet loss detection/recovery mechanisms in an OS sensitive manner. Second, we evaluate different parameter settings to identify the optimal setting to achieve best possible connection duration in real world connections.

1.4 TCP Loss Prediction and Avoidance

While the previous section discusses the possibility of detecting and recovering from losses efficiently, an alternate (and perhaps more desirable) approach would be to predict upcoming losses and react to avoid these from occuring. We next study ability of TCP to predict losses. Below, we discuss the advocated use of delay for predicting losses and formulate the problem of analyzing these predictors.

Ability of delay signals to predict losses: In order to avoid packet losses and the associated performance costs of TCP loss-recovery, several *Delay-based Congestion Estimators* (DBCEs) have been proposed [Jai89, WC91, WC92, BOP94, AR98, MNR03, WJLH06, YQC04, LSM+07]. These DBCEs rely on the assumption that during periods of congestion, packets within a connection will experience higher queuing delays at the congested link—this should translate to an increase in packet round-trip times (RTTs). By sampling per-packet RTTs, and comparing them to a base RTT value (measured in the absence of congestion), a DBCE infers the onset as well as alleviation of congestion. The hope is that DBCEs can detect the onset of congestion much earlier than the occurrence of packet loss and the corresponding congestion avoidance (CA) mechanisms can potentially avoid the loss. Existing DBCEs differ primarily in the RTT-derived metric and the base metric used for estimating congestion.

The performance of a DBCE is governed by how well it can predict congestion and help avoid packet loss. Specifically, two factors determine the overall efficacy of a DBCE:

- *Ability to predict losses:* Packet losses have been shown to significantly impact the durations of TCP connections, mainly due to the time spent in loss detection/recovery [RKS07a]. The larger

the fraction of losses in a connection that can be predicted and avoided[3], the larger the likely reduction in connection duration.

- *Ability to avoid false predictions:* Note that an aggressive DBCE that signals congestion most of the time is likely to predict more losses. However, the corresponding CA mechanism would react to this signal and keep reducing the connection send rate—even when no loss is likely to happen—and degrade the connection duration. Thus, the lower the rate of false loss predictions made by a DBCE, the better the connection duration is likely to be.

Unfortunately, due to the inherent noise in TCP RTT estimates, the two factors mentioned above are often in conflict with each other. An aggressive DBCE is likely to predict more losses, but also have a high rate of false predictions. On the other hand, a conservative DBCE will seldom give false predictions, but would miss out on many losses. Consequently, it is natural to ask: *how well do existing DBCEs perform along these two factors?* And perhaps more importantly, *what DBCEs perform the best in terms of achieving maximum reduction in connection duration?*

Secondly, the loss prediction ability is also a characteristic of the connection and the network. A connection that consumes a significant portion of bandwidth on a congested link is likely to witness high correlation between the losses and end-to-end delays. On the other hand, connections that traverse a highly multiplexed path and occupy only a small fraction of the link bandwidth may witness low correlation. In our evaluation of DBCEs, therefore, we would also like to address the question,: *what connection and network properties affect the performance of a DBCE in real-world TCP transfers?*

While DBCEs have been evaluated in the past [JWL03, PJD04, BV03, MNR03, BV98a, BV98b], the issue of what DBCEs are likely to improve the overall timeliness performance of TCP connections has not been adequately addressed. To understand these issues we first conduct a comprehensive evaluation of several prominent DBCEs. Our evaluation explicitly models the impact of a DBCE on the duration of these connections. Next, we study the characteristics of these connections in order to analyze their influence on the efficacy of delay-based congestion estimation.

[3]Note that a DBCE can aid in merely predicting losses; a corresponding CA mechanism is expected to additionally help avoid the impending loss. We discuss this assumption in more detail later.

1.5 Thesis Statement

While it is generally accepted that TCP, the most popular transport protocol in use in the Internet, is adversely affected by the packet losses on the network, the actual impact of these losses on TCP performance in the real world is not well understood. Furthermore, the ways to mitigate this impact have not been systematically evaluated. In this dissertation, I carry out a systematic evaluation of the impact of packet losses on real world TCP connections and also evaluate two approaches to mitigate the impact of losses, namely (i) optimizing the configuration of current loss detection algorithms and (ii) evaluating the efficacy of delay signals in predicting losses. This dissertation develops a detailed OS-sensitive passive analysis tool to facilitate the above analysis.

The main theses of my dissertation are as follows

- Real world TCP connections can accurately be analyzed to study the impact of packet losses on TCP.
- We can improve upon the recommended and in use configurations of TCP loss detection parameters to improve TCP performance.
- Aggressive DBCEs can help in predicting the packet losses for connections with high throughput but have limited use for other connections.

1.6 Contributions

The main contributions for this dissertation are as follows

- We developed a detailed, OS-sensitive passive analysis tool (*TCPdebug*) for accurate classification of retransmission in a TCP connection. We carefully validated the tool using both controlled and real world test cases and found it to be accurate in 99+% of cases. To the best of our knowledge, *TCPdebug* is the only OS-sensitive tool available for passive TCP analysis. We have made significant advances by explicitly including TCP implementation-specific factors for those operating systems that are currently (and likely to be for the foreseeable future) the dominant end points for TCP connections (Windows, Linux, Solaris and FreeBSD/Mac OS X). While many of the trace analysis insights used in the tool are not new, this is the first time all have been

integrated into a single, carefully validated tool. We have also been careful to cover many of the "corner cases" and boundary conditions that are missing in prior work, choosing to rely on explicit sender-state tracking rather than approximations or heuristics where possible.

- Using *TCPdebug* we analyzed a large number of diverse traces to provide a detailed view of current state of losses in the Internet. We find that while a large percentage of connections do not experience any loss at all. For connections that do experience losses, the main detection mechanism used to detect these losses is RTO and not FR/R. The main cause of prevalence of RTO is the small flight sizes of typical connections, which do not generate significant duplicate ACKs to trigger FR/R based loss detection. Finally, we find that a significant number (3.7-19%) of all retransmissions are unnecessary as the packets are not actually lost.

- We performed a detailed analysis of TCP loss detection using *TCPdebug*. Specifically, we study the impact on connection performance of the configuration of parameters associated with TCP loss detection. To facilitate this analysis, we develop models to capture the impact of change in each parameter setting on the total duration of a connection. Based on this analysis we conclude that:

  - Most of current implementations of RTO estimators are conservative in incorporating variability in TCP RTT. Reducing the influence of RTT variability can help significantly reduce the connection durations of TCP connections.
  - Unlike in the past, timer granularity and the minimum RTO no longer significantly limit connection performance.
  - Based on our analysis of a large number of connections, we found that the best-performing configuration of TCP loss detection can be obtained by lowering the value of K from current default value of 4 to 2 and making the dupack threshold D adaptive according to the rule $D = max\{1, min\{3, F-2\}\}$ where F is the current flight size of the connection.
  - The Linux RTO estimator converges fast and is the most efficient. If properly configured, this estimator has the greatest potential for improving connection durations.

- We evaluated the ability of several prominent DBCEs to predict losses. We developed analytical models to passively predict the impact of these predictions/mis-predictions on the connection duration. We find that

  - CIM is the overall best estimator. It is likely to reduce the duration of large connections significantly.

  - The estimator used by the prominent Vegas protocol is fairly conservative. It has almost no impact on the performance of TCP connections that do not transmit large flights of segments.
  - Tri-S and DECA are two estimators which, in most cases, worsen connection performance.

- Finally, we study the influence of connection characteristics on the performance of DBCEs. We find that connections with a high throughput and large flight sizes are likely to benefit the most from any DBCE . Connections which have very few packets in flight are least likely to see any improvement in their performance.

1.7 Overview

In Chapter 2, we first discuss the basic functioning of TCP, its congestion control algorithm and alternate congestion control algorithms. Next we present a survey of various passive and active analysis tools. Finally, we summarize research on TCP analysis as well as modeling of TCP performance.

Chapter 3 presents our tool *TCPdebug*. In this chapter we present the basic approach for the tool, its major challenges, and their solutions. Finally, we validate the accuracy of the tool and compare it with other prominent tools in this area.

Chapter 4 focuses on analyzing the efficacy of current TCP implementation's loss detection mechanisms. We show results from systematic evaluation of the impact of design parameters associated with TCP loss detection/recovery mechanisms on the performance of real world TCP connections. This chapter concludes by suggesting changes to current TCP implementations and its potential impact on TCP performance.

Chapter 5 studies the ability of nine prominent DBCEs to predict or mis-predict losses. It quantifies the potential change in connection duration each of these DBCEs results in for a prediction or mis-prediction and uses it to evaluate the overall impact of each DBCE . We also study the impact of connection duration on the performance of these DBCEs .

Chapter 6 presents ours conclusions and discusses future work.

CHAPTER 2

Background and Related Work

A scientific theory should be as simple as possible, but no simpler.

— Albert Einstein (1879–1955)

The outcome of any serious research can only be to make two questions grow where only one grew before.

— Thorstein Veblen (1857–1929)

TCP, being the most popular transport protocol, has received a lot of attention in the literature. In the first part of this chapter we will provide a brief background on TCP and its operation. In this part we discuss the mechanism used by most current implementation of TCP. With this background, in the second part of the chapter we highlight some of the past work related to this dissertation. This covers past work on

- TCP reliability and congestion control. We look at the literature which proposes and evaluated alternate approaches for implementing congestion control in TCP. We then discuss work related to enhancing and evaluating the accuracy of TCP's loss detection mechanism which is the backbone of achieving reliability in TCP as well as look at how several other protocols achieve reliability.
- TCP analysis tools and methodologies. Next, we present various tools and methodologies developed for understanding TCP's behavior and performance and highlight their advantage and drawbacks.
- TCP analysis and modeling. Finally, we present the conclusions of several studies analyzing TCP as well as present models used to capture TCP's behavior.

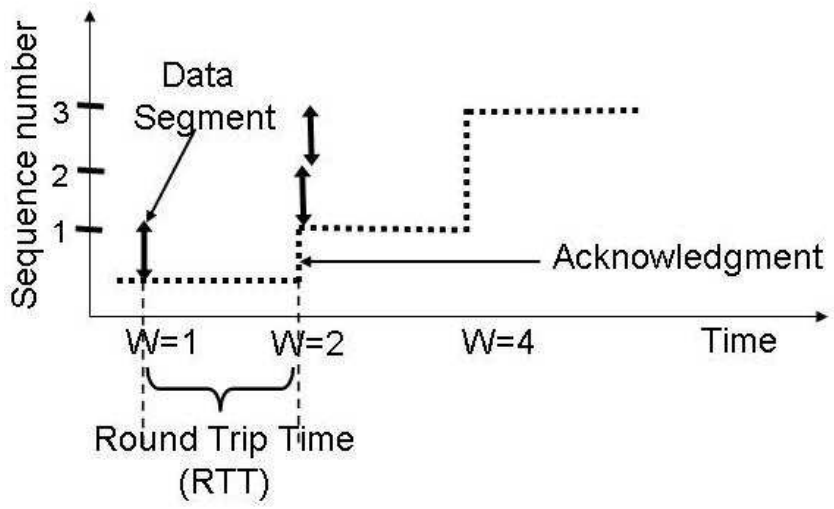

Figure 2.1: Time Sequence Plots for TCP: these plots are used to present the exchange of data and acks between the sender and the receiver

2.1 Background

In this section, we will discuss the basic operation of TCP protocol as well as some of the popular modifications proposed or implemented for this protocol. TCP provides a connection oriented data transfer service to applications communicating over the Internet. The main reasons for TCPs popularity are that it provides two attractive services: reliability and congestion control. Reliability ensures that all data reaches the receiver while congestion control makes sure that a TCP connection does not overload the network. Below we discuss how TCP implements these services.

2.1.1 TCP Reliability

In order to provide reliability, TCP relies on the mechanism of assigning sequence number to all the data it sends out and requires the receiver to send acknowledgment for the data it receives. TCP sends out new packets on receiving the acknowledgment for packets it has send out. This interaction can be represented by a time sequence plot as shown in figure 2.1. The x-axis plots the time and the y-axis plots the sequence number of packets send by the sender or the acks received. The data packets are depicted by an arrow and the cumulative acknowledgment received so far is represented by a dotted line.

In absence of loss, TCP keeps sending new packets as old ones are acknowledged until all the data is transferred. Problems occur when a packet is lost. Since TCP guarantees reliability, it is responsible

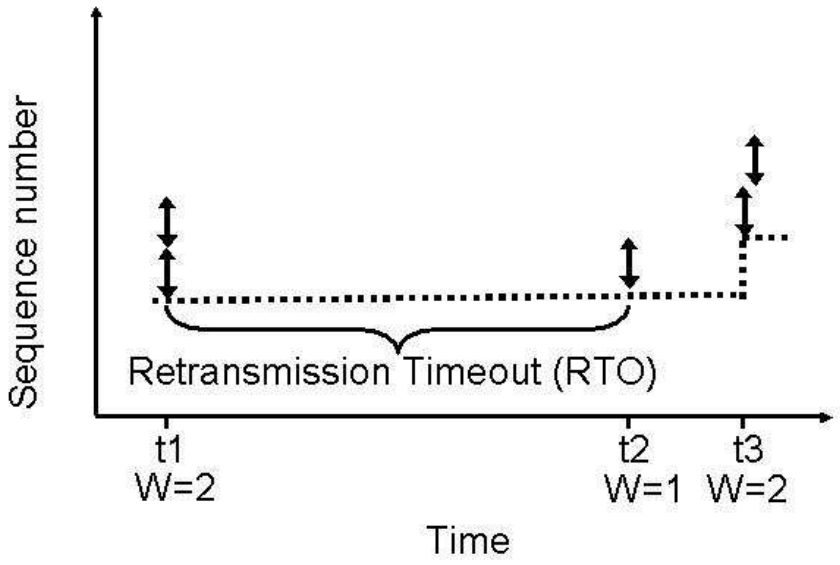

Figure 2.2: Retransmission Timeout (RTO) based loss detection in TCP

for detecting any packet loss and retransmitting the packets. TCP relies on implicit signals from the network or the receiver to make an educated guess about whether a packet is lost or not. It detects the first of a series of packet losses using one of the following mechanisms.

Retransmission Timeout

TCP starts a timer when it sends out a data packet and waits for an acknowledgment. The timer is referred to as the retransmission timer. If an acknowledgment arrives before the timer expires, TCP resets the timer and measures the time difference between the data packet send and its ack received as the Round Trip Time (RTT) for the connection (as shown in figure 2.1). However, if an acknowledgment is not returned within the timeout period, the sender times-out, assumes that the packet is lost in the network and retransmits it. Fundamental to this strategy is the act of setting the retransmission timeout (RTO). In order to adjust to the network delays over different types of network, the retransmission timeout is not set to a fixed value but is calculated as a function of the connections RTT. Also, as the RTT itself may vary during the duration of the connection, the retransmission timeout is recomputed with each measurement of the RTT.

The main problem with waiting for the retransmission timeout to detect a loss is that for the duration when a connection is waiting for the RTO to expire, the sender is not sending any data. Figure 2.2 depicts this. During interval $t1$ to $t2$ TCP is waiting for the timer to expire and in absence of acks, the sender does not send out any new data packets. This stalling in effect increases the duration of the

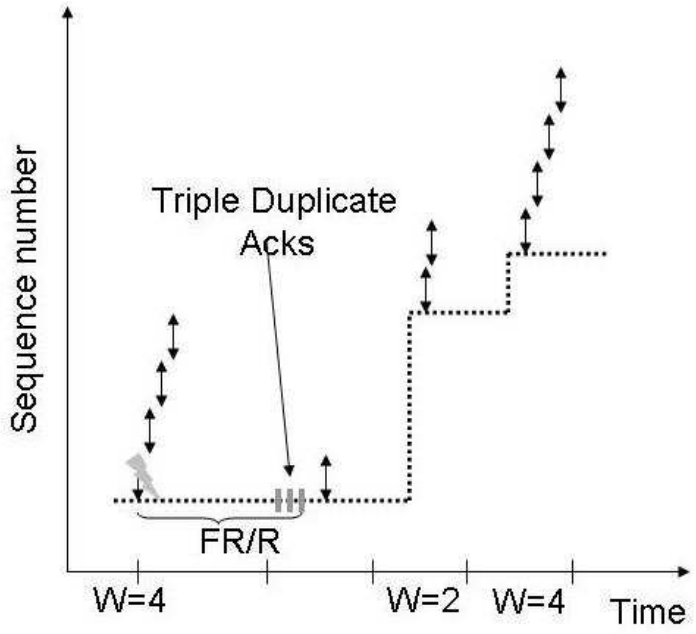

Figure 2.3: Duplicate acknowledgments based loss detection in TCP

connection[1].

Duplicate Acks Based Detection

To reduce the amount of stalling TCP undergoes when it has to wait for a timeout, it relies on a second, much faster, detection mechanism whenever possible. A TCP receiver sends out implicit negative acknowledgment on receiving a higher sequence number packet than what it was expecting. For TCP these negative acks are simply inferred from the duplicate acks for the last in-sequence packet. Reception of the duplicate ack indicates to the sender that a packet may be lost or delayed as the only way a duplicate ack is generated is if the receiver is missing a packet. TCP waits for a few of these before concluding that the packet is indeed lost and resends[2]. Figure 2.3, shows a dupack triggered loss detection. RTO based detection is very conservative and cause connection stalling for several RTTs, the dupack based detection is more aggressive (as it has more information to make an informed decision.) and leads to loss detection in slightly more than an RTT. As is indicated in the figure 2.3, the amount of stalling in this case is much less compared to the RTO based detection.

Next we discuss the congestion control algorithm used by TCP.

[1]Note that smaller connection duration is usually desirable as it means that all the data sent reaches the receiver earlier.

[2]The recommended number of duplicate acks to wait for is three but some implementations use a value of two while others use a variable value.

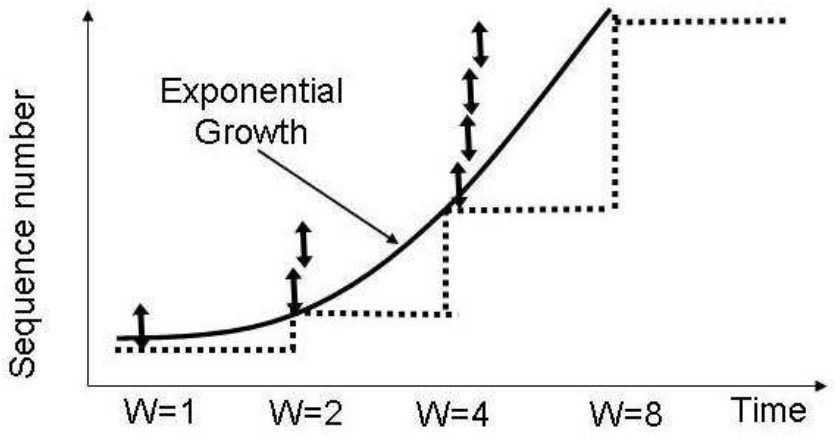

Figure 2.4: TCP Slow Start: The congestion window increases exponentially.

2.1.2 TCP Congestion Control

Early TCP implementations relied on a simple go-back-n mechanism without any congestion control. It had a notion of a fixed congestion window(cwnd) size and the whole window size worth of data is sent out without waiting for any acknowledgment. The sender waits for a predefined time (timeout interval) for an acknowledgment from the receiver and in absence of any ack, it resends all the packets again starting with the unacked packet.

The problem with this approach is that if TCP's sending rate causes a router in the path to get overloaded and its queue to overflow causing packet losses, TCP would simply timeout and resend all the packets at the old rate, again overloading the network. To prevent this effect, TCP implements a congestion control approach where it is continuously trying to adjust its sending rate to match the available capacity on the network. The algorithm used by TCP consists of several distinct phases and the rate adjustment in these phases and transition to and from these phases is described below.

Slow Start

When a connection is established, it starts off in a slow start phase. The initial size of the congestion window is between one to four packets[3]. The connection sends out the "congestion window" worth of data and waits for the acks. For every ack the sender receives, it increases the congestion window by one. Contrary to its name, slow start is not very slow. It increases the congestion window exponentially as shown in figure 2.4. This exponentially increases the load on the network and at some point the

[3]The initial recommended value was one but recent proposals have suggested using a higher value. Most TCP implementations currently use a value of two

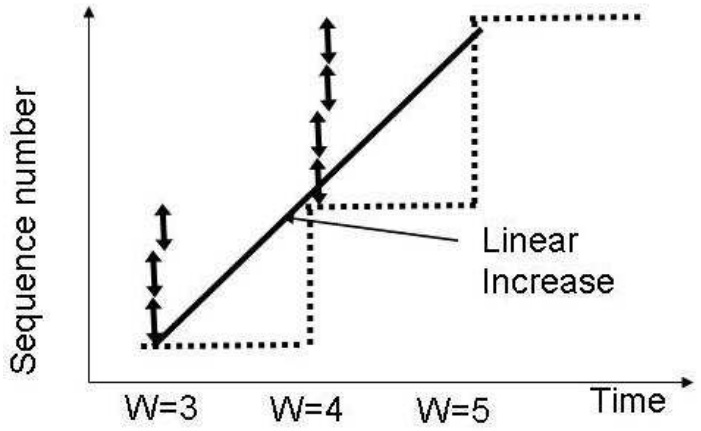

Figure 2.5: TCP Congestion Avoidance: The congestion window increases linearly.

network queues overflows and leads to packet losses. This loss is detected using either the retransmission timeout (RTO) or the duplicate ack-based detection.

TCP considers the encountered loss as an indication of network congestion and to alleviate the congestion it reduces its sending rate. The amount by which it reduces its sending rate depends on the mechanism used to detect the loss. If the loss is detected using a retransmission timeout it reduces the sending rate down to one packet and then uses a combination of slow start and congestion avoidance described below to increase its congestion window again. On the other hand, if the loss is detected using duplicate acks it reduces the sending rate to half of its current rate and enters the fast retransmit and recovery phase as described below. This reduction in sending rate again impacts the connection's duration. For retransmission timeout based detection the reduction is more drastic and hence may impact the connection more severely.

Congestion Avoidance

TCP uses a variable "ssthresh" to transition from an exponential increase to a linear increase. The slow start phase continues until it sees a loss or until the congestion window reaches the ssthresh limit. Once the congestion window crosses ssthresh, the connection enters the "congestion avoidance" phase. In congestion avoidance the sender increases its congestion window (cwnd) by 1/cwnd for every ack it receives. As shown in figure 2.5, this is equivalent to increasing the congestion window by 1 packet per RTT if every packet sent out is acked.

Ssthresh determines the point at which the rate of increase in the congestion window changes. The value of ssthresh is reset every time the connection detects a loss using the retransmission timeout

or duplicate acks. The ssthresh is set to maximum of 2 or half the number of packets in flight. As mentioned above when a loss is detected using RTO, the congestion window is reduced to 1, it then uses slow start to increase the window until it reaches ssthresh, and following that enters the congestion avoidance phase. The recovery scheme used after a dupack-based detection is described below.

Fast Retransmit and Recovery (FR/R)

When the sender receives a predefined number (recommended value is three) of duplicate acks, it concludes that the packet acked by the duplicate acks is lost and retransmits it. This is referred to as fast retransmit(FR). Following the loss detection, the ssthresh is updated and the congestion window is reduced to half the number of packets in flight. At this point the connection enters fast recovery instead of congestion avoidance.

In fast recovery, the sender keeps track of the number of duplicate acks it receives and since they are triggered by packets reaching the receiver, it inflates the congestion window and retransmits new packets to replace the ones which have left the network. This keeps the data flowing in the network. The sender comes out of fast recovery when it receives an ack for the segment which was retransmitted.

If multiple packets are lost from the same flight the losses following the first loss have to be individually detected using one of the two loss detection mechanism. This may increase the time it takes to detect the losses as well as reduce the sending rate multiple times. To avoid this, TCP relies on two mechanisms to detect multiple packet losses in the same flight. These mechanisms are detailed below.

Partial Acks

When the sender is using the Partial Acks(PAs) mechanism to detect multiple packet losses after Fast Retransmit, it remains in recovery until all the packets that were in flight when the loss is detected are acked. Figure 2.6 shows a connection's behavior when it uses PAs. While the sender is in recovery, if a partial ack (i.e. an ack which acks new data but not all the data that was in flight) is received, the sender concludes that the packet acked by the partial ack was lost in the network and it is immediately retransmitted. If more partial acks are received the corresponding packets are retransmitted. On detecting the losses using partial acks, the congestion window or ssthresh is not modified and the connection enters congestion avoidance phase when all packets which where in flight when the first packet loss was detected using duplicate acks are acknowledged.

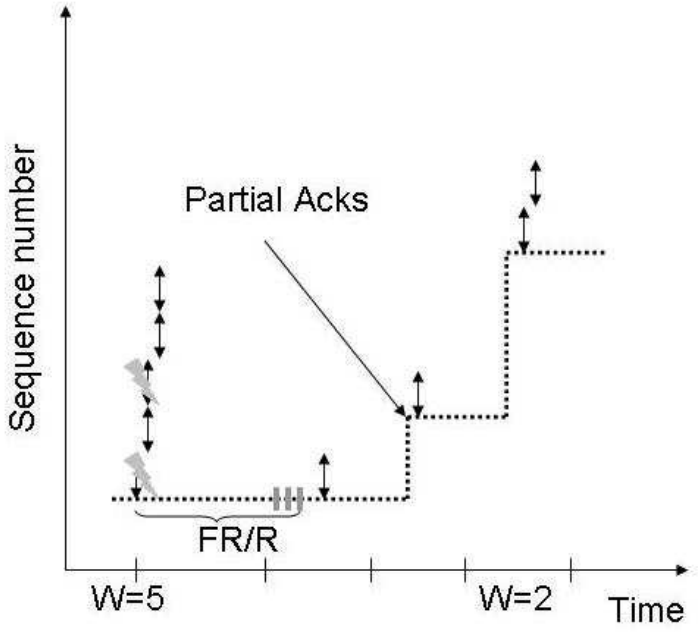

Figure 2.6: TCP Partial Acks.

Selective Ack (SACK)

Another way to deal with multiple losses is for the receiver to explicitly inform the sender about the packets it has received. This is exactly what selective acks do. TCP receiver uses SACK blocks, which are additional fields in an ack packet, to inform the sender what packets have reached it. After a dupack based loss detection, the sender retransmits all the packets indicated by the SACK block as missing at the receiver as long as the congestion window allows it.

We have so far covered the current popular implementations of TCP congestion control which are packet loss based. An alternate delay based scheme, which is slowly gaining popularity, is described below.

Delay Base Congestion Control for TCP We will now describe an alternate congestion control scheme which relies on packet delays. This scheme is popularly referred to as the Delay Based Congestion Control (DBCCs). In this section we will simply discuss the basic concept of this scheme and leave detailed discussion of the various Delay Based Congestion Estimators (DBCEs) proposed in the literature for the related work section.

Internet is a store and forward network. If the network is not congested, the network queues are relatively empty (as shown in figure 2.7). Hence the time it takes a data packet to reach the receiver

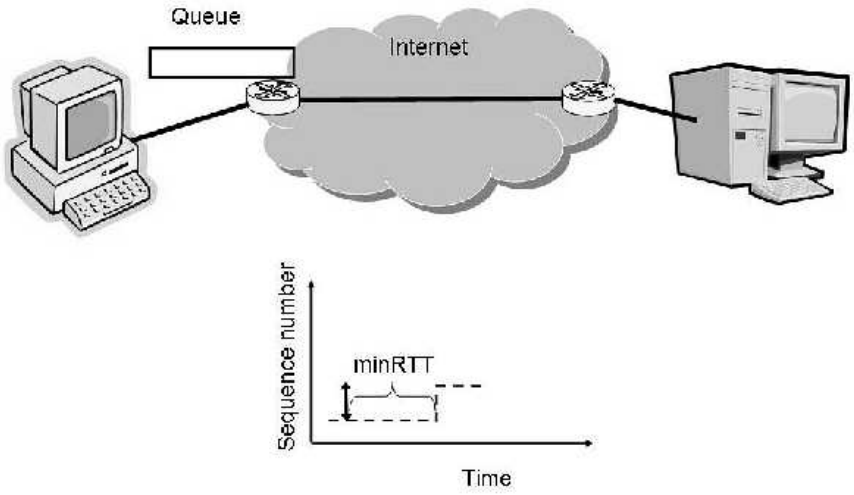

Figure 2.7: Uncongested Network

and the ack to get back to the sender is simply the time it takes for these packets to propagate through the network. This results in the data packets experiencing the minimum RTT for the network. On the other hand if the network is congested, the network queues will be relatively full (as shown in figure 2.8), the data packets will spend some time queued behind other packets and hence the total time it takes for the data packet to reach the receiver and the ack to get back is more than the minimum RTT.

DBCEs exploit this very fact. It measures the RTT and whenever it finds that the current RTT is larger than the minimum RTT by a certain factor, it infers congestion. DBCEs are therefore likely to infer congestion before the queues in the network completely fill up. Thus, it can avoid losses from occurring. This ability to avoid losses can greatly improve the performance for a connection as now the connection neither stalls waiting for a loss detection, nor has to undergo the drastic reduction in sending rate following a loss.

However, in practice there are several problems with this approach. The most prominent being the noise in the RTT measurement. There are several queues in the network and so the overall delay experienced by a packet is the sum of all delays and may not correctly reflect the situation at any one queue. Similarly, the end-system may introduce additional processing delays which adds more noise to the delay signal. It is not clear how well the proposed schemes would behave in the Internet.

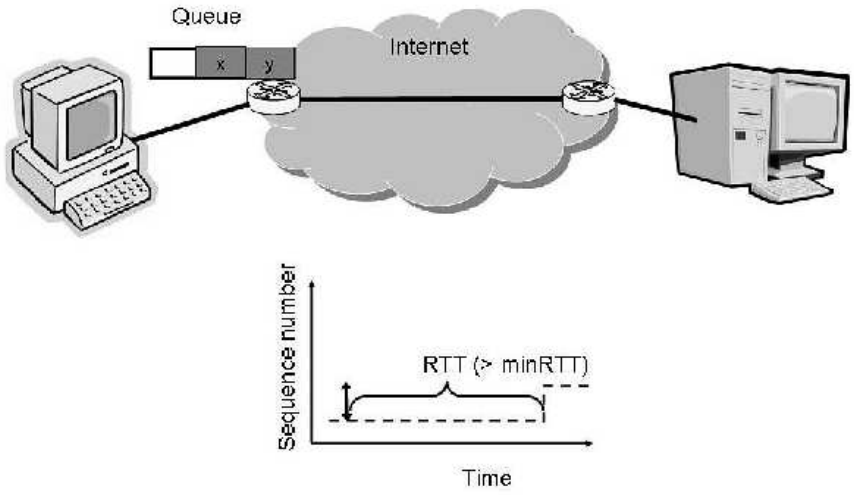

Figure 2.8: Congested Network

2.2 Related Work

With the above background on TCP, we will now present some research related to this dissertation. The first step involved in improving any system is to understand it. Hence first we will look at the evolution of TCPs congestion control algorithms and work related to providing reliability with emphasis on the motivations behind the various design decisions. Next, we look at some of the tools used in the past for understanding TCP's behavior. Than we look at some approaches to TCP analysis. This involves the various evaluations of TCP as well as work related to modeling the behavior of TCP. Both of these approaches help us better understand the performance of TCP.

2.2.1 Past Work on TCP Congestion Control and Reliability

In this section, we discuss the work related to TCP congestion control and reliability. The default packet loss based congestion control mechanism and its evolution is described first followed by the delay based congestion schemes, which are slowly gaining popularity. We then briefly look at other alternate methods of congestion control. While reliability in TCP is quite straightforward and has been discussed in the Section 2.1, we will briefly discuss work related to evaluating the accuracy of TCP's loss detection and its impact of TCP performance. Finally, we look at some alternate techniques proposed for implementing reliability in transport protocols.

Congestion Control

The current implementations of TCP use packet loss as an indicator of congestion. We will first briefly discuss these congestion control schemes and then look at other congestion control schemes based on delay, mathematical equations or sending rate.

Packet Loss Based Congestion Control

TCP was designed in the late seventies and early eighties as a reliable protocol. It achieved reliability by detecting packet losses and retransmitting the packets. It also provided flow control mechanism to prevent the sender from overrunning the receiver and sequencing of packets [CDS74, Pos81]. In October of 1986, the Internet observed a series of congestion events during which the performance of the Internet drastically declined [Jac95]. The cause was identified to be overflowing queues at the routers coupled with the use of continuous retransmissions by TCP to guarantee retransmission. In 1988 Jacobson proposed a series of algorithms based on the principle of "conservation of packets" to overcome these shortcomings [Jac95, APS99]. The proposed algorithms used packet drop as an indicator of congestion and employed exponential back-off mechanism to reduce its transmissions rate and thus mitigating the congestion. There were other proposed modifications like the use of slow start (for better estimation of available bandwidth at the start), better estimation of retransmission timeouts using variation in RTT, dynamic window sizing, etc. These modifications were deployed in a version of TCP that is popularly know as "TCP Tahoe".

TCP Tahoe waits for a coarse timeout every time a packet is lost. This long wait coupled with reducing the sending rate to a low value of one packet drastically degrades the performance of TCP connections. To overcome this problem TCP Reno [Ste97] was proposed. Reno relies on the fact that a TCP receiver send cumulative acks for packets it receives. So if a packet is lost, all the packets with higher sequence numbers will generate a duplicate acks for the lost packet. Duplicate acks will also be generated if the packet is not lost but simply reordered causing higher sequence packets to reach the receiver first. Reno assumes that the level of reordering is such that it will usually generate only one or two duplicate acks. Hence, if more than that many duplicate acks are seen by the sender, it can safely assume that the packet is lost and retransmit it. Furthermore, since higher sequence packets were able to reach the receiver without having been lost, the level of congestion in the network is relatively mild and hence it just halves its sending rate as against going down to a sending rate of one packet as done by Tahoe. These two changes reduce the detection and recovery period for packet losses and improve

TCP performance.

TCP Reno works well in low loss conditions where only one packet is lost in the network at a time. However, if multiple packets are lost then TCP Reno is able to recover the loss quickly only for the first packet, the remaining packets still have to be recovered using the much more costly timeout mechanism. TCP New-Reno [FHG04] was proposed to overcome this drawback. When multiple packets are lost in the flight, a retransmission for the first lost packet triggers an ack indicating that the next packet expected is the second lost packet. This ack is called as a partial ack (PA) (as it does not ack all packets that where in flight when the loss occurred). In Reno a partial ack ends the fast recovery. New-Reno does not exit fast recovery on partial acks. It uses the partial ack as an indicator that the second packet (acked by the partial ack) was actually lost in the network and retransmits it. Thus when multiple packets are lost, New-Reno can recover without another fast-retransmit or timeout, retransmitting one lost packet per RTT.

All the above modifications used an "intelligent interpretation" of the ack stream to detect and recover from lost packets. While duplicate acks or Partial acks indicate that an intermediate packet is missing and a higher packet has reached the receiver, but it is not able to inform the sender which packets are exactly missing or have reached the receiver. To overcome this shortcoming TCP SACK [MMFR96, BAFW03] was proposed. The SACK option in a TCP header contains a number of sack blocks. Each sack block reports a non-continuous block of data received by the receiver. The first sack block includes the most recently received packet and the other block simply repeat the most recent sack blocks. The receiver can use the information supplied in the sack blocks to maintain a list of packets that have reached the receiver and the ones which are missing. Sack does not change the basic congestion control algorithm or TCP robustness to delayed/reordered packets by waiting for multiple duplicates acks before triggering fast retransmit. Its behavior is different only in presence of multiple losses in a flight. When multiple packets are lost and fast retransmit is triggered for the first packet, Sack TCP uses the information in the sack blocks to identify the other lost packets and retransmit them.

The above description captures the TCP variants used by most of current TCP implementations. New modifications [FMMP00, BA02, BRAB06, LK00, ZKFP03] to TCP's congestion control have been proposed to overcome few specific problems with its performance. We will look at a few of these in Section 2.2.1.

All the above modifications still relied on packet losses as an indication of congestion in the network. Next we look at proposals which uses packet delays as an indication of congestion.

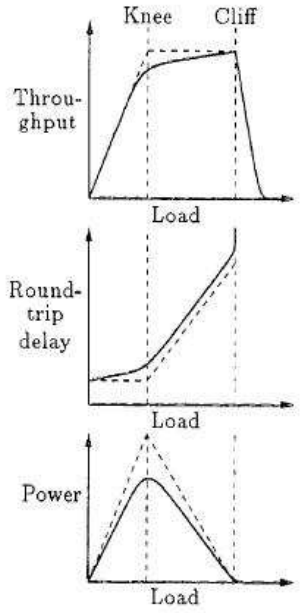

Figure 2.9: Network Performance as a function of load

Delay Based Congestion Control

There are several proposed modifications to TCP which use increase in delay on a path as an indicator of congestion. These schemes are collectively referred to as Delay Based Congestion Avoidance (DBCA) techniques. The basic idea behind these schemes is described in Section 2.1.2. These schemes are based on the following principle. The increase in delay is due to queuing at the routers. The increase in queuing results from the load been greater than the network capacity. This is an indicator of congestion. These delay based techniques use the round trip time (RTT) as the delay estimate of a path instead of the one way delay, which is difficult to measure. Below we discuss a few prominent schemes proposed in this area followed by studies evaluating these schemes.

Proposed DBCEs (Delay Based Congestion Estimatators

CARD [Jai89] was proposed in 1989 and was the first of the DBCE schemes. It is based on the argument that when the network is lightly loaded, the connection throughput increases with an increase in sending rate. But as network load reaches saturation, connection throughput does not increase further and RTTs start to increase. This can be viewed as a power graph as shown in figure 2.9[4]. CARD uses the increase in delay associated with increased network load as an indicator of congestion. It measures a gradient of delay and based on whether the gradient is increasing on decreasing it reduces or increases

[4]Figure from the paper [Jai89]

the congestion window.

The *Tri-S* scheme [WC91] is similar to this scheme but it uses a throughput gradient. It compares the current throughput gradient to the initial throughput gradient. Depending on whether the former is greater or small, Tri-S infers the absence or presence of congestion and decides on the change in congestion window.

While both *CARD* and *Tri-S* involve gradients, *Dual* [WC92] is based simply on comparing the maximum and minimum RTT. *Dual* is based on the assumption that the minimum RTT of a connection corresponds to the propagation delay along the network path, and the maximum RTT is the sum of this minimum RTT and the maximum queuing delay. When the current RTT exceeds the average of the min and max RTTs, Dual estimates congestion and its CA algorithm decreases the sending rate. *Delay-based End-to-end Congestion Avoidance* [YQC04] (DECA) also tried to maintain the RTT at the midpoint of the minimum and maximum RTT. However, instead of using the current RTT for comparison, it uses the maximum RTT observed in the last RTT interval.

While all the above schemes were promising, the idea of using delay as congestion indicator really came into the limelight following the introduction of *TCP Vegas* [BOP94]. *TCP Vegas* attempts to maintain enough data in the network such that it exceeds the delay-bandwidth product by a small amount. It compared the observed throughput in a flight of packet to the expected throughput, where the expected throughput is the throughput that would result in if all packets are acked within the minimum RTT. Its congestion-estimator relies on the fact that if the sending rate is much larger than that required to maintain only a few additional packets in the network, the connection RTT would increase (and its throughput would decrease). The decrease in connection's throughput is an indicator of congestion. *TCP FAST* [WJLH06] is a variant of Vegas designed for high-speed networks. Its congestion estimator is similar to that of Vegas; although it uses increase in RTT as an indicator of congestion, its DBCE can be shown to be a simple derivative of the Vegas throughput-based DBCE.

Delay based Additive Increase Multiplicative Decrease [LSM+07] (DAIMD) is also based on comparing the observed RTT to the minimum RTT for a connection. DAIMD assumes that the minimum RTT is simple the propagation delay. It measures the queuing delay as the difference between the smoothed RTT and the minimum RTT. However unlike *Dual* which uses a relative metric to estimate the congestion level, DAIMD compares the queuing delay to a fixed threshold (20-50ms) and if the queuing delay is above the threshold, it predicts congestion. To avoid oscillating between congestion and no congestion indicators, it uses a lower threshold to change back to no-congestion state. Another

algorithm that uses a fixed threshold is the *TCP Buffer Fill Avoidance* [AR98] (BFA). BFA measures the RTT variability similar to TCP-Reno but maintains its sign (direction of change). If the variation is above 10ms, it predicts congestion. It predicts no-congestion when the variability falls below -10ms.

None of the above DBCEs use history of the RTT in its estimation. Next we look at two DBCEs which uses history of RTT for a connection to estimate the congestion. Sync-TCP [WJS05] is based on TCP Reno but uses one way delay to detect network congestion. It detects congestion and its severity as follows. Sync-TCP defines the difference between observed RTT and the minimum RTT as the queuing delay. It measures the trend in the queuing delay over a certain number of packets (usually 9) and if the trend is increasing it compares the measured queuing delay with the max queuing delay seen so far. Based on this comparison, the estimator predicts congestion severity. The *Congestion Indication Metric* (CIM) was proposed in [MNR03] as a DBCE metric. This metric compares the most recent RTT samples (typically 1-2 samples) to the average RTT of several immediately preceding samples (typically, 20). If the most recent sampled RTT is greater than the average RTT, it concludes congestion has occurred.

Apart from these DBCE, there are several other DBCEs that have been proposed for specific environments [SS06, PLK05, Qiu05], or that use delay in conjunction with packet loss signals [LBS06], or use a parameterized model in conjunction with a DBCE to predict losses [HR06, MOM02]. We have not considered these DBCEs as they are not generic enough.

Evaluation of DBCEs

We now look at research evaluating the efficiency of DBCEs. The efficacy of a DBCE depends on its ability to predict the onset of congestion (or predict an impending loss) accurately and in a timely manner. The first question to be asked is whether the DBCCs even have a hope of succeeding in the real network. We look at a few studies which argue that DBCCs will not be effective in the real network.

In [PJD04], the authors briefly discuss a set of conditions under which DBCC will fail. They argue that DBCC is bound to fail if: (i) the max queuing delay at the bottleneck link is too small compared to the connection RTT, or (ii) the RTT sampling rate is less than the required Nyquist rate, or (iii) there is high degree of aggregation along a path and a connection's contribution to the total load is too small, or (iv) packet loss is not handled effectively. The paper, however, merely presents arguments and does not conduct a detailed investigation to validate the conclusions.

[JWL03] argues that the loss-based congestion control is inherently unstable in high-bandwidth

networks. It contends that the binary signal of loss/no-loss is too coarse to allow fine adjustment of send-rates, which is needed for stability at high speeds. A multi-bit signal can be obtained by using queuing delays as an additional indicator of network conditions. The authors, however, do not present any experimental results to illustrate/prove their point.

While both of the above studies present arguments against DBCE they do not evaluate DBCE in a real world experiment. Next, we look at some real world evaluations of DBCCs.

In [MNR03] the authors argue that if the increase in delay is due to the congestion in the queues, then a correlation should be present between observing higher delay for the packets and the probability of seeing a loss in the queue. They defines several metrics for quantifying correlation between packet losses and high RTTs. The authors conducted passive analysis of a large number of connections instantiated over 7 paths using the CIM DBCE defined in the paper. They find that using CIM as a congestion estimator can reduce connection loss rates. However, it also results in a large number of false positives resulting in a 37% reduction in the aggregate throughput. Based on this observation they conclude that DBCC is in general not a viable solution. This study is severely limited in its size. It also does not analyze the influence of connection characteristics on the performance of CIM.

In [BV03], the authors compare the correlation between the packets in flight and RTT unlike [MNR03], which compares the correlation between RTT and packet loss. The idea is that DBCC is likely to be effective only if its congestion avoidance can alleviate congestion by reducing the sending-rate (that is, the connection is *self-congesting*). This is not likely to happen if the correlation between sending rate (or packets in flight) and observed RTTs is low. The authors passively analyze 14,218 connections instantiated over 737 different paths. They found that, in general, the coefficient of correlation between RTT and packets in flight is weak [BV03]. However, this study does not evaluate the ability of DBCEs to predict loss in general.

Apart from these real network studies, there have been several studies using network simulations to evaluate DBCEs. In [BV98a, BV98b] the authors have used network simulations to evaluate the ability of Vegas, CARD, and Tri-S to predict loss. They found that all three methods are rarely better than a "random coin-tossing" estimator. However, the Vegas method was found to be slightly better than the other two methods. Unfortunately, their use of simulations prohibits evaluation under a wide range of connection characteristics, as well as prohibits sampling the wide variety of real world network conditions which may affect the performance of an estimator.

Both loss based and delay based congestion control schemes implements both congestion control and

reliability components of TCP. Several real world application, like live video streaming, do not need the reliability component or simply want to have more control on the congestion control mechanism than what TCP allows protocols build on top of it. These protocols are TCP-friendly in the sense that they do not aggressively consume the network resources but share the network like TCP. However, these protocols do not directly implement all of TCPs features. Below we discuss several of these protocols.

TCP Friendly Congestion Control

The TCP friendly congestion control protocols are described as those for which the long-term throughput does not exceed the throughput of a conformant TCP connection under the same conditions. A slightly alternate definition is used by some protocols. This definition considers a protocol to be TCP friendly if it does not reduce the long-term throughput of any co-existent TCP flow more than another TCP flow on the same path would do under the same network conditions. TCP friendly protocols can be broadly classified as multicast or unicast protocols [WDM01]. Designing multicast protocols is much more difficult because the congestion action has to take into consideration the feedback from all the hosts. We will first look at some unicast protocols and then discuss some simple multicast protocols.

We will first look at some techniques that rely on detecting packet losses to sense congestion in the network. An obvious first step to detect packet losses and be TCP-friendly is to use the default TCP congestion control method directly but remove its reliability mechanism. This approach was used in [JE96] for video streams. Rate Adaptation Protocol [RHE99](RAP) further expanded this idea. It implemented an AIMD scheme similar to TCP in a rate based manner but does not retransmit lost packets. Like TCP, RAP uses ack streams and timeouts to detect packet losses. However, on detecting the losses, it reduces its sending rate only by half each time. The decision of increasing or decreasing the rate of sending data packets is done only once every RTT. General AIMD congestion control [YL00] (GAIMD) is another scheme using an additive increase and multiplicative decrease behavior like TCP. However, instead of increasing congestion window by one packet and reducing it by half, GAIMD uses variable parameters. This paper found that to remain friendly to TCP, the increase in the rate(α) and the decrease the rate (β) should maintain a ratio of $\alpha = 4(1 - \beta^2)/3$.

Loss-Delay Based Adaptation Algorithm [SW00] LDA+ relies on the RTCP feedback mechanism provided by the Real time Transport Protocol [FHG04] (RTP) instead of using TCP like methods to measure losses on the network. LDA+ uses a packet pair technique [KLDL04] to estimate the available bandwidth and uses this to dynamically set its additive rate increase. Its multiplicative rate decrease is set as $1 - \sqrt{p}$ where p is the observed packet loss rate in the network.

An alternative to reacting like TCP to losses is to model the behavior of TCP and modify the sending rate according to this model. This is exactly the approach used by the TCP-Friendly Rate Control Protocol [PKT99] (TFRCP). It uses the model for TCP throughput developed in [PFTK98] to adjust its sending rate rather than reacting to losses directly. This protocol divides time into rounds with fixed durations and in each round the parameters required for the model are calculated. If any packet loss is seen during a round, the rate is recalculated using the new set of parameters. If no loss is observed in a round, the sending rate is doubled. This doubling of sending rate makes this protocol more aggressive than TCP and leads to unfairness as well as oscillation in the rate. TCP-Friendly Rate Control Protocol [FHPW00] (TFRC) evolved from [PKT99]. [FHPW00] uses the complex equations developed in [PFTK98] as well, but it uses more complex methods to measure the parameters used in the model. Most notably, the packet loss rate is measured as a moving average to avoid sudden increase in sending rate when no loss is seen in a round.

Finally, TCP Emulation At Receiver [ROY00] (TEAR) also measures losses like TCP, but does so at the receiver instead of at the sender. The receiver calculates a fair receive rate and sends it back to the sender, who adjusts its sending rate accordingly. To calculate the receive rate, the receiver has to replicate the congestion window changes at the sender which it does by tracking the congestion window reduction that would be triggered by any triple duplicate acks it sends out and by estimating the timeout events at the receiver. It uses a moving average to smoothen the rate changes resulting in a much smoother rate change than TFRC.

We will now discuss a few of the multicast protocols. This review is not exhaustive but covers a representative set of protocols. The Loss Tolerant Rate Controller [Mon97] (LTRC) uses an AIMD scheme which is different than TCP. Given the difficulty of measuring network characteristics to be used in the model as parameters for a large number of receivers, the protocol proposes using preset values for these parameters. The presets include even the RTT expected for a packet. All receivers report the losses using negative acks (NACKs). LTRC uses the worst loss rate seen by the receiver to calculate TCP throughput using the simple model presented in [MSM97]. It uses this as an upper bound on its sending rate. LTRC does not react to each loss it experiences. Periodically, based on the timers used to detect loss, it evaluates the number of losses seen so far and if this is over a certain threshold, it decreases its sending rate else it increases it rate. Since this protocol uses preset values for some of its parameter including RTT, it cannot adapt to the dynamics of the network. The use of explicit packet loss indication using NACKs, which cannot be suppressed, limits the scalability of these protocols.

To streamline the process of estimating network loss rates, Tree-based Reliable Multicast protocol [CHKW98] (TRAM) uses a dynamic tree structure for reliable multicast. The receivers (leaves of the tree) periodically generate a report of the losses they experience. The intermediate nodes can also generate a congestion report if their buffer occupancy exceeds a certain threshold. These congestion reports are passed towards the root node and aggregated as they are passed upward. The sending rate to be used by the sender is bounded by a maximum and minimum sending rate set independently for the tree. If the final congestion report reaching the receiver indicates congestion the sender halves its sending rate else it increases its sending rate as a fraction of the difference between its current sending rate and the maximum sending rate.

The authors in [GS99] propose a protocol which is a hybrid of the approach proposed in TEAR [ROY00] and TRAM [CHKW98]. The receiver keeps track of the congestion window adjusted similar to the congestion window at the sender. Based on this it calculated the highest sequence number it can receive and propagates this to the sender using a tree structure similar to that used in TRAM. The aggregated information is received at the sender and it uses the minimum sequence number it has received to set its congestion window.

The protocols discussed so far use a single sending rate and adjust it according to the feedback received from the receiver(s). There are certain classes of protocols which allow the sender to have different rates for different receivers allowing each receiver to get data at the best bandwidth it can support. This is especially important for multimedia content where the quality of the content is directly proportional to the bandwidth available. A number of protocols [VRC98, BFH$^+$00, JAA00, LPPA97] use variant of the basic approach proposed by Receiver-driven Layered Multicast [MJV96] (RLM). In RLM, the sender splits the data (in this case video) into multiple layers. Each higher layer carries more information than a lower layer. A receiver starts by subscribing to the first layer. If it is able to sustain the bandwidth for its current layer, it subscribes to the next higher layer. It does this until it subscribes to all layers or starts experiencing losses. This allows, each receiver to self determine the amount of bandwidth it can support and control the data transfer accordingly. In Layered Transmission Schemes [TPB97] (LTS) and TCP-Friendly Transport Protocol [TZ99] (TFRP) the above approach was modified slightly to reduce the amount of losses experienced. Rather than probing for bandwidth by joining higher layers, these schemes use the TCP model in [MSM97] to calculate the bandwidth they can support and join the corresponding number of layers directly. These schemes measure the RTT using a timestamped message sent to the sender which the sender replies to. This limits the scalability of these schemes.

Reliability

TCP detects losses using one of it many detection mechanisms (RTO, FR/R, PA, and SACK) and retransmits the packets to achieve reliability. While retransmission of lost packets to achieve reliability is obvious, accurately detecting lost packets is a challenge. Incorrect loss detection leads to spurious retransmissions and unnecessary reduction of TCP's sending rate. We will first look at proposals to mitigate the impact of spurious retransmissions and work related to evaluation of the loss detection mechanisms. Next, we will look at other protocols which achieve reliability by detecting and retransmitting lost packets.

Mitigating impact of spurious retransmissions

There are several approaches used to mitigate the impact of spurious retransmissions. The first approach is to optimize the TCP parameters to try and reduce the number of spurious retransmissions. The second approach is to modify the TCP loss detection mechanism itself to reduce the number of spurious retransmissions. Finally, the last approach is to identify the spurious retransmissions after they occur and to mitigate their impact. Lets look at each of these approaches below.

Optimizing TCP parameters to avoid spurious retransmissions

In [Pax97a], Paxson investigated the effect of changing dupack threshold on the number of spurious retransmissions. The data was collected by actively establishing approximately 20,000 connections sending 100KB of data between multiple machines and capturing the packet flow using tcpdump [JlM]. He found that increasing dupack threshold to 4 improves ratio of needed FR/R to spurious FR/R by a factor of 2.5. However, it also reduces the chance of detecting a loss by FR/R by 30%. Reducing the dupack threshold to 2 increases the number of FR/R by 65-70% but the ratio of needed to unneeded FR/R falls by a factor of 3. Since, reordering leads to spurious ack generation at the receiver, Paxson investigates the usefulness of delaying dupack generation at the receiver to avoid generating spurious dupacks. He found that for his dataset, reducing the dupack threshold to 2 and waiting for 20ms at the receiver results in same fraction of spurious retransmissions as a threshold of 3 but allows 65-70% more FR/R. While, this paper studies the impact of changing FR/R parameters on its ability to avoid spurious retransmission and its ability to detect packets by FR/R instead of RTO, it does not consider in much detail the impact of these changes on the performance (i.e. response times) of the connections. More recently [BRAB06] suggested similar changes to TCP. [BRAB06] recommends waiting for a certain time τ before reacting to the duplicate acks. The recommended value of τ is one

RTT. However, there is again no substantial evaluation of the effect of this recommendation on TCP performance.

In [AP99], the authors investigate the effect of changing several parameters related to RTO-estimation on the accuracy and timeliness of RTO detection. The data used for this study is same as that used for [Pax97a] and hence suffers from the same lack of diversity/representativeness. This study was done in 1999 when the most popular *minRTO* value was 1 second and the timer granularity was 500ms. They found that the *minRTO* and timer granularity has the most impact on the accuracy and timeliness of the equation while the other parameters related with the exponential moving average calculation of RTO had little impact on the overall performance. This study did present some insights on the parameters of the RTO equation but it did not evaluate the effect of changing these parameters on the response time of the connection. Eight years after this study, we find that timer granularity no longer impacts the performance of TCP loss detection (indeed, several current OSes use a 10ms timer). While the *minRTO* does limit performance in some OSes (Solaris and BSD), it is not a dominant factor for most connections. Instead, we find that the multiplicative factor, k, is quite significant and a low value of k can help many connections achieve close to their Best-Case reduction in response times. Finally, we explicitly model and evaluate the impact of timeliness and accuracy of RTO-based loss detection on overall connection response times.

Modifying TCP algorithms to avoid spurious retransmissions

In [FC05], the authors argue that TCP's loss detection is more of an inference rather than a known entity and hence should be viewed in a standard inference framework. They propose the use of Bayesian detector to identify whether a dupack is triggered by an actual packet loss or a network reordering event. The detector bases its prediction on the RTT samples measured before a loss event. They were able to predict 80% of the losses triggered by a single dupack using their detector with a false positive rate of 15%. They were able to test this detector on a small dataset and while they used an analytical model to investigate the behavior of their method under different circumstances; however, a broader network measurement based evaluation of the method needs to be done.

In [BHL+03], the authors suggest a very aggressive approach where they completely neglect the dupacks received but use a very aggressive adaptive timer (similar to RTO) to decide when to send a packet. The aggressiveness and effectiveness of the new algorithm is tested to a limited extent using simulations but a complete analysis of the effect of the changes made is not performed. [KM05] also suggested changing the RTO equation. In this paper, the authors suggest changing the RTO equation

to base it on the window size to optimize the throughput. They use the model suggested in [PFTK98] to analytically show the benefit of their proposal. They do not evaluate the proposal using any network measurements or simulations nor do they consider the effect of the proposed change on the number of spurious retransmissions.

Identifying spurious retransmissions and mitigating their impact

While SACK provided a good way to exactly identify which packets were lost in the network to avoid unnecessary retransmissions it does not help at all if there is indeed an unnecessary retransmission. In [FMMP00], the authors propose an extension to SACK called DSACK to detect spurious retransmissions. On receiving a packet that the receiver already has (i.e. the sender retransmitted unnecessarily), the receiver uses the SACK blocks to indicate this back to the sender. In [BA02] several responses to a DSACK notification were proposed. The simplest response proposed was to restore the original congestion window (congestion window before the spurious retransmission was detected). Apart from this, [BA02] also suggested strategies to adjust dupack threshold to avoid spurious retransmission due to FR/R. The different responses suggested were (i) increasing dupack by a constant, (ii) setting new threshold to average of current threshold and the number of dupacks caused leading to the spurious retransmission, and (iii) setting threshold to an exponentially weighted moving average of the number of dupacks received at the sender. RR-TCP [ZKFP03] is a more recent extension of the DSACK study which suggests mechanisms to avoid spurious retransmissions by adjusting the dupack threshold. It increases the dupack threshold on experiencing spurious retransmission and decreases it on experiencing RTO.

In [LK00], Ludwig and Katz propose the Eifel algorithm. It proposes the use of timestamp option in TCP [MMFR96] to differentiate among acks generated in response to the original transmission and a retransmission in order to detect spurious RTO and FR/R. It recommends restoring the congestion window state to its state prior to the spurious retransmission. FRTO [SKR03] was suggested to detect spurious RTO without the use of any TCP options. It monitors the acks after a timeout to determine if the timeout was spurious. The algorithm does not attempt to recover the congestion window on detecting the spurious retransmission but is aimed at avoiding sending out any more unnecessary retransmissions. The benefits of both these proposals have not been quantified.

Reliability in non-TCP protocols

We will now discuss the loss detection and retransmission mechanism used in different protocols to

achieve reliability. We will first look at three UDP [Pos80] based reliable protocols.

The first protocol, Reliable Blast UDP [HLYD02, HAE+03], provides reliability without congestion control. It does not use a per packet ack as done by TCP, instead acks are aggregated and delivered only at the end of the data transmission. The sender indicated the end of data transfer by sending a done signal. On receiving the done signal the receiver send a special ack consisting of a bitmap tally of received packets. Sender uses this to identify the lost packets and retransmit them. This process continues till all packets are transmitted.

The second protocol, UDP based Data Transfer [GG07, GGH+03] (UDT), builds its own reliability and congestion control. It has a set of API which allows both reliable data streaming and partial reliable messaging. In UDT, rather than generating an ack for each data packets, a selective ack is send periodically (every 0.01 second) and a negative ack (NACK) is generated explicitly when a packet is lost. The sender then retransmits the message to achieve reliability. In partial reliability scenario the sender will send a "message drop" signal to the receiver if the retransmission is not successful for a certain interval and the receiver will then marked those packets as received even if it is not actually received it.

Finally, UDP lite [LDP+04] does not provide loss detection and recovery but improves on UDP's perceived performance by allowing packets with a single bit error to be passed to the application instead of dropping it. This is helpful especially for multimedia applications. However, since several link layers themselves are known to drop a packet with bit errors in it the utility of this protocol is questionable.

Next we look at some TCP like protocols. Stream Control Transmission Protocol [SXM+00] (SCTP) as well as DCCP [KHF06b, KHF06a] use loss detection and recovery mechanisms similar to TCP. SCTP also provides an extension which allows partial reliability [SRX+04] similar to UDT. In DCCP, the congestion control and reliability mechanisms are separated. Hence it can also be tuned to allow different level of reliability requirement.

Detecting and recovering from losses is a popular way to implement reliability. However, this approach is not always timely as a considerable time is wasted in detecting the loss to begin with. To overcome this problem several schemes have been proposed to use Forward Error Correction [CC81] (FEC) schemes or redundancy to achieve partial or complete reliability. SmartTunnel [LZQL07] is one of the many protocols [PPM07, Bie92, APRT96, WCK03, RS99] that use FEC to achieve reliability. SmartTunnel uses a FEC encoder to generate redundancy packets for its data. The original packets and the redundancy packets are then send to the sender through different paths. (other schemes may choose

to send all packets on the same path). If any data packet is lost, the receiver can use the redundant packets to rebuild the original data stream.

2.2.2 Past Work on TCP Analysis Tools and Methodology

Several tools have been developed to understand the behavior of TCP. These tools can be roughly divided into two categories. The first category consists of tools which analyze a pre-captured trace of TCP. These are referred to as passive tools as they do not actively inject packets in the network. The second category belongs to the active tools which actively instantiates TCP connections to study their behavior. In this section, we present both these types of tools. We also discuss some tools which are developed to remotely identify the Operating System for a machine. These tools in conjunction with the active or passive analysis tools can provide additional insights in the performance of TCP implementations.

Passive Trace Analysis Tools

Tcptrace [tcpc] is perhaps the most widely used among the many tools available for passive analysis. Tcptrace is able to identify several characteristic of a flow like throughput, elapsed time, number of bytes/segments send or received, RTT, number of out-of-order segments, advertised window, and packet duplication. However, tcptrace, like many other tools, does not maintain enough state to accurately identify and classify TCP packet losses. *tstat* [MCC] is another tool similar to tcptrace. It captures statistics such as number of IP addresses in the trace, and different options on SYN packets in addition to everything else that tcptraces does. However, *tstat* is also limited in its analysis ability as it does not perform any loss classification.

Using heuristics in the analysis is a popular approach used by several tools to overcome the limitations of the above stateless tools. In [ZBPS02], the authors propose *TRAT*, a passive tool to study TCP flow rates and identify the probable causes that limit TCP throughput. Instead of directly measuring the characteristics of a flow they develop heuristics to estimate the RTT, MSS and other characteristics of the flow. This approach allows them to analyzed connections even when they do not have access to both the data stream and ack stream of a flow. While they do study flow behavior, the limited state information they have do not allow them to study a connection's behavior at the granularity of individual congestion events. They also do not study exactly which TCP mechanism limits the performance

and whether it could be improved.

Next step in improving the analysis accuracy and granularity was achieved by including more state in the analysis tools. The first category of such tools used additional state to identify spurious retransmissions. Both [AP99] and [LK00] were designed to deal with timeout-triggered spurious retransmissions. While [AP99] relies on keeping enough state to measure the time difference between the retransmitted segment and the ACK to identify spurious timeouts, [LK00] requires end-system cooperation to implement a new algorithm called Eifel, which actively includes timestamps to detect spurious timeouts. Both these approaches only work for timeout-based spurious retransmissions. Allman Et. al presented an improved algorithm called *LEAST* [AEO03] for passive estimation of unnecessary timeouts (for Reno implementations) or any unnecessary retransmission in presence of SACK blocks. We find that their method underestimates unnecessary retransmissions in Reno implementations because they do not address additional retransmissions in Fast Retransmission/Recovery. Further, the limited state maintained is insufficient for tracking unneeded retransmissions when duplicate ACKs are lost.

A significant improvement in the level of details used for TCP analysis was achieved in the *tcpflows* [JID+04] tool which uses a state-machine based approach for analysis of TCP connections. This state-machine design is based on RFC specifications for congestion response, retransmissions, and RTO calculations. *tcpflows* can perform passive analysis of traces taken anywhere in the network and attempts to characterize the causes of packet losses using inferences of RTO and the sender's congestion window. Though this tool was a significant advance in passive analysis, we found that this method has practical limitations because widely used TCP implementations vary significantly from the RFC specification. Furthermore, since the primary purpose of [JID+04] was not to study packet losses in detail, their analysis tool is limited in the granularity with which OOS segments are classified. The results from their method (especially the RTT calculation and the subsequent RTO calculation) are dependent on the frequency with which RTT is measured (per packet vs. per flight), and the inferences necessary to track the sender's congestion window. All of these details vary across TCP implementations. To overcome these problems, in our analysis, we rely on OS-specific state machines to more robustly infer TCP sender state in passive analysis. TCP *mystery* [KKB+04] is another state-machine based tool which identifies loss events and classify them as necessary or unnecessary. This tool uses a subset of the algorithms used in *tcpflows* and hence suffers from the same limitation of being OS agnostic.

OS-specific analysis is not a new idea. As early as in 1997, Paxson implemented a stateful implementation specific analysis in his tool, *tcpanaly* [Pax97b], for passive analysis of traces at the end

system. The primary limitation of this tool is that it has not been extended to handle traces taken in the middle of the network. Since the analysis was performed on end system traces, there was no need to address several practical challenges such as packets lost between the trace point and end system. Further, the analysis did not have to *infer* the specific TCP implementation characteristics because the end system OS was known in advance. Given the above reasons and the significant pace of changes to TCP implementations since the time the tool was developed, we believe that *TCPdebug* represents a substantial advancement.

Estimating a connection's Round-Trip Time (RTTs) is fundamental to estimating the retransmission timeouts (RTO), which in turn impact a connections performance. We will next look at some of the passive techniques used for estimating a connection's RTT.

Several analysis techniques do not rely on a per-packet RTT estimate. These techniques simply obtain a single estimate of the connections RTT and use it throughout the connection's lifetime. One of such techniques was proposed in [JD02]. In [JD02], the authors present a simple passive method for getting a single estimate for a connection's RTT using the three-way TCP handshake [Pos81] and first few flights of data. Their method works even if the trace is not bi-directional. For traces containing the SYN, the RTT is measured as the difference between the SYN and the ack generated in response to the SYN-ACK. For traces which do not see the SYN, the proposed technique works only when the flow has at least 5 segments and 4 of these are of MSS size. The initial congestion window for the connection is assumed to be one or two packets. They apply heuristics to divide the first 5 segments into two flights and measure the RTT as the difference between the send times of these two flights. While this technique provides an RTT estimated even when we have only one direction (either the ack or the data packets) of a connection, it provides only one estimate instead of a running series of estimates of RTTs throughout the lifetime of the connection. In [VLL05], the authors propose a slightly different approach for estimating RTT both in bi-directional as well as uni-directional traces. For bi-directional traces, the approach relies on using timestamps included in the packets to match the data segments and their acks and estimate the RTT using these. For uni-directional traces, the authors use autocorrelation techniques to detect patterns caused by self-clocking that repeats every RTT. Like [JD02] this method also provides only a single RTT estimate for a connection.

The more interesting problem is that of estimating an RTT for every packet in a connection. This provides more details about the evolution of RTT throughout a connections lifetime and hence will provide a better estimate of a connection RTO as well. Several stateful and heuristic based approaches

have been proposed to do this. The main challenge for these approaches is to maintain enough state to match the data and ack packets to estimate the RTT and at the same time to get a robust estimate of RTT even in presence of loss. When an ack arrives for a segment that has been retransmitted, there is no indication whether the ack is for the original packet transmission or for the new retransmitted packet. This result in an ambiguity about the RTT inferred from this ack. [KP88] proposed a simple solution to this problem. They recommend neglecting the RTT obtained using this ack and not updating the RTO. The RTO is updated only when a packet is acknowledged without an intervening retransmission. This algorithm is used in all the popular implementation of TCP. Implementing this algorithm and using a Finite State Machine (FSM), the authors of [JID+04] proposed tracking the correspondence between the data and ack packets as follows. The RTT is calculated in two parts the forward path RTT and the reverse path RTT. The forward path RTT estimation is simple. For each data packet going in the forward direction its corresponding ack is tracked and the forward RTT is estimated as the time difference between the data and ack packet. The reverse RTT estimation is slightly more complicated. The FSM tracks the congestion window at the sender and by doing this it estimates the data packets that would be triggered by each ack reaching the receiver. By tracking the acks and the corresponding data packets triggered by it the reverse RTT is estimated as the time difference between the ack packet and the data packet. The problem with this approach is the inaccuracies in the RTT estimation resulting for the inaccuracies in tracking the senders congestion window. This paper found the error in RTT estimation because of these inaccuracies to be limited to less than 10% for 90% of the flows.

Active Analysis Tools

While passive analysis tools are beneficial as they provide access to a large quantity of real world data, they do not allow controlled interaction with the network to study its impact. We will now look at some of the active tools developed to study either TCP or network behavior using controlled active experiments.

The first set of tools we look at were developed to measure the network properties and not really the TCP properties though they may rely on TCP to do so. In [Sav99], the authors describe an active tool, *Sting*, for measuring end-to-end path properties. This tool overcomes problems with using ICMP packets (such as blocking, rate-limiting, and spoofing) to measure the network properties. The tool estimates one-way end-to-end network loss rate by careful manipulation of TCP behavior. Bolot [Bol93]

and Sommers [SBDR05] used active probes to measure the network loss rate. In [BS02], the authors design a tool for actively measuring reordering on a network path. This tool exploits the relative sequence number spacing between segments transmitted and can not be easily adapted for passive monitoring of the network. All of these tools are not designed to be generic enough to be used for studying other TCP characteristics.

In [PF01],, the authors developed a versatile tool called *TBIT* which emulates TCP behavior to test the behavior of the network or the end-host TCP stack. *TBIT* queries the TCP stack of the end-host to explore its deployment of different TCP features. The end-host behavior that is tested includes SACK information generation, MTU discovery, ECN and other TCP options. Their key observations were that (i) SACK is prevalent in two-thirds of servers and nine-tenth of clients, (ii) DSACK is supported by 40% of servers, (iii) ECN is not prevalent, and (iv) 1460 bytes is a popular size of MSS. The active nature of the tool and the fact that it requires the other end of the connection to run a webserver limits the scalability of this method to study network behavior. In our research we use *TBIT* to extract several default parameter settings of TCP implementations, as well as to validate our tool *TCPdebug*.

End Host Operating Systems Fingerprinting

As we have emphasized earlier, estimating the source Operating System (OS) of a connection is an important step in accurately understanding a connections behavior. In this section we will look at some of the tools which are used for identifying the source OS for remote machines. This information is also of interest for several other reasons such as intrusion detection, servicing OS specific content, maintaining inventories, building representative models and, in our case, verifying the results reported by our tool.

There are several methods proposed for identifying the OS passively. [Zal06] relies on the difference in the option field of SYN, SYN-ACK and RST packets for different OSes to identify the source OS. It maintains a fingerprint of all OSes option fields and then compared the observed packets to find a match. While, this tool is capable of giving an exact answer most of the time, its main drawback is that it has to keep updating the fingerprint list to make sure it matches the latest change in the TCP stack for any OS. It is also not able to predict the OS if the option field is slightly modified (either by the sender or an intermediate router). To overcome this drawback of rule-based fingerprinting techniques, [Bev04], developed a Bayesian classifier to passively infer a host OS even if the option fields are scrambled a bit.

There are also several active fingerprinting tools which explicitly set-up a connection with an oper-

ating system to identify its OS. While these techniques are usually more accurate, their active nature limits the scalability of these techniques to identify the OS for a large number of hosts. [Lyo97] is probably the most popular of these active tools. It sends a series of IP and TCP packets to a host and examines its response to obtain a fingerprint for a given host. It then compares this to a list of known fingerprints for a large number of OSes to identify the OS. SYNSCAN [Tal04] is similar to [Lyo97] in its operation though the level of details it provides for the end-system differ.

2.2.3 Past Work on TCP Analysis and Modeling

In this Section, we review work on TCP analysis and modeling. We will first present some work on analysis of TCP loss detection and recovery mechanism and evaluation of various DBCEs . Research related to measurement of various TCP statistics and performance is reported next. Finally, we conclude with a look at TCP performance modeling, including studies on TCP loss modeling.

TCP Measurement and Analysis

Understanding the cause of retransmissions and the RTTs observed for connection is essential to optimize TCPs loss detection mechanisms. For e.g. if the RTT variability is really high the algorithm used for RTO estimations would have to be modified to account for it. Similarly if the number of spurious dupack based loss detection is high then that part of the algorithm has to be modified. In this Section we will look at past work related to analysis of out-of-sequence segment and RTT.

Analysis of Losses

Understanding network loss properties would help in designing better loss detection algorithm. There is considerable work on characterizing the losses on a network path using active methods. [Bol93] and [Pax97a] studied the correlation between packet losses. Both studies found that the conditional probability of seeing a packet loss if previous packet is lost is considerably high. Similar observations were made in [YMKT99], which developed a model for the loss patterns as well. In [Pax97b] and [ZD01] the authors studies the duration of time over which all packets were lost (loss periods). In [ZD01], the authors found that 95% of all loss periods where smaller that 250ms. In [Pax97b], the author found that the loss periods span an order of magnitude. While 10% of the loss periods lasted less than 33ms there were 10% of loss periods which lasted more than 3.2 seconds.

While the network loss properties helps algorithm design, a post-hoc analysis of current TCP implementations would help even more in improving the implemented designs. Understanding the frequency of occurrence for different loss detection algorithms as well their efficiency is the first step in modifying the algorithms. In [JID+02], the authors used *tcpflows* [JID+04] to study the retransmissions in over 17 million connections from the middle of the network. They found that 7-25% of all out-of-sequence segments occurred as a result of packet reordering in the network. They also found that 9-19% of retransmissions seen in the traces were unnecessary. [AEO03], presented a method, referred in this dissertation as *LEAST*, to improve the loss estimation for TCP by identifying unneeded retransmission. They found that on an average 33% of all retransmission in TCP Reno and 2% of retransmission for SACK TCP were unnecessary. They also investigated the burstiness of losses and found that 60% of all losses where single loss events.

Analysis of RTTs

Several studies have looked at the distribution of Round-Trip Times (RTTs) observed in the internet. We look at results from several of these studies below.

- In [JD02], the authors present a technique to obtained one RTT measurement per connection. 95% of RTT were seen to be less than 500ms. 75-90% of RTTs were less than 200ms. This study present interesting statistic about the distributions of RTT in the network, however, it does not provide any insight about the variability in RTT on a per packet basis.

- In [JID+02], the authors used *tcpflows* to study the distribution of RTT in over 17 million connections. 80% of connection in their dataset had a RTT of over 100ms. While the tool did calculate per packet RTT, this paper did not study the overall RTT variability in the trace but compared the RTT obtained using the Syn/Syn-Ack packet to the other RTTs in the flow and found that the Syn/syn-Ack RTT underestimates the RTT observed by most of the packets for a trace. This observation was later verified by [AKSJ03].

- In [All00], the author presents the distribution of per-packet RTTs obtained using tcptrace [tcpc] for over 500,000 connections collected at a web-server at NASA's Glenn Research Center. They found that 75% of connections had an average RTT greater than 100ms and 40% of connection had an average RTT greater than 200ms. While the RTT distributions varied from a few ms to a few seconds, 85% of RTT were between 15-500ms. They also found that for their dataset 90% of connection had a median RTT which was less than twice the minimum RTT.

- A much more detailed study of distribution of RTT and its variability was reported in [AKSJ03]. In [AKSJ03], the authors studied the per-packet RTT distribution of over 1 million connections. They also found that the RTT distribution varies between a few ms to 200s. In their dataset 55% of connection had RTT greater than 100ms and 35% of connections had an RTT of more than 200ms. The also found that in 70% of the connections the median RTT was less than twice the minimum RTT and 25% of connection see a median RTT which is 2 to 10 times the minimum RTT.

TCP Modeling

To understand (and predict) TCP performance, several detailed models have been proposed for TCP. We will briefly describe some of these general TCP models as well as some models associated with the loss process.

TCP performance modeling

In [MSM97], the authors develop a model to describe TCP's throughput as a function of connections RTT and loss rate. It assumes that the connection always has data to send (bulk transfer) and that the losses are either random or periodic. For a loss rate of p the throughput B is given as

$$B(p) = \frac{MSS}{RTT}\frac{C}{\sqrt{p}} \tag{2.1}$$

where, MSS is the packet size, RTT is the connections average RTT and C is a constant which depends on whether losses are periodic or random and whether delayed ack is employed or not. The model was shown to match the performance quite well in a low loss rate scenario. However, the model assumes that all TCP losses are recovered by an FR/R and no RTO based retransmissions are seen. Considering that most retransmission are RTO based and its affects the performance much more than FR/R this assumption is the key cause for the model not matching the actual observed performance in many cases. This was the first paper to show that TCP's performance is inversely related to a connection RTT and the square root of its loss rate.

To overcome the limitations of [MSM97] a new model was proposed by Padhye Et. al in [PFTK98] which models both the FR/R based retransmission and the RTO based retransmission. [PFTK98] present a simple model for TCP Reno's throughput as a function of the loss rate of a connection. For

a given loss rate p the approximate model for throughput B is

$$B(p) = \frac{1}{RTT\sqrt{\frac{2bp}{3}} + T_o\ min(1, 3\sqrt{\frac{3bp}{8}})\ p(1+32p^2)} \tag{2.2}$$

where, b is the delayed ack threshold (2 if every second packet is acked), RTT is the average RTT of the connection and T_o is the timeout interval for a RTO based retransmission. This model matches the actual performance for connections with large number of losses quite well. This model also confirmed that TCP's performance is inversely related to its RTT and square root of its loss rate for low loss rate scenarios.

While the above models worked reasonably well for large connection with a sustained loss rate, they failed to model the performance of short connections which see few or no losses. The performance of such flows is dominated by start-up effect of TCP. [CSA00] extended the model in [PFTK98] to capture the start-up effect. They present a model for a connection's latency as a function of its transfer size, RTT and loss rate. The model is developed in two stages. First the connection establishment time is models in terms of the forward and reverse loss rates for the connection. The second stage models the data transfer time as sum of the time spend in slow start, time to recover from first loss, time required to transfer rest of the data, and the time delay introduced by use of delayed acks. This model was shown to be more accurate than that mentioned in [PFTK98], especially for small connections. For large connection the difference in the performance of these models was negligible.

TCP loss model

The pattern of losses experienced by a connection can severely impact its performance. For e.g. a burst of packet losses experienced by a Reno TCP connection results in the later losses to be detected using the much costlier RTO if the first loss is detected using a FR/R. On the other hand for connections using SACK TCP these losses can be recovered in a few RTTs.

We will first briefly look at the loss models assumed by some of the popular studies. [MSM97] was one of the earlier papers assuming a periodic loss model to derive a simple stochastic model for TCP behavior. (The model assumed a steady state loss probability "p" and assumed that every $1/p$ packet is dropped.) Cardwell et.al [CSA00] also assumed an independent loss model while modeling TCP latency. Other studies which assume an independent loss model are [ARA00, Kum98, LM97]. [PFTK98], one of the most widely cited and influential of TCP modeling papers, assumes that packet losses are independent across flights but highly correlated within a flight. [MGT99] modeled the losses

as a Poisson stream of arrivals. [AAB00] used a correlated loss model to model TCP performance. A two state Markov model was used in [ARA00, AT99].

While, the above studies did explore several popular and expected loss models and its impact on TCP performance, the interesting question to be considered is which of these loss patterns are actually present in the Internet. Next we will look at some empirical studies aimed at identifying the loss patterns in the network.

In [Bol93], Bolot et al. studied the conditional probability of losses using UDP packet streams. While they did find that the conditional loss probability of the second packet being lost given that the first was lost is slightly higher, the overall loss process was argued to be best described as random.

The use of UDP packets may prohibit using the insights from the above paper for TCP analysis. Packets pattern in TCP is very different than the UDP packet patterns and hence to understand the loss patterns that would be experienced by TCP we need to look at a TCP based analysis study. In [Pax97a], Paxson used TCP connections to measure loss rate on the network. Both end points of the connection were monitored to identify the packets which were dropped in the forward as well as reverse direction. He found, that the second packet was more likely to be lost if the first packet is lost. This is one of the few studies which use actual TCP connections to study the loss process on the network. Similarly, [BSUB98] used a similar approach in which it used a client-server program to measure losses in both the forward and reverse direction of a transfer. The client sends a stream of packets to the server and the server echoes them back. They found that the loss process was very bursty but the number of packets lost in a burst had a long tail distribution. A large number of packets were lost in a small number of bursts. They showed that while there was dependence among losses the degree of dependence varied a lot.

A more detailed study for loss patterns where the losses were modeled as a multi-state process were reported in [YMKT99] and [ZD01]. In [YMKT99], the authors looked at conditional loss probabilities of losses at different timescales. They collected 128 hours of end-to-end unicast and multicast data to analyze losses. 76 hours worth of data over 38 runs was found to have stationary losses and was considered for analysis. They evaluated three different models: Bernoulli, 2-state Markov chain model, and k^{th}-order Markov chain model. They found that 18% of the runs matched Bernoulli model (i.e. the losses were independent). 26% of the runs could be modeled by a simple 2-state Markov model while 55% were best modeled by a Markov model with more then two states. [ZD01] studied loss behavior on a path using Poisson probes. They found that only 27% of the traces could be modeled to have

independent losses. They then combined all "near-by" losses into a single loss *episode*. The idea was that all losses in a loss episode would be caused by the same congestion event. A test of independence on these loss episodes showed that 64% of the traces could be modeled as independent events. Thus it shows that the congestion events (loss episodes) are independent but the losses within the event are bursty. Within a loss episode they found that 40% of the losses could be modeled as independent, 49% out of the rest fit a 2-state Gilbert model, 9% of the rest were modeled by a 3-state Gilbert model and 1% by 4-state Gilbert model.

CHAPTER 3

TCP*debug*

Truth is what stands the test of experience.

— Albert Einstein (1879–1955)

If the only tool you have is a hammer, you tend to see every problem as a nail.

— Abraham Maslow (1908–1970)

In this Chapter we describe our tool *TCPdebug* developed for the passive analysis of TCP packet traces. The purpose of this tool is to provide more complete and accurate results for identifying and characterizing out-of-sequence TCP segments than those provided by prior tools such as tcpanaly, tcpflows, LEAST, and Mystery [AEO03, JID+02, JID+04, Pax97b].

The methodology presented here classifies each segment that appears out-of-sequence (OOS) in a packet trace into one of the following categories: network reordering or TCP retransmission triggered by one of—timeout, duplicate ACKs, partial ACKs, selective ACKs, or implicit recovery. Further, each retransmission is also assessed to determine whether it was needed or not.

This tool provides significant improvement over current state-of-the-art tools for passive analysis of TCP. One of the crucial factors that limits the accuracy of prior tools is that different TCP implementations (for different operating systems) have unique parameters (e.g., timer granularity, minimum RTO, duplicate ACK thresholds, etc.) or algorithms that influence what can be inferred about out-of-sequence segments. Our approach is to analyze each TCP segment trace from the perspective of each of four implementations (Linux, Windows, FreeBSD/Mac OS-X, and Solaris) and determine which specific implementation behavior best explains the out-of-sequence segments and timings observed in the trace.

We validate *TCPdebug* using controlled lab experiments and real world traces. Using *TCPdebug* we analyze packet traces of more than 52 million Internet TCP connections collected from 5 different vantage points across the globe. and present the results.

Given that prior tools have been shown to provide reasonably good results, one might question whether the additional completeness and accuracy justifies creating a new tool. We believe that it does so for the following reasons. First, as discussed in Chapter 1 and 2, each of these prior tools has particular strengths and weaknesses for analyzing some aspect(s) of out-of-sequence segments but none deal with all aspects at the desired level of accuracy. Second, a number of potential uses for the analysis results are much enhanced when they are accurate. For example, while the TCP loss detection and recovery mechanisms are quite mature and unlikely to undergo major design changes, there may still be opportunities for "fine-tuning" to improve certain cases. Prior studies have indicated that retransmissions are triggered much more frequently by timeouts than by duplicate ACKs, and that significant numbers of retransmissions are unnecessary. Having accurate data on issues such as these is necessary for quantifying the potential benefits of fine-tuning these TCP mechanisms. Another example where accurate results from analysis of out-of-sequence segments are needed is in validating and evaluating models of TCP performance; such models are based on the evolution of TCP's congestion window as it changes along with retransmissions, and according to how the need for a retransmission was detected (timeout or duplicate ACKs) [CSA00, FF96, PFTK98]. An inaccurate classification of such retransmission can mislead such evaluations.

In this chapter, we elaborate on the methodology already discussed in Chapter 1. We then present validation results for the tool. Next, through real world trace analysis we show the impact of the details incorporated in this tool while also comparing it to other related tool.

3.1 Passive Inference of TCP Losses

A packet trace of a TCP connection is a time-ordered sequence of data segments and acknowledgments (ACKs) exchanged (and observed at the trace-collecting monitor) between the TCP sender and the TCP receiver. Our objective is to find out, given a packet trace, which TCP segments were lost in the network. Below we describe our passive loss inference methodology.

3.1.1 Passive Loss Inference Methodology

TCP uses a well-known combination of detection and recovery mechanisms to deal with packet losses—retransmission timeouts (RTOs), fast retransmit/recovery (FR/R), partial acks (PAs), and selective acks (SACKs). Each of these mechanisms is used to detect and *retransmit* segments that

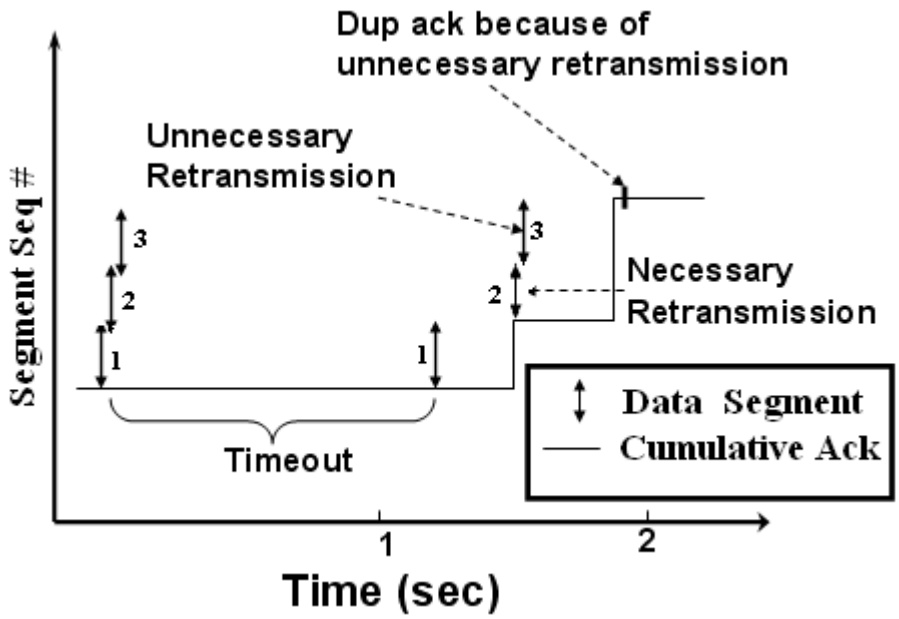

Figure 3.1: Implicit TCP Retransmission. Segment 1 is retransmitted due to a timeout. Segment 2 is a necessary implicit retransmission while segment 3 is an unnecessary implicit retransmission triggered simply due to TCP's recovery mechanism.

are perceived to be lost [APS99, BAFW03, FF96, FHG04, MMFR96, Ste93]. Since TCP is actively detecting losses, the simplest way to identify losses in a trace would be to count the number of packets which are retransmitted in a trace. We explore this and several similar approaches to identifying losses below and highlight their drawback and propose solutions to them.

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 somewhere between the monitor and the receiver [KSE+04]. However, this approach can lead to over-estimation of losses as illustrated in Fig 3.1, which depicts part of a TCP connection selected from the *unc* trace. Segment 3 is retransmitted during a post-timeout period, although the original transmission was successfully received (as is confirmed by the subsequent duplicate ACK). In [AEO03], Allman proposes an algorithm, *LEAST*, that accounts for such unneeded retransmissions in computing the true loss rate of a connection, by simply subtracting the count of duplicate ACKs that are received after timeouts. However, such an approach does not help identify *which* retransmissions were unneeded.

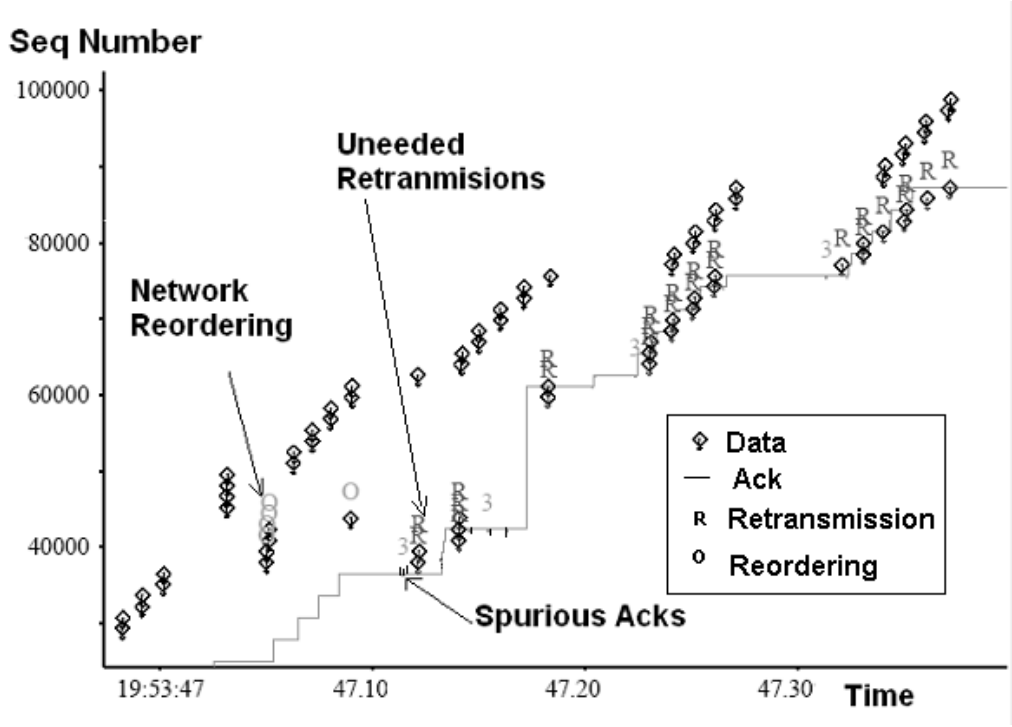

Figure 3.2: Unneeded Retransmission. This visualization of a real connection from the *unc* trace shows how a single occurrence of network reordering results in some spurious duplicate ACKs, that ultimately trigger 64 subsequent phases of unnecessary retransmissions.

Note that while segment 3 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 from all segment retransmissions, it is important to *track the explicit triggering of TCP's loss detection mechanisms—namely, RTO, FR/R, PA, and SACK.*

Why not simply track TCP sender state?

It turns out that even simply tracking the triggering of loss detection/recovery mechanisms in a TCP sender—as is done in [JID+04]—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 (RTO, FR/R, PA, SACK). Fig 3.1 illustrates this for segment 2, 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 *identify* ***implicit*** *retransmissions that are* ***needed*** *for recovering from packet losses.*

Note that if history about all previously transmitted data packets is maintained, then the ACK stream can help identify such retransmissions (in Fig 3.1, the cumulative ACK received after retransmission of segment 1 indicates that segment 2, which was previously transmitted, was lost).

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 3.2, which again depicts part of a TCP connection selected from the *unc* trace (and visualized using the *tcptrace* utility [tcpc]), 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 *identify* ***explicit*** *retransmissions that are* ***not needed*** *for recovering from packet losses.*

Such unneeded explicit retransmissions are not identified by *LEAST* [AEO03]—our analysis of Internet TCP connections in Section 3.3 shows that more than 90% of unneeded segment retransmissions in the Internet may occur due to explicit loss detection/recovery actions by TCP. Note that an explicit retransmission can be identified as unneeded if an ACK is received within a fraction of the connection's minimum RTT after the segment is retransmitted—we use a fraction of 0.75 in our analysis.

Basic Approach

As reasoned above, if the timing and history about all previously transmitted packets are maintained for each connection, then the ACK stream can help achieve each of the three goals outlined above. Based on this intuition, our basic approach for passive inference of TCP losses is to: **(i)** replicate partial state machine for a TCP sender that uses the data and ACK streams 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) FR/Rs, (iii) PAs, (iv) SACKs, and (v) implicit. Furthermore, each

retransmission is also classified as *needed* or *unneeded*. Fig 3.3 depicts this classification taxonomy.

A similar approach is taken in [JID+04] for developing a tool, *tcpflows*, for studying congestion window behavior of TCP connections. However, due to the different objective, *tcpflows* does not focus on accurately identifying and classifying segment losses. In particular, it classifies retransmissions into RTO-triggered, FR/R-triggered, RTO-recovery, and FR/R-recovery. It does not analyze implicit retransmissions (RTO-recovery) to see if these are needed or not. In Section 3.3, we show that up to 30% of needed (and up to 40% of unneeded) segment retransmissions in the Internet occur during such an RTO-recovery phase.

3.1.2 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 can differ across OSes. This results in two main challenges in implementing our basic approach. First, the difference in implementations on different OSes necessitates that we implement different analysis tools 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 trivial to replicate these mechanisms in our OS-specific analysis programs.

This challenge has not been addressed in *tcpflows* [JID+04], which replicates only the TCP standards specification [APS99, BAFW03, FHG04, MMFR96, PA00]. *tcpflows* has been validated only against connections with FreeBSD senders (that follow the standards closely). Our analysis of general Internet

connections in Section 3.3 reveals that more than 80% of real world connections involve either a Windows or Linux sender. More importantly, we find that analyzing such connections with a FreeBSD-based tool can introduce significant inaccuracy in identifying and classifying TCP losses.

Our Approach:

We consider and incorporate 4 prominent OS stacks in our analysis tools—namely, Windows XP, Linux2.4.2, FreeBSD 4.10, and Solaris. The TCP sender stack in MacOS is identical to the FreeBSD stack; hence this OS is also implicitly incorporated in our analysis. We used the popular passive fingerprinting tool, *p0f* [Zal06], in order to identify the sender OS in three of our traces (*unc*, *ibi* and *jap*)—we found that nearly 90% of TCP connections originated from one of these 5 sender-side OSes.

We extract sufficient details about the implementation of loss detection/recovery in the above OS stacks using three different approaches: (i) by studying the source code when publicly available, (ii) through direct communication with OS Vendors, and (iii) by using an approach similar to the *TBIT* approach described in [PF01] (in order to infer non-public details). To extract OS information using *TBIT* we install all four above-mentioned OSes on experimental lab machines and run the Apache web-server on each machine. We then implement an application-level TCP receiver (by borrowing from the *TBIT* 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 FR/Rs, RTOs, PAs, and SACKs). We then use the manner in which the server responds to the ACK stream to infer 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 Table 3.3 and in [RKS05]. 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 and classify each retransmission event.

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, duplication, 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 either (i) detecting if an OOS segment appears within a fraction (0.75) of the connection's minimum RTT after the segment with the next higher sequence number, or (ii) detecting reordering in the *IP-id* field of packets seen from a given TCP source. Finally, we infer network duplication of packets by detecting repetition in the *IP-id* field of reoccurring segments seen from a given TCP source. We remove such duplicated OOS segments from further analysis.

[1]The RTT measured at the monitor (monitor-receiver-monitor) is less than that measured at the sender (sender-receiver-sender). We address this issue (i) by estimating the monitor-sender-monitor delay during the initial three-way SYN/SYN+ACK handshake, and (ii) by adding this quantity to each estimate of the monitor-receiver-monitor delay, in order to obtain the sender-receiver-sender RTT. The initial sub-RTT obtained from the SYN/SYN+ACK exchange is a good approximation of the minimum monitor-sender-monitor delay [AKSJ03]. If subsequent delays on this sub-path vary significantly, 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. Therefore, a smaller-than-actual value of RTO would simply lower the threshold and still be able to correctly identify retransmissions that occur due to timeouts.

Non-availability of SACK Blocks in Traces

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 the absence of these blocks, the monitor will fail to accurately identify the cause of these retransmissions.

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 evaluate this heuristic using the *unc* and *jap* traces. We first run our analysis tools with the SACK blocks available and log all OOS segments that were SACK-triggered. Then we remove the SACK blocks from these traces and run the tools with the above heuristic. The heuristic-based analysis identified all of the OOS segments identified as SACK-triggered by the analysis based on SACK blocks; however, it also marked 6.9% and 15.3% of the unexplained events as being SACK-triggered, in the *unc* and *jap* traces respectively. Our analysis of Internet TCP connections in Section 3.3 shows that only a small fraction (less than 7%) of all OOS segments are SACK-triggered—the possible overestimation introduced by the above heuristic, therefore, is not significant.

3.1.3 Summary of Our Methodology

Our methodology for reliably inferring and classifying TCP losses can be summarized as follows.

1. We first extract the implementation details of four prominent TCP stacks (Windows XP, Linux 2.4.2, FreeBSD 4.10 (MacOS), and Solaris) using the approaches described in Section 3.1.2. These details primarily include the initial RTO, the minimum RTO, the RTO estimation algorithm, the number of duplicate ACKs that trigger FR/R, and the responses to partial ACKs and SACKs. In addition, some OS-specific peculiarities are included—for instance, if a segment with options

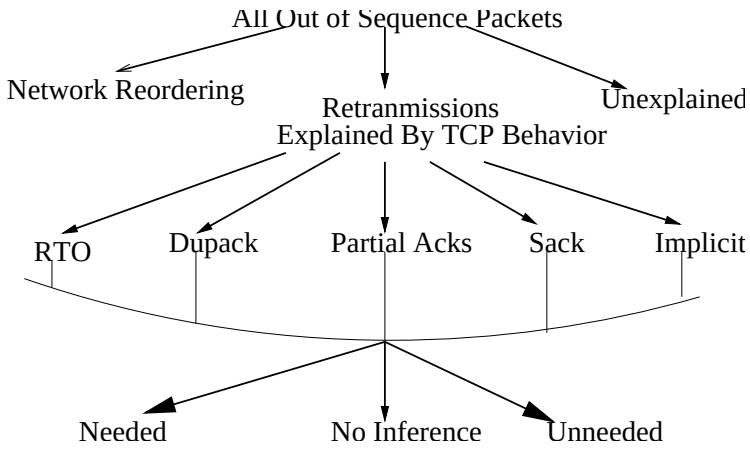

Figure 3.3: Classification Taxonomy.

fields is to be retransmitted in FR/R, some versions of Windows transmit a small packet equal to the size of the options field.

2. We then replicate the loss detection/recovery 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. For instance, on detecting an RTO-based retransmission, the state machine will enter an "RTO-recovery" state. A new RTO is calculated, any pending RTT measurements are canceled and the SACK block, if present, is cleared. The machine exits this state on receiving an ACK for the highest packet that was in flight when RTO was detected.

3. We then augment these machines with extra logic and state about all previously-transmitted packets, in order to classify retransmissions as needed or unneeded and 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 and classify all of the observed OOS segments. In case more than one machine matches this criteria, we check if the classification of each OOS segment is the same in each machine. If not, we discard the connection. We also discard the connection in case none of the machines can explain each OOS segment.

Our methodology classifies all OOS segments that appear within the packet trace of a TCP connection, according to the taxonomy depicted in Fig 3.3.

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
Japan-155Mbps-2004 (jap)	4h	1.93 Mbps	0.3 M	3.5 G	3.7 M
UNC-wireless-2005 (wls)	178h	1.58 Mbps	20.2 M	126.9 G	157.6 M
UNC-wired-2005 (wrd)	178h	2.18Mbps	6.8M	175.1 G	217.5 M
Liepzig-1Gbps-2003 (lei)	2h 45m	9.53 Mbps	2.4 M	11.8 G	17.3 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 3.1: General Characteristics of Packet Traces. The trace name indicates the location, link speed, the year data was collected and the acronym used for the trace. The remaining columns describe the duration of the trace, average load on the link, and the number of connections, bytes, and packets.

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. The source code is available online via [tcpa].

In the next two sections, we validate our methodology and compare its performance with past work.

3.2 Validation

Our primary validation method is to compare the output from the analysis tools for TCP connections where the "ground truth" about the classification of each OOS segment is known. To do this, we modified the TCP Behavior Inference Tool (*TBIT*) [PF01] in order to observe the sender's responses under additional controlled conditions. We supplement this validation by comparing the determination made by the tools for identifying a specific OS implementation with the results from *p0f* [Zal06] - a well-known passive fingerprinting tool.

Below, we first describe the sources of Internet packet traces used throughout this Chapter and then we will present our validations. methodology.

3.2.1 Data Sources

Table 3.1 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 [abi] are collected from a backbone link of the Internet-2 network (Abilene); the *jap* trace [jap] is collected off a trans-Pacific link connecting Japan to the US by the MAWI working group; the *unc* trace is collected at the campus-

to-Internet links of the University of North Carolina; and the *wls* and *wrd* traces are captured inside the UNC campus. The *wls* trace captures wireless TCP connections from over 600 wireless access points while the *wrd* trace captures just the wired network. The *lei* [lei] traces are collected at the campus-to-Internet links of University of Leipzig; the *ibi* trace captures traffic served by a cluster of high-traffic web-servers (mirror for ibiblio.org). All traces except the one from the link to Japan were collected using Endace DAG cards [dag]; the *jap* trace was collected using tcpdump [JlM]. The *abi* and *lei* traces are from the NLANR repository. The *unc*, *ibi*, and *jap* traces include TCP options as well. Our trace set is fairly diverse in its geographic location, proximity to TCP senders, as well as types of users represented.

	All Connections			OOS Connections			All OOS Segments Explained		
Trace	# Conn	# Bytes	# Packets	# Conn	# Bytes	# Packets	# Conn	# Bytes	# Packets
abi	389.0 K	180.1 G	148.4 M	66.1 K	120.1 G	100.0 M	40.5 K	55.8 G	45.0 M
jap	58.0 K	5.0 G	4.8 M	29.8 K	4.2 G	4.1 M	23.1 K	1.3 G	1.5 M
wls	329.8 K	121.7 G	144.1 M	101.3 K	113.3 G	122.1 M	63.3 K	28.0 G	40.1 M
wrd	290.9 K	171.3 G	208.8 M	98.0 K	167.7 G	200.0 M	73.3 K	36.7 G	63.1 M
lei	75.4 K	10.5 G	12.6 M	14.0 K	7.8 G	9.7 M	10.7 K	3.1 G	4.4 M
unc	774.8 K	121.3 G	129.5 M	168.1 K	94.7 G	100.5 M	131.7K	46.0 G	49.1 M
ibi	287.5 K	161.8 G	157.2 M	78.5 K	135.6 G	129.5 M	59.8 K	57.4 G	64.9 M

Table 3.2: Characteristic of Connections That Transmit More Than 10 Segments. Connections that transmit at least 10 data segments are described under "All Connections". Out of these, the connections with traces that contain at least one OOS segment are described under "OOS Connection". The final set of columns describe the characteristics of the connections for which our tool was able to unambiguously explain and classify all OOS segments.

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). Table 3.2 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.

3.2.2 Validation Against *TBIT* Controlled Conditions

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. Because the state-machine analysis critically depends on inferring the TCP sender's RTO to identify retransmissions triggered by timeouts, we 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 constant 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 very large file. *TBIT* then implemented the desired validation scenario with a specifically generated ACK stream. Unless stated otherwise, each validation scenario was repeated 100 times because not all sources of variation in timing 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 each of four different TCP implementations on the server machine – Windows XP, Solaris, Linux 2.4.2, and FreeBSD 4.10. Bidirectional tcpdumps of all packets were taken on these server machines and the traces were then used as input to our validation procedures. The procedures have two parts – (1) to verify that each TCP implementation responds in real operation as expected (thus establishing the "ground truth") , and (2) to verify that the state-machine analysis programs correctly emulate each implementation's responses. For part (1) we processed the tcpdump traces with *tcptrace* [tcpc] and other tools to verify the implementations' responses by inspection. For part (2), 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 the inferences about conditions at the sender were. We also used the tcpdumps as input to the analysis program, *tcpflows*, presented in [JID+04] but report the results from this only when they differ substantially from ours. In addition, we implement the *LEAST* algorithm from [AEO03] for identifying unneeded retransmissions.

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, *TBIT* causes all retransmissions to be triggered by

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

Table 3.3: Values of key parameters in different TCP Stacks

timeouts (by withholding ACKs). The analysis 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 elements are verified as part of the validation results. Table 3.3 gives the values used in the state machine for each TCP implementation.[2] [3]

RTT estimation:

Dummynet was used in experiments with constant minimum RTTs—of 50, 100, 150, 200, 400, 1000, and 2000 ms—between the two machines. All RTTs estimated for segment/ACK pairs by our state machines were within +/- 10 milliseconds of the value set by dummynet (these differences are consistent with the inherent variable delays in the switches).

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 OS-specific state machines correctly identified the SYN+ACK retransmission as being triggered by RTO; further, the measured RTO was equal to the value expected +/- the timer granularity (also shown in Table 3.3).

[2]Details about the RTO computation (srtt and rttvar) are taken from RFC 2988 [PA00]. 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 [RKS05].

[3]Some parameters for Windows are based on private communication with engineers at Microsoft Corp.

Trace	# OOS Conn.	p0f-id Conn.	% Linux		% Windows		% FreeBSD		% Solaris		% Other OSes
			Correct	Wrong	Correct	Wrong	Correct	Wrong	Correct	Wrong	
jap	23923	21260 (89%)	25.79	0.02	21.21	0.32	41.51	0	1.98	0.05	9.11
ibi	59713	59713 (100%)	99.80	0.20	0	0	0	0	0	0	0
unc	138214	136524 (99%)	7.26	0	78.08	0.69	5.02	0	8.52	0.17	0.25

Table 3.4: Validation using *p0f*. The third column lists the number (and percent) of connections for which p0f was able to identify the source OS. For each OS, we next list the percent of connections for which our estimation of sender OS was correct or wrong. The last column lists the percent of connections which did not belong to any of the OSes that we model. All percent values are with respect to the second column.

Minimum RTO setting:

No delays were added to the actual RTT (typically 1 millisecond) over the switched Ethernet. *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 extremely small RTT and the stabilization of the RTO before we simulate a dropped packet ensure that RTO should occur after an interval approximately equal to the minimum RTO. The OS-specific state machines correctly identified these retransmissions as triggered by RTO using these minimum values and timer granularities.

RTO Estimation

These validations were conducted with both near-constant and highly variable random delays. For the experiments with near-constant delays (varying by only 1-10 ms caused by switch delays), we used dummynet to set a target minimum RTT ranging from 10 to 1000 ms between the two machines. For experiments with highly variable delays, ACKs were delayed randomly by *TBIT* to vary the RTT from 0 to 400% above the dummynet minimum delays described above. In both sets of experiments *TBIT* triggered RTO retransmissions by withholding ACKs after a randomly selected packet.

Figure 3.4 summarizes the results of all the RTO experiments. It shows the CDF of the error between the actual RTO extracted from the tcpdump and the RTO value predicted by the state machines, normalized to the timer-granularity of each OS. We see that the errors fell well within the timer granularity for a particular OS except for Windows. Windows exhibits a strong instability in its RTO calculation. We contacted engineers at Microsoft who attributed our observations to a "rounding issue"

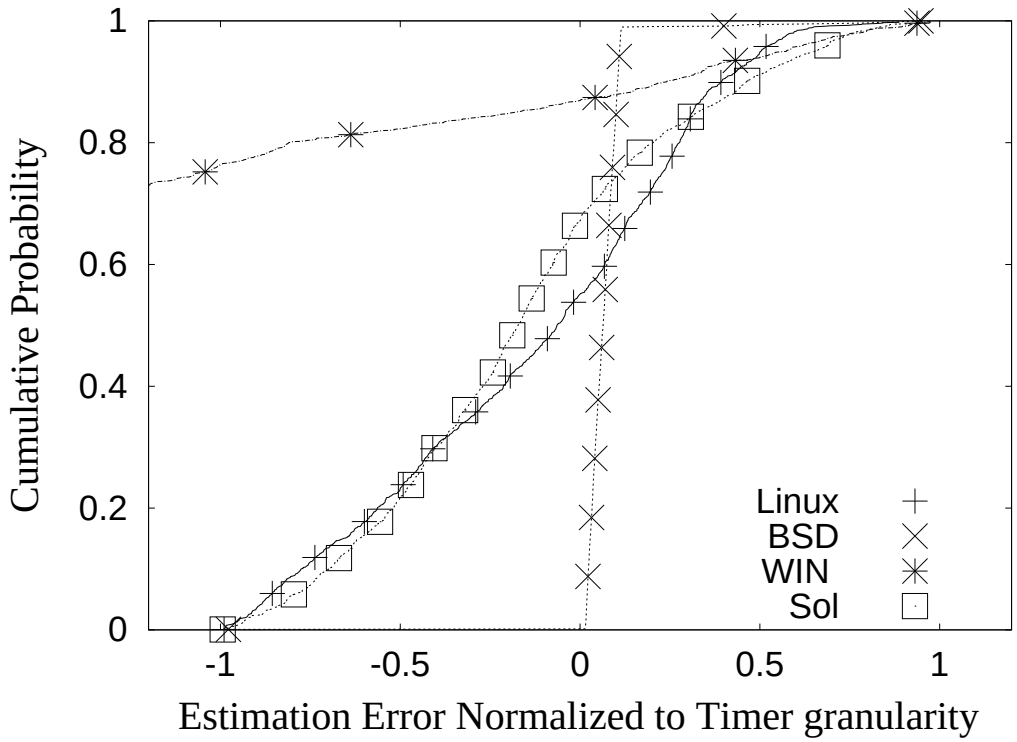

Figure 3.4: Error in RTO estimation for different OSes

with the OS, the details of which were not revealed due to copyright issues. However, our heuristics for timeout detection in the state machine for Windows are conservative enough to not be affected by the error. In terms of absolute numbers, the difference between the observed and state machine RTO was within $20ms$ for Linux and within $2ms$ for FreeBSD and Solaris.

These experiments also allowed us to estimate the percentage of RTO timeout events that would have been missed if we used only the RFC specifications in the analysis tools as is done for *tcpflows* [JID+04]. We found that if only the RFC specification was used, we would miss 85% of RTO events in Linux TCP connections, 55% in Windows, and 100% in Solaris. This is perhaps the most important reason that OS-specific logic needs to be incorporated in the analysis tools.

FR/R classification:

The second group of *TBIT* 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. In all cases, *TBIT* received and ACKed a randomly chosen number of segments before creating a specific loss scenario.

Number of duplicate ACKs to trigger retransmission:

To simulate this case, *TBIT* sent duplicate ACKs (without delays) in response to subsequent segments (thus simulating loss of a random segment). The number of duplicate ACKs was varied from 1 to 4. We repeated each of these experiments 4 times with different random number seeds. In the absence of enough duplicate ACKs, the sender times out and this is detected correctly by our OS-specific state machines. In the presence of enough duplicate ACKs, the retransmission was by a Fast retransmit and our OS-specific state machines also classified these events correctly. For a windows connection, the *tcpflows* tool failed to identify the retransmissions triggered by 2 duplicate ACKs. This is because it assumes that 3 duplicate ACKs are needed, as is recommended in the RFC specification.

Response to Partial ACKs in Fast Recovery:

TBIT triggered a retransmission by sending sufficient duplicate ACKs (as described above) and then sent partial ACKs for a randomly chosen segment from among those transmitted between the original and retransmission. We repeated this experiment 4 times with different random seeds. Our OS-specific state machines correctly identified these Partial ACK events. 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 sending sufficient duplicate ACKs and generated several different cases of SACK block contents indicating gaps in the received segments beyond the simulated loss. In all cases, our OS-specific state machines correctly classified such retransmissions. There are minor differences in the way Windows responds to SACK. This can cause a RTO-triggered retransmission even in presence of correct SACK blocks. These packets were correctly classified by our Windows-specific state machine. The *tcpflows* tool, which does not use SACK blocks, classified the above as simply retransmissions during "FR/R recovery".

Unneeded and Needed Retransmissions:

TBIT simulated instances of the implicit retransmission scenario of Fig 3.1. In a second set of scenarios, it sends spurious duplicate ACKs to trigger an unneeded retransmission (similar to Fig 3.2). Our OS-specific state machines correctly classified the corresponding retransmissions as *needed* or *unneeded*. These experiments also allowed us to compare our state-machine results with those we obtained by implementing the algorithm used in *LEAST* [AEO03]. *LEAST* correctly identified the unneeded retransmissions in the first scenario but failed to identify them in the second case.

3.2.3 Validation Against Real TCP Connections

Next, we validate our tool-set against traces of real world TCP connections. In this case, since we do not have access to either the TCP sender or the receiver for these connections, the ground truth about the classification of each OOS segment is not known. Consequently, we can not use the same validation tests as those used in Section 3.2.2. Instead, we use our tool to identify the sender OS (as the one corresponding to the state machine that is able to explain all OOS segments). Our validation evaluates how accurately does our tool-set identify the sender-OS (and hence, is able to accurately model the sender state machine and classify OOS segments).

For establishing the ground truth about the sender-OS, we rely on *p0f* [Zal06]—a popular passive fingerprinting tool which uses the information present in the option fields of SYN, SYN+ACK, or Reset segments to identify the source OS for the packet. We use *p0f* to identify the sender-side OS for all OOS connections in the *jap*, *ibi*, and *unc* traces that were successfully classified by our tool-set. These traces include TCP option fields and, hence, can be analyzed by *p0f*.

We compare our estimate of the sender-OS to that reported by *p0f*. Table 3.4 reports the comparison results. The numbers listed under the OS-specific columns report the percent of *p0f*-identified connections for which our tool-set correctly or incorrectly identified the sender-OS. We observe that:

- *p0f* is able to identify the sender-OS for 89-100% of the connections. The relative mix of sender-OS is quite different across the three traces. This is to be expected; *ibi* represents connections to a cluster of web-servers, all of which run Linux; *unc* represents members of an academic and medical community, most of whom use Windows PCs; *jap* represents trans-continental connections made by a generic mix of users in Japan.

Trace	# OOS Segments	% Network Reorder	% Segment Retransmissions					
			Total	RTO	Dupack	PA	SACK	Implicit
abi	339.2 K	14.1	85.9	33.4	17.9	4.4	7.1	23.1
jap	121.7 K	4.2	95.8	46.9	13.9	5.8	2.4	26.9
wls	1119.5 K	5.8	94.2	52.9	10.7	6.1	2.4	22.1
wrd	1250.4 K	5.3	94.7	66.0	9.6	2.5	1.1	15.6
lei	110.5 K	0.2	99.8	53.5	9.9	2.8	5.0	28.6
unc	1327.4 K	12.9	87.1	40.3	13.2	6.1	6.5	20.9
ibi	787.4 K	0.2	99.8	32.8	17.3	8.7	0.4	40.8

Table 3.5: Classification of OOS segments by *TCPdebug* . These are from connections for which we were able to unambiguously explain and classify all OOS segments.

Trace	# OOS Segments	*tcpflows*				
		% Network Reorder	% Segment Retransmissions			
			RTO	Dupack	RTO recovery	FR/R recovery
lei	110.5 K	0.8	55.2	7.5	34.7	1.8
unc	1327.4 K	13.8	39.5	7.0	36.0	3.6
ibi	787.4 K	0.27	26.5	21.2	29.7	22.3

Table 3.6: Classification of OOS segments by *tcpflows* . These are from connections for which we were able to unambiguously explain and classify all OOS segments. *tcpflows* classifies an OOS segment as one of: network reordering, retransmission triggered by RTO, duplicate ACKs, or during FR/R or RTO-recovery.

- Our estimate of sender-OS matches that of *p0f* for more than 99% of the connections—accuracy is high for all four OSes. We attribute this high level of success to two factors: (i) our in-depth modeling of sender-state as well as high granularity of analysis of OOS segments; and (ii) our conservative approach of filtering out connections with even a single OOS segment that is not robustly explained.[4]

A natural question to ask is: *in practice, how important is it to correctly model the sender OS?* In particular, if an RFC-based analysis tool is used, how different would the results be. We investigate this and other issues in the next section.

3.3 Impact

We believe the reason for the high degree of accuracy of our tool is that we insist on unambiguously explaining and classifying all OOS segments that appear within a connection. In order to be able to

[4]This also means that we can directly use *p0f* to identify the source OS for a connection.

Trace	# OOS Segments	% Network Reordered	% Segment Retransmissions						
			Total	RTO	Dupack	PA	SACK	Implicit	Unexplained
abi	1345.0 K	11.4	88.6	26.9	17.1	4.0	7.3	17.4	16.0
jap	340.8 K	6.3	93.7	35.6	14.6	5.0	4.2	22.1	12.2
wls	2927.5 K	7.7	92.3	39.9	11.4	5.9	2.7	22.3	10.0
wrd	4177.3 K	7.3	92.7	43.9	11.4	2.2	0.8	19.2	15.3
lei	294.5 K	0.4	75.6	40.6	11.6	3.1	5.6	24.0	14.7
unc	2752.9 K	12.6	87.4	32.9	12.7	5.7	6.0	20.1	10.0
ibi	2383.9 K	0.7	93.0	26.3	19.6	10.9	0.2	34.8	7.5

Table 3.7: Classification of all OOS segments (including unexplained events) by our tool-set. These are all connections irrespective of whether we were able to explain all events or not.

Trace	# Total Retran	Our Tool-set							*LEAST* [AEO03]	
		% Needed			% Unneeded			% No Inference	% Needed	% Unneeded
		Total	Implicit	Explicit	Total	Implicit	Explicit			
abi	291.9 K	79.1	13.1	66.0	12.0	4.9	7.1	8.8	89.6	10.4
jap	116.6 K	82.4	4.4	78.0	15.6	6.6	9.0	2.0	92.7	7.3
wls	1054.2 K	86.7	22.3	64.4	13.2	1.1	12.1	0.1	98	2.0
wrd	1184.2 K	96.2	15.9	80.3	3.7	0.5	3.2	0.1	97.7	2.3
lei	110.3 K	82.5	19.6	62.9	12.7	4.2	8.5	4.8	87.7	12.3
unc	1155.9 K	91.2	21.4	69.8	7.7	1.1	6.6	1.5	96.2	3.8
ibi	785.5 K	76.6	23.2	53.4	18.9	13.1	5.8	4.5	85.0	15.0

Table 3.8: Needed and Unneeded Retransmissions (for connections with all OOS segments unambiguously explained).

do so, our tool encodes significant amount of state and logic and it incorporates much of the diversity across TCP implementations. It is natural to ask: *in practice, how much difference does this make?* In particular, if prior tools are used to analyze real world OOS connections, how different would the classification results be? We investigate this issue by raising several questions below—we address each question by analyzing all of the seven Internet TCP trace-sets described in Section 3.2.1.

- ***How many OOS segments can we successfully classify?***

  Table 3.2 reports the number of OOS connections in which *all* OOS segments were unambiguously classified by our analysis. We find that in nearly 25-35% of OOS connections, at least one OOS segment could not be classified. Two main factors are responsible for the failure to completely classify a connection.

  – First, we specifically model only 5 sender OS versions. In order to study the prevalence of these OSes, we ran *p0f* against all connections (whether or not they had any OOS segments) that appear in the *jap*, *unc*, and *ibi* traces. While more than 80% of connections in each trace originated from a Windows or Linux machine, we found that nearly 10% of connec-

tions in each trace originated from an OS different from the above five—such connections, consequently, may not be successfully modeled by our state machines.

– Second, recall that we apply a conservative filter for accepting a connection classification: (i) each OOS segment that appears in a connection trace must be explained, and (ii) the explanations must match if more than one state machine explains all such segments.

  More that 50% of the discarded connections are discarded only due to the second rule above. In Section 3.2.3 we saw that *p0f* can be used quite effectively for identifying the source OS of connections. This allows us to eliminate the second filtering rule—if more than one state machines explain all OOS segments of a connection, *p0f* can be used to identify the sender OS, and the corresponding OOS classification can be accepted. We are currently incorporating this feature in our tool-set.

Figure 3.5 plots the distribution of the number of unexplained OOS segments in each OOS connection. We see that *all* segments are classified in 62-95% of the connections and these are accepted by our filters (as is also indicated in Table 3.2). More interestingly, the number of unexplained OOS segments are less than 5 in most of the remaining connections. Since the total number of unexplained segments is small, it may be worthwhile to include in our analysis the explained OOS segments even from the discarded connections. We study below how our classification results change if we do so. For the rest of the analysis in this section, however, we do not include results from such connections.

- ***How important is it to replicate TCP sender state?***

  Tables 3.5 and 3.8 report our classification for OOS segments in the seven traces, according to the taxonomy of Fig 3.3. Table 3.8 shows that the fraction of retransmissions that are unneeded (the original transmission of the segment was not lost) ranges from 3-19%. This suggests that, in practice, TCP loss rate would be significantly overestimated if every segment retransmission is taken as an indicator of packet loss—this underscores the importance of modeling and replicating TCP sender state.

  In order to study the effect of our connection filter—that requires that each OOS segment be unambiguously classified—we present in Table 3.7, our classification results when the explained OOS segments are included from all connections (including the connections discarded by our filter). We find that the number of unexplained segments are small (7-16%) in each trace. More importantly, Tables 3.5 and 3.7 are quite comparable in the distribution of elements within the

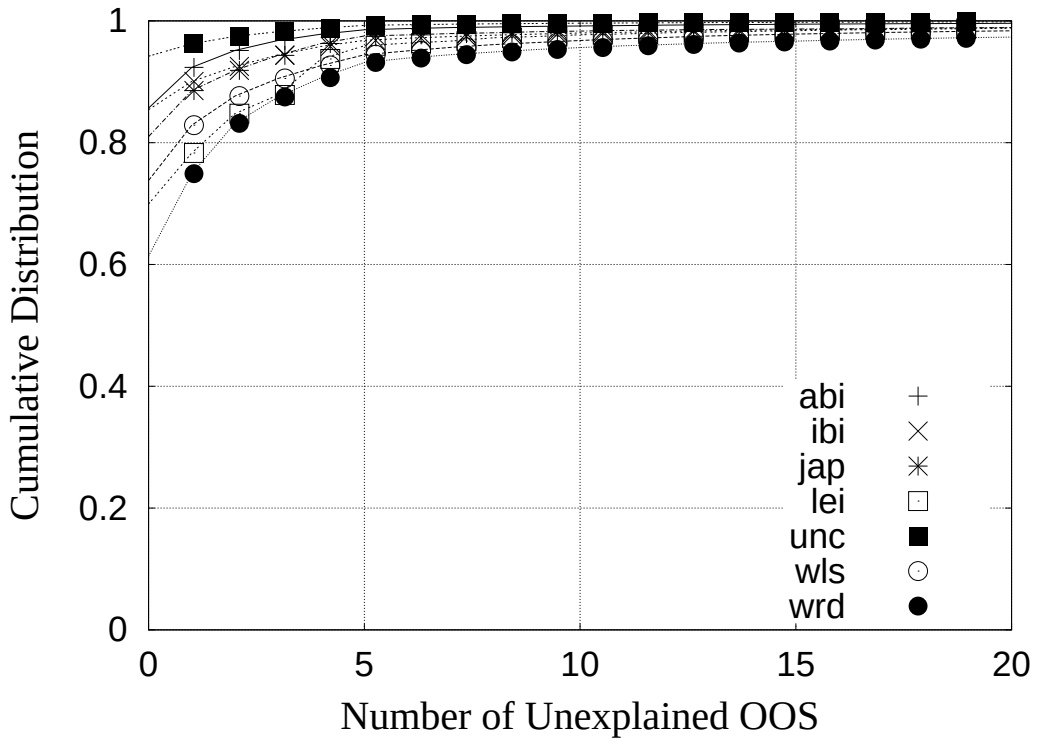

Figure 3.5: The distribution of the number of unexplained OOS segments in each OOS connection.

classification tree.[5] This suggests that our filter does not bias our classification results significantly.

- ***How important is it to identify unneeded explicit retransmissions?***

  Table 3.8 also compares the classification of needed and unneeded retransmissions made by our tool-set to that made by *LEAST* [AEO03]. We find that the number of unneeded retransmissions reported by *LEAST* is always lower (sometimes by more than 50%) than that reported by our tool-set. There are two reasons for this. First, as illustrated in Section 3.1 and as demonstrated in Section 3.2, *LEAST* does not identify some explicit retransmissions that are unneeded. Table 3.8 indicates that a majority of unneeded retransmissions occur due to explicit TCP loss detection-recovery actions. Second, when duplicate ACKs generated by unneeded implicit retransmissions are lost in the network, *LEAST* fails to conclude that the retransmission was not needed. While this is true even for our tool-sets, our additional analysis of the timing between the retransmission and the ACK (ACK arrives within a fraction of the minimum RTT) for the segment helps us identify some of these retransmissions.

- ***How important is it to classify implicit retransmissions?***

[5] We also find, although not reported here, that the values in Table 3.8 (including those reported for *LEAST*) do not change significantly if explained OOS segments from discarded connections are also included.

Implicit retransmissions are not analyzed for whether these are needed or not by *tcpflows* [JID+04]. Table 3.5 indicates that the fraction of retransmissions that are sent implicitly by TCP is significant (16-40%). More importantly, Table 3.8 indicates that, in practice, up to 30% of needed (and up to 40% of unneeded) segment retransmissions occur implicitly. Classifying these is, therefore, important for any study of either TCP losses or the effectiveness of TCP mechanisms.

- ***How important is it for the analysis to be OS-sensitive?***

  tcpflows [JID+04] is based on TCP standards specified in RFCs and does not incorporate variations that exist in TCP implementations across different OSes. In order to assess the impact of being OS-insensitive, we analyze using *tcpflows* all OOS connections that were explained by our tool-set in the *lei*, *unc*, and *ibi* traces (the other traces could not be processed by *tcpflows* due to incompatible trace formats). Table 3.6 includes the results—note that the "FR/R" classification of *tcpflows* is a combination of our PA- and SACK-triggered categories, and that "RTO-recovery" is captured by our implicit category. The classification differs significantly from that of our tool-set reported in the same table, underscoring the need for incorporating popular implementations.

  We also evaluate the need for OS-sensitive analysis using our tool-set. For this, we again consider all OOS connections in the above three traces that were explained by our OS-sensitive tool-set, and observe the classification results when only our FreeBSD-specific state machine (which follows the TCP standards fairly closely) is used on these. This state machine was unable to explain around 50% of all OOS segments in each trace!

- ***How important is it to incorporate delays and losses between the monitor and the sender?***

  Table 3.5 shows that significant number of OOS segments occur due to network reordering between the sender and the monitor. We have also observed that a significant fraction of losses occur between the sender and the monitor. It is, therefore, important to incorporate such network anomalies in the analysis.

  In the *abi* and *unc* traces,[6] nearly 13-14% 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 3.6, 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

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

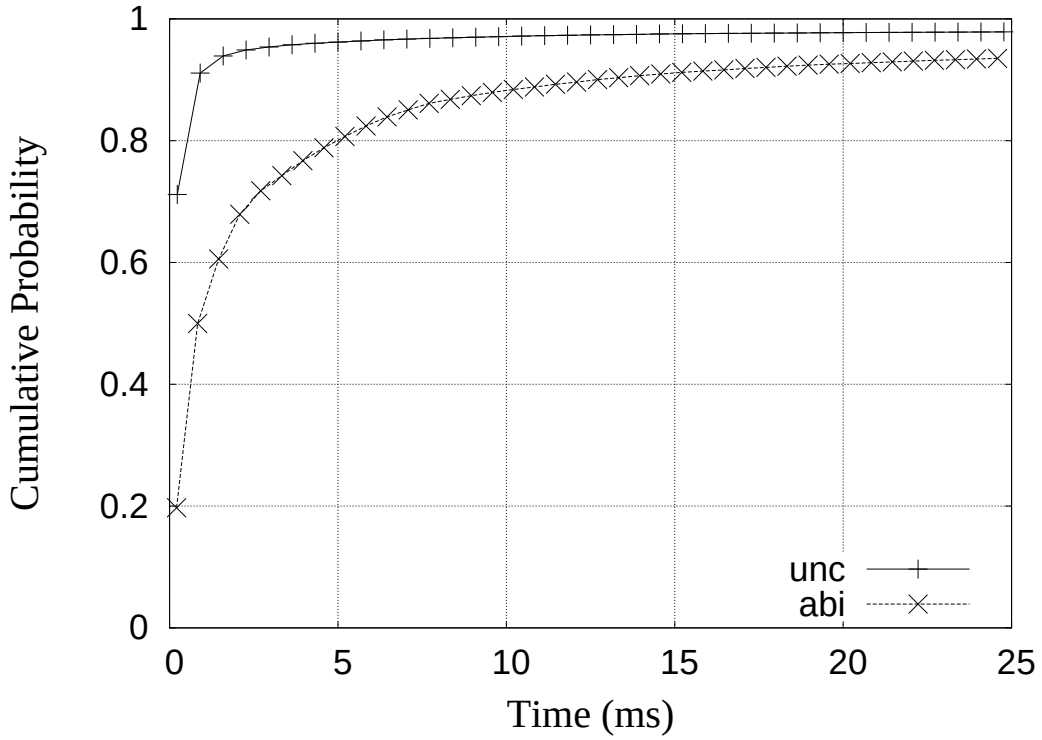

Figure 3.6: Resequencing delays for reordered segments

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.

3.4 Concluding Remarks

The primary contribution of our work is the implementation and validation of a new suite of tools for passive analysis of TCP connections. These tools are freely available to the networking research community and we hope they will encourage others to contribute to our understanding of TCP behavior "in the wild" by analyzing larger and more diverse sets of traces. While many of the ideas used in these tools are not new (see the discussion of related work), we believe this is the first time all have been integrated into a single, carefully validated, analysis approach. Further, we have made significant advances by explicitly including TCP implementation-specific factors for those operating systems that are currently (and likely to be for the foreseeable future) the dominant end points for TCP connections (Windows, Linux, FreeBSD/MAC OS X, Solaris). We have also been careful to cover many of the "corner cases" and boundary conditions that are missing in prior work, choosing to rely on explicit

sender-state tracking rather than approximations or heuristics where possible.

We believe the accuracy and high classification granularity of our tools will facilitate research to address issues related to the efficacy of TCP's loss detection/recovery mechanisms, to develop new models for the underlying loss processes that TCP must deal with, and to better understand the impact of network congestion on real world TCP performance. For example, the analysis of real world TCP connections may suggest important implications for the refinement of analytic models of TCP throughput as a function of loss rates [CSA00, PFTK98].

CHAPTER 4

TCP Detection Recovery

In the middle of difficulty lies opportunity.

— Albert Einstein (1879–1955)

Believe those who are seeking the truth. Doubt those who find it.

— Andre Gide (1869–1951)

While it is generally known that segment losses can adversely impact the connection duration of TCP connections, the extent to which they do so in the Internet has never been quantified. We address this issue by evaluating the impact of the design of TCP loss detection/recovery mechanisms on the performance of real world TCP connections.

As discussed in Chapter 1, two performance related goals guide the design of TCP loss detection mechanisms. First, TCP should *accurately* identify segment losses. In particular, if TCP erroneously infers that a segment was lost, it would unnecessarily invoke loss recovery and increase the connection response time. Second, TCP should *quickly* identify segment losses. A longer detection period adversely impacts connection response time as well. This fundamental trade-off between accuracy and timeliness is controlled by several design parameters associated with RTO and FR/R based loss detection—these include the dupACK threshold, the minimum RTO, the RTT-smoothing factor, the weight of RTT variability in the RTO-estimator, and the RTO estimator algorithm itself. In this chapter, we will systematically vary these parameters and (i) study the accuracy and timeliness of TCP loss detection/recovery in real world TCP connections originating from different sender OS stacks, and (ii) study the impact of packet loss detection/recovery on overall response time of these connections.

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

and traffic conditions.

We will first summarize the problem as formulated in Chapter 1. Next we present our data source and methodology. We present the detailed analysis of loss detection mechanism next and then conclude the chapter with the summary of our findings.

4.1 Problem Formulation

In Chapter 1, we highlighted the objectives of TCP loss detection/recovery analysis. The key questions we address here are: *Are the parameter settings in different TCP implementations working well in reducing connection response times? Are the decade-old recommended settings in the TCP standards optimal for the current Internet?*

These questions have been partially addressed in a couple of key studies—however, as described in detail in Chapter 2, most studies were conducted nearly a decade ago; consequently, these do not incorporate diversity in properties of real world TCP implementations, Internet paths, and application behavior. More importantly, to the best of our knowledge, no previous study has modeled the impact of TCP configuration on the overall response times of TCP connections.

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

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

Trace	Duration	Avg TCP Load	# Connections	# Bytes	# Packets
japan-155Mbps-2004 (jap)	4h	1.93 Mbps	0.3 M	3.5 G	3.7 M
UNC-1Gbps-2005 (unc)	4h	74 Mbps	14.5 M	133.3 G	151.0 M
ibiblio-1Gbps-2005 (ibi)	4h	90.64 Mbps	0.9 M	163.2 G	158.9 M
wireless-2006 (wls)	178h	1.58 Mbps	20.2 M	126.9 G	157.6 M

Table 4.1: General Characteristics of Packet Traces: We present the average TCP load, number of connections, number of bytes and number of packets in the traces. The name is brackets will be used to represent the traces in the rest of this chapter.

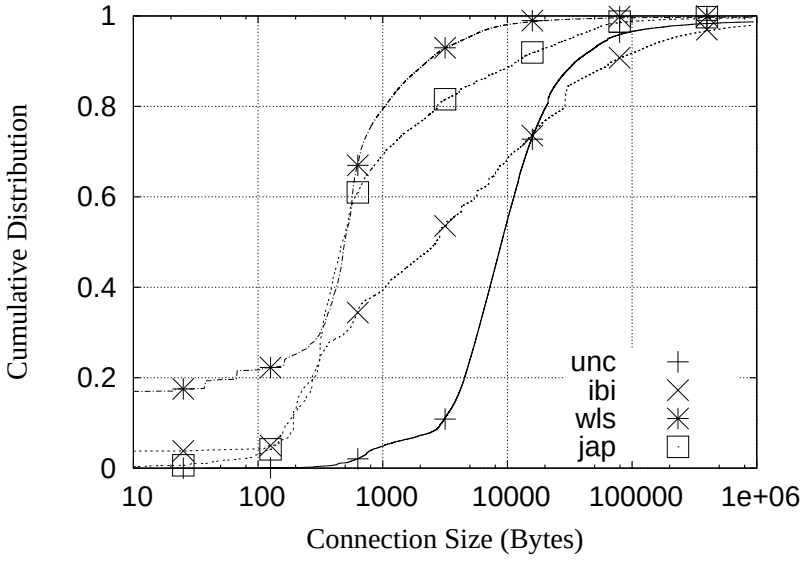

Figure 4.1: Distribution of Bytes Transmitted in Each Connection

4.2 Data Sources

Table 4.1 describes the bi-directional traces used in our analysis. These traces are collected from links with transmission capacities ranging from 155 Mbps to OC-48. The *jap* trace [jap] is collected off a trans-Pacific link connecting Japan to the US by the MAWI working group; the *unc* trace is collected at the campus-to-Internet link of the University of North Carolina (UNC); the *wls* trace captures wireless TCP connections from over 600 wireless access points within the UNC campus; and the *ibi* trace captures traffic served by a cluster of high-traffic web-servers (mirror for ibiblio.org). All traces except the one from the link to Japan were collected using Endace DAG cards [dag]; the *jap* trace was collected using tcpdump [JlM]. This trace set is fairly diverse in its geographic location, proximity to TCP senders, as well as types of users represented. The traces also vary significantly in the distribution of bytes transmitted per connection (see Fig 4.1).

Trace	All Connections			Lossy Connections			Lossy Connections with OS identified		
	Conn	Bytes	Packets	Conn	Bytes	Packets	Conn	Bytes	Packets
jap	58 K	5 G	4.8 M	29.8 K (51.37 %)	4.2 G (84 %)	4.1 M (85.42 %)	29.8 K (51.38 %)	4.2 G (84 %)	4.1 M (85.42 %)
unc	774.8 K	121.3 G	129.5 M	168.1 K (21.69 %)	94.7 G (78.07 %)	100.5 M (77.61 %)	122.65 K (15.83 %)	93.74 G (77.28 %)	87.71 M (67.73 %)
ibi	287.5 K	161.8 G	157.2 M	78.5 K (27.30 %)	135.6 G (83.81 %)	129.5 M (82.38 %)	30.68 K (10.67 %)	134.54 G (83.15 %)	128.28 M (81.6 %)
wls	329.8 K	121.7 G	144.1 M	101.3 K (30.72 %)	113.3 G (93.10 %)	122.1 M (84.73 %)	13.95 K (4.23 %)	94.32 G (77.5 %)	104.27 M (72.36 %)

Table 4.2: Characteristics of Connections That Transmit More Than 10 Segments. Connections that transmit at least 10 data segments are described under "All Connections". Out of these, the connections with traces that contain at least one OOS segment are described under "Lossy Connections". The final set of column describes the characteristics of the connections for which we where unambiguously able to identify the OS. The number in the brackets is the percentage of total connections, bytes or packets in that column.

4.3 Methodology

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

4.3.1 Identifying Loss Detection Attempts and Configuration:

We rely on *TCPdebug* [RKS06, tcpa] to (i) identify all segment retransmissions in a TCP packet trace and classifies them based on the corresponding loss-detection mechanism—including RTO, FR/R, and SACK, (ii) identify if the retransmission was necessary or spurious (depending on whether the original segment was actually lost or not) and (iii) identify the sender-side OS for each connection—more relevantly, identify the configuration of the 6 design parameters of interest to us.

We run this tool against all TCP connection traces and select those for which the tool can unambiguously identify the sender-side OS. We have already evaluated the OS identified by *TCPdebug* against *p0f*, a passive fingerprinting tool [Zal06], and found the OS-identification accuracy to be more than 99.9%. Since our objective is to study loss detection/recovery, we consider traces of only those

Trace	Explained Lossy Connections			Distribution across OSes			
	Conn	Bytes	Packets	Windows	Linux	Solaris	FreeBSD
jap	23.10 K (39.83 %)	1.30 G (26.00 %)	1.50 M (31.25 %)	5.47 (23.70 %)	6.56 (28.40 %)	0.52 (2.23 %)	10.55 (45.67 %)
unc	115.52 K (14.91 %)	47.80 G (39.41 %)	50.74 M (39.18 %)	97.45 (84.36 %)	2.56 (2.22 %)	12.35 (10.69 %)	3.15 (2.73 %)
ibi	23.37 K (8.13 %)	42.23 G (26.10 %)	52.57 M (33.44 %)	0.00 (0.00 %)	23.37 (100.00 %)	0.00 (0.00 %)	0.00 (0.00 %)
wls	13.82 K (4.19 %)	25.95 G (21.32 %)	39.40 M (27.34 %)	13.77 (99.63 %)	0.00 (0.00 %)	0.00 (0.00 %)	0.05 (0.37 %)

Table 4.3: Characteristics of Connections Used in Our Analysis and the Distribution of OSes in Them: The first column titled "Explained Lossy Connections" describe characteristics of connections for which we were able to explain all the losses and identify the OS unambiguously. The second set of columns describe the distribution of these explained connections within the different OSes. For the connections characteristics, the numbers in brackets represent the percentage of total connections, bytes and packets represented by those columns. For the OS distribution the number in brackets describe the percentage of explained lossy connections for that OS.

connections that experience at least one segment loss.[1] Table 4.2 and 4.3 summarizes the impact of applying these filters to our traces—a total of more than 2.8 million connections—as well as the distribution of connections across the 4 OSes. Our traces provided a large set of Windows and Linux connections, but relatively few Solaris or FreeBSD connections.

4.3.2 Studying Accuracy and Timeliness of Loss Detection:

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

4.3.3 Studying Impact of Design Parameters:

For this, we create several different instances of the emulated sender-side state-machine—one instance for each of several different configurations of the 6 design parameters. We then reprocess our traces using these modified state-machines and estimate for each sender configuration: (i) whether a

[1]Ideally, it would be interesting to study even connections that do not experience any losses to measure the false-positives rate of TCP loss detection (when TCP erroneously infers losses in these).

segment loss would be detected by either FR/R or RTO, (ii) whether a spurious retransmission would be avoided, and (iii) whether a segment would be spuriously retransmitted due to a premature RTO or spurious dupACKs. We use this data, to compute the accuracy and timeliness of the corresponding sender configuration.

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

4.3.4 Studying Impact on Connection Response Times:

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

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

$$F(x) \quad = \quad x + 1 - \frac{3x^2 - x}{5x + 1} \tag{4.1}$$

- *A spurious RTO-based loss detection is avoided.* When a sender avoids a spurious RTO-based retransmission, it saves time spent on recovering the flight size. To see by how much, assume that the flight size was $2x$ before the RTO expired; on RTO-expiry, the flight size is reduced to 1. It would take the sender $log(x) + x - 1$ RTTs to recover the flight size. However, the sender also achieves some goodput during this time —in Appendix A, for each such avoided spurious RTO-based loss recovery, we derive an estimate of the overall saving in connection duration in

units of RTT to be:

$$R(x) \quad = \quad x + \log x - 1 - \frac{3x^2 + x - 4}{5x + \log x - 1} \tag{4.2}$$

- *A loss is detected by FR/R, instead of an RTO.* Assume that the flight size was $2x$ when a loss occurred. If the loss is detected by FR/R instead of an RTO, there are two types of savings in response time. The first is in the time it takes to detect the loss, and is given by the difference between the RTO and the time at which the dupACKs are received (usually around 1 RTT). The second savings is due to the fact that the TCP sending rate after exiting from FR/R (flight size is reduced to x) is usually higher than after an RTO expiry (flight size is reduced to 1). It would take the receiver $log(x) - 1$ RTTs to gain a flight size of x after an RTO. However, the sender would also achieve some goodput during this time —in Appendix A, for each loss that is detected by FR/R, instead of an RTO, we derive an estimate of the overall saving in connection duration in the post-loss-recovery period in units of RTT to be:

$$RF(x) \quad = \quad \log x - 1 - \frac{2x - 4}{2x + \log x - 1} \tag{4.3}$$

- *A needed RTO-based loss detection takes a different amount of time.* For non-spurious RTO-based retransmissions, we compute the change (increase or decrease) in loss-detection time (difference between the original value of RTO and the new estimated value)—this also equals the change in connection duration for each such event.

Based on the above methodology, we next present our analysis and results for existing TCP implementations, as well as variants created by varying the 6 design parameters. We present only the most significant results here—a complete tabulation of all results can be found in [RKS07b].

4.4 Analysis of TCP Loss Detection

4.4.1 Baseline Performance of real world TCP Implementations

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

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

Table 4.4: Classification of TCP Retransmissions: We divide the observed losses in a trace into needed and unneeded retransmissions and further classify these losses as triggered by RTO and FR/R. The figure in brackets represent the percentage of total losses represented by that column.

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

Table 4.5: Values of key parameters in different TCP Stacks. Timer granularity is the granularity of clock used in the OS to measure RTT and RTO. Initial RTO is the initial value of RTO used. $minRTO$ is the minimum value of the RTO permitted by the OS. a, b, m, k are the parameters of RTO equation used by the OS. D is the dupack threshold used by the OS. RTO equation is the outline of the equation used by the OS.

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

Accuracy

Table 4.4 summarizes the total number of spurious retransmissions that were triggered by RTOs as well as FR/Rs, across connection traces originating from each of the 4 sender-side OSes. We observe that:

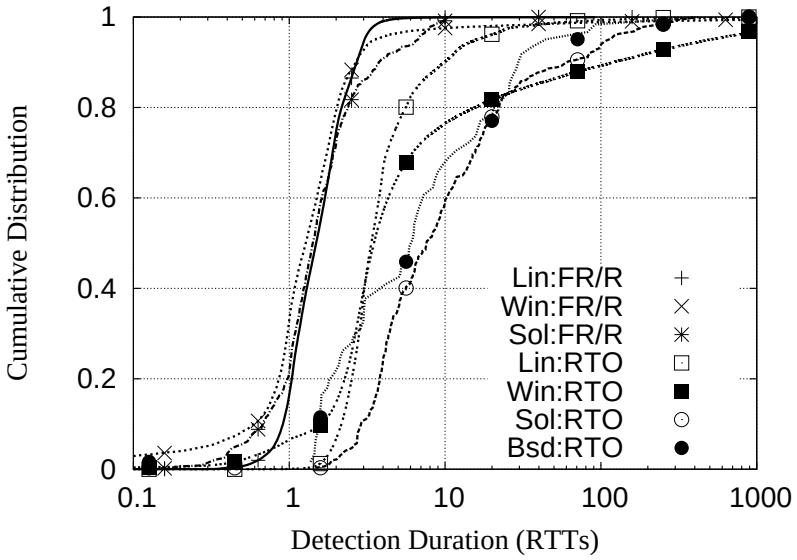

Figure 4.2: Distribution of Detection duration for FR/R and RTO (Normalized with RTT)

- Nearly 7-35% of all TCP retransmissions are spurious. In all of these cases, TCP inaccurately inferred that a segment was lost and retransmitted it. Most of the spurious retransmissions are triggered due to the expiry of an RTO (rather than an FR/R).

- The frequency of spurious RTOs varies significantly across OSes. This is somewhat to be expected, since the implementations and configurations of RTO estimators differ across current OSes—the Linux RTO estimator differs most significantly from the rest. We find that among Windows connections, nearly 30% of all RTO-triggered retransmissions are spurious, while for Linux, only about 6% of all RTO-triggered retransmissions are spurious.

- The fraction of all retransmissions that are based on spurious FR/R events is much smaller (3 - 5%), and do not differ much across OSes. It is important to note that the spurious FR/R-based retransmissions occur only when network reordering events result in the generation of D or more duplicate ACKs—the occurrence of such events is independent of the sender-side OS. All OSes (other than Windows that uses a value of 2) use 3 as the value of D—see Table 4.5[2]. We find that this value does not result in a large fraction of retransmissions due to inaccurate loss inferences.

Recall that the accuracy of loss detection can be improved by increasing the dupACK threshold and the RTO.

[2]Dupack Threshold for Linux is actually adaptive depending on number of duplicate ack triggered spurious retransmissions

Timeliness

Detection Durations RTO-based loss detection is, in general, more time-consuming that that based on FR/R—Figure 4.2 plots the distribution of loss detection times in units of the moving average of RTT,[3][4] for all FR/R- and RTO-based retransmissions. We find that:

- Most FR/R-based loss detection takes about 1-2 RTTs for all OSes.[5] For the Solaris TCP connections, however, around 7% of FR/R detections take more than 5 RTTs.
- RTOs take much longer than FR/Rs to detect losses. Also unlike FR/Rs, the RTO-based detection durations differ significantly across OSes. While the median RTO for Windows and Linux is around 4 RTTs, it is more than 20 RTTs for Solaris and FreeBSD.

  The Solaris RTO-estimator uses a minimum RTO of 400 ms and an srtt multiplier of 1.25; FreeBSD uses a minimum RTO of 1200 ms—these values are significantly higher than for Windows or Linux. As a result, for connections with relatively small and stable RTTs, the RTOs computed by Solaris and FreeBSD tend to be higher—TCP connections on these OSes, therefore, take longer to detect a loss using RTOs.
- The tail of the distribution of RTO detection duration for Windows differs significantly from that of Linux. For instance, while only around 10% of RTOs are larger than 10 RTTs for Linux, nearly 25% of Windows RTOs are larger than this amount. On close inspection of our traces, we find that several of these larger RTOs in Windows correspond to losses at the beginning of the respective TCP connections, when the RTO is primarily governed by the initial RTO and has not converged to a value representative of the network path. Linux updates its estimates of RTT and RTO at a much higher frequency (once per segment) than Windows (once per flight) and converges faster.

Overall, we find that the Linux RTO estimator converges fast and results in the smallest fraction of spurious RTO-based retransmissions.

Table 4.4 summarizes the relative frequency of occurrence of RTO-based vs. FR/R-based loss detection. We find that 60-88% of TCP retransmissions are triggered by RTOs. Our observation above

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

[4]The detection time is below 1 RTT in some cases due to using a moving average instead of just previous RTT. For windows additional rounding issues resulting in the retransmission timer to expire early.

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

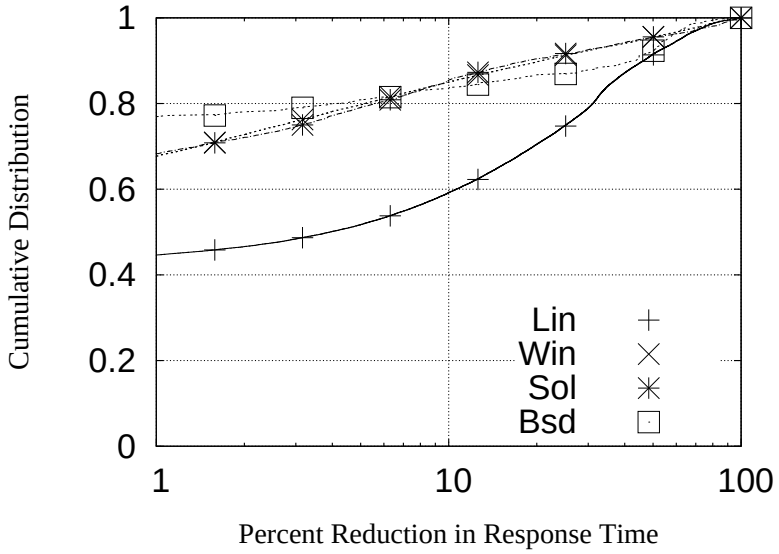

Figure 4.3: Distribution of Best-Case Reduction in Response Time

shows that all such RTO-based detection can be quite time-consuming. The prime reason for the high frequency of RTOs rather than FR/Rs is that there are often not enough segments in flight to trigger duplicate ACKs for a connection that experiences a loss. Thus, reducing the value of D should increase the likelihood of FR/R-based detection when losses occur; and reducing the value of RTO should reduce the loss-detection times when RTOs are unavoidable.

Recovery Durations The time spent by a TCP connection recovering from segment losses is independent of the loss-detection mechanism, and depends primarily on the number of segments lost within a flight. We analyze our traces to study recovery durations and find that these are also relatively independent of the sender-side OS. We also find that that use of *selective acknowledgments* (SACKs) helps reduce recovery times, but only when 3 or more segments are lost in a TCP flight. We refer the reader to [RKS07b] for details.

Most relevantly, note that none of the six design parameters considered in this paper impact loss recovery; TCP immediately retransmits segments upon inferring a loss—a TCP-like protocol can not do better than that. Thus, in this paper, what we are really studying is the design of loss detection mechanisms.

Scope for Improving Connection Response Times

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

In order to address these questions, we attempt to characterize the *Best-Case* reduction in connection durations that can be achieved by an ideal set of loss detection mechanisms. Our analysis is optimistic and assumes that in an *ideal* setting, (i) all spurious retransmissions (based on either FR/R or RTOs) are avoided, and (ii) all RTO-based loss detection takes no more than the maximum RTT of a connection.[6]

Specifically, for each connection in our traces: (i) we identify all instances of spurious FR/R-based loss detection and use Equation (4.1) to compute the savings in connection duration if the inaccurate loss inference is avoided; (ii) we identify all instance of spurious RTO-based loss detection and use Equation (4.2) to compute the savings in connection duration if the inaccurate loss inference is avoided; and (iii) we identify all needed RTO-based loss detection and compute the savings in connection duration if the RTO was instead equal to the maximum RTT of the connection (as described in Section 4.3.4). For each connection, we compute the total savings in connection duration as the sum of each of the above.

Figure 4.3 plots the distribution of the percent reduction in connection duration for connections belonging to each of the 4 OSes. We observe that:

- Nearly 45-75% connections have little potential (less than 1%) for improving their connection duration by improving the configuration of loss detection.

  However, a significant fraction (15-40%) of connections can see greater than 10% reduction in their connection durations by improving loss detection.

- The potential for improving connection durations differs across OSes. While more than 40% of Linux connections can see greater than 10% improvement in connection durations, less than 15% of Windows, FreeBSD, and Solaris connections have a similar opportunity.

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

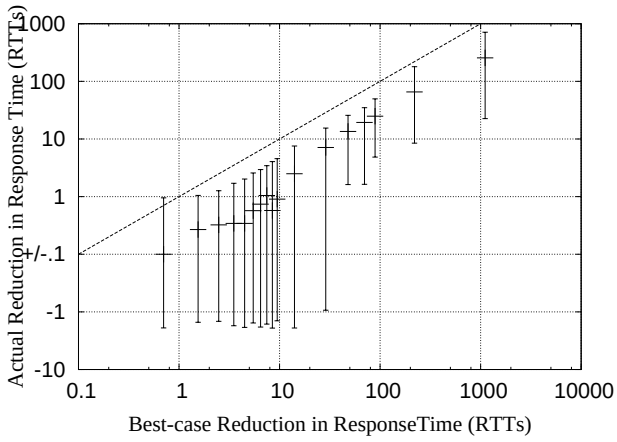

Figure 4.4: Linux: Impact of $k = 2$ on Connection Response Times

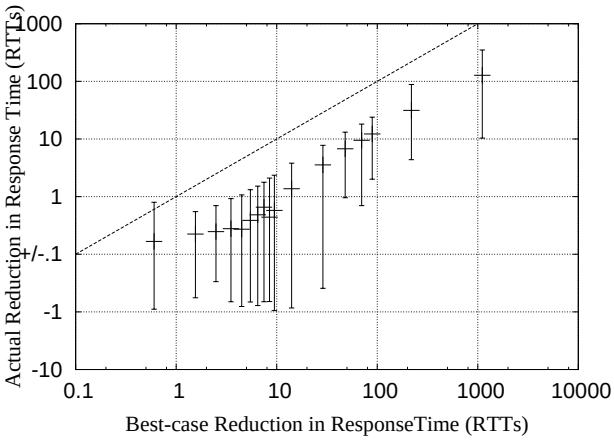

Figure 4.5: Linux: Impact of $k = 3$ on Connection Response Times

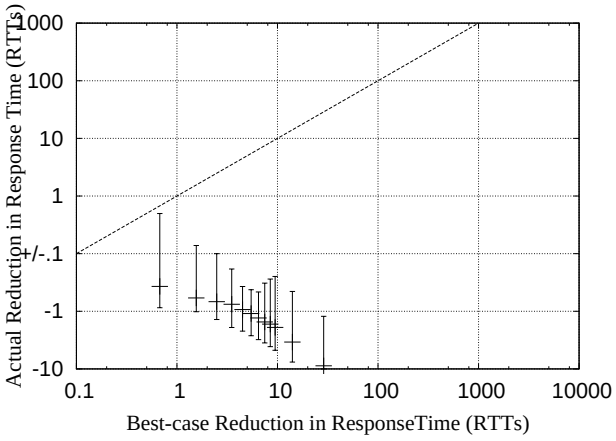

Figure 4.6: Linux: Impact of $k = 6$ on Connection Response Times

Based on the above, we expect to be able to significantly reduce connection durations of up to 40% of TCP connections by improving the configuration of loss detection parameters. In the rest of this section, we carefully compare the impact of each of these parameters on the performance of a connection with respect to the upper-bound computed above.

4.4.2 Impact of The RTO Estimator

The value of RTO is controlled by the 4 parameters: $k, minRTO, a, b$ [7]. If the value is large, the number of spurious RTO-based retransmissions reduce and help improve connection duration. Furthermore, there is increased likelihood of detecting losses by FR/R (rather than RTO). On the other hand,

[7]Current default for parameter m was observed to be optimal; results for varying m are not discussed in this paper

if the value of RTO is small, the time spent on detecting the loss (given by the RTO) decreases and helps improve the connection duration. We vary the above 4 parameters (k from 2 to 8, minRTO from 100ms to 1000ms, a from 1 to 1/32, and b from 1 to 1/32), and for each combination of their values, estimate the RTOs that would be computed within each connection. We then estimate what segments would be retransmitted spuriously due to premature RTOs, which RTO-detected losses would instead be detected by FR/R, and which RTO-detected losses would incur different detection durations.

We then study how each of these different phenomena interplay to impact the connection duration of each connection, by asking three questions: (i) *What is the reduction in connection duration when a spurious RTO is avoided?* (ii) *What is the reduction in connection duration when a loss is detected by FR/R (instead of an RTO)?* (iii) *What is the reduction in connection duration when the value of RTO is small?* The first two questions have already been addressed in Section 4.3.4. For the third, we compute the saving in connection duration simply as the difference between the default connection RTO and the new estimated RTO. For each combination of RTO-related parameters, and for each connection, we then list all instances of any of the above phenomena, and use the above formulations to compute the total reduction in connection duration[8].

Recall from Figure 4.3 that the scope for reduction in connection duration can differ by several orders of magnitude across different connections. To put our observations in perspective, therefore, throughout this section we plot the total connection duration reduction for each connection (y-axis) as a function of the Best-Case reduction for that connection (x-axis), each computed in units of the average connection RTT. For improved readability of these plots, we first divide the x-axis (Best-Case reduction in connection duration) into logarithmically-sized bins. We then consider all connections that fall within each bin, and compute the average and the 95-percentile values of the actual reduction in connection durations. We then plot these per-bin average and 95-percentile values (plotted as error-bars) against the average Best-Case reduction in connection duration for that bin.

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

Impact of k

Figures 4.4, 4.5, and 4.6 plot the average and 95-percentile improvements in connection durations of connections as a function of their best-case improvement, for k equal to 2, 3, and 6, respectively. We

[8]A negative value of reduction imply an increase in connection duration

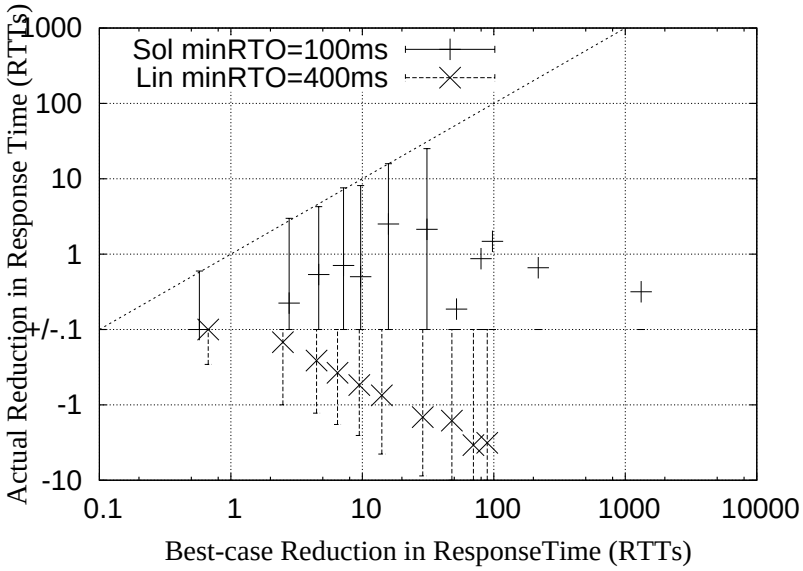

Figure 4.7: Linux: Impact of $minRTO$ = 100ms, 400ms

find that:

- The value of k significantly impacts the connection durations. We find that a small value of k (2 or 3) can help significantly improve the connection durations of most connections. A value of 2, in fact, comes fairly close to achieving an average connection duration improvement similar to the best-case improvement for connections. This suggests that perhaps k is the single-most influential parameter related to TCP loss detection and that setting it to a small value of 2 (rather than the in-use and recommended value of 4) can help significantly reduce connection durations.

  k = 6 consistently increases the connection durations.

  Some connections do seem to witness an increase in their connection durations even with a small value of k—this is especially true for connections with a small potential for best-case improvement. However, the average connection in all of these cases does witness a good reduction in connection durations. This suggests that only a small fraction of connections is adversely affected.

- The impact of k on other OSes is similar in trend, but not as pronounced as for Linux.

  It is important to note that Linux tracks RTT on a finer time-scale and hence its RTO estimate is robust even with a small value of k.

Impact of $minRTO$

Figures 4.7 plot the average and 95-percentile improvements in connection durations as a function of their best-case improvement, for Solaris with $minRTO$ equal to 100ms and Linux with $minRTO$ equal to 400ms, respectively. We find that a larger value of $minRTO$ adversely impacts the connection durations of almost all connections. Even the 95-percentile connections do not witness a reduction in connection duration. This suggests that the overall performance of TCP loss detection is significantly adversely impacted by a large value of $minRTO$. This observation was also made in [AP99], although the $minRTO$ evaluated was a very large value of 1 second. Fortunately, both Linux and Windows use a minRTO of $200ms$; FreeBSD and Solaris, however, use larger values. We find that reducing the Solaris $minRTO$ from 400ms to 100ms improves the connection durations of the Solaris connections—the improvement, however, is not as significant as that observed using small values of k. Reducing the Linux $minRTO$ to 100ms had negligible impact on the connection durations of most connections.

Impact of Smoothing Factors, a ***and*** b

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

Figure 4.8 plots the average and 95-percentile improvements in connection durations of the Linux connections as a function of their best-case improvement, for $a = 1/2$. We find that a larger value of a helps improve connection durations for most Linux connections, especially those that have a large Best-Case potential for reduction. This corroborates well with the fact that the Linux RTO estimator runs at a higher frequency (once per segment) than most other OSes (once per flight) and hence is less sensitive to high fluctuations in the measured RTT. This implies that in computing the RTT variation, the most recent sample of *rttvar* should be given a weight of at least 25%. $a = 1/32$ (or anything smaller than 1/4) increases the connection durations of most Linux connections. Changing the value of b (to a larger or smaller value than 1/8) has negligible impact—less than 1 RTT–on the connection durations of most Linux connections, independent of their Best-Case reductions.

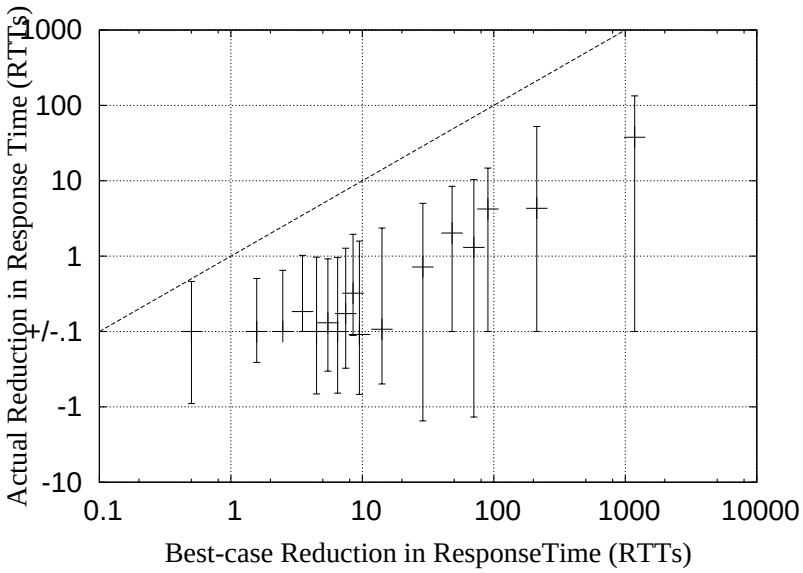

Figure 4.8: Linux: Impact of $a = 1/2$

4.4.3 Impact of The dupACK Threshold

Recall that the value of D can impact the connection durations in opposing ways. Table 4.6—that lists the changes in total number of RTOs, FR/Rs, and spurious FR/R-based retransmissions, for $D = 2, 3, 4$—highlights this fact. For instance, we find that increasing D to 4 avoids 1.2% of Linux retransmissions due to spurious FR/R events, but at the same time causes 7.2% of retransmissions due to FR/R to be converted into RTOs. In order to see which of these factors has a more pronounced effect on connection durations, we need to answer two questions: (i) *By how much does a spurious FR/R increase connection duration?* (ii) *By how much does a detection by FR/R (instead of RTO) reduce connection duration?* Both of these questions have already been addressed in Section 4.3.4, where Equations (4.1) and (4.3) formulate these effects respectively. Using these formulations, for each connection, we compute the total reduction in connection duration when the dupACK threshold is varied from 4 to 2.

Note that the likely impact of D on a connection depends on its average flight size. A larger flight is likely to benefit from a large value of D that helps avoid spurious retransmissions due to dupACKs caused by network reordering. When the flight is small, however, a large D does avoid spurious FR/R retransmissions, but also implies that a genuine segment loss can not be detected by the faster FR/R mechanism and has to suffer an RTO-based detection. To put this observation in perspective, Figure 4.9 plots separately for small and large Linux connections, the total connection duration reduction as a

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

Table 4.6: Impact of the dupACK Threshold: First set of columns show the impact of changing D from 3 to 2 for all OSes except Windows for which it shows the impact of changing D from 2 to 3. The second set of columns show the impact of changing D to 4. Figures in bracket represent the percentage of total losses represented by that column.

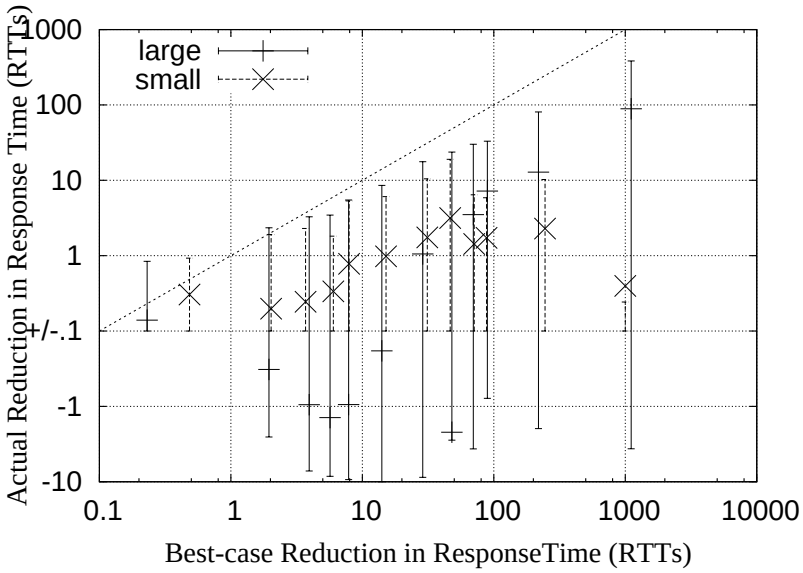

Figure 4.9: Linux: Impact of $D = 2$

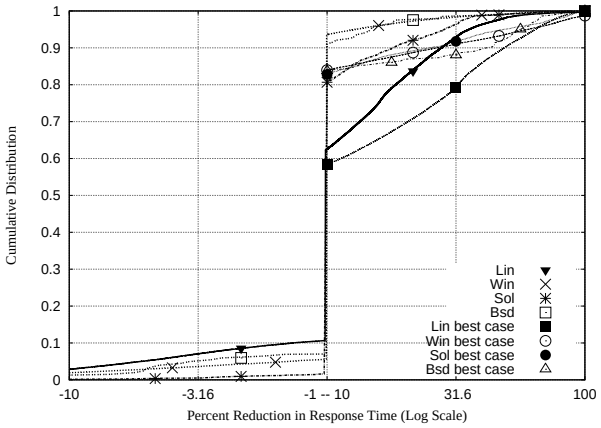

Figure 4.10: Smart Config: Relative Change in Response Time

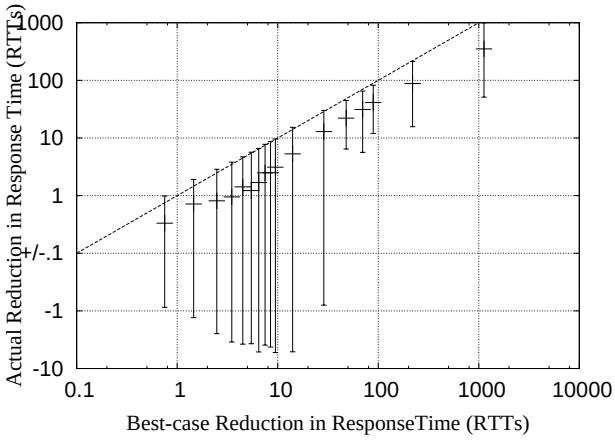

Figure 4.11: Smart Config: Actual Response Time vs. Upper Bound

function of the Best-Case reduction, when D is changed from 3 to 2—we refer to connections that transmit 1500KB ($\sim$ 10 MSS-sized segments) or less as "small" connections; such connections do not achieve a flight size larger than 4. We find that:

- Reducing D from 3 to 2 reduces the connection duration of most (including 95-percentile) of the small connections. Generally speaking, larger is the Best-Case reduction, larger is also the average reduction in connection durations for connections—however, the average reduction is always more than an order-of-magnitude smaller than the Best-Case. Thus, D comes only after k in its ability to help reduce connection durations.

  Reducing D to 2 helps reduce the connection duration of only those large connections that have large values of Best-Case potential reduction—other large connections witness an increase in connection durations using $D = 2$.

  The impact of using $D = 2$ is similar for connections emanating from other sender-side OSes.

- Increasing the dupACK threshold to 4 consistently worsens the performance of most connections (including the 95-percentile performance), irrespective of the connection size, the potential Best-Case reduction, or the sender-side OS.

4.4.4 Impact of The Smart Configuration

In this section, we adopt the best-performing configurations—referred to as the *Smart-Config*—for each of the 5 parameters and quantify the total improvement in connection durations. Specifically, we

set $k = 2$, $minRTO = 100ms$, $a = 1/4$, and $b = 1/8$. The dupACK threshold is set according to the rule: $D = \max\{1, \min\{3, F - 2\}\}$, where F is the current flight size of a connection.

Fig 4.10 plots the percentage change in per-connection connection duration with this configuration. We find that with the Smart-Config, the connection durations reduce by more than 10% for nearly 40% of Linux connections. A smaller percentage of connections—7% for Windows, 20% for Solaris, and 10% for BSD—witness a similar improvement in the other OSes. The Smart-Config also results in an *increase* in connection durations for some connections. However, less than 3% of connections in each OS suffer an increase of more than 10%.

Fig 4.11 plots the connection duration change (in units of RTT), as a function of the best-achievable savings. We find that the *average* observed reduction in connection duration closely matches the theoretical upper bound computed in our estimate of the Best-Case reduction in connection duration. There is a significant variability in the actual reduction in connection duration among connections with a similar estimate of the Best-Case upper bound. The variability, however, is smaller at larger values of the Best-Case estimate.

4.5 Concluding Remarks

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

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

Our analysis suggests that by re-tuning the configuration of TCP parameters, up to 40% of Linux

connections can witness a significant reduction (more than 10%) in their connection durations. For a majority of connections, this is close to the best-achievable reduction.

CHAPTER 5

Delay Based Congestion Avoidance

If everything seems under control, you're not going fast enough

— Mario Andretti (1940–)

Confusion is a word we have invented for an order which is not understood.

— Henry Miller (1891–1980)

In this Chapter, we will investigate the use of delay instead of packet loss as a network congestion indicator. Most deployed versions of TCP rely on *packet loss* in order to detect network congestion and respond to it by drastically reducing their sending rate. Consequently, packet losses significantly degrade the connection performance. An alternate strategy is *delay-based congestion-control* (DBCC), which attempts to avoid packet losses by — (i) detecting congestion early through increase in the packet round-trip times (RTTs), and (ii) reducing the connection sending rate in order to alleviate congestion before packet losses can occur.

As discussed in Chapter 1, two key issues determine the effectiveness of DBCC mechanisms: *can RTTs be used to reliably predict impending packet losses*; and *can the subsequent reduction in sending rate prevent packet losses from occurring*. In this Chapter, we concentrate on answering the first of the two issues mentioned above. Specifically, we — (i) evaluate well-known delay-based congestion estimators (DBCEs) for their effectiveness in signaling congestion before packet loss occurs, (ii) investigate which connection characteristics are likely to influence the performance of such estimators, and (iii) evaluate the potential improvements in connection's performance with the use of DBCC mechanisms.

We analyze more than 1.8 million real world TCP connection traces, captured at five different locations around the world. We first extract reliable estimates of per-segment RTTs and packet losses for each connection. We then run the sequence of extracted RTTs for each connection through each DBCE in order to evaluate its efficacy in predicting losses, as well as in reducing connection duration.

We then study the characteristics of connections for which the estimators fare well, as well as of those for which they do not fare well.

In the remaining sections of this chapter, we summarize our objectives for the study, present the dataset used and then discuss the methodology and evaluation results at length. We also study the influence of connection characteristics on the performance of the Delay Based Congestion Estimator (DBCE). We conclude this Chapter with the summary of our key observations and contributions.

5.1 Problem Formulation

In order to avoid packet losses and the associated performance costs of TCP loss-recovery, several *Delay-based Congestion Estimators* (DBCEs) have been proposed. These DBCEs rely on the assumption that during periods of congestion, a connection's packets will experience higher queuing delays at the congested link—this should manifest itself in increased packet round-trip times (RTTs). By sampling per-packet RTTs, and comparing them to a base RTT value (measured in the absence of congestion), a DBCE infers the onset as well as alleviation of congestion. The hope is that DBCEs can detect the onset of congestion much earlier than the occurrence of packet loss and the corresponding congestion avoidance (CA) mechanisms can potentially avoid the loss. Existing DBCEs differ primarily in the RTT-derived metric and the base metric used for estimating congestion. Table 5.1 lists the choice of these metrics for some of the prominent DBCEs—these DBCEs are described in more detail in Chapter 2.[1] Here we simply note that while Vegas and Tri-S study the *relative* ratio of the current RTT sample to the minimum RTT of a connection, most of the other DBCEs study the *absolute* difference between recent RTTs and the minimum or mean RTT. Also, of all of the DBCEs listed in Table 5.1, CIM is the only one that relies on a *history* of recent RTT samples to assess network congestion—all of the rest use only the most recent RTT sample.

As formulated in Chapter 1, the performance of the DBCE depends on its ability to predict a large number of losses and at the same time not unnecessarily predict congestion where there is none. However, these two goals conflict. An aggressive DBCE is likely to predict more losses, but also have a high rate of false predictions. On the other hand, a conservative DBCE will seldom give false predictions, but would miss out on many losses. Consequently, it is natural to ask: *how well do existing DBCEs perform along these two factors?* And perhaps more importantly, *what DBCEs are the best-performing*

[1] CIM is proposed only as a DBCE, the rest have been proposed with a corresponding CA mechanism.

Estimator	Metric Used	Compared to	Condition Used To Estimate Congestion
CARD [Jai89]	Delay Gradient	Previous RTT	$\frac{RTT-prevRTT}{RTT+prevRTT} * \frac{Window+prevWindow}{Window-prevWindow} > 0$
Tri-S [WC91]	Throughput Gradient	Initial RTT	$\frac{firstRTT}{Window-prevWindow} * (\frac{Window}{RTT} - \frac{prevWindow}{prevRTT}) < 0.5$
Dual [WC92]	Delay	Min and Max RTT	$\frac{minRTT+maxRTT}{2} < RTT$
Vegas [BOP94]	Throughput	Min RTT	$window * (1 - \frac{minRTT}{RTT}) > 3$
BFA [AR98]	Delay variability	fixed thresholds	$signed - rtt - variability > 0.01$
CIM [MNR03]	Delay	Previous 20 RTTs	$avg(RTT_2) > avg(RTT_{20}) + RTT_{std-dev}$
FAST [WJLH06]	Delay	Min RTT	$1 - \frac{100}{window} <= \frac{minRTT}{avgRTT}$
DECA [YQC04]	Delay	Min and Max RTT	$\frac{(maxRTT+minRTT)}{2} - perFlightMaxRTT < 0$
DAIMD [LSM+07]	Delay	Min RTT	$(pktInFlight > x)\&(sRTT - minRTT) > 0.02$

Table 5.1: Estimator Descriptions: References in first column provide more details on the estimtors considered.

in terms of achieving maximum reduction in connection duration?

While DBCEs have been evaluated in the past [JWL03, PJD04, BV03, MNR03, BV98a, BV98b], the issue of what DBCEs are likely to improve the overall timeliness performance of TCP connections has not been adequately addressed. It is our goal to do so. Specifically, we make two key contributions. First, we conduct a comprehensive evaluation of all DBCEs listed in Table 5.1 using more than 1.8 Million real world TCP connections, captured at a diverse set of locations around the world. Our evaluation explicitly models the impact of a DBCE on the duration of these connections. We believe that this is the first DBCE evaluation of such a large scale, diversity, and comprehensiveness. Second, we study the characteristics of these connections in order to analyze their influence on the efficacy of delay-based congestion estimation. To the best of our knowledge, this issue has not been addressed before.

Note that it is not our objective to design an optimal DBCE, but rather to evaluate prominent estimators and investigate the conditions under which they succeed or fail.

5.2 Data Sources

We analyze TCP connection traces collected from 5 different global locations. Table 5.2 describes the traces used in our analysis. These traces are collected from links with transmission capacities ranging from 155 Mbps to OC-48. The *abi* traces [abi] are collected from a backbone link of the Internet-2 network (Abilene); the *wls* trace captures wireless TCP connections from over 600 wireless access points within the UNC campus; the *unc* and *lei* [lei] 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

Trace	Duration	Avg TCP Load	# Dst.	# Conn.	# Bytes	# Pkts	# RTT
Abilene-OC48-2002 (abi)	2h	211.41 Mbps	3452.5 K	7.1 M	190.3 G	160.1 M	81.3 M
Liepzig-1Gbps-2003 (lei)	2h 45m	9.53 Mbps	1430.2 K	2.4 M	11.8 G	17.3 M	11.8 M
UNC-1Gbps-2005 (unc)	4h	74 Mbps	1082.9 K	14.5 M	133.3 G	151.0 M	85.0 M
Ibiblio-1Gbps-2005 (ibi)	4h	90.64 Mbps	88.7 K	0.9 M	163.2 G	158.9 M	75.6 M
Wireless-2006 (wls)	178h	0.61 Mbps	768.6 K	9.7 M	48.5 G	68.9 M	50.6 M

Table 5.2: General Characteristics of Packet Traces: We present the average TCP load, number of connections, number of bytes and number of packets in the traces. The name is brackets will be used to represent the traces in the rest of this chapter.

Trace	Connections with more than 10 segments		
	# Connections	# Bytes	# Packets
abi	388.9 K	180.1 G	148.5 M
lei	75.4 K	10.5 G	12.6 M
unc	774.8 K	121.3 G	129.6 M
ibi	287.5 K	161.8 G	157.2 M
wls	327.7 K	41.3 G	55.0 M
Total	1.85 M	515 G	503 M

Table 5.3: Statistics for Large Connections: We present the number of connections, bytes and packets for connections with atleast 10 segments in them.

served by a cluster of high-traffic web-servers (*ibiblio.org*). All traces were collected using Endace DAG cards. The traces represent a fairly diverse and large population. The traces are also fairly diverse in the distribution of connection RTTs as well as the number of bytes transmitted per connection.

For our analysis in the rest of this Chapter, we use only those connections that transmit at least 10 segments—most DBCEs need several RTT estimates before they can robustly infer the state of network congestion. Table 5.3 lists, for each trace, the number of connection that have greater than 10 packets and the number of bytes and packets that they contain. We find that while very few (less than 6%) connections carry at least 10 segments, these connections carry most of the bytes (more than 94%).

Figure 5.1 plots the distribution of loss rates observed for each connection used. 60-85% of connections have no losses. Among others, the loss rate varies from less than 0.1% to more than 80% with most in the range of 1-10%.

For brevity, we combine all traces for the purpose of presenting our DBCE evaluation results. We have closely examined the per-trace performance of DBCEs and find the results to be remarkably similar across traces.

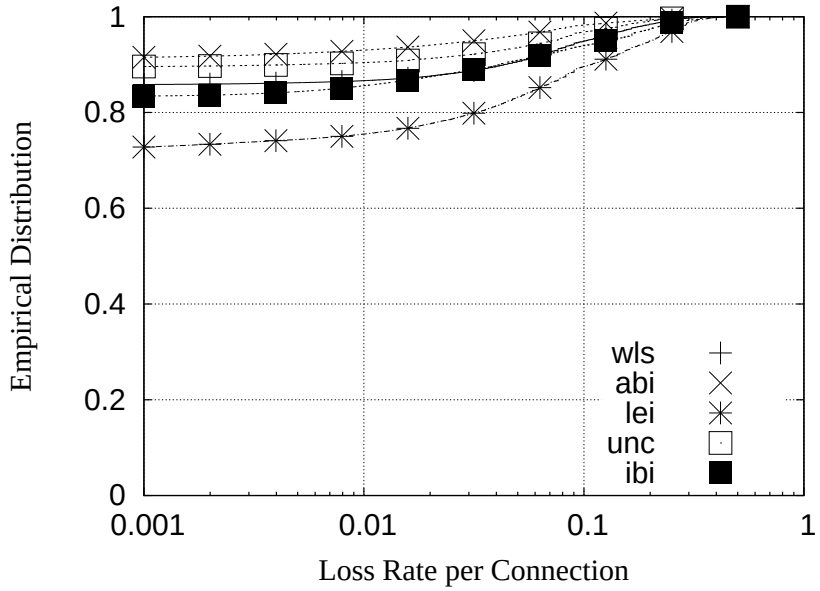

Figure 5.1: Distribution of Per-connection Loss Rate

5.3 Evaluation Methodology

Our basic approach for evaluating DBCEs is to analyze these against passively-collected traces containing large numbers of real world TCP connection. Below we describe our basic trace-processing steps and our DBCE-evaluation methodology in detail.

5.3.1 Trace Pre-processing

Basic Approach For each connection, we first compute reliable estimates of its segment RTTs and segment losses. The segment loss information is computed as a series of *loss episodes*, where we group all losses that occur within the same flight of segments into a single loss-episode. We then run the connection against prominent DBCEs and use the RTT estimate to infer the predictions made by the DBCE . Using this we divide each connection into alternating phases of congestion and no-congestion periods, interspersed with the above-defined loss episodes. In this section, we describe each of the above steps in some detail.

Extracting Valid RTT Samples

For reliably extracting all valid RTT samples from a TCP connection trace, we use *TCPdebug* . This work extends Karn's algorithm in several ways to deal with complications arising from lost and reordered segments, especially those that occur between the sender and tracing monitor. In dealing with these complications, our guiding principle has been to include only those RTT values for which there is an unambiguous correspondence between an acknowledgment and the data segment that triggered its generation.

Extracting Packet Losses

For reliably identifying segment losses within TCP connection traces, we again rely on *TCPdebug*. *TCPdebug* classifies all out-of-sequence segments as either retransmissions or reordering of packets. It further classifies all retransmission as needed (real losses) or unneeded (spurious retransmission).

Defining Per-DBCE Connection States

We next divide the entire duration of each connection into a sequence of three types of phases. First, we group losses into *loss-episodes*, in which we group all losses that occur within the same *flight*[2] of segments into a single loss-episode; each loss episode starts with the transmission of the first segment in a flight that is subsequently lost, and ends with the transmission of the last segment from that flight that gets lost. Thus, the analysis of each connection yields a series of loss-episodes with well-defined beginning and ending segment transmissions.

Note that reliable RTT estimates cannot be obtained during a loss-episode [AKSJ03]. Conversely, all of the segments whose transmissions do not lie within a loss-episode do yield an RTT sample. We next run these extracted RTT samples of each connection through each of the DBCEs we are evaluating—for each RTT sample, a DBCE either signals congestion (CN) or no-congestion (NCN). Thus, for each connection-estimator pair, we obtain a corresponding series of CN and NCN indications. We use these indications to divide each connection into a series of three *mutually-exclusive* phases (on a per-DBCE basis): congestion-phase, no-congestion phase, and loss-episode. Loss-episodes are defined as mentioned above. A congestion (or no-congestion) phase begins with the transmission time of any data segment

[2]At any given point in time, a flight is defined as the set of segments that have been transmitted, but not yet acknowledged. A flight roughly corresponds to a congestion-window worth of packets.

that signals CN (or NCN), but whose preceding segment either belongs to a loss-episode or signals NCN (or CN). The CN (or NCN) phase ends with the first subsequent segment that signals the beginning of any other phase.

At the end of the above classification for each connection-estimator pair, the connection is divided into a sequence of these three phases. Below we describe our metrics for evaluating a DBCE based on this sequence of phases.

5.3.2 Evaluation Approach

Base Metrics

For each connection, we define a set of two evaluation metrics for each DBCE:

- *Loss Prediction Ability (LPA):* The first metric captures the ability of the DBCE to predict an impending loss. It is defined as the fraction of loss-episodes that are immediately preceded by a congestion-phase. The higher the value of this metric, the better the corresponding DBCE at predicting a loss.
- *False Positive Avoidance (FPA):* We define the rate of false positives (FP) to be the fraction of congestion-phases that are not succeeded by a loss-episode (but by a no-congestion phase, or the end of connection). Note that FP is a lower-is-better metric—the lower is the value of this metric, the better is the corresponding DBCE. In order to interpret both metrics consistently, we define the rate of *false positive avoidance* as the inverse of this as: FPA = 1 - FP.

Note that both of the above metrics take values between 0 and 1.

Assessing Impact on Connection Durations

Finally, we assess the collective impact of LPA and FPA on overall connection durations. Note that when a DBCE predicts and helps avoid a loss, TCP saves the time spent in detecting and recovering from the loss, as well as the lost throughput due to a loss-related decrease in flight-size—the total savings differ according to whether the loss is detected by TCP's *retransmission-timeout* (RTO) mechanism or the *fast-retransmit/recovery* (FR/R) mechanism [RKS07a].

The impact on duration is also governed by what CA policy is used along with a DBCE to respond to a congestion signal. Two such policies have been defined in the literature:

- Additive Decrease (add): Vegas [BOP94] decreases its flight-size by one segment for every flight in which its DBCE signals congestion. Tri-S and BFA also employ additive decrease.
- Multiplicative Decrease (mult): The DAIMD [LSM+07] proposal decreases the flight-size by the multiplicative factor: $RTT_{min}/currRTT$ on receiving a congestion signal, where RTT_{min} is the minimum RTT of the connection and $currRTT$ is the current RTT sample. CARD, DUAL, and DECA also rely on multiplicative decrease.

We use both of these policies to assess the impact of a DBCE on the duration of a connection as follows. The use of a DBCE can result in one or more of the following events, in a given flight of events, that impact connection duration:

- *The flight completely*[3] *lies within a congestion phase:* The overall impact on connection duration in this case depends on the CA policy:
  - Additive decrease (add): In this case, the congestion window is reduced by one. If the TCP flight size was x before the congestion event, the new window size becomes $x - 1$, hence it takes the connection one RTT to recover from this reduction, However, the sender also achieves some goodput during this time —in Appendix B, for each such flight, we derive an estimate of the overall increase in connection duration in units of RTT to be:

$$A(x) = \frac{1.5}{x + 0.5} \tag{5.1}$$

  - Multiplicative decrease (mult): In this case, the congestion window is reduced by the factor $\beta = minRTT/RTT$. If the TCP flight size was x before the congestion event, the new window size becomes bx, hence it takes the connection $x - \beta x$ RTTs to recover from this reduction in flight size. However, the sender also achieves some goodput during this time —in Appendix B, for each such flight, we derive an estimate of the overall increase in connection duration in units of RTT to be:

$$M(x) = \frac{\beta(2\beta x + 1)}{2 + \beta} \tag{5.2}$$

[3]We assume that the CA policy reacts only to one indication of congestion per-flight. This is certainly true for most DBCEs that compute only a single estimate of RTT per flight. When a DBCE uses all RTT estimates from a flight, we assume that the corresponding CA reacts only if *all* RTTs signal congestion.

- *A loss, that was subsequently detected by the FR/R mechanism, is predicted (and avoided):* In this case, the sender does not spend any time on detecting and recovering the segment.[4] More significantly, the sender would not reduce its sending rate after "recovering" from the loss. If the TCP flight size was $2x$ before the segment was retransmitted, it will take the sender $x+1$ RTTs to recover the same flight size after exiting FR/R. However, the sender also achieves some goodput during this time —in Appendix B, for each such avoided spurious FR/R-based loss recovery, we derive an estimate of the overall reduction in connection duration in units of RTT to be:

$$F(x) \quad = \quad x + 2 - \frac{3x^2 - x}{5x + 1} \tag{5.3}$$

- *A loss, that was subsequently detected by the RTO mechanism, is predicted (and avoided):* When a sender avoids an RTO-based loss detection/recovery, it saves time spent in detecting the loss equal to the retransmission timeout,t, for that connection (quite significant for RTOs) and in recovering the flight size. To see by how much, assume that the flight size was $2x$ before the RTO expired; on RTO-expiry, the flight size is reduced to 1. It takes the sender $log(x) + x - 1$ RTTs to recover the flight size. However, the sender also achieves some goodput during this time —in Appendix B, for each such avoided spurious RTO-based loss recovery, we derive an estimate of the overall reduction in connection duration in units of RTT to be:

$$R(x) \quad = \quad t + x + \log x - 1 - \frac{3x^2 + x - 4}{5x + \log x - 1} \tag{5.4}$$

For each DBCE, we augment *TCPdebug* to process the trace of each connection and (i) classify each event as one of the above possible types, and (ii) compute the total reduction in connection duration.

Limitations of the Above Approach

While the use of a passive analysis methodology provides us with the opportunity to study a fairly large and diverse set of real world TCP connections, it suffers from the following key limitations that may not arise while using an active experimental approach:

- One of the key assumptions behind the formulation of our evaluation metrics (LPA and FPA) is that the prime purpose of a DBCE is to accurately predict impending loss. In reality, the use of

[4]Recall that we are assuming that the sender would be able to avoid TCP-based loss detection-recovery—see Section 5.3.2.

delay-based congestion control (DBCC) serves two purposes. The first is mentioned above; from the perspective of individual TCP connections, DBCC facilitates early detection of congestion (before network buffers overflow and cause losses) and helps reduce packet losses—this can result in significant reduction in overall connection duration. However, congestion may not result in segment losses in each and every connection that encounters it. However, the use of a DBCC is still useful since all connections react early to congestion and buffer occupancy in routers remains small. This not only reduces queuing delays for traffic but also facilitates the design of high-speed routers that use small on-chip memory for buffering packets [AKM04]. Unfortunately, our metrics do not capture this second purpose of a DBCE—to predict congestion (and not only impending packet loss). Since we rely on a passive analysis of connection traces, it is not possible for us to know the ground truth about network congestion. We hope to use active controlled experiments to evaluate this aspect of DBCEs as part of future work.

- Another key assumption we make in assessing the impact of DBCEs on connection durations is that, if a TCP connection employs delay-based congestion-control, its CA policy can avoid all losses that are predicted by its DBCE. This, however, is likely to happen only if most connections in the network rely on delay-based congestion control (DBCC) and reduce their sending rates on inferring congestion. If the competing traffic instead uses regular loss-based TCP implementation, then the loss can not be avoided. However, even in this case, the DBCC connection can rely on forward-error-correction techniques that transmit redundant data to help recover from losses without requiring the time-consuming TCP loss detection/recovery.

Despite the above limitations, we do believe that our evaluation is useful in (i) assessing the relative performance of existing DBCEs; (ii) for studying what connection characteristics impact it; and (iii) for studying the extent to which real world TCP connections can benefit from DBCC, if it is universally deployed.

5.4 Evaluation Results

5.4.1 Evaluation of Loss Prediction Ability

Each of the nine algorithms mentioned in Table 5.1 was simulated on each connection that appears in our traces. For a given DBCE, and for each connection in a given trace, we computed the loss-prediction

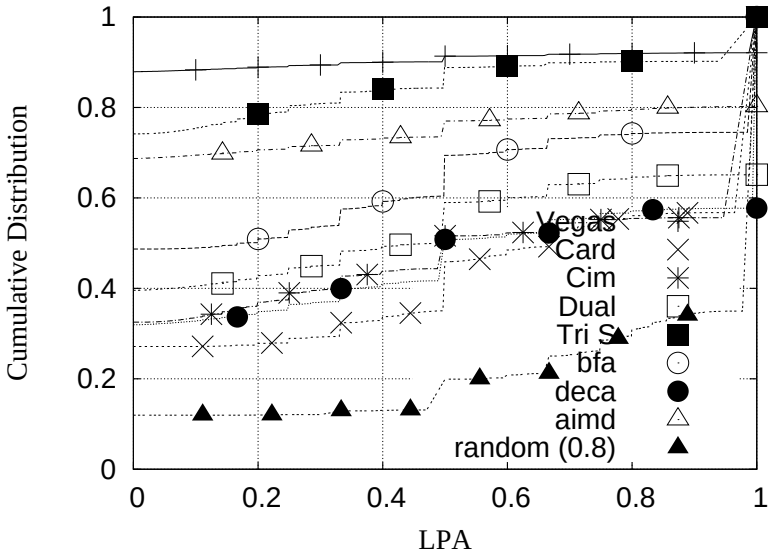

Figure 5.2: Loss Prediction Ability

ability (LPA) metric.

Figure 5.2 plots, on a per-DBCE basis, the cumulative distribution function (CDF) of the LPA values observed across all connections. We find that:

- CARD has the best LPA-performance among existing DBCEs, followed closely by DECA, CIM, and DUAL. These DBCEs can quite successfully predict an impending loss in more than 45% of connections. However, even in CARD, a similar fraction of connections (nearly 45%) have an LPA smaller than 0.5. Thus, even in the best-case, existing DBCEs predict less than half of the losses for many connections.

- More striking is the observation that the Vegas, Tri-S, and DAIMD estimators are inefficient in indicating congestion before a loss—these DBCEs predict none of the losses in nearly 70-90% of connections.

Another issue to consider in evaluating these algorithms is whether a correct indication of congestion (one that precedes a loss event) is sufficiently timely to allow a congestion control mechanism to take action based on the indicator. For many congestion-control designs an indicator of congestion that precedes a loss by at least one RTT should be timely enough to allow effective adjustment of the connection sending rate. Figure 5.3 shows the distribution of the durations of all congestion-phases that precede a loss-episode. The durations are expressed normalized to the current exponentially-

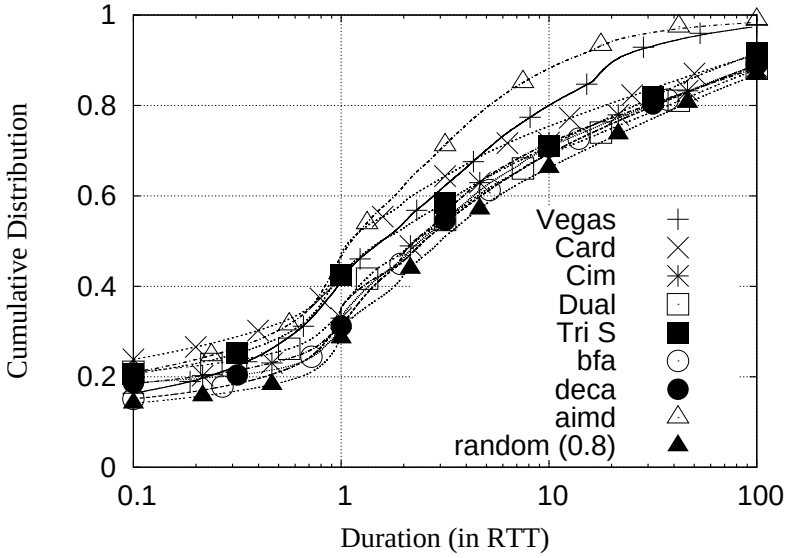

Figure 5.3: Congestion Phase Durations Before Loss

weighted average of the RTT for the corresponding connection (using a weight of 1/8 for the most recent observation). The results are quite consistent for all nine algorithms. They show that 50-70% of congestion-phases that are actually followed by a loss-episode begin at least one RTT before the first loss. This indicates that a congestion control mechanism reacting to the congestion indicator should potentially be able to react in time in order to reduce congestion.

Considering that even the best LPA-performing DBCEs did not perform well for half of the connections, we compared the performance of all the estimators with a *random* congestion indicator, *Random(0.8)*, that predict congestion randomly with a probability of 0.8. To do this, we re-processed the UNC-Campus trace replacing the nine algorithms in our analysis with a simple simulation of a random estimator. For each valid RTT obtained that could be used by one of the algorithms, we generated a random number between 0 and 1. If the number was 0.2 or greater, the connection state was set to indicate congestion and if less than 0.2 it was set to indicate no congestion. This result is also plotted in Figure 5.2. Interestingly, the random estimator performed better than existing DBCEs in terms of the LPA metric. Note that this is to be expected—this estimator is fairly aggressive and indicates congestion 80% of the time. Such an estimator, however, is also likely to yield many false predictions of loss—we evaluate this property next.

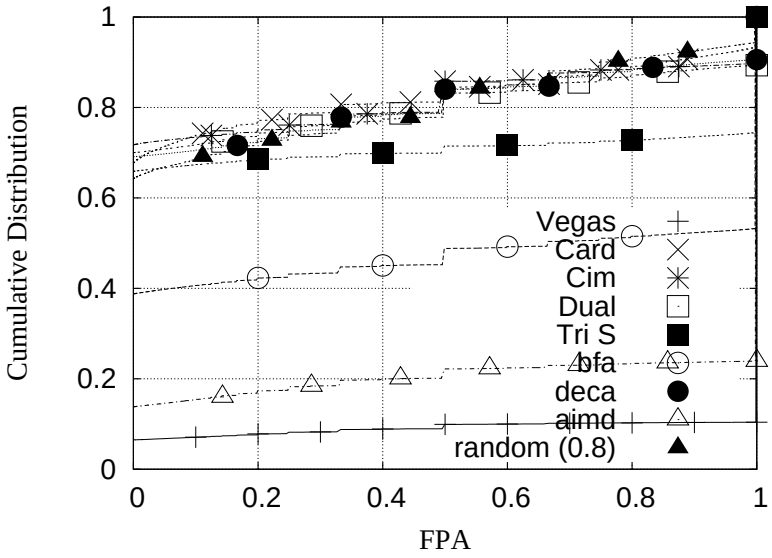

Figure 5.4: False Positives Avoidance Ability

5.4.2 Evaluation of False Positive Avoidance

Figure 5.4 shows the cumulative distribution of FPA over all connections for each estimator. We find that:

- Vegas has the best ability to avoid false positives—it yields no false predictions of loss in more than 90% of connections. This is closely followed by DAIMD that has no false positives in nearly 80% of connections.

  Considering, however, that the LPA-performance of both Vegas and DAIMD is quite poor, the above seems to be an indication that these algorithms are simply too conservative—these are rarely wrong when they indicate congestion, but they miss many instances of congestion severe enough to lead to losses.

- Most of the rest of the DBCEs—including CARD, CIM, DUAL, Tri-S, and DECA—have a poor FPA-performance. Loss predictions are incorrect in 65-75% of connections. In fact, the FPA-performance of these DBCEs is only marginally better than that of the aggressive Random(0.8) estimator. Considering that Random(0.8) has a significantly better LPA-performance than any of these DBCEs, this suggests that a simple random predictor outperforms any of these! We further investigate this issue below by assessing the net effect of LPA and FPA on the duration of each connection.

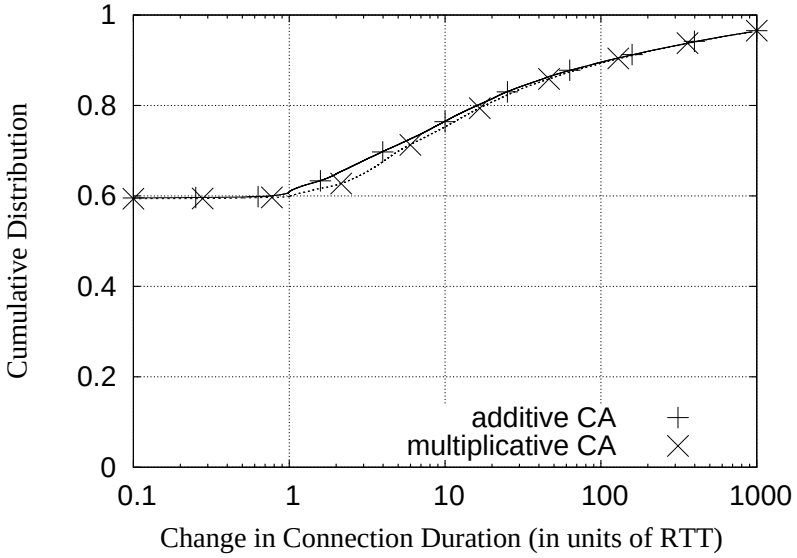

Figure 5.5: Best-Case Savings in Connection Durations

5.4.3 Impact on Connection Durations

We next evaluate the collective impact of LPA and FPA on connection durations using the analysis described in Section 5.3.2. We first characterize an upper-bound on improvement and then compare our estimated performance of DBCEs to this bound. We present each of these analyses below.

Scope for Improving Connection Durations

The LPA- and FPA-related observations made above suggest that existing DBCEs can do much better in terms of predicting losses accurately. A natural question to ask, though, is: *how much better can they do?* Or more relevantly, *what is the maximum amount by which one can hope to improve connection durations by designing a good DBCE?*

In order to address these questions, we attempt to characterize the *best-case* reduction in connection durations that can be achieved by an ideal DBCE. Our analysis is optimistic and assumes that in an *ideal* setting, (i) all losses are preceded by a congestion phase (LPA = 1), and (ii) all congestion phases end in a loss (FPA = 1). In such a setting, we compute the corresponding change in duration for each connection according to the formulation of Section 5.3.2.[5] Figure 5.5 plots the total reduction in duration achieved with the ideal estimator, when the additive and multiplicative CA policies are used, respectively. We find that:

[5]Note that for connections with no losses, the ideal estimator will yield no change in duration.

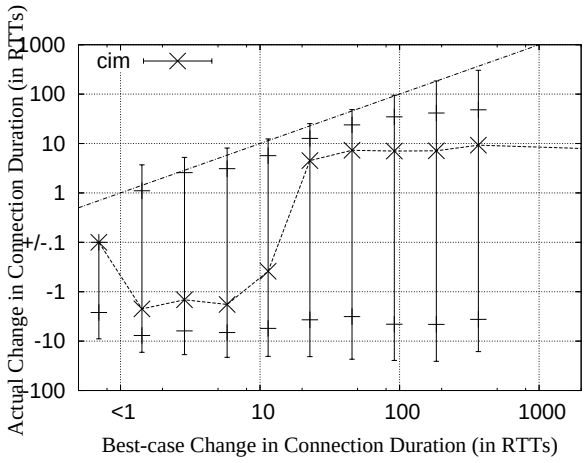

Figure 5.6: Impact of CIM on Connection Durations when an additive congestion avoidance approach is used

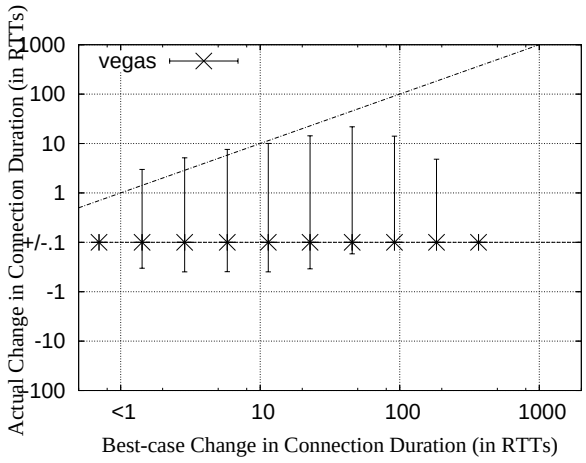

Figure 5.7: Impact of VEGAS on Connection Durations when an additive congestion avoidance approach is used

- The best case saving in connection duration is same irrespective of whether the additive or multiplicative algorithm is used. This is because of the fact that most flows have a very low flight size when they experience a loss. Consider a flow with 3 packets in flight. On experiencing a congestion event the additive algorithm will reduce the congestion window to two. For a multiplicative algorithm the reduction is $minRTT/RTT$ which when greater than 0.66 will also result in a congestion window of two. For a larger ratio of $minRTT/RTT$, the flight size will also have to be larger for the multiplicative algorithm to be more aggressive than the additive algorithm.

- 60% of the connection can gain less than 10% of a RTT. Most of these flows have zero or very few losses and hence cannot gain much by avoiding them.

Below, we compare the performance of existing DBCEs to that of this ideal estimator.

Impact of Existing DBCEs on Connection Durations

As observed above, the scope for reduction in duration can differ by several orders of magnitude across different connections. To put the performance of an existing DBCE in perspective, therefore, in Figures 5.6–5.11, we plotthe estimated duration reduction for each connection (y-axis) as a function of the Best-Case reduction for that connection (x-axis), each computed in units of average connection RTT. For improved readability of these plots, we first divide the x-axis (Best-Case reduction in duration) into logarithmically-sized bins. We then consider all connections that fall within each bin, and compute the median, the 25- and 75-percentiles, and the 5th- and 95th-percentile of the y-axis values (actual

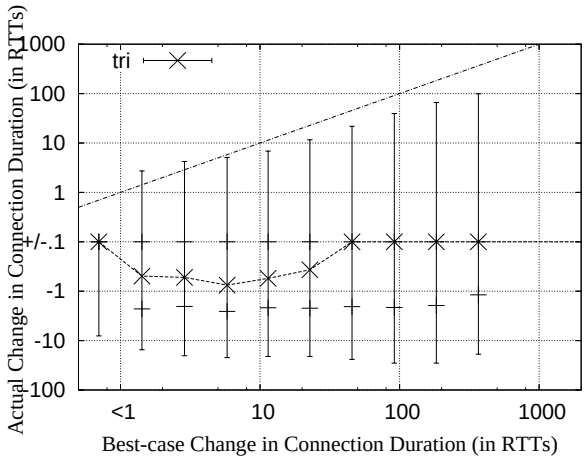

Figure 5.8: Impact of Tri-S on Connection Durations when an additive congestion avoidance approach is used

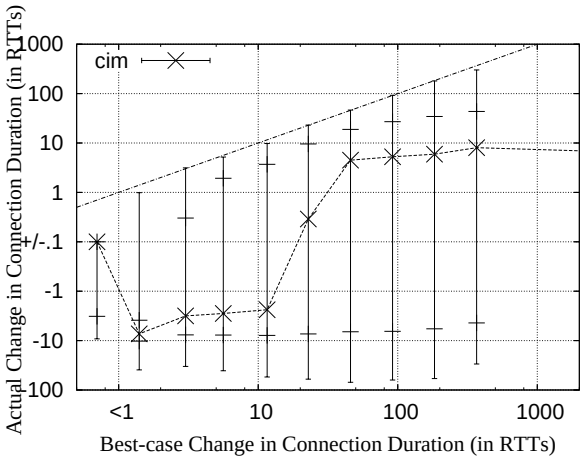

Figure 5.9: Impact of CIM on Connection Durations when an multiplicative congestion avoidance approach is used

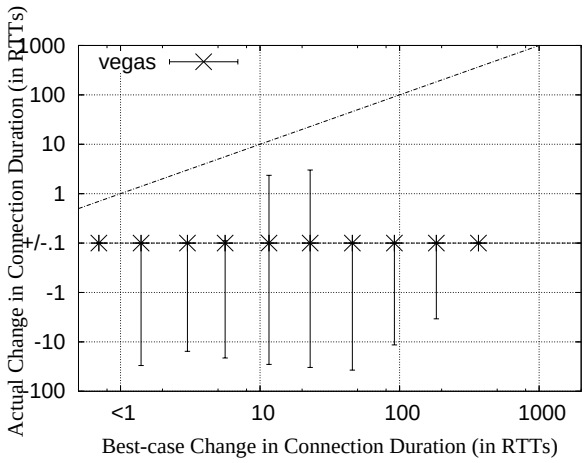

Figure 5.10: Impact of VEGAS on Connection Durations when an multiplicative congestion avoidance approach is used

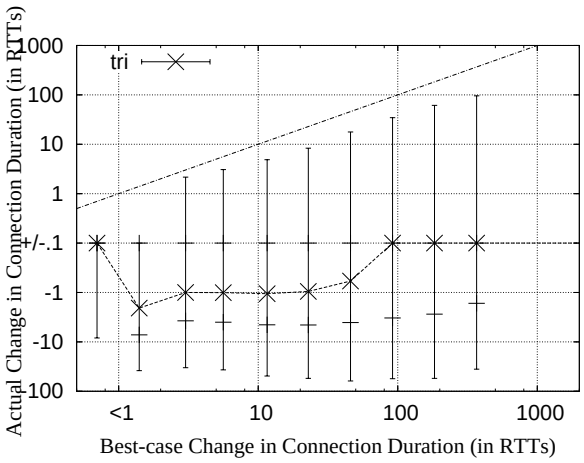

Figure 5.11: Impact of Tri-S on Connection Durations when an multiplicative congestion avoidance approach is used

reduction in duration). We then plot these per-bin median (plotted using the "X" marker), quartiles (plotted using the "−" marker), and percentile values (plotted as error bars) against the average Best-Case reduction in duration for that bin. In Figures 5.6–5.11, we include plots for only Vegas, CIM, and Tri-S—the performance of the other DBCEs are similar to one of these three and are mentioned below. We find that:

- Based on their performance, the DBCEs can be grouped into three categories:
  - The performances of Vegas and DAIMD are similar. Both of these estimators are fairly conservative and rarely predict congestion; the median change in connection duration for all connection-bins is nearly zero!
  - Tri-S and DUAL fall into the second category. Both of these cause more harm than good, increasing the durations of most connections.
  - The rest of the DBCEs fall in the third category. These show considerable improvement in more than 50% of the connections. CIM performs the best in this category. Surprisingly, Random(0.8) performs almost as well as the others—CARD, BFA, DECA.
- The impact of the additive and multiplicative CA policies on most DBCEs are fairly similar. The exception is the conservative Vegas estimator—the rare congestion predictions can cause an increase in connection duration with the multiplicative CA policy. Note that the Vegas estimator signals congestion only if the flight size is large (greater than 4); for large flights, the multiplicative CA policy adversely impacts connection durations to a greater extent.
- Connections with a low potential for best-case reduction in connection duration(XXXX WHAT ARE THESE CONNECTIONS?), have very little to gain by predicting losses; on the contrary, these might suffer an increase in connection duration due to false predictions. A conservative algorithm like Vegas performs the best for these connections.

  On the other hand, for connections with a large potential for reducing connection duration,(XXX AGAIN EXPLAIN THIS) a conservative algorithm like Vegas fails to improve the performance. For such connections, CIM-like estimators are able to improve the performance significantly. We believe that this category consists of large connections that experience a large number of loss-episodes (and hence, have greater potential for improving connection durations)—it is well-known that most of the bytes in the Internet is carried by large connections. Thus, a CIM-like estimator is likely to be a better choice for the Internet than a conservative estimator like that of Vegas.

The above observations suggest that the most prominent estimator (Vegas) is quite conservative to be useful. More aggressive estimators such as CIM are likely to benefit large connection significantly, at the possible expense of small connections—in fact, the most aggressive Random(0.8) estimator, that does not predict congestion only 20% of the time also performs reasonably well for such connections.

5.4.4 RTT Estimates: All or once-per-flight?

Many original DBCE proposals compute only one RTT estimate per flight, as is the common practice in several current TCP implementations. Consequently, the DBCE makes predictions with lower frequency —this is true even for the Vegas estimator. It has been argued that this low frequency of RTT estimation is also less noise-prone and consequently many DBCEs are likely to perform better when they use it.

In order to validate this claim, we repeat all experiments described so far using the low frequency (one per flight) RTT signal. Specifically, we compute a new set of once per flight RTT estimates in exactly the same manner as most of the current implementations of TCP-NewReno do. We then run this signal through each of the candidate DBCEs and observe their performance in terms of the per-connection FPA, LPA, and the total duration savings.

Interestingly, for each DBCE, we find that the distributions of the LPA metric are nearly identical in the two cases: when only once-per-flight RTTs are used, and when all RTT estimates are used. The FPA-performance of all DBCEs are marginally better when the once-per-flight RTT signal is used. However, the overall impact on connection duration is practically identical in the two cases. We conclude that the use of once-per-flight RTT signals neither benefits nor adversely impacts the performance of any DBCE.

We next examine more closely the relationship between characteristics of individual TCP connections (including those of the network path they traverse) and the ability of DBCEs to identify congestion and no-congestion conditions.

5.5 Influence of Connection Characteristics

Several connection characteristics are likely to influence the ability of prominent DBCEs in predicting impending loss (as also argued in [PJD04]). For instance, if the min RTT of a connection is much larger

than the maximum queuing delay experienced at the bottleneck link, a DBCE that uses the minimum RTT as a base value is unlikely to accurately detect connection. Furthermore, if the sending rate (and thus, the RTT-sampling rate) of a connection is too small, a DBCE is unlikely to sample enough RTTs before a segment loss occurs. We study the influence of these and other connection characteristics on the performance of a DBCE. For this, we rely on a connection-clustering approach—our methodology is described in detail below.

5.5.1 Connection Characteristics of Interest

Several characteristics of a TCP transfer are likely to influence the efficacy of a DBCE used for the transfer. We summarize the prominent ones below:

- *Queuing Delays vs. RTTs:* Many estimators use the *relative* increase in RTT as a congestion-predictor. On high-bandwidth paths with long propagation delays, even the maximum queuing delay may be significantly smaller than the minimum path RTT—this limits the magnitude of the relative increase in RTT, and limits the ability of a DBCE to predict congestion. We capture this effect by considering the *ratio of the* 90^{th} *percentile of connection RTT to the minimum RTT.*

- *Loss Rate and Patterns:* The pattern of packet losses in a connection affects a DBCE in several ways. First, for a connection with a high loss rate (high loss episode frequency), an aggressive DBCE that often predicts congestion will perform well w.r.t. the LPA and false positives metrics; while a conservative estimator will perform poorly w.r.t. LPA. At low loss-rates, on the other hand, the performance of these estimators will reverse. Hence, we include the *packet loss rate* of a connection as a characteristic of interest.

  Also, if the gap between consecutive loss-episodes is small, a DBCE may not obtain enough RTT samples to reliably infer congestion. We capture this effect by considering the *average loss distance*, measured in units of RTT, between loss-episodes for each connection.

- *Degree of Self-induced Congestion / Degree of Aggregation:* A key argument used against the use of delay-based congestion-control is that it is likely to work well only when losses are due to self-induced congestion. Connections that traverse highly-aggregated paths might encounter random losses or those caused by congestion due to cross-traffic (as against self-induced congestion). It is not clear if reducing the sending rate of a connection is likely to result in alleviation of congestion

on such paths [PJD04].[6] Furthermore, most TCP transfers are small [CAI] and are likely to experience losses of the latter type.

For self-congested connections (that contribute significantly to the total load on the congested network link), an increase in the number of packets in flight would result in queue-buildup, which in turn will lead to an increase in observed RTTs. We use the *correlation between the number of packets in flight and the RTT* to characterize the degree of self-induced congestion experienced by a connection.

- *RTT Sampling Rate:* A DBCE uses RTT samples to predict congestion. If network congestion evolves slowly and the estimator is able to collect enough RTT samples during this evolution, it should be able to better predict congestion. On the other hand, an estimator that does not get enough RTT samples before a packet loss may not be able to predict the loss. In fact, [PJD04] argues that if the delay sampling frequency is less than the Nyquist frequency, a DBCE may not be effective in predicting losses. They conclude that the frequency should be much larger than the frequency with which buffer overflow occurs at a bottleneck router queue. We characterize the RTT sampling rate of a connection by considering the *average throughput* of a connection.

- *Typical Flight Size:* Finally, we consider the *median flight-size* of a connection. This characteristic is likely to influence the impact of a DBCE on the connection duration—this is because the larger is the flight size when a loss occurs, the larger is the time spent by TCP in recovering the congestion-window (see Section 5.3.2). Thus, connections with larger flights can potentially benefit more from the use of a DBCE.

5.5.2 Clustering of TCP Connections

Our aim is to cluster the TCP connections represented in our traces along the above-mentioned characteristics. We need to address two challenges in order to do so:

- *Range-diversity and skew in the distribution of characteristics:* Each of the connection characteristics spans a fairly distinct range of values; for instance, while RTTs can be as large as several 1000s of milliseconds, the loss rate is never larger than 1.0. Further, the distribution of a given

[6] It is important to note that the above argument is actually against the efficacy of delay-based congestion-control (and not against the efficacy of delay-based congestion *estimation*). Even with respect to congestion-control, the above argument applies only to scenarios in which only a few connections employ delay-based congestion control and compete with regular TCP connections that rely on losses to detect congestion. If on the other hand, delay-based congestion-control is widely deployed, responding to any type of congestion (whether self-induced or not) should help reduce the possibility of buffer overflow (and losses).

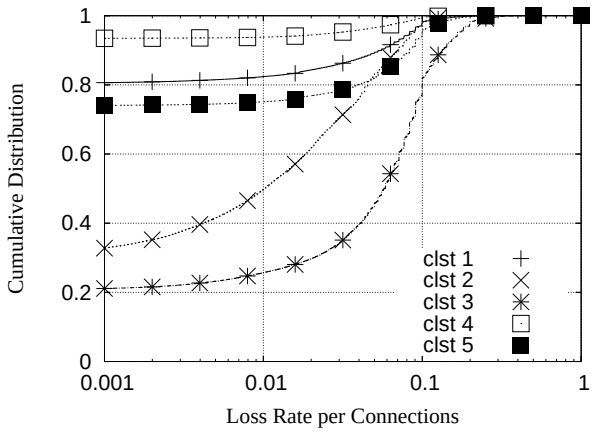

Figure 5.12: Distribution of Loss Rate Across Clusters

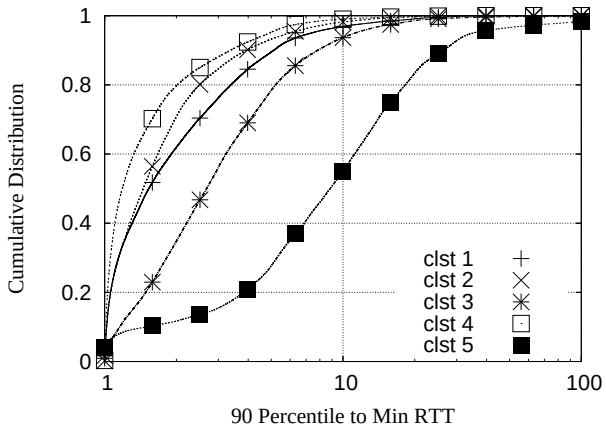

Figure 5.13: Distribution of RTT Variability Across Clusters

characteristic across connections can be quite skewed. In order to eliminate any clustering bias due to this range-diversity and skew, we need to first transform the characteristics into those that have distributions that are less skewed and are similar in span. We do this by first taking the log of each characteristic, C_i, and then normalizing the logs to the range $[0, 1]$ as follows:

$$X_i = \frac{log(C_i) - min\{log(C_i)\}}{max\{log(C_i)\} - min\{log(C_i)\}}$$

It is easy to see that all X_is vary from 0 to 1.

- *Non-independence of characteristics:* Several of the connection characteristics identified above are not independent. For instance, the average loss distance of a connection is likely to be large if its loss-rate is small. Further, the median flight size, the average RTT, and the correlation between RTTs and flight size are also dependent characteristics. In order to eliminate any clustering bias due to such inter-dependence of characteristics, we first transform the set of characteristics into a set of independent dimensions using the technique of *principle component analysis* (PCA) [Fuk90]. This technique yields a set of independent dimensions, which also represent the characteristics that have the maximum variability across connections. We then cluster TCP connections along these transformed dimensions.

We use the Matlab [Inc92, mat] programming environment to conduct PCA and clustering. For the latter, we rely on the K-mean algorithm [McQ67]—we experiment with different values of the number of clusters, K, and present results only the most insightful setting of $K = 5$. More details of our clustering approach can be found in [RKS07b].

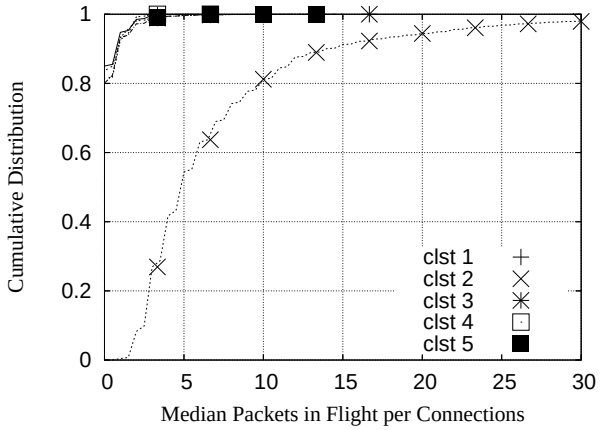

Figure 5.14: Distribution of Median Flight Size Across Clusters

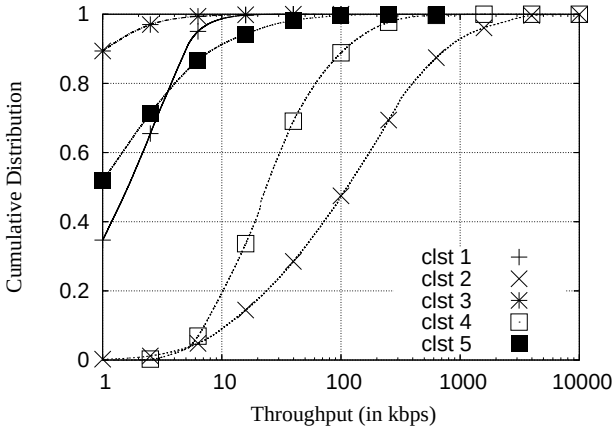

Figure 5.15: Distribution of Throughput Across Clusters

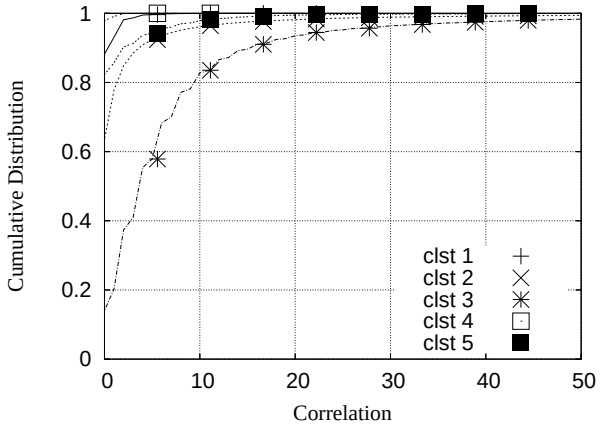

Figure 5.16: Distribution of Correlation between Flight Size and RTT Across Clusters

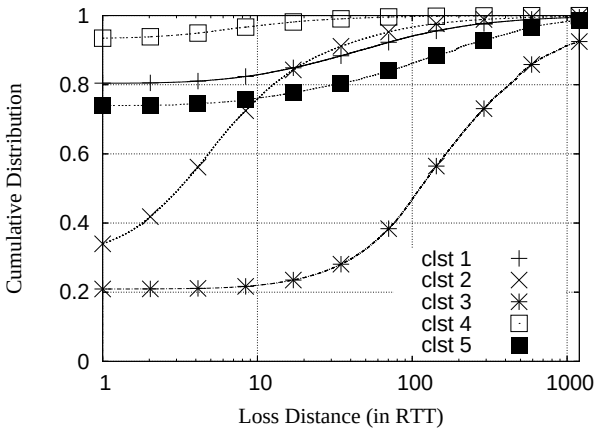

Figure 5.17: Distribution of Loss Distance Across Clusters

Cluster #	# connections	Summary of Key Characteristics
Cluster 1	261898	Low throughput, low loss rate, small flight size
Cluster 2	229718	Very high throughput, high loss rate, large flight size
Cluster 3	184244	Very low throughput, very high loss rate, small flight size, large loss-distance
Cluster 4	281550	High throughput, very low loss rate, small flight size
Cluster 5	88682	Low throughput, low loss rate, small flight size, large RTT variability

Table 5.4: Key Characteristics of Connection Clusters

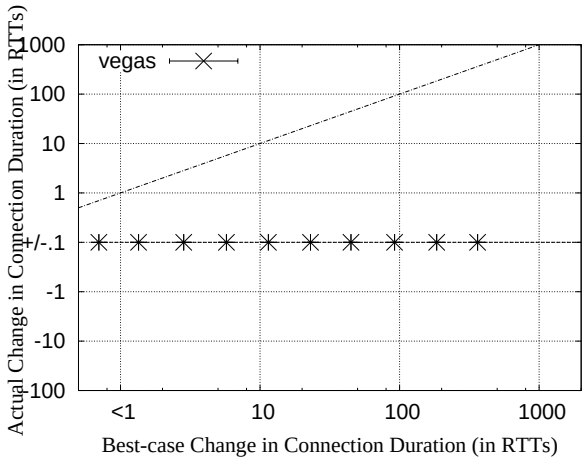

Figure 5.18: VEGAS: Impact on Connection Duration (Add CA) on Cluster 1

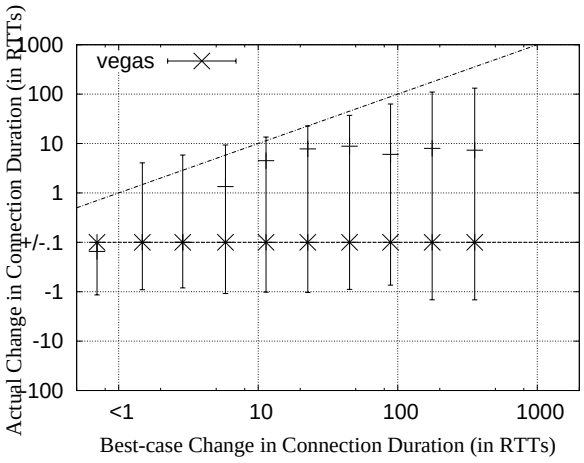

Figure 5.19: VEGAS: Impact on Connection Duration (Add CA) on Cluster 2

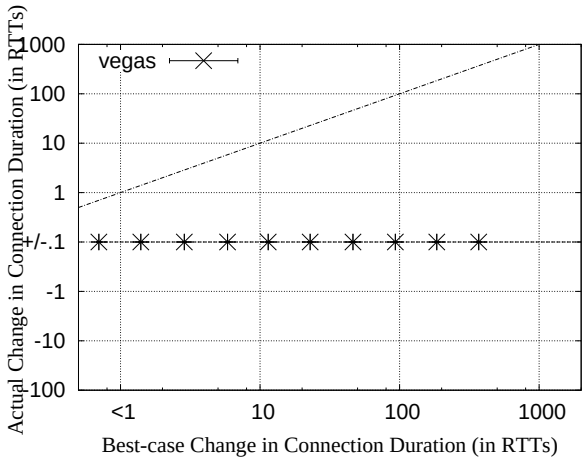

Figure 5.20: VEGAS: Impact on Connection Duration (Add CA) on Cluster 3

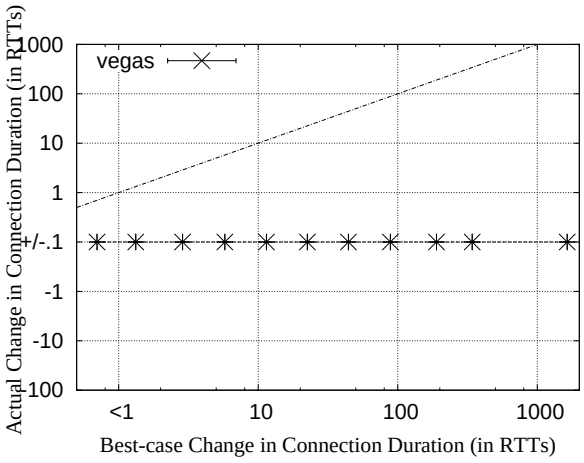

Figure 5.21: VEGAS: Impact on Connection Duration (Add CA) on Cluster 4

Figures 5.12–5.17 plot the distribution of the connection characteristics across the 5 clusters. Interestingly, and somewhat surprisingly, we find that the clusters can differ significantly across most of the characteristics (note the log-scale of the x-axis in most of these plots). We summarize the key distinguishing characteristics of each cluster in Table 5.4.

5.5.3 Performance of DBCEs Across Clusters

We next study the per-cluster performance of all the candidate DBCEs. Figures 5.18–5.29 plot the reduction in connection duration estimated using the additive CA policy for the Vegas, CIM and Tri-S DBCEs (the observations for multiplicative CA are similar). Cluster 5 behaves very similar to Cluster 1 and is omitted. The observations seen with the other DBCEs are similar to one of these three DBCEs

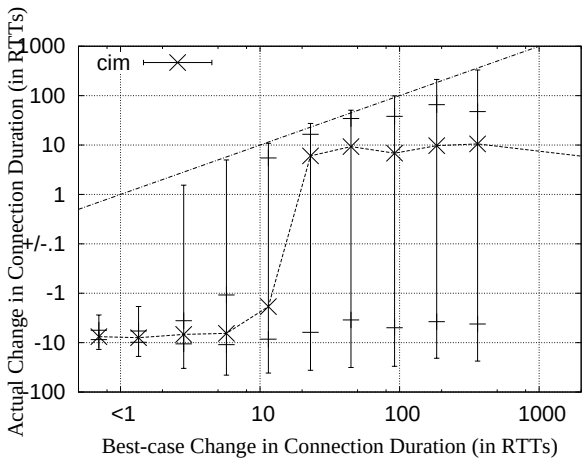

Figure 5.22: CIM: Impact on Connection Duration (Add CA) on Cluster 1

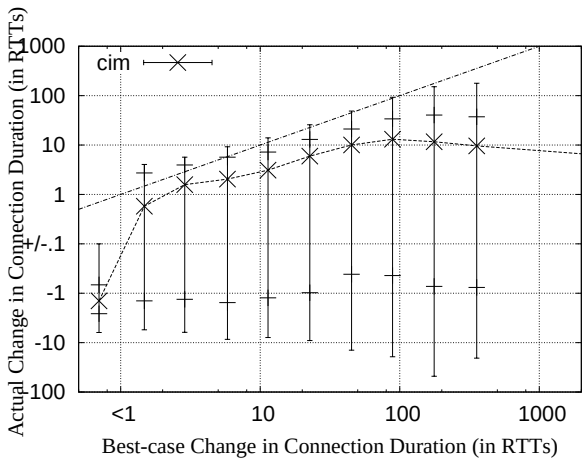

Figure 5.23: CIM: Impact on Connection Duration (Add CA) on Cluster 2

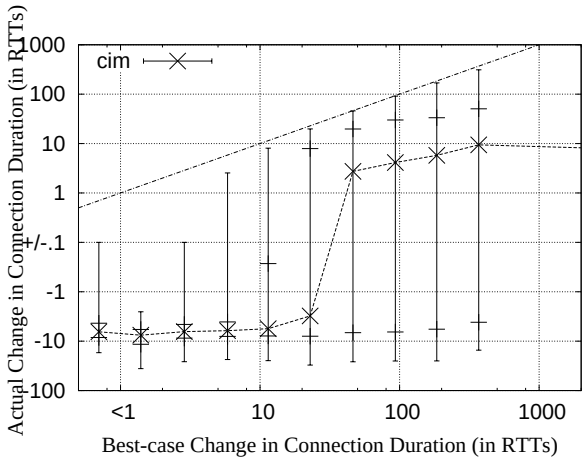

Figure 5.24: CIM: Impact on Connection Duration (Add CA) on Cluster 3

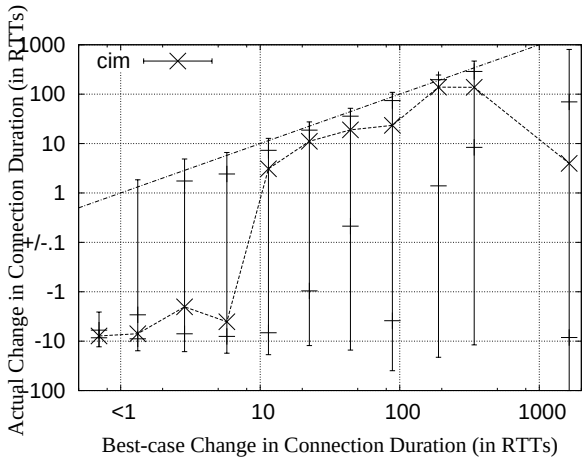

Figure 5.25: CIM: Impact on Connection Duration (Add CA) on Cluster 4

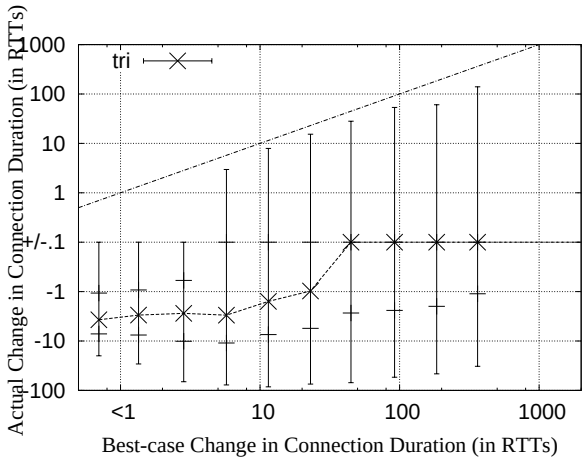

Figure 5.26: Tri-S: Impact on Connection Duration (Add CA) on Cluster 1

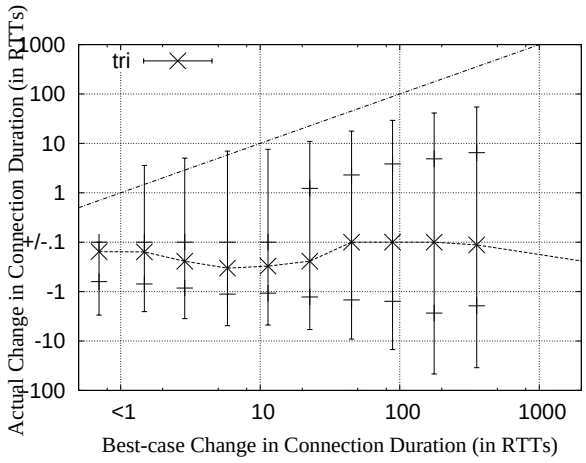

Figure 5.27: Tri-S: Impact on Connection Duration (Add CA) on Cluster 2

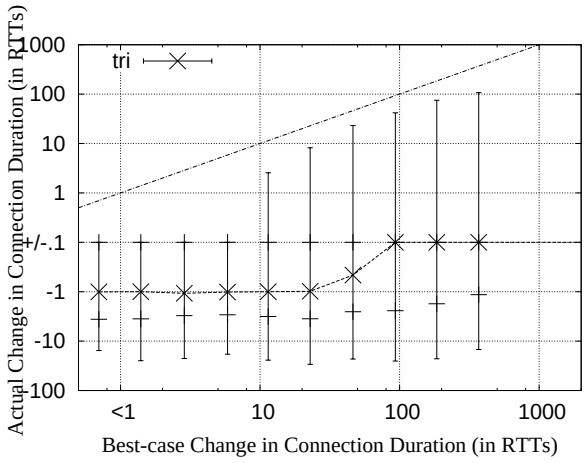

Figure 5.28: Tri-S: Impact on Connection Duration (Add CA) on Cluster 3

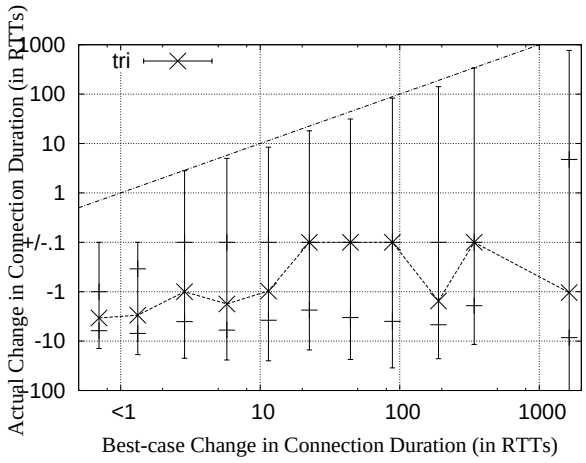

Figure 5.29: Tri-S: Impact on Connection Duration (Add CA) on Cluster 4

(in exactly the same way as seen in Section 5.4.3). We find that:

- Across all DBCEs, Cluster 2 has the best performance in terms of the number of connections that witness a reduction in connection duration; this is especially true for connections with best-case reduction in connection duration within 100 RTTs.
- For connections with a larger best-case potential, Cluster 4 performs better with CIM—in fact, in this case the median reduction in connection duration is fairly close to the best-case.

  For Vegas and Tri-S, the performance of Cluster 4 is similar to that of Clusters 1, 3, and 5.
- The impact on connection duration with Vegas is zero for almost *all* connections in Clusters 1, 3, 4, and 5. Only in Cluster 2, do a significant fraction of connections see a significant reduction in connection durations (by more than 10 RTTs). The only distinguishing characteristic of Cluster 2, not present in any other cluster, is the noticeably large median flight sizes.

From the above observations and Table 5.4, we conclude that (i) connections with high throughput and large flight-sizes (Cluster 2) are likely to benefit the most from any DBCE; (ii) for DBCEs (such as CIM) that perform well on an average among existing DBCEs, a high-throughput connection can perform better with smaller flights if its loss-rate is low (Cluster 4); and (iii) the estimator of the prominent Vegas protocol is fairly conservative and has an impact only on connections that have large flight sizes. We find that the other connection-characteristics do not have any significant impact on DBCE performance.

5.6 Concluding Remarks

We conduct a large scale study with a diverse set of TCP connection traces where we extract the RTTs seen by each connection and used them to evaluate eight prominent delay-based congestion estimators (DBCEs). We tested each estimator's performance in terms of (i) the loss prediction ability (LPA), (ii) fraction of erroneous congestion prediction (false positives), and (ii) the overall impact on connection duration when the DBCE is used in congestion with congestion-avoidance. We also cluster our connection traces using several connection characteristics that are likely to impact the performance of DBCEs; we then study the per-cluster performance of DBCEs. Our main findings are:

- CIM is the overall best estimator. It is likely to reduce the durations of large connections significantly, though at the possible expense of small connections.
- The estimator used by the prominent Vegas protocol is fairly conservative. It has absolutely no impact on the durations of TCP connections that do not transmit large flights of segments.
- Connections with a high throughput and large flight-sizes are likely to benefit the most from any DBCE.

Our high-level conclusion is that the state-of-the-art in delay-based congestion estimation can be improved upon by designing an adaptive DBCE that considers the characteristics of a TCP connection and those of the path it traverses to select either an aggressive or a conservative estimator. Our passive analysis approach does not allow us to study the interaction between delay-based congestion-control and network cross-traffic. We hope to address this using an active experimental framework in the near future.

CHAPTER 6

Conclusions and Future Work

The important thing is not to stop questioning.

— Albert Einstein (1879–1955)

A conclusion is the place where you got tired of thinking.

— Harold Fricklestein

Even though TCP has been around for decades and several studies have looked at the performance of TCP under different conditions, its actual performance in the "real world" is not really well understood. This is especially true for its loss detection and recovery mechanism. The main reason for this is the difficulty of studying TCP behavior in a detailed manner on the real Internet. In this dissertation, I have made a systematic attempt to address this shortcoming to conduct detailed analysis. I believe this work is a significant step in the right direction and provides several insights to modify current TCP implementations as well as to design new better protocols. However, this work is by no means complete. In this Chapter, I will highlight some of the key contributions/results of this dissertation, discuss the implications of these results and suggest future avenues to explore.

6.1 *TCPdebug*

One of the key contributions of this dissertation is the *TCPdebug* tool we developed for the passive analysis of TCP. This tool can be used to analyze large traces quickly and accurately. *TCPdebug* not only integrates several important analysis techniques proposed in the literature but also covers several corner cases ignored by the previous tools. However, the main innovative feature of this tool is the use OS specific TCP implementation details for TCP analysis. *TCPdebug* models the TCP implementation-specific behavior of four Operating Systems (Windows, Linux, FreeBSD/MAC OS X, Solaris) that are currently (and likely to be in the foreseeable future) the dominant end-systems for TCP connections.

The tool is designed to provide a very detailed classification of the out-of-sequence segments seen in real world traces. It also tracks several key characteristics of a trace such as the Round Trip Times (RTTs), packets in flight and advertised window.

This tool is extensively validated using a controlled setting as well as using large number of real traces. The tool is found to be accurate in 99+% of the cases. This is almost a 100% improvement over the current state of the art for TCP analysis tools. *TCPdebug* is freely available to the networking research community and we hope they will encourage others to contribute to our understanding of TCP behavior "in the wild" by analyzing larger and more diverse sets of traces. The tool is also designed to be easily extensible and new features like tracking congestion windows can be easily implemented. In fact, we ourselves have extended this tool to include the algorithms used by several delay based congestion control schemes to predict network congestion. The tool is quite fast and can process several gigabytes of compressed packet header traces in minutes once they are sorted (sorting itself is not a hard requirement and the tool can be easily modified to remove this requirement).

6.2 Loss Detection/Recovery

TCP packet losses severely impact the performance of TCP connections. We use *TCPdebug* to understand the accuracy and timeliness of TCP's current TCP loss detection and recovery mechanisms. We make several interesting observations which are listed below

- 50-80% of TCP loss detections are triggered by the costly RTO rather than FR/R. The main reason for the prominence of RTO is the lack of enough packets in flight to trigger a FR/R based detection.
- 7-35% of all retransmission in TCP are spurious (i.e. TCP inaccurately infer that a segment was lost and retransmits it). Most of the spurious retransmissions are due to RTO based detection as against due to FR/R based detection. Only 3-5% of all retransmission are due to spurious FR/R based loss detection. The number of spurious FR/R based detection was observed to be same for all OSes but the number of spurious RTO based detection was OS dependent. Windows had nearly 5 times more spurious retransmissions than Linux in our traces.
- Up to 14% of all out-of-sequence segments were due to reordering of packets in the network.
- As expected RTO based detection is more time consuming that FR/R based detection. The

time required for FR/R based detection is OS agnostic and is approximately equal to 1-2 RTTs in most of the cases. However, the RTO based detection time depends on the source OS as it depends on the equation used to measure the retransmission timer which differs from one OS to another. The median RTO based detection time for Windows and Linux is 4 RTTs while for Solaris and FreeBSD it is larger than 20 RTTs. RTO based detection for Windows and Linux differ significantly in its tail. While only 10% of Linux connections takes longer than 10 RTT, 25% of Windows connections take longer than 10 RTTs. On close inspection of our traces, we find that several of these larger RTOs in Windows correspond to losses at the beginning of the respective TCP connections, when the RTO is primarily governed by the initial RTO and has not converged to a value representative of the network path. Linux updates its estimates of RTT and RTO at a much higher frequency (once per segment) than Windows (once per flight) and converges faster.

- Overall, the RTO based detection for Linux was observed to converge the fastest and had the least number of spurious retransmissions.

- The time spend in recovering from lost packets is independent of the detection mechanism used and only depends on the number of packets lost within a flight. The recovery time was also found to be independent of the sender side OS. Use of selective acknowledgment (SACK) help reduce recovery time but only when 3 or more segments are lost from a flight.

The above observations indicate that there is much scope to improve TCP performance by improving its accuracy and timeliness. While studying the impact of changing the connection's loss detection parameters on its accuracy and timeliness is straight forward, especially when we use a powerful tool like *TCPdebug* , and has been attempted in the past, understanding impact of changing parameters on the overall performance (connection duration) of a connection is quite difficult in a passive setting. One of the key contributions in this dissertation is the concept of using analytical models to predict the impact of changing parameters on the overall performance of a connection. To the best of our knowledge this is the first attempt to passively study the impact of changing detection parameters on the overall performance of a connection. Using these analytical models we first compute the upper bound on the potential improvement in TCP. We found that while 45-75% of the connections in our traces had very little scope (less than 1%) for improvement in their connection duration, a significant fraction (15-40%) of connections can see more than 10% improvement in their connection durations. However, to achieve this upper bound we have to achieve 100% accuracy in loss detection along with

high efficiency in the timeliness of detection. We studied the actual impact of various parameters on the performance and identified the best settings for these in the real world. Our main observations from this study are as follows

- Current RTO estimators are typically too conservative in incorporating RTT variability. We find that reducing the weight given to the RTT variability (K) from its current default value of 4 to 2 can significantly improve the overall performance of a connection. In fact the achieved improvement is quite close to the upper bound on the improvement indicating that the performance of TCP loss detection is influenced a lot by the weight given to the RTT variability.

- Unlike the observations made in the past studies, the $minRTO$ and timer granularity were found to have almost no influence on the performance of current TCP loss detection mechanism. This is due to the fact that default $minRTO$ used in most current implementations of TCP is usually much lower (200-400ms) than the values used in the past (1000ms). The timer granularity in most current implementations is also much finer (10ms) than that in the past(100-500ms).

- Other parameters used in RTO computations (a, b, m) did not have significant impact on the overall performance of a connection.

- Reducing the dupack threshold (D) helps connections which have fewer packets in flight but harms connections which have more packets in flight. The ideal value of the dupack threshold is thus adaptive depending on the number of packets in flight. The dupack threshold is set according to the rule: $D = max\{1, min\{3, F - 2\}\}$, where F is the number of packets in flight to achieve maximum possible improvement in performance.

- Using the best set of values for all parameters of loss detection mechanism allowed us to almost achieve the upper bound on the performance improvement for a significant number of connections.

6.3 Ability of DBCEs to Predict Losses

We investigate the ability of several prominent Delay Based Congestion Estimators (DBCEs) to predict TCP packet losses with a view to avoid them. This would help reduce the impact on performance that results from detecting and recovering from losses. We considered several prominent DBCEs such as CARD, Tri-S, Dual, Vegas, BFA, CIM, FAST, DECA and DAIMD. We modified *TCPdebug* to

incorporate the equations used by these DBCEs to predict congestion and used it to study the efficacy of each of the DBCE in predicting/mis-predicting losses in a connection. We find that

- CARD has the best performance in predicting TCP packet losses, followed closely by DECA, CIM, and Dual. These DBCEs predicted an impending loss in 45% of the connections. However, in a similar fraction of connections (45%), the DBCEs predicted less than half of the losses in a connection.
- Vegas, DAIMD, and Tri-S did not predict a single loss in nearly 70-90% of the connections.
- Vegas has the best ability to avoid mis-predicting losses. In 90% of cases it does not mis-predict losses even once. This is closely followed by DAIMD that has no mis-prediction in 80% of the connections.
- CARD, CIM, Dual, Tri-S, and DECA erroneously predict losses in a large number of cases.
- Another issue to consider is the timeliness of the loss prediction/mis-prediction. We found that in 50-70% of the cases the prediction/mis-prediction of loss lasted at least for 1 RTT, giving the connection ample time to react to it.

There is much scope to improve the connection's performance by correctly predicting and avoiding packet losses. However, the DBCEs which show a better loss prediction ability also have a large number of mis-predictions while DBCEs which have good performance in terms of mis-predicting losses do not fare well in predicting the losses. We need to know which of the DBCEs strike a good balance and improve the overall performance of a connection. In this dissertation we develop analytical model to study the overall impact of a correct prediction or mis-prediction on a performance of the connections. To the best of our knowledge this has not been done in any of the past work and provides a good metric to compare the performance of these as well as any new DBCEs that may be proposed. Using these analytical models we first computed the upper bound on improvement that could be achieved by a connection if all losses are predicted and avoided and there are no mis-predictions. We computed the upper bound assuming two different reaction policies to the loss prediction - (i) additive decrease in which the sending rate is reduced by 1 packet and (ii) multiplicative decrease in which the sending rate is reduced by a factor calculated as the ratio of minimum RTT to the current RTT ($minRTT/RTT$). We find that the best case saving is same for both the additive and multiplicative decrease policy. This is because, most of the times, a connection has very few packets in flight and a multiplicative decrease of $minRTT/RTT$ amount to a decrease of just one packet. While 60% of connections has negligible

scope for improvement in their performance, 25% of connections could see a saving of more than 10 RTT in their duration.

Using the analytical models mentioned above we studied the performance of the various DBCEs and found that

- Based on their performance, the DBCEs can be grouped into three categories:
  - Both Vegas and DAIMD are fairly conservative and rarely predict congestion; the median change in connection duration is nearly zero!
  - Tri-S and DUAL form the second category. Both of these causes more harm than good, increasing the connection duration for most connections.
  - The rest of the DBCEs form the third category. These DBCEs show considerable improvement in more than 50% of the connections. CIM performs the best in this category. Surprisingly, Random(0.8) predictor of congestion, which predicts congestion with 80% probability, performs almost as well as the others—CARD, BFA, DECA.
- Connections which have low potential to improve their performance still have a lot to lose by mispredicting losses. A conservative algorithm like Vegas performs quite well for these connections. On the other hand, for a connection with large potential for improvement, Vegas do not improve performance much but a more aggressive DBCE like CIM performs well.

Another interesting observation from our study was the realization that the choice of measuring RTT (and hence making a congestion prediction) on a per-flight or a per-packet basis does not impact the performance of the DBCEs much.

Next, we studied the impact of the connection characteristics on the overall performance of a DBCE. We considered a wide range of connection characteristics such as loss rate, flight sizes, throughput, and loss distances. We found that a connection with high throughput and large flight size benefit the most from any DBCE followed by connections with either of these two characteristics. This suggests that DBCEs hold a lot of promise in high bandwidth networks emerging today.

6.4 Future Work

TCPdebug is a very detailed tool and this along with some of the other observations and methodologies used in this dissertation opens up several new avenues to explore in the future. Below I will briefly summarize some of the future direction that I find most interesting.

In this dissertation, I have developed *TCPdebug* as a versatile tool. By tracking several characteristics of the end host *TCPdebug* can accurately classify the cause of retransmission. But this tracking ability can also be used to gain insights into several different aspects of TCP's performance. Below I list a few of these studies

- Using *TCPdebug* we can identify the main performance limiting cause for a connection. If the connection is experiencing losses than the main performance limiting cause, in most likelihood, are the losses. However, there are several other potential causes such as (i) lack of data to send, (ii) advertised window limitation because of a small default advertised window, (iii) advertised window as a limitation because the receiver is not removing packets fast enough and is shrinking the window, and (iv) implementation problems. Studying the main cause of performance limitation will help identify the right set of changes to TCP which will improve its performance.
- *TCPdebug* provides a detailed look at losses in the network. This can be used to investigate models for losses seen in the network. As a first step we can compare the loss patterns observed on the network to those proposed in the literature [MSM97, PFTK98, ARA00, Kum98, LM97, AAB00, ARA00, AT99, MGT99, Pax97a, YMKT99, ZD01]. Given the diversity in the Internet, I believe that a single loss model would not suffice to model losses for different connections. In this case, we should investigate the impact of connection characteristics on the loss pattern seen in the connection. Using these steps, I envision evolution of a hybrid loss model which can explain the patterns observed in a large number of real world connections.

In Chapter 4, we studied the efficacy of current TCP loss detection/recovery mechanism in real world connections and suggested changes which will improves TCP performance in the real world. Most changes we suggested are static. We can imagine a more reactive analysis where *TCPdebug* monitors the network in real-time (we have already noted that this is feasible) and use the information to keep tuning the detection parameters on the go. For example, if a particular connection has seen several spurious RTO based retransmission (identified using *TCPdebug*), we can change the weight of RTT variation for the connection to a higher value to improve its performance.

A more specialized application of the methodology of optimizing TCP loss detection and recovery mechanism proposed in Chapter 4 would be to optimize TCP for a particular environment (such as wireless) or application (such as multimedia streaming). The basic idea is that we can capture a trace of the particular environment or application we are interested in studying and repeat our analysis (with maybe a few specific changes) and derive the set of parameter setting which will work for that trace. A purely 802.11 [Gas02] network may have different network characteristics such as large RTT variability which may require an entirely different set of detection parameters than the general TCP connections. Similarly, a video streaming application may have a larger throughput on an average and may need slightly different optimization for the loss detection mechanism.

Finally, we have investigated the ability of different DBCEs to predict TCP losses and improve TCP performance. Again we used an analysis approach where a single DBCE was used throughout a connections lifetime. We have already noted that a conservative DBCE worked well when the connection has a low throughput overall while an aggressive DBCE works better for high throughput connections. Using this information and *TCPdebug*'s tracking of a connections behavior we can build a modified version of DBCE which can change its aggressiveness depending on the connections characteristics.

Appendix A
Analytical Model: Changing Parameters for Detection Mechanisms

In this appendix, we present the analytical models developed to calculate the change in connection duration for changing parameters of the detection mechanism. We first present derivation of equations to calculate the best possible improvement in connection duration. Next we present 4 different analytical models for (i) an increasing dupack threshold, (ii) a decreasing dupack threshold, (iii) an increasing RTO timer, and (iv) a decreasing RTO timer.

A.1 Model for Best Possible Improvement in Connection Duration

We attempt to characterize the *best-case* improvement in connection durations that can be achieved by an ideal set of loss detection mechanisms to understand the maximum possible benefits from improving the detection mechanisms. Our analysis is optimistic and assumes that in an ideal setting, all spurious retransmissions are avoided, and all loss detection takes no more than the maximum RTT of a connection. We also assume that if we get at least one dupack the connection can detect the loss using FR/R.[1] To compute this metric, we augment the tool to re-process the trace of each connection and compute the savings in connection duration for each of the following four cases:

- *A spurious FR/R-based loss detection is avoided.*

  In this case, the sender would not unnecessarily retransmit a segment. More significantly, the sender would not reduce its sending rate after "recovering" from the perceived loss. We need to calculate the change in recovery time on avoiding the spurious retransmission. Figure A.1 shows a time-sequence graph for the congestion window behavior in presence and absence of the FR/R triggered spurious retransmission. Let the number of packets in flight before the spurious

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

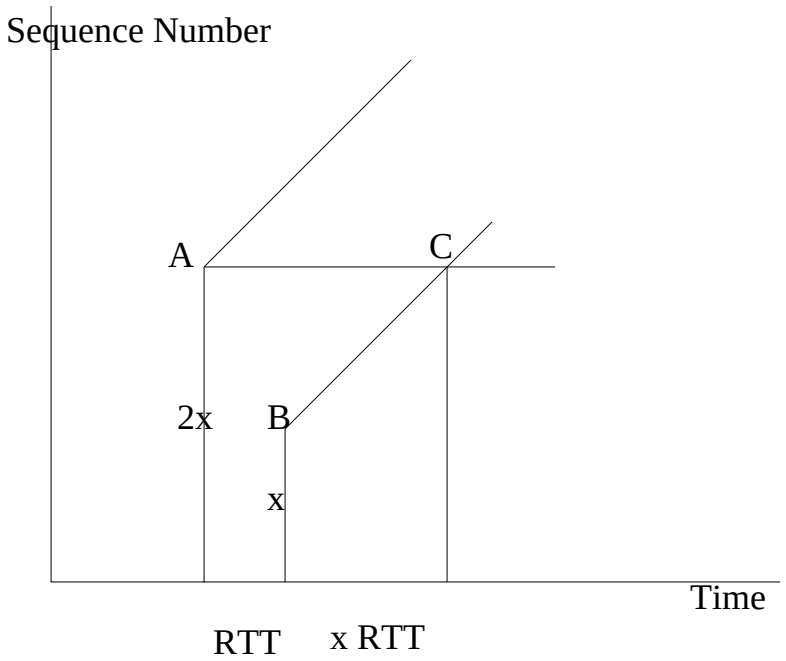

Figure A.1: Congestion window before and after a FR/R based spurious retransmission

retransmission be $2x$ (point A). At time t, the sender experiences a spurious detection and resent the packet and reduces the congestion window from $2x$ to x (point B) and enter congestion avoidance stage [APS99]. Assuming that each packet is acked (i.e. no delayed ack), it will take $x + 1$ RTTs (as shown in figure) to get back to its original congestion window size of $2X$. This is point C in the figure A.1. We assume that from this point onwards the connection behaves exactly as it would have behaved from point A. This is an overly simplistic assumption but is the best we could do in a passive analysis. However, the Difference between point A and C does not capture the complete change in recovery time. In this interval, the sender achieved some useful goodput and we need to subtract this from the increase in connection duration between A and C to get the actual increase in connection duration. The number of packets send out in this time is

$$packet\ count = x^2 + \sum_{i=0}^{x-1} i = x^2 + \frac{(x-1) * x}{2} = 1.5x^2 - 0.5x$$

The average sending rate between A and C in absence of the spurious retransmission would have been

$$sending\ rate = \frac{2x + 3x + 1}{2} = 2.5x + 0.5$$

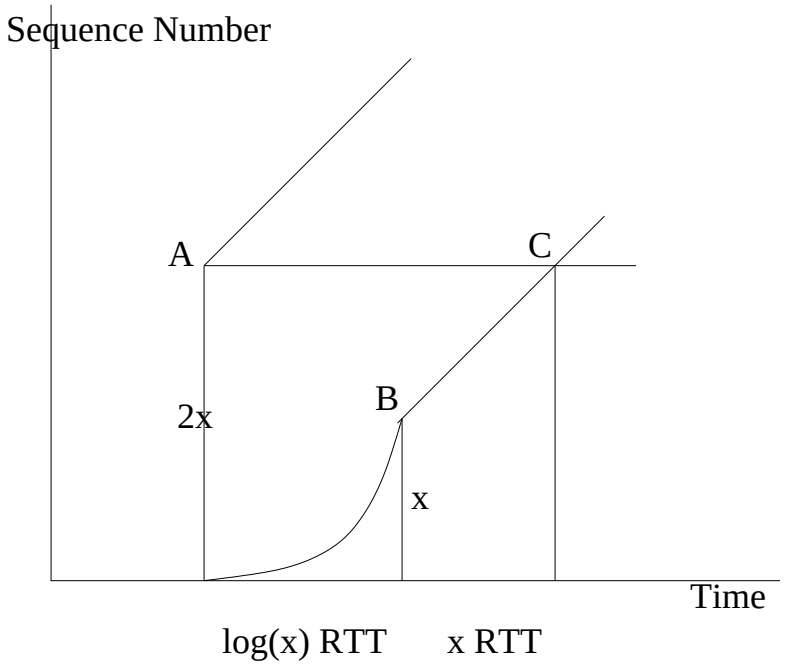

Figure A.2: Congestion window before and after a RTO based spurious retransmission

Hence the time to send out *packet count* packets at the rate of *sending rate* would be

$$send\ time = \frac{1.5x^2 - 0.5x}{2.5x + 0.5}$$

Thus the overall estimate of the saving in connection duration will be

$$F(x) = (x + 1) - \frac{1.5x^2 - 0.5x}{2.5x + 0.5} \tag{A.1}$$

- *A spurious RTO-based loss detection is avoided.* When a sender avoids a spurious RTO-based retransmission, it saves time spent on recovering the flight size. Figure A.2 shows a time-sequence graph for the congestion window behavior in presence and absence of an RTO triggered spurious retransmission. Again, let the number of packets in flight before the spurious retransmission be $2x$ (point A). At time t, the sender experiences a spurious detection and resent the packet and reduces the congestion window from $2x$ to 1 and enter slow start [APS99]. Assuming that each packet is acked (i.e. no delayed ack), it will take $log(x) - 1$ RTTs (as shown in figure) to get to a congestion widow of x and then the connection will enter congestion avoidance (point B). From this point it will take another x RTTs to get back to its original congestion window size of $2x$ (point C). Again, we assume that from this point onwards the connection behave exactly as

it would behave from point A. In the interval A to C, the sender achieved some useful goodput and we need to subtract this from the increase in connection duration. Let $n = log(x)$. Then the number of packets send out in this time is

$$packet\ count = 2^n - 2 + x^2 + \frac{(x-1) * x}{2} = 2^n - 2 + 1.5x^2 - 0.5x$$

The average sending rate between A and C in absence of the spurious retransmission would have been

$$sending\ rate = \frac{2x + 2x + x + n - 1}{2} = 2.5x + 0.5n - 0.5$$

Hence the time to send out *packet count* packets at the rate of *sending rate* would be

$$send\ time = \frac{2^n - 2 + 1.5x^2 - 0.5x}{2.5x + 0.5n - 0.5}$$

Thus the overall estimate of the saving in connection duration will be

$$R(x) = (x + n - 1) - \frac{2^n - 2 + 1.5x^2 - 0.5x}{2.5x + 0.5n - 0.5} \tag{A.2}$$

- *A needed RTO-based loss detection takes only max RTT worth of time.* For non-spurious RTO-based retransmissions, we compute the reduction in loss-detection time (difference between the corresponding value of RTO and the maximum RTT of the connection)—this also gives us the saving in connection duration for each such event.
- *A RTO-based loss detection is converted in FR/R based detection on receiving at least one dupack*

Figure A.3 shows the time-sequence graph for the congestion window when a RTO triggered detection is converted to FR/R. Assume that the flight size was $2x$ when a loss occurred. If the loss is detected by FR/R instead of an RTO, there are two types of savings in connection duration. The first is in the time it takes to detect the loss, and is given by the difference between the time the retransmission timer expired and the time at which the dupacks are received. The second savings is due to the fact that the TCP sending rate after exiting from FR/R (flight size is reduced to x – point A) is usually higher than that after an RTO based detection (flight size is reduced to 1 – point B). From point B, the connection will take $log(x) - 1$ RTTs to get to the window size of x –point C. The difference between A and C is one part of the change in connection duration when RTO gets converted to FR/R. In this interval sender would also achieve some goodput. Let

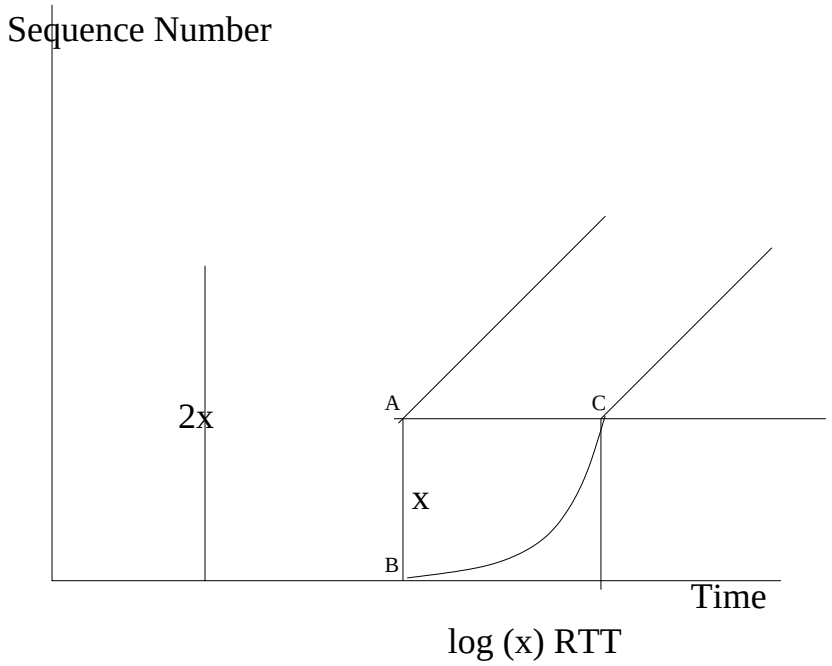

Figure A.3: Congestion window for an RTO and FR/R event

$n = log(x)$. Then the number of packets send out in this time is

$$packet\ count = \sum_{i=0}^{n-1} 2^i - 1 = 2^n - 2$$

The average sending rate between A and C would be

$$sending\ rate = \frac{x + x + n - 1}{2} = x + 0.5n - 0.5$$

Hence the time to send out *packet count* packets at the rate of *sending rate* would be

$$send\ time = \frac{2^n - 2}{x + 0.5n - 0.5}$$

Thus the overall estimate of the saving in connection duration will be

$$RF(x) = (n-1) - \frac{2^n - 2}{x + 0.5n - 0.5} \tag{A.3}$$

A.2 Increasing Dupack Threshold

Increasing the dupack threshold will affect the connection in following three ways

- *Some FR/R will be detected by RTO:* If an FR/R detection is triggered by exactly dupack threshold, D, worth of dupacks, increasing the dupack threshold will cause this FR/R to be detected by RTO. We need to find out, by how much would this change from FR/R to RTO change the connection duration. Figure A.1 captures the difference between a FR/R and the corresponding RTO. Equation (A.3), could be used to capture the change from FR/R to RTO as well. The only difference is that in this case the connection duration will increase instead of decrease by the amount indicated by Equation (A.3).
- *Some Spurious FR/R will be avoided:* If a spurious FR/R is triggered by exactly D dupacks the increase in dupack threshold will cause this spurious retransmission to be avoided. Equation (A.1) formulates the decrease in connection duration when a spurious FR/R is avoided.
- *Slight increase in average detection time for FR/R:* For FR/R triggered detections which had enough dupacks to still be detected by FR/R even after the increase in dupack threshold, there will be a slight increase in the detection duration as the sender has to wait for more dupacks before it can conclude that a packet is lost. This increase is the difference in the dupack time for the new threshold and the old threshold.

A.3 Decreasing Dupack Threshold

The effect of decreasing the dupack threshold will be exactly opposite of that for increasing the dupack threshold. The three ways a decreasing threshold affects a connections is.

- *Some RTO will be detected by FR/R:* If a loss event is accompanied by some dupacks but not enough to trigger FR/R, reducing the dupack threshold can now trigger an FR/R. Equation (A.3), formulated the decrease in connection duration when a RTO based detection gets converted to FR/R.
- *Some Spurious FR/R will be triggered:* Decreasing the dupack threshold will trigger more spurious FR/Rs which were earlier avoided due to higher dupack threshold. Equation (A.1) formulates the increase in connection duration when a spurious FR/R is caused.

- *Slight decrease in average detection time for FR/R:* The sender will have to wait for fewer dupacks and hence will detect the loss slightly earlier. The decrease in connection duration will be the difference in the dupack time for the new threshold and the old threshold.

A.4 Increasing RTO

Changing the various parameters for the RTO equation can cause an increase in the RTO timer. This can affect the connection in three ways

- *Some RTO gets converted to FR/R:* If some losses were detected by RTO simply because the corresponding dupacks were delayed in the network, increasing the RTO could allow these losses to be detect by FR/R. Equation (A.3) Formulates the change in connection duration due to this.
- *Some Spurious RTO is avoided:* Spurious retransmission can happen during RTO based detection, if the ack is simply delayed and the timer expires before the sender gets the ack. By increasing the timer the sender may be able to avoid the spurious retransmission. Equation (A.2) formulates the decrease in connection duration due to this.
- *Increase in detection time for RTOs:* If the RTO is increase, all RTO based detection will take longer to detect the loss. This can be easily measured as the difference between the new and old RTO timer.

A.5 Decreasing RTO

Decrease in RTO can affect the connection in three ways

- *Some FR/R gets converted to RTO:* Triggering RTO earlier could result in some FR/R to be converted to an RTO based detection if the RTO timer expires before the dupacks triggering FR/R arrives at the sender. Equation (A.3) formulates the change in connection duration due to this.
- *Some Spurious RTO is triggered:* Reducing the RTO could cause the timer to expire before the ack for a packet reaches the receiver. This can cause spurious RTOs. Equation (A.2) formulates the increase in connection duration due to this.

- *Decrease in detection time for RTOs:* Decreasing RTO timer will reduce the detection time for an RTO based event. The benefit for this can be measured as the difference between the new and old RTO timer.

Appendix B

Analytical Model: Efficacy of DBCEs

In this appendix, we present the analytical models developed to calculate the change in connection duration when a loss is correctly or incorrectly predicted by a DBCE. We develop models for both the additive and multiplicative response to a congestion estimation as explained in Section 5.3.2 of Chapter 5.

B.1 Model for Change in Connection Duration when congestion is detected

The overall impact on connection duration depends on the congestion avoidance policy used. We present a model for both the additive decrease and multiplicative decrease policy. We employ the corresponding equation for the policy when we encounter congestion. The sending rate is reduced only once per RTT.

B.1.1 Additive Decrease

Figure B.1 shows the time-sequence graph for the case when an additive decrease policy is used to decrease the congestion window by 1 on detecting congestion. If the TCP flight size was x before the congestion event(point A), then the new window will be $x - 1$(point B). Assuming that every packet is acked (i.e. no delayed acks are used), the connection will take 1 RTT to get back to its original sending rate (point C). We assume that from this point onwards the sender behaves exactly as it would have behaved from point A. This is the best we can do in a passive analysis model. However, the difference between point A and C does not capture the complete change in connection duration. In this interval the connection has achieved some goodput. We need to subtract this to get the actual change in connection duration. The number of packets send out during this time is

$$packet\ count = x - 1$$

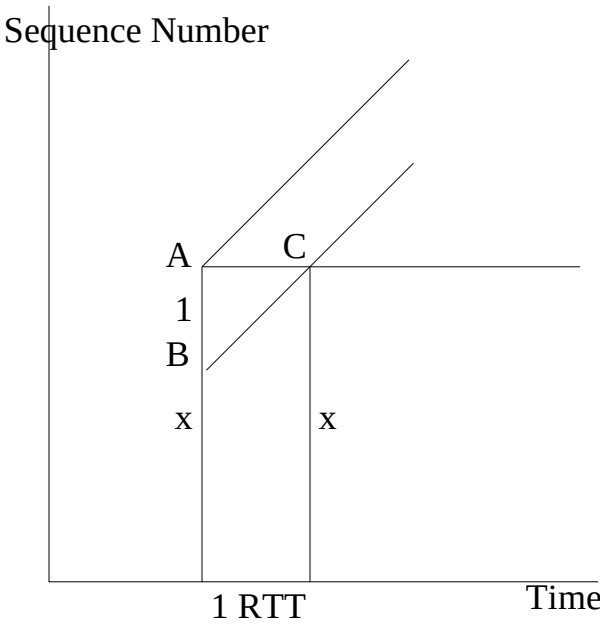

Figure B.1: Congestion window before and after an Additive decrease

The average sending rate between A and C in absence of the congestion event would have been

$$sending\ rate = \frac{x + x + 1}{2}$$

Hence the time to send out *packet count* packets at the rate of *sending rate* would be

$$sending\ time = \frac{x - 1}{x + 0.5}$$

Thus the overall change in connection duration will be

$$\begin{aligned} A(x) &= 1 - \frac{x - 1}{x + 0.5} \\ &= \frac{1.5}{x + 0.5} \end{aligned}$$

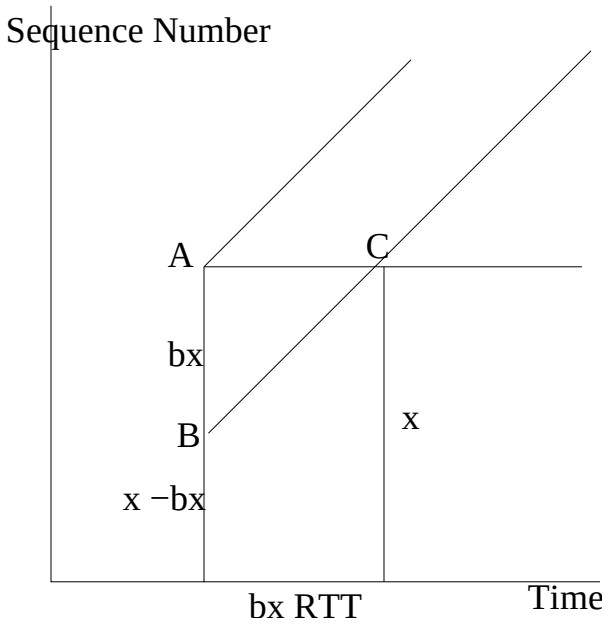

Figure B.2: Congestion window before and after a multiplicative decrease

B.1.2 Multiplicative Decrease

Figure B.2 shows the time-sequence graph for the case when a multiplicative decrease policy is used to decrease the congestion window on detecting congestion. The congestion window is reduced by a factor of $\beta = minRTT/RTT$. If the TCP flight size was x before the congestion event (Point A), then the new window will be $x - \beta x$ (point B). Assuming that every packet is acked, it will take the connection βx RTTs to recover from this reduction (point C). However, the sender also achieves some goodput in this duration. We subtract this to get the actual change in connection duration. The number of packets send out during this time is

$$\begin{aligned} packet\ count &= (x - \beta x)(\beta x) + \sum_{i=0}^{\beta x - 1} i \\ &= \beta x^2 - (\beta x)^2 + \frac{(\beta x - 1)(\beta x)}{2} \\ &= \frac{2\beta x^2 - (\beta x)^2 - \beta x}{2} \end{aligned}$$

The average sending rate between A and C in absence of the congestion event would have been

$$sending\ rate = \frac{x + x + \beta x}{2}$$

Hence the time to send out *packet count* packets at the rate of *sending rate* would be

$$send\ time = \frac{2\beta x^2 - (\beta x)^2 - \beta x}{2x + \beta x}$$

Thus the overall change in connection duration will be

$$\begin{aligned} M(x) &= \beta x - \frac{2\beta x^2 - (\beta x)^2 - \beta x}{2x + \beta x} \\ &= \frac{\beta(2\beta x + 1)}{2 + \beta} \end{aligned}$$

B.2 Model for Change in Connection Duration when loss is predicted correctly

When loss is correctly predicted, we would reduce the congestion window either additively or multiplicatively (depending on the policy used) instead of by half (for FR/R) or to one (for RTO). Thus a accurately loss prediction can be modeled as the saving that would occur if the connection had not spend anytime detecting and recovering from the loss minus the increase in connection duration due to reducing the congestion window according to the policy used. This can be expressed as

$$total\ saving = saving\ in\ detection + saving\ in\ recovery - new\ recovery\ time$$

In 4.1 and 4.2 we have already developed model for the saving in connection duration when the FR/R and RTO based loss is avoided. The saving in detection time for FR/R based loss detection can be measured from a trace as the difference between the data packet and the last duplicate ack triggering the retransmission. Similarly the saving in detection time for RTO based loss detection is equal to the

RTO timer value at the time the packet is retransmitted. The final component (new recovery time) for both additive and multiplicative policy is derived above (equation B.1 and B.1)

B.3 Best Case Change in Connection Duration

The best case change in response time will occur when all the losses are correctly predicted and there is no mis-prediction of loss. This is simply the equations for change in connection duration for loss predictions applied for all losses in a flow.

BIBLIOGRAPHY

[AAB00] Eitan Altman, Konstantin Avrachenkov, and Chadi Barakat. A stochastic model of tcp/ip with stationary random losses. In *SIGCOMM '00: Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, volume 30, pages 231–242, New York, NY, USA, October 2000. ACM Press.

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

[AEO03] Mark Allman, Wesley M. Eddy, and Shawn Ostermann. Estimating loss rates with TCP. *SIGMETRICS Perform. Eval. Rev.*, 31(3), 2003.

[AKM04] G. Appenzeller, I. Keslassy, and N. McKeown. Sizing router buffers. In *SIGCOMM - Special Interest Group on Data Communications*, August 2004.

[AKSJ03] J. Aikat, J. Kaur, D. Smith, and K. Jeffay. Variability in TCP round-trip times. In *Proceedings of the ACM SIGCOMM Internet Measurement Conference*, October 2003.

[All00] Mark Allman. A web server's view of the transport layer. *SIGCOMM Comput. Commun. Rev.*, 30(5):10–20, 2000.

[AP99] Mark Allman and Vern Paxson. On estimating end-to-end network path properties. In *SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 263–274, New York, NY, USA, 1999. ACM Press.

[APRT96] Ender Ayanoglu, Pramod Pancha, Amy R. Reibman, and Shilpa Talwar. Forward error control for mpeg-2 video transport in a wireless atm lan. *Mob. Netw. Appl.*, 1(3):245–257, 1996.

[APS99] M. Allman, V. Paxson, and W. Stevens. RFC 2581: TCP congestion control, 1999.

[AR98] A. Awadallah and C. Rai. TCP-BFA: Buffer fill avoidance. In *HPN '98:Eigth International Conference on High Performance Networking*, pages 575–594, Deventer, The Netherlands, The Netherlands, 1998. Kluwer, B.V.

[ARA00] A. Abouzeid, S. Roy, and M. Azizoglu. Stochastic modeling of tcp over lossy links. In *infocom*, pages 1724–1733, 2000.

[AT99] Farooq Anjum and Leandros Tassiulas. On the behavior of different tcp algorithms over a wireless channel with correlated packet losses. In *SIGMETRICS '99: Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, volume 27, pages 155–165, New York, NY, USA, June 1999. ACM Press.

[BA02] Ethan Blanton and Mark Allman. On making TCP more robust to packet reordering. *SIGCOMM Comput. Commun. Rev.*, 32(1):20–30, 2002.

[BAFW03] E. Blanton, M. Allman, K. Fall, and L. Wang. RFC 3517: A conservative selective acknowledgment (SACK)-based loss recovery algorithm for TCP, April 2003.

[Bev04] Robert Beverly. A robust classifier for passive tcp/ip fingerprinting. In *PAM*, pages 158–167, 2004.

[BFH+00] John Byers, Michael Frumin, Gavin Horn, Michael Luby, Michael Mitzenmacher, Alex Roetter, and William Shaver. Flid-dl: congestion control for layered multicast. In *COMM '00: Proceedings of NGC 2000 on Networked group communication*, pages 71–81, New York, NY, USA, 2000. ACM Press.

[BHL+03] Stephan Bohacek, Joao P. Hespanha, Junsoo Lee, Chansook Lim, and Katia Obraczka. TCP-PR: TCP for persistent packet reordering. In *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 222, Washington, DC, USA, 2003. IEEE Computer Society.

[Bie92] Ernst W. Biersack. Performance evaluation of forward error correction in atm networks. In *SIGCOMM '92: Conference proceedings on Communications architectures & protocols*, pages 248–257, New York, NY, USA, 1992. ACM Press.

[Bol93] 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.

[BOP94] L.S. Brakmo, S.W. O'Malley, and L.L. Peterson. TCP vegas: New techniques for congestion detection and avoidance. *SIGCOMM Comput. Commun. Rev.*, 24(4):24–35, October 1994.

[BRAB06] S. Bhandarkar, A. L. N. Reddy, M. Allman, and E. Blanton. RFC 4653: Improving the robustness of TCP to non-congestion events, 2006.

[BS02] J. Bellardo and S. Savage. Measuring packet reordering. In *IMW '02: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 97–105, New York, NY, USA, 2002. ACM Press.

[BSUB98] Michael S. Borella, Debbie Swider, Suleyman Uludag, and Gregory B. Brewster. Internet packet loss: Measurement and implications for end-to-end qos. In *ICPPW '98: Proceedings of the 1998 International Conference on Parallel Processing Workshops*, page 3, Washington, DC, USA, 1998. IEEE Computer Society.

[BV98a] S. Biaz and N.H. Vaidya. Performance of TCP congestion predictors as loss predictors. Technical report, College Station, TX, USA, 1998.

[BV98b] S. Biaz and N.H. Vaidya. Sender-based heuristics for distinguishing congestion losses from wireless transmission losses. Technical report, College Station, TX, USA, 1998.

[BV03] S. Biaz and N.H. Vaidya. Is the round-trip time correlated with the number of packets in flight? In *IMC '03: Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 273–278, New York, NY, USA, 2003. ACM Press.

[CAI] CAIDA. Characterizing traffic workload. http://www.caida.org/analysis/learn/trafficworkload/.

[CC81] George C. Clark and J. Bibb Cain. *Error-Correction Coding for Digital Communications*. Perseus Publishing, 1981.

[CDS74] Vinton Cerf, Yogen Dalal, and Carl Sunshine. RFC 675:specification of internet transmission control program, September 1974. DARPA Internet Program Protocol Specification.

[CHKW98] D. Chiu, S. Hurst, M. Kadansky, and J. Wesley. Tram : A tree-based reliable multicast protocol, July 1998.

[CMT98] K. Claffy, Gj Miller, and K. Thompson. The nature of the beast: Recent traffic measurements from an internet backbone. In *INET'98*, July 1998.

[CSA00] Neal Cardwell, Stefan Savage, and Thomas Anderson. Modeling TCP latency. In *INFOCOM (3)*, pages 1742–1751, 2000.

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

[FC05] Nahur Fonseca and Mark Crovella. Bayesian packet loss detection for TCP. In *Proceedings of Infocom 2005*, March 2005.

[FF96] K. Fall and S. Floyd. Simulation-based comparisons of tahoe, reno, and SACK TCP. *ACM Computer Communication Review*, 26(3), July 1996.

[FHG04] S. Floyd, T. Henderson, and A. Gurtov. RFC 3782: The newreno modification to tcp's fast recovery algorithm, 2004.

[FHPW00] Sally Floyd, Mark Handley, Jitendra Padhye, and Jörg Widmer. Equation-based congestion control for unicast applications. *SIGCOMM Comput. Commun. Rev.*, 30(4):43–56, 2000.

[FMMP00] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. RFC 2883: An extension to the selective acknowledgement (SACK) option for TCP, July 2000.

[Fuk90] Keinosuke Fukunaga. *Introduction to statistical pattern recognition (2nd ed.)*. Academic Press Professional, Inc., San Diego, CA, USA, 1990.

[Gas02] Matthew S. Gast. *802.11 Wireless Networks: The Definitive Guide*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[GG07] Yunhong Gu and Robert L. Grossman. Udt: Udp-based data transfer for high-speed wide area networks. *Comput. Networks*, 51(7):1777–1799, 2007.

[GGH+03] Robert L. Grossman, Yunhong Gu, Don Hamelburg, Dave Hanley, Xinwei Hong, Jorge Levera, Dave Lillethun, Marco Mazzucco, Joe Mambretti, and Jeremy Weinberger. Experimental studies using photonic data services at igrid 2002. *Future Gener. Comput. Syst.*, 19(6):945–955, 2003.

[GS99] S. Jamaloddin Golestani and Krishan K. Sabnani. Fundamental observations on multicast congestion control in the internet. In *INFOCOM*, pages 990–1000, 1999.

[HAE+03] Eric He, Javid Alimohideen, Josh Eliason, Naveen K. Krishnaprasad, Jason Leigh, Oliver Yu, and Thomas A. DeFanti. Quanta: a toolkit for high performance data delivery over photonic networks. *Future Gener. Comput. Syst.*, 19(6):919–933, 2003.

[HLYD02] Eric He, Jason Leigh, Oliver Yu, and Thomas A. DeFanti. Reliable blast udp: Predictable high performance bulk data transfer. In *CLUSTER '02: Proceedings of the IEEE International Conference on Cluster Computing*, page 317, Washington, DC, USA, 2002. IEEE Computer Society.

[HR06] M. Haeri and A. H. M. Rad. Adaptive model predictive tcp delay-based congestion control. *Computer Communications*, 29(11):1963–1978, 2006.

[Inc92] The Math Works Inc. *MATLAB, High-performance Numeric Computation and Visualization Software. User's Guide*. 1992.

[JAA00] S. Jagannathan, K. Almeroth, and A. Acharya. Topology sensitive congestion control for real-time multicast, June 2000.

[Jac95] V. Jacobson. Congestion avoidance and control. *SIGCOMM Comput. Commun. Rev.*, 25(1):157–187, 1995.

[Jai89] R. Jain. A delay-based approach for congestion avoidance in interconnected heterogeneous computer networks. *SIGCOMM Comput. Commun. Rev.*, 19(5):56–71, October 1989.

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

[JD02] Hao Jiang and Constantinos Dovrolis. Passive estimation of tcp round-trip times. *SIGCOMM Comput. Commun. Rev.*, 32(3):75–88, 2002.

[JE96] S. Jacobs and A. Eleftheriadis. Providing video services over networks without quality of service guarantees. In *WWW Consortium Workshop on Real-Time Multimedia and Web*, October 1996.

[JID+02] 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.

[JID+04] Sharad Jaiswal, Gianluca Iannaccone, Christophe Diot, Jim Kurose, and Don Towsley. Inferring tcp connection characteristics through passive measurements. In *IEEE Infocom*, Hong Kong, March 2004.

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

[JWL03] Cheng J., D.X. Wei, and S.H. Low. The case for delay based congestion control. In *IEEE computer communication workshop*, October,2003.

[KHF06a] E. Kohler, M. Handley, and S. Floyd. RFC 4340: Datagram Congestion Control Protocol (DCCP), 2006.

[KHF06b] Eddie Kohler, Mark Handley, and Sally Floyd. Designing DCCP: congestion control without reliability. In *SIGCOMM '06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 27–38, New York, NY, USA, 2006. ACM Press.

[KKB+04] S. Katti, D. Katabi, C. Blake, E. Kohler, and J. Strauss. MultiQ: automated detection of multiple bottleneck capacities along a path. In *IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 245–250, New York, NY, USA, 2004. ACM Press.

[KLDL04] S. Kang, X. Liu, M. Dai, and D. Loguinov. Packet-pair bandwidth estimation: Stochastic analysis of a single congested node, October 2004.

[KM05] Alexander Kesselman and Yishay Mansour. Optimizing TCP retransmission timeout. In *ICN ' 05: Proceedings of The 4th International Conference on Networking*, pages 133–140, 2005.

[KP88] P. Karn and C. Partridge. Improving round-trip time estimates in reliable transport protocols. In *SIGCOMM '87: Proceedings of the ACM workshop on Frontiers in computer communications technology*, pages 2–7, New York, NY, USA, 1988. ACM Press.

[KSE+04] Rajesh Krishnan, James P. G. Sterbenz, Wesley M. Eddy, Craig Partridge, and Mark Allman. Explicit transport error notification (eten) for error-prone wireless and satellite networks. *Comput. Networks*, 46(3):343–362, 2004.

[Kum98] Anurag Kumar. Comparative performance analysis of versions of tcp in a local network with a lossy link. *IEEE/ACM Transactions on Networking*, 6(4):485–498, 1998.

[LBS06] S. Liu, T. Başar, and R. Srikant. Tcp-illinois: a loss and delay-based congestion control algorithm for high-speed networks. In *valuetools '06: Proceedings of the 1st international conference on Performance evaluation methodolgies and tools*, page 55, New York, NY, USA, 2006. ACM Press.

[LDP+04] L-A. Larzon, M. Degermark, S. Pink, L-E. Jonsson, and G. Fairhurst. RFC 3828: The lightweight user datagram protocol (UDP-Lite), July 2004.

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

[LK00] Reiner Ludwig and Randy H. Katz. The Eifel algorithm: making TCP robust against spurious retransmissions. *SIGCOMM Comput. Commun. Rev.*, 30(1):30–36, 2000.

[LM97] T. V. Lakshman and Upamanyu Madhow. The performance of tcp/ip for networks with high bandwidth-delay products and random loss. *IEEE/ACM Trans. Netw.*, 5(3):336–350, 1997.

[LPPA97] X. Li, S. Paul, P. Pancha, and M. Ammar. Layered video multicast with retransmission (LVMR): Evaluation of error recovery schemes. *Proceedings of the Sixth International Workshop on Network and Operating System Support for Digital Audio and Video*, 1997.

[LSM+07] D. Leith, R. Shorten, G. McCullagh, J. Heffner, L. Dunn, and F. Baker. Delay-based aimd congestion control. In *PFLDnet 2007*, 2007.

[Lyo97] Gordon Lyon. Nmap: URL http://www.insecure.org/nmap/nmap-fingerprinting-article.txt., 1997.

[LZQL07] Yi Li, Yin Zhang, Lili Qiu, and Simon S. Lam. Smarttunnel: Achieving reliability in the internet. In *INFOCOM*, pages 830–838, 2007.

[mat] URL http://www.mathworks.com/.

[MCC] M. Mellia, A. Carpani, and R. Lo Cigno. TStat: TCP STatistic and analysis tool.

[McQ67] J. B. McQueen. Some methods of classification and analysis of multivariate observations. In *5th Berkeley Symp. on Mathematical Statistics and Probability*, pages 281–297, 1967.

[MGT99] V. Misra, W. Gong, and D. Towsley. Stochastic differential equation modeling and analysis of tcp-windowsize behavior, 1999.

[MJV96] Steven McCanne, Van Jacobson, and Martin Vetterli. Receiver-driven layered multicast. In *SIGCOMM '96: Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*, pages 117–130, New York, NY, USA, 1996. ACM Press.

[MMFR96] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. RFC 2018: TCP selective acknowledgement options, 1996.

[MNR03] J. Martin, A. Nilsson, and I. Rhee. Delay-based congestion avoidance for TCP. *IEEE/ACM Trans. Netw.*, 11(3):356–369, June 2003.

[MOM02] M. Morita, H. Ohsaki, and M. Murata. Designing a delay-based adaptive congestion control mechanism using control theory and system identification for tcp/ip networks. volume 4865, pages 132–143. SPIE, 2002.

[Mon97] T Montgomery. A loss tolerant rate controller for reliable multicast, August 1997.

[MSM97] Matthew Mathis, Jeffrey Semke, and Jamshid Mahdavi. The macroscopic behavior of the tcp congestion avoidance algorithm. *SIGCOMM Comput. Commun. Rev.*, 27(3):67–82, 1997.

[PA00] V. Paxson and M. Allman. RFC 2988: Computing TCP's retransmission timer, November 2000.

[Pax97a] V. Paxson. End-to-end Internet packet dynamics. In *Proceedings of ACM SIGCOMM*, volume 27, pages 139–152, New York, NY, Oct 1997.

[Pax97b] V. Paxson. *Measurements and Analysis of End-to-End Internet Dynamics*. PhD dissertation, University of California, April 1997.

[PF01] J. Pahdye and S. Floyd. On inferring TCP behavior. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, 2001.

[PFTK98] 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.

[PJD04] R.S. Prasad, M. Jain, and C. Dovrolis. On the effectiveness of delay-based congestion avoidance. In *PFLDnet*, 2004.

[PKT99] J. Padhye, J. Kurose, and D. Towsley. A model based TCP-friendly rate control protocol. June 1999.

[PLK05] T. Park, J. Lee, and B. Kim. Delay-based congestion control for networks with large bandwidth delay product. In *Fifth International Conference on Information, Communications and Signal Processing*, pages 1245–1248, 2005.

[Pos80] J. Postel. RFC 768: User Datagram Protocol, August 1980.

[Pos81] J. B. Postel. RFC 793:Transmission Control Protocol, September 1981. DARPA Internet Program Protocol Specification.

[PPM07] B. Praveen, J. Praveen, and C. Siva Ram Murthy. On using forward error correction for loss recovery in optical burst switched networks. *Comput. Networks*, 51(3):559–568, 2007.

[Qiu05] B. Qiu. An intelligent algorithm for improving qos in a delay-based end-to-end congestion avoidance scheme. In *Conference on Information Technology and Applications (ICITA'05)*, pages 653–658, Washington, DC, USA, 2005. IEEE Computer Society.

[RHE99] Reza Rejaie, Mark Handley, and Deborah Estrin. RAP: An end-to-end rate-based congestion control mechanism for realtime streams in the internet. In *INFOCOM (3)*, pages 1337–1345, March 1999.

[RKS05] S. Rewaskar, J. Kaur, and F.D. Smith. Passive inference of TCP losses using a state-machine based approach. *Technical Report TR06-002, Department of Computer Science, University of North Carolina at Chapel Hill*, October 2005.

[RKS06] S. Rewaskar, J. Kaur, and F.D. Smith. A passive state-machine approach for accurate analysis of TCP out-of-sequence segments. *ACM Computer Communication Review*, 36(3):51–64, July 2006.

[RKS07a] S. Rewaskar, J. Kaur, and F.D. Smith. A performance study of loss detection/recovery in real-world tcp implementations. In *Proceedings of IEEE ICNP*, Oct 2007.

[RKS07b] S. Rewaskar, J. Kaur, and F.D. Smith. A performance study of loss detection/recovery in real-world TCP implementations. *Technical Report, Department of Computer Science, University of North Carolina at Chapel Hill*, April 2007.

[ROY00] I. Rhee, V. Ozdemir, and Y. Yi. Tear: Tcp emulation at receivers – flow control for multimedia streaming, April 2000.

[RS99] J. Rosenberg and H. Schulzrinne. RFC 2733 an RTP payload format for generic forward error correction, 1999.

[Sav99] S. Savage. Sting: A TCP-based network measurement tool. In *USENIX Symposium on Internet Technologies and Systems*, 1999.

[SBDR05] Joel Sommers, Paul Barford, Nick Duffield, and Amos Ron. Improving accuracy in end-to-end packet loss measurement. In *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, volume 35, pages 157–168, New York, NY, USA, October 2005. ACM Press.

[SKR03] Pasi Sarolahti, Markku Kojo, and Kimmo Raatikainen. F-RTO: An enhanced recovery algorithm for TCP retransmission timeouts. *SIGCOMM Comput. Commun. Rev.*, 33(2):51–63, 2003.

[SRC84] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4):277–288, 1984.

[SRX+04] R. Stewart, M. Ramalho, Q. Xie, M. Tuexen, and P. Conrad. RFC 3758: Stream Control Transmission Protocol (SCTP): Partial reliability extension, 2004.

[SS06] J. Sing and B. Soh. Optimising delay based congestion control for geostationary satellite networks. In *ICCS 2006. 10th IEEE Singapore International Conference on Communication systems*, pages 1–5, 2006.

[Ste93] W. Richard Stevens. *TCP/IP illustrated (vol. 1): the protocols.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[Ste97] W. Stevens. RFC 2001: TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms, January, 1997.

[SW00] Dorgham Sisalem and Adam Wolisz. LDA+: A TCP-friendly adaptation scheme for multimedia communication. In *IEEE International Conference on Multimedia and Expo (III)*, pages 1619–1622, June 2000.

[SXM+00] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. RFC 2960: Stream Control Transmission Protocol, 2000.

[Tal04] G. Taleck. Synscan: Towards complete tcp/ip fingerprinting, 2004.

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

[tcpb] Netflow weekly traffic report, URL http://netflow.internet2.edu/weekly/.

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

[TMW97] K. Thompson, Gj Miller, and R. Wilder. Wide-area internet traffic patterns and characteristics. *Network, IEEE*, 11(6):10–23, November 1997.

[TPB97] Thierry Turletti, Sacha Fosse Parisis, and Jean-Chrysostome Bolot. Experiments with a layered transmission scheme over the Internet. Technical Report 3296, INRIA, 1997.

[TZ99] Wai-Tian Tan and A. Zakhor. Error control for video multicast using hierarchical fec. In *International Conference on Image Processing*, pages 401–405, October 1999.

[VLL05] Bryan Veal, Kang Li, and David K. Lowenthal. New methods for passive estimation of tcp round-trip times. In *PAM*, pages 121–134, 2005.

[VRC98] Lorenzo Vicisano, Luigi Rizzo, and Jon Crowcroft. Tcp-like congestion control for layered multicast data transfer. In *INFOCOM*, pages 996–1003, 1998.

[WC91] Z. Wang and J. Crowcroft. A new congestion control scheme: slow start and search (Tri-S). *SIGCOMM Comput. Commun. Rev.*, 21(1):32–43, January 1991.

[WC92] Z. Wang and J. Crowcroft. Eliminating periodic packet losses in the 4.3-Tahoe BSD TCP congestion control algorithm. *SIGCOMM Comput. Commun. Rev.*, 22(2):9–16, April 1992.

[WCK03] Huahui Wu, Mark Claypool, and Robert Kinicki. A model for mpeg with forward error correction and tcp-friendly bandwidth. In *NOSSDAV '03: Proceedings of the 13th international workshop on Network and operating systems support for digital audio and video*, pages 122–130, New York, NY, USA, 2003. ACM Press.

[WDM01] J. Widmer, R. Denda, and M. Mauve. A survey on TCP-friendly congestion control. *IEEE Network*, 15(3):28–37, 2001.

[WJLH06] D.X. Wei, C. Jin, S.H. Low, and S. Hegde. FAST TCP: Motivation, architecture, algorithms, performance. *IEEE/ACM Trans. Netw.*, 14(6):1246–1259, 2006.

[WJS05] M.C. Weigle, K. Jeffay, and F.D. Smith. Delay-based early congestion detection and adaptation: Impact on web performance. In *Computer Communications*, volume Vol 28/8,, pages pp. 837–850, May 2005.

[YL00] Y. R. Yang and S. S. Lam. General aimd congestion control. In *ICNP '00: Proceedings of the 2000 International Conference on Network Protocols*, page 187, Washington, DC, USA, 2000. IEEE Computer Society.

[YMKT99] M. Yajnik, S. Moon, J. Kurose, and D. Towsley. Measurement and modeling of the temporal dependence in packet loss. In *INFOCOM*, 1999.

[YQC04] Li Yan, Bin Qiu, and Lichang Che. A delay-based end-to-end congestion avoidance scheme for multimedia networks. In *Advances in Multimedia Information Processing - PCM 2004*, pages 389–389, 2004.

[Zal06] M. Zalewski. Passive OS fingerprinting tool: URL http://lcamtuf.coredump.cx/p0f.shtml., 2006.

[ZBPS02] Yin Zhang, Lee Breslau, Vern Paxson, and Scott Shenker. On the characteristics and origins of internet flow rates. In *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 309–322, New York, NY, USA, 2002. ACM Press.

[ZD01] 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.

[ZKFP03] Ming Zhang, Brad Karp, Sally Floyd, and Larry Peterson. RR-TCP: A reordering-robust TCP with DSACK. In *ICNP '03: Proceedings of the 11th IEEE International Conference on Network Protocols*, page 95, Washington, DC, USA, 2003. IEEE Computer Society.

[ZSC91] Lixia Zhang, Scott Shenker, and Daivd D. Clark. Observations on the dynamics of a congestion control algorithm: the effects of two-way traffic. In *SIGCOMM '91: Proceedings of the conference on Communications architecture & protocols*, pages 133–147, New York, NY, USA, 1991. ACM Press.