IMPROVING TRANSMISSION CONTROL PROTOCOL PERFORMANCE WITH PATH ERROR RATE INFORMATION

A thesis presented to

the faculty of

the College of Engineering and Technology of Ohio University

In partial fulfillment

of the requirements for the degree

Master of Science

Wesley M. Eddy

March 2004

This thesis entitled

IMPROVING TRANSMISSION CONTROL PROTOCOL PERFORMANCE WITH PATH ERROR RATE INFORMATION

BY

WESLEY M. EDDY

has been approved for

the School of Electrical Engineering and Computer Science

and the Russ College of Engineering and Technology by

Shawn D. Ostermann

Chair of Electrical Engineering and Computer Science

R. Dennis Irwin

Dean, Russ College of Engineering and Technology

EDDY, WESLEY M. M.S. March 2004.
Electrical Engineering and Computer Science

Improving Transmission Control Protocol Performance with Path Error Rate Information (64pp.)

Director of Thesis: Shawn D. Ostermann

The Transmission Control Protocol (TCP) is designed to reliably transmit data over a wide range of network conditions while responding fairly to other traffic when given an indication of congestion. TCP's inability to distinguish between packet losses due to congestion and those due to corruption, however, makes it perform inefficiently on links with a high rate of packet errors. We describe methods for notifying TCP senders of a network path's packet error rate and ways for using this information to increase TCP's performance while still behaving reasonably in response to congestion signals.

Approved:

Shawn D. Ostermann

Chair & Associate Professor of Electrical Engineering and Computer Science

ACKNOWLEDGEMENTS

The NASA Glenn Research Center's Satellite Networks and Architectures Branch was kind enough to fund me to do much of this work as an intern. Mark Allman served as my mentor there and has provided an unending stream of useful comments and criticicms, helped find and fix bugs in simulator code, provided opportunities to hone my writing and presentation skills, and bought lunch on occasion. The rest of NASA's Satellite Networks and Architectures Branch has also given useful feedback. Specifically, Mike Cauley can be credited for the comments that led to our development of $CETEN_A$.

Shawn Ostermann convinced me to come to graduate school and served as my advisor. His tips on writing and presentation have been tremendously useful, and he provided me with a nice office and a teaching assistant position that allowed me to think about networks rather than C++. His discussions in lab meetings have often conveyed both technical information and philosophical perspective on issues as well. Once in a while, he has even been known to answer email.

The past and present members of Dr. Ostermann's Internetworking Research Group have made time in the lab entertaining and never cried when I broke their mail server.

Rajesh Krishnan was a useful resource for learning more about the original BBN CETEN work and explained some of the design decisions that were made and questions that were left open.

TABLE OF CONTENTS

Page

ABSTRACT 3

ACKNOWLEDGMENTS 4

LIST OF TABLES 7

LIST OF FIGURES 8

1 Introduction 10

1.1 The Transmission Control Protocol 13

1.1.1 Reno TCP 13

1.1.2 SACK TCP 15

1.2 TCP Throughput Model 16

2 Basis for Cumulative Explicit Transmission Error Notification 20

2.1 TCP Modifications for High BER Paths 20

2.1.1 Explicit Transmission Error Notification 22

2.1.2 Cumulative Explicit Transmission Error Notification 23

2.2 Estimating Packet Loss and Packet Error Rates 24

2.2.1 Loss Estimation AlgorithmS for TCP (LEAST) 24

2.2.2 Cumulative Explicit Transmission Error Notification 28

2.2.3 CETEN Simplification 29

3 TCP Congestion Control Modifications with CETEN 32

3.1 Probabilistic CETEN TCP 32

3.2 Deterministic CETEN TCP 37

3.3 Simulation Results 41

3.3.1 Implementation Details 42

3.3.2 CETEN Throughput Gain 43
3.3.3 CETEN Fairness and Friendliness 46
4 Conclusions and Future Work 54
4.1 Conclusions 54
4.2 Future Work 55
BIBLIOGRAPHY 58

APPENDIX

A LEAST Code 62
A.1 Reno TCP LEAST Code 62
A.2 SACK and DSACK TCP LEAST Code 64

LIST OF TABLES

Table Page

2.1 CETEN Header Fields 29

3.1 Congestion Control Event Probabilities 34

LIST OF FIGURES

Figure Page

1.1 Stock TCP Performance from Padhye's Model 17

1.2 Comparison of TCP Congestion Window Behavior 18

2.1 Cumulative Distribution of Actual Loss Rates on the NIMI Mesh 26

2.2 Accuracy of Reno LEAST Algorithm and Raw Retransmissions, Compared to Actual Packet Losses 27

2.3 Accuracy of DSACK LEAST Algorithm and Raw Retransmissions, Compared to Actual Packet Losses 28

3.1 $CETEN_C$ Correct Guesses vs e/p 34

3.2 Congestion Window Behavior After a Corruption Loss 35

3.3 Window Reduction Coefficient Function 38

3.4 Average Congestion Window Size, $p = 2\%$ 40

3.5 Throughput vs Relatative Error Rate, $p = 2\%$ 41

3.6 Percent Error in LEAST Estimate 43

3.7 Simulation Topology 44

3.8 Bulk-Transfer Goodput 45

3.9 Bulk-Transfer Goodput (No Ack Loss) 46

3.10 Bulk-Transfer Goodput (With Congestion) 47

3.11 Fairness Indices - 20 Flows 48

3.12 Fairness Indices - 100 Flows 49

3.13 Aggregate Throughput - 20 Flows 50

3.14 Aggregate Throughput - 100 Flows 50

3.15 CETEN (Un)Friendliness 52

4.1 Example MDF Function Candidates for Future Work 56

1. Introduction

In recent years, wireless networking technologies have been rapidly advancing on many fronts. Notable examples are Bluetooth [41], the suite of 802.11 wireless Ethernet protocols [19], and 3G CDMA cellular telephones [42]. Since by design the Internet Protocol (IP) [33] can run on top of virtually any lower-level network, we see the Internet coming to include a wide range of wireless networks. As the Internet becomes a more important part our lives, having access to it at all times via the devices we carry around in our pockets is a popular idea. Growing demand for Internet-enabled wireless devices like cellphones, laptop computers, and personal digital assistants is one source adding to the number of wireless links present in the Internet. Another comes in geographic areas where "broadband" Internet access isn't available from phone or cable television providers, so in lieu of traditional wired links, many consumers must use wireless terrestrial and satellite services for fast Internet connectivity. We also see military operations increasingly depend upon mobile, wireless, IP-based networks. NASA has also been using and testing Internet protocols in space environments, with projects like the Interplanetary Internet [11] promising to one day add long stretches of wireless hops into the Internet. High bit-error rates (BERs) are typical of most wireless links, and may lead to a large number of packets being discarded due to uncorrectable errors when the forward error correction (FEC) [4] in use is not strong enough to repair all of the damaged bits. This is not a problem generally faced by the wired Internet.

The transport protocol used by the majority of Internet traffic is the Transmission Control Protocol (TCP) [34] which lacks the ability to distinguish between packet

losses due to errors and those due to network resource contention (congestion). The protocol's goal is only to recover from packet losses by retransmission, so distinguishing the causes of losses isn't requiried. By design, standard TCP implementations assume congestion as the cause of all packet losses and slow their sending rates in response [3]. The motivation for slowing down for perceived congestion was that, at the time, there were several congestion collapse [21] events during which the Internet was unusable, and so fixing TCP to send more conservatively in the presence of losses was important. This was not an indefensible decision on the TCP designers' part given that at the time most Internet links were wired, so it was a safe assumption that congestion caused the vast majority of packet losses. This assumption, however, leads to suboptimal performance when TCP is used over networks where packets are dropped due to corruption and in which corruption losses are independent of the congestion level. Since most applications in common use today (web, email, file transfer) run over TCP, we see them perform more poorly than necessary over wireless networks. Our goal is to extend TCP's useful operational range to include networks with high levels of packet loss due to corruption.

There are generally three ways used to improve TCP performance:

1. Make lower layer protocols do more work, for example link-layer retransmissions [5], stronger forward error correction (FEC) [4], etc.

2. Insert processing into the network to transparently fix things, for example proxies [30], spoofing boxes [20], protocol translators [12], etc.

3. Address shortcomings directly and fix the protocol itself. This may be done by adding features (eg SACK [29]) and/or by modifying the TCP algorithms.

Although all three alternatives listed above are effective, we can provide some evidence that the third option may be the best approach. Adding complexity to lower layer protocols goes against much of the acquired wisdom in building complex

scalable systems [37], and middle-boxes that add complexity inside the network rather than at the edge make debugging and maintenance difficult [9]. According to the end-to-end principles [37], the edge of the network is where complexity belongs - in the end-host TCP stacks. In this thesis, we propose such a solution.

Several middleware and end-to-end solutions requiring varying degrees of modification to infrastructure and end-hosts have been proposed, studied, and standardized to improve TCP's performance in the face of high packet error rates [13]. We will describe several of these and their benefits and drawbacks in chapter 2. Then we describe means by which a TCP sender might passively obtain the total packet loss rate with minimal state and how to integrate this with a simplified version of Cumulative Explicit Transmission Error Notification (CETEN) [27] to closely determine the rate at which packet losses are due to errors. In chapter 3, we outline two ways this information can be used to modify TCP's congestion control behavior in order to increase performance without unjustly penalizing other traffic. We show both a probabilistic and a deterministic algorithm and examine the performance of each with regard to both gain in throughput and preservation of fairness to competing traffic via simulations. The remainder of this chapter is a short introduction to the present-day TCP in section 1.1 and its performance deficit when packet errors are introduced in section 1.2.

The work presented in this thesis closely builds on the previous CETEN work by researchers at BBN Technologies [27]. The specific refinements we provide over their work are:

1. We introduced the LEAST algorithms to passively measure the total packet loss rate of a network path (section 2.2.1).
2. We coupled LEAST with the CETEN error rate reports, and in doing so, absolved the need for CETEN corruption rate reports (section 2.2.3).
3. We proposed and tested the $CETEN_A$ congestion control modification, which

has some advantages over the original $CETEN_C$ algorithm that the BBN work presented (section 3.2).

4. We explored both modified TCP congestion control algorithms in the context of fairness and friendliness with regard to competing traffic flows (section 3.3.3).

1.1 The Transmission Control Protocol

The goal of the Transmission Control Protocol (TCP) is to provide reliable in-order delivery of data in a byte stream between two hosts. It is generally implemented as an operating system service that programmers may easily use without knowing all the details of its operation. For many years, TCP has been widely available in modern operating systems, and its use in several popular Internet applications makes it the dominant transport protocol on today's Internet. Over time, many additions and enhancements have been made to the original TCP specification to fix various problems that have come up while maintaining interoperability with old versions (e.g. congestion control [21], selective acknowledgements [29], partial acknowledgements [15], etc). In section 1.1.1 and section 1.1.2, we discuss the common standardized Reno and SACK variations of TCP.

1.1.1 Reno TCP

The name *Reno* comes from the Net/2 release of the 4.3 BSD operating system. This is generally regarded as the least common denominator among TCP flavors currently found running on Internet hosts. Reno TCP's features include slow-start, congestion avoidance, and fast retransmit [40] which were carried over from the previous Net/1 *Tahoe* release, and additionally the fast recovery algorithm [40]. These features are all congestion control related. Originally TCP used the window advertised by the receiver as the amount of data it would send unacknowledged. This was problematic in that the receiver advertises how much buffer space it has made available in memory. Since bytes of end-host memory might be plentiful while router memory may not be, the advertised window can be greater than the buffering capacity of the network.

A TCP sender pushing out a full receive window of data (or many senders pushing many receive windows worth of data) will then cause packet losses as the network's buffering capacity is exceeded. After several instances of congestion collapse which rendered the Internet unusable, an additional state variable, the congestion window [21], was added to the TCP sender and the amount of outstanding data was then limited to the minimum of the advertised receive window and the congestion window.

The congestion window's purpose is to estimate the capacity of the network. It starts out at a small number of segments and uses acknowledgements indicating successful receptions (i.e. sufficient end-to-end capacity) in order to grow. There are two states that determine the rate at which acknowledgements increase the congestion window: whether exponentially in the "slow-start" state, or linearly in the "congestion avoidance" state.

Slow-start refers to the phase used by TCP senders to quickly raise their congestion window to a value approximating the capacity the network path is able to provide. A round-trip time (RTT) is measured as the time between when a packet was sent and when an acknowledgement was received for it[1]. In slow-start, the congestion window doubles with every round-trip time[2]. A TCP sender leaves slow-start when it detects a packet loss, entering the congestion avoidance state. In congestion avoidance, the congestion window grows linearly with each RTT (at most one segment per RTT). Losses occurring in either state result in the congestion window being reduced by half, thus slowing the rate at which packets are sent.

There are two heuristics by which TCP assumes that a packet has been lost, the retransmit time-out (RTO) timer and the duplicate acknowledgement counter. The RTO timer provides a heuristic for how long to wait for a transmitted packet to be acknowledged before assuming it was lost. The duplicate acknowledgement counter

[1] This is a very rough definition, in reality Karn's algorithm [24] is employed to prevent error caused by the ambiguity of determining what triggered acknowledgements of retransmitted segments.

[2] Many TCP implementations support delayed acknowledgements [10], where one acknowledgement is sent for every b data packets received. This slows down the rate of congestion window growth [1].

tracks the number of times packets with the same cumulative acknowledgement are consecutively sent by the receiver while other transmitted data remains outstanding. When this counter exceeds some threshold (usually three), the packet after the one whose receipt has been acknowledged multiple times is assumed to be lost and is retransmitted. This is termed a *fast retransmit* because it doesn't involve waiting for the full RTO timer to expire.

After either a timeout or a fast retransmit, the congestion window is reduced. The fast recovery algorithm in Reno allows it to grow back to its previous level more quickly by using further duplicate acknowledgements after a fast retransmit to increase the congestion window. This is a valid course of action since such acknowledgements indicate that some subsequent segments after the lost one were successfully received.

1.1.2 SACK TCP

The cumulative acknowledgements TCP uses are not capable of indicating if any data in the transmitted window above the cumulatively acknowledged point was successfully received. This can result in the sender retransmitting data that wasn't lost, which wastes time and utilizes resources inefficiently. By using the Selective Acknowledgement (SACK) option, TCP receivers are able to report ranges of data above the cumulative acknowledgement that were not lost and don't need to be resent. This allows the TCP sender to know exactly what holes in the sequence space need to be filled and to efficiently retransmit the segments needed to fill those holes.

The SACK option is defined in RFC 2018 [29]. It could, in theory, be implemented on any flavor of TCP, but generally it augments systems with at least Reno capabilities, so we will use the name SACK TCP to describe stacks with both Reno features and support for the SACK option. The SACK option is widely available in modern operating systems (Windows 98 and 2000, Linux, Solaris, IRIX, OpenBSD, and AIX to name several) and may be expected to become nearly ubiquitous in the future, although implementing the proper exchange of SACK information is not equivalent

to implementing an efficient retransmission algorithm that uses the acquired SACK information [8]. In addition, there is an enhancement to the SACK information that a TCP receiver may implement called Duplicate SACK (DSACK) [17] which allows senders to be notified of unnecessarily-retransmitted segments. The implementation involved for sending DSACKs is trivial and currently available in some open-source operating systems (for example, Linux), although again, intelligent use of received DSACK information is not widely deployed.

1.2 TCP Throughput Model

The throughput a TCP sender can achieve is limited by the amount of data it can have outstanding (unacknowledged). In terms of packets, this is the number of maximum-sized segments that can be sent per round trip time. For simplicity, the maximum segment size and round trip time are assumed to be fixed properties of the given network path. The amount of outstanding data allowed at any time is the minimum of the receiver's advertised window and the sender's congestion window. For reasonable bulk-transfer performance, the advertised window (the amount of buffer space the receiver is willing to devote to a connection) should be large [39], and we generally assume it to be large enough that the throughput of a TCP connection will be determined mainly by the congestion window as a measure of network capacity unbounded by the advertised window.

With several reasonable assumptions, it's possible to derive accurate models predicting the average congestion window over time. Padhye *et al* [31] develops one such model and uses it to obtain a simple equation (equation 1.1) relating TCP throughput to the overall packet loss rate.

$$B(p) = \frac{MSS}{RTT}\sqrt{\frac{3}{2bp}} + o(\frac{1}{\sqrt{p}}) \tag{1.1}$$

In equation 1.1, $B(p)$ is the throughput attained in bytes per second at packet loss rate p, with maximum segment size of MSS bytes, round-trip time RTT seconds,

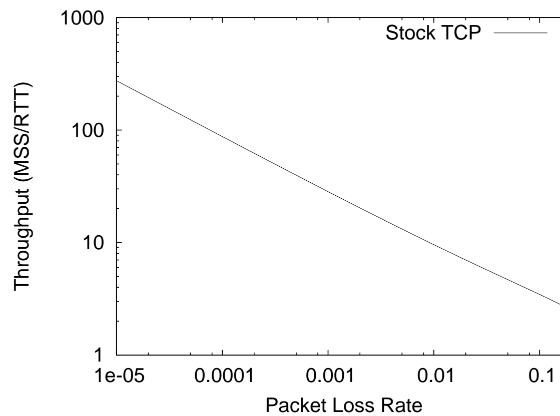

Figure 1.1. Stock TCP Performance from Padhye's Model

and b the delayed acknowledgement threshold (the number of data packets that immediately generate an acknowledgement when delayed acknowledgements are used). The $o(\frac{1}{\sqrt{p}})$ term is used to simplify the form of the equation and indicate that there are less significant additional terms which are on the order of $\frac{1}{\sqrt{p}}$. This relation is represented graphically in figure 1.1 on a log-log scale.

Figure 1.1 and equation 1.1 clearly show that the achievable TCP throughput falls off very quickly as the packet loss rate grows. The packet loss rate p has two components, the congestion rate and the packet error rate, c and e respectively, such that $p = c + e$. In some types of shared-access networks, a high level of congestion can cause corruption as multiple sources attempting to transmit interfere with each other. This is only true of a small subset of network types however, and more commonly packets are corrupted independently of the congestion level. If corruption events are uncorrelated to the congestion level, then TCP need only reduce its congestion window as if the loss rate were c (where $c < p$), and can obtain better throughput than with current congestion control algorithms slowing down for the full loss rate p.

Assuming that an application provides enough data to be sent to fill an entire congestion window at all times, the integral of the congestion window over time

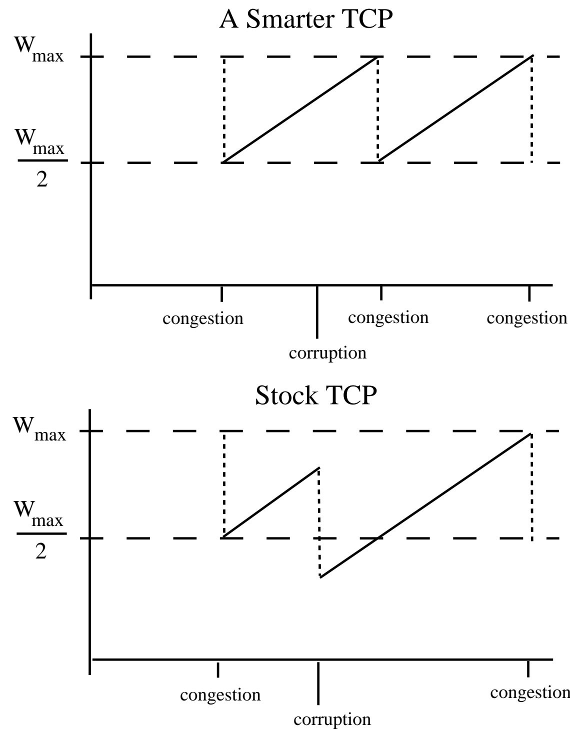

Figure 1.2. Comparison of TCP Congestion Window Behavior

yields the amount of data transmitted. This means that the area under a plot of the congestion window should be maximized for best throughput. Figure 1.2 shows how a modified TCP that doesn't slow down for packet errors outperforms a stock TCP by this principle, where W_{max} is the buffering capacity of the network and thus the maximum achievable TCP window. Congestion and corruption events are marked in time on the x-axis. Since the x-axis is time, and the y-axis represents the amount of data allowed to be outstanding, the area under the curve gives the total amount of data transmitted which we can see is clearly larger for the TCP that is able to distinguish corruption losses from congestion losses. This plot represents a small slice of a connection, in a long-lived bulk-transfer situation, this scenario will be repeatedly played out, meaning the difference in throughput over the course

of the connection might be quite dramatic. In this thesis, we specifically explore bulk-transfer connections and means for keeping packet errors from reducing their congestion windows to unnecessarily small values.

2. Basis for Cumulative Explicit Transmission Error Notification

2.1 TCP Modifications for High BER Paths

In wireless channels with high bit-error rates, TCP packets may frequently be lost due to corruption rather than congestion, and TCP's response of slowing its sending rate for all losses can severely and unnecessarily degrade performance. This TCP problem is well known among the research community and several solutions have been proposed attempting to mitigate it [28]. From figure 1.1 it is clear that reducing the rate at which packet losses (p) cause congestion window reductions to be made will lead to significant throughput gains.

The spectrum of solutions can be grouped into three categories according to Balakrishnan *et al* [6]: split-connections, link-layer mechanisms, and end-to-end solutions. Each of these three classes has its own advantages and disadvantages.

- *Split-Connections*

  The split-connection approach hides a lossy link by sandwiching it between two middle-boxes that locally buffer and retransmit lost data packets transparently to the endpoints. This prevents the TCP endpoints of a connection from seeing those losses. Packet errors on the link appear only as variations in the round trip time. The main advantages of these solutions are that they are easy to deploy by the owners of the lossy link without requiring coordination with end-users, and they can repair holes caused by corruption more quickly than end-to-end retransmission. These approaches can cause a large amount of variation (jitter) in RTT which is a problem for TCP as it may lead to unreasonable

RTO timer values. In some retransmission-based schemes for correcting packet errors, reordering of packets within a TCP stream may be introduced, which can also be detrimental to TCP performance.

- *Link-Layer Mechanisms*

  Link-layer mechanisms for improving TCP performance use either Automatic Repeat Request (ARQ) retransmissions (similar to the split-connection approach), or heavy forward-error correction (FEC). The Snoop Protocol [7] is one example of an ARQ technique that is TCP aware and locally buffers and retransmits TCP packets that are corrupted. Such ARQ techniques suffer the same problems as split-connection approaches, and if they treat TCP-carrying packets differently than other IP packets, may introduce fairness and quality-of-service issues among traffic classes. These would also prove ineffective when traffic is encrypted at the network layer, such as in IPSec [26]. Strong FEC is generally more friendly but reduces the amount of bandwidth available by transmitting the same data multiple times in the form of redundancy bits for error-correction , while also introducing some additional amount of buffering and delay (dependent on the coding scheme and block size used) for encoding and decoding.

- *End-to-End Schemes*

  End-to-end schemes involve modifying only the end hosts' TCP stacks. The main advantages to this approach are that it may function without requiring help from routers and middle-boxes (and thus be more scalable), it may be easier to modify end-hosts than the network infrastructure, and it minimizes the amount of "voodoo[1]" occurring in the network that could lead to other unforeseen problems. The disadvantages are that end-to-end solutions require

[1] Any transparent components of the network can make debugging problems more difficult or cause unexpected (seemingly magical) results when changes occur.

more testing because they optimize for an entire path rather than a single link and rely on operating system vendors and consumers for deployment.

Of these three types of approaches, the end-to-end methods may in some sense be regarded as the most correct as they have the least impact on the network's design, do not involve preferentially treating classes of traffic, scale well across heterogeneous link technologies and administrative domains, and should be easily deployable via operating system patches. For these reasons, the methods we develop in this thesis are of the end-to-end variety.

2.1.1 Explicit Transmission Error Notification

The suite of solutions described by Explicit Transmission Error Notification (ETEN) developed by Krishnan *et al* [27] are end-to-end approaches that involve modifying TCP to understand control messages from routers that indicate packet corruption in a TCP stream. The idea is that when a router notices a checksum failure or other sign of corruption on an incoming packet frame, it can attempt to extract the address of one of the connection end points and notify that endpoint before discarding the packet, rather than silently throwing the packet away. With this type of system in place, individual packet losses could be handled as corruptions if an ETEN message was received or as congestion otherwise. Full deployment of Explicit Congestion Notification (ECN) [36] across a path would also allow such a decision in reverse, treating losses as congestion given an ECN signal (such as an ICMP source quench packet) and as corruption otherwise.

The advantage of per-packet ETEN mechanisms is that the correct congestion control decisions can be made with good certainty, and even if ETEN isn't fully deployed through the path, the TCP will still benefit (although more conservatively). One disadvantage is that it requires a change to the routers that need to identify and notify sources before dropping packets. The major drawback however is that by the nature of corruption, packets are mangled and it may not be possible to correctly

extract the source or destination addresses and ports due to corruption. It may not even be possible to tell if these fields are in the portion of a packet that has suffered corruption, making their validity ambiguous. Extraction of this information also may be impossible if encryption methods like IPSec are in use.

2.1.2 Cumulative Explicit Transmission Error Notification

To overcome the difficulty of correctly identifying a corrupted packet's source and destination endpoints, Cumulative Explicit Transmission Error Notification (CETEN) uses a distributed computation to propagate a packet error rate across the entire network path of a connection. This rate can then be used by a TCP sender to help determine how often it should be responding to congestion when it detects losses. It doesn't, however, provide per-packet loss information, so the TCP sender is still unsure of the cause of any given packet loss, only knowing the probability of a packet error. It also doesn't provide explicit notification of packet losses, leaving that decision to the standard TCP mechanisms of the RTO timer and duplicate acknowledgment counter.

The original work we present in this thesis is based on the CETEN of Krishnan *et al* [27]. This original CETEN work has routers relay a path's overall rates of packet loss due to congestion (c) and corruption (e) to connection endpoints. With this information, after inferring a packet loss, the TCP sender probabilistically takes the standard congestion response at rate $\frac{c}{e+c}$ and otherwise does not reduce its sending rate. We propose and test several modifications to this scheme. Notably, we refine the method for computing the error rate for a link, introduce a deterministic congestion control algorithm in addition to the probabilistic version, and remove the need to compute and forward the path congestion rate in tandem to the error rate by introducing passive sender side algorithms for overall packet loss rate estimation.

2.2 Estimating Packet Loss and Packet Error Rates

In this section, we describe a set of algorithms that can be used to passively measure the overall packet loss rate, p, as observed by a TCP sender. We also describe the Cumulative Explicit Transmission Error Notification (CETEN) [27] scheme for relaying packet error rates measured from inside the network to the end hosts.

2.2.1 Loss Estimation AlgorithmS for TCP (LEAST)

The **L**oss **E**stimation **A**lgorithm**S** for **T**CP (LEAST) provide a passive means by which TCP senders can determine their end-to-end packet loss rates with high accuracy [2]. Since this is done passively, it is without any alteration to the behavior of the TCP stack. No extra bytes of traffic are injected by a sender running LEAST, no modifications to well-known and debugged procedures are made, nor is the traffic generated at all different from that of a TCP not performing LEAST measurements. In section 2.2.1.1, we present the LEAST algorithm for Reno TCP and demonstrate its measured accuracy, and in 2.2.1.2 we do the same for a SACK TCP with a DSACK-capable receiver and show it to be even more accurate.

2.2.1.1 Reno TCP Algorithm

The basic idea behind LEAST is that the number of packets lost in the network is equivalent to the number of *needed* retransmits made by TCP. A counter of the number of retransmitted packets is a reasonable rough guess, as TCP doesn't retransmit a packet unless there is some indication it may have been lost. Some retransmissions aren't needed, though, as the cumulative acknowledgment doesn't give much information when there are only a few small holes in a window of data rather than a large continuous run of losses. In addition some timeouts or fast retransmits may be spurious due to large variations in delay, reordering, or loss of acknowledgments. However, we can use clues from the stream of acknowledgments from the receiver to infer the number of unneeded retransmits. By subtracting this value from the number of total retransmissions we obtain an estimate on the number of lost packets. Dividing this

by the total number of packets sent yields the overall packet loss rate. The Python programming language code we use to analyze the accuracy of the LEAST algorithms using captured packets from live connections is available in appendix A.1.

For Reno TCP, a counter is kept indicating the estimated number of lost data segments. This counter starts at zero in the beginning of a connection. The lost segment counter is incremented each time a segment is retransmitted. If the RTO time expires, then we keep counters of the number of duplicate acknowledgements and the number of retransmissions. These two counters are kept until the sender leaves loss-recovery and then their minimum is substracted from the lost segment counter. This description is a somewhat simplified version of the complete algorithm detailed in appendix A.1, but serves to show that the LEAST algorithms are computationally very simple.

To validate our LEAST algorithms, we used a number of tests over the NIMI mesh, where we traced both the sender and receiver sides of packet transfers of 5000 segments, using both Reno and DSACK TCPs. In post-processing, we are then able to match up the packets sent and received to determine how many were lost and compare this to the LEAST-computed value. Using the NIMI mesh gives us a diverse sampling of network paths. Of the 14 hosts used, 8 are in the United States, 4 in Europe, 1 in the Far East, and 1 in South America. Figure 2.1 illustrates the wide range of actual loss rates we observed and gives some indication of typical loss rates of Internet paths. Specifically, over 20% of the transfers experience no losses, while 0.6% see loss rates of over 10%. Allman, Eddy, and Ostermann [2] show that the loss periods and loss distances also sampled a diverse range of loss behavior.

Of the NIMI transfers, 2546 of them were valid Reno connections. Figure 2.2 plots the cumulative distribution of the percent error in both the raw number of retransmits and the Reno LEAST estimate from the actual loss rate as computed by combining our sender and receiver packet traces. We see that over 56% of the transfer loss rates as computed by LEAST are exactly correct, and only 3% err by more than

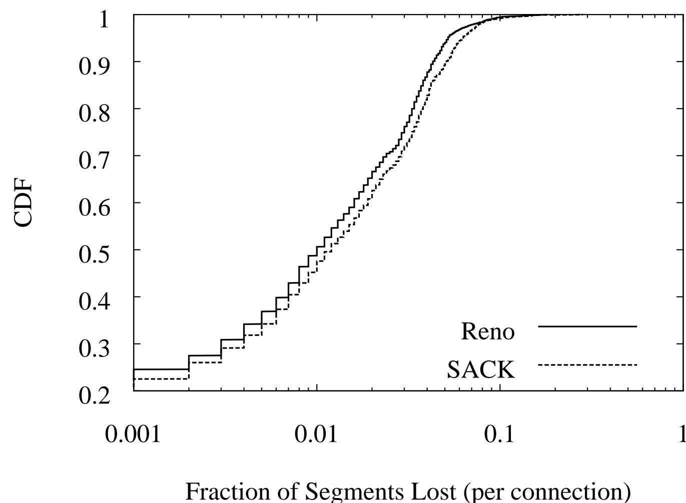

Figure 2.1. Cumulative Distribution of Actual Loss Rates on the NIMI Mesh

10%. Considering the much larger error present in using the retransmission counter as a rough estimate for the number of losses, it is clear that this method of inferring unneeded retransmissions from the acknowledgment patterns is valid.

There are numerous sources of errors in the Reno LEAST algorithm, which are mostly uncorrectable. These include: spurious retransmissions, dropped duplicate acknowledgments, partial acknowledgments triggered by spurious retransmits, dropped retransmissions, and spurious retransmissions that end a loss recovery period. These events are (for the most part) fairly rare, although clearly not impossible or unexpected [2]. Given the minimal amount of computational overhead involved in running LEAST, we feel that it comes sufficiently close in accuracy without any accounting for these rarities.

2.2.1.2 SACK and DSACK TCP Algorithms

The LEAST algorithm for use with a DSACK TCP is the simplest we present. With the DSACK option, duplicate receptions of the same data trigger transmission of a DSACK block from the receiver indicating the range of data unnecessarily re-

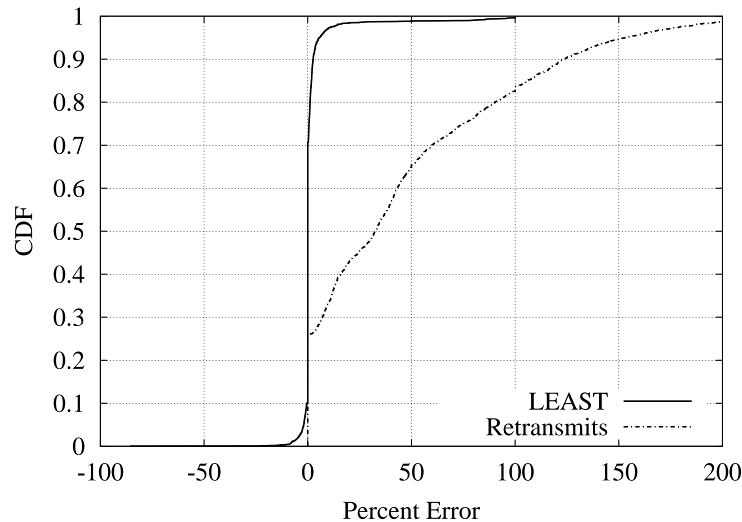

Figure 2.2. Accuracy of Reno LEAST Algorithm and Raw Retransmissions, Compared to Actual Packet Losses

transmitted. The sender may assume to see one DSACK block for each segment with duplicate data, assuming no loss in the ACK path. The number of retransmissions minus the number of DSACK blocks seen is an excellent and easily computable estimate of the number of packets lost. The code for an implementation of our DSACK LEAST algorithm can be found in appendix A.2. For SACK TCPs without DSACK, the algorithm is less accurate, but it helps that since use of the SACK option greatly decreases the number of unneeded retransmissions, the retransmission counter alone is a fairly accurate estimate of the number of lost segments.

Figure 2.3 plots the accuracy of the DSACK TCP LEAST algorithm over 2577 NIMI transfers, confirming that our estimate is very good for DSACK TCPs. Dropped acknowledgments carrying DSACKs are the only known source of error in the DSACK LEAST algorithm. Loss and reordering in the acknowledgment path are the primary causes of error in the SACK without DSACK LEAST algorithm.

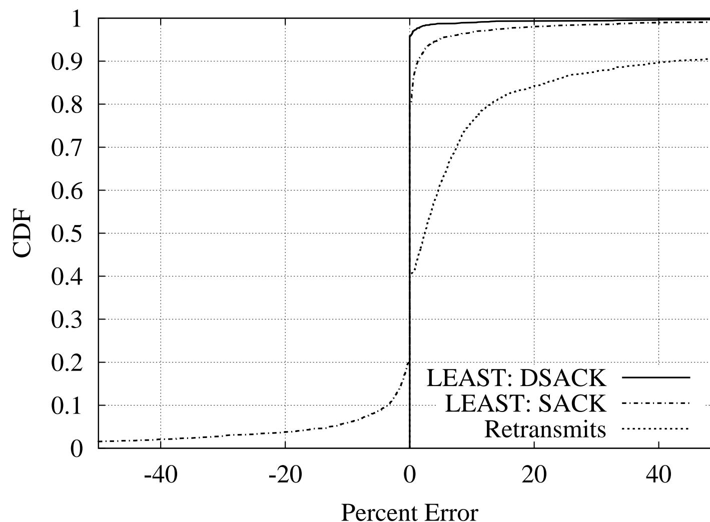

Figure 2.3. Accuracy of DSACK LEAST Algorithm and Raw Retransmissions, Compared to Actual Packet Losses

2.2.2 Cumulative Explicit Transmission Error Notification

In the Cumulative Explicit Transmission Error Notification (CETEN) work, Krishnan *et al* [27] outlines a simple means by which the packet error rate and the congestion rate for a network path may be relayed to TCP senders with some assistance from the routers in the path. The CETEN scheme involves defining an additional packet header with four fields. This is most naturally tacked onto an IP packet as IP options bytes. Table 2.2.2 describes the four fields in a CETEN packet header. As implemented, CETEN uses 8 byte double precision floating point representations for each field.

The $ferr$ and $fcong$ fields are initially set to 1 by the end-host sending the packet. Routers in the network path track their loss rates due to both packet errors and congestion. For a router numbered i, call these e_i and c_i. Equivalently these might be stored as survival probabilities $1 - e_i$ and $1 - c_i$. The survival rates for the entire path of n links then are given by equations 2.1 and 2.2.

Table 2.1 CETEN Header Fields

Field	Description
ferr	The probability a packet survives corruption in the forward direction (from source to destination)
berr	The probability a packet survives corruption in the reverse direction (from destination to source)
fcong	The probability a packet survives congestion in the forward direction
bcong	The probability a packet survives congestion in the reverse direction

$$1 - e = \Pi_{i=1}^{n}(1 - e_i) \tag{2.1}$$

$$1 - c = \Pi_{i=1}^{n}(1 - c_i) \tag{2.2}$$

These products can be computed sequentially by each routing node i by taking the values of $ferr$ and $fcong$ in each packet and multiplying them by $1-e_i$ and $1-c_i$. In this way the complete path survival rates $1 - e$ and $1 - p$ are obtained by the end hosts in $ferr$ and $fcong$. The *berr* and *bcong* fields are then used to carry echoes of the forward values in acknowledgments back to the sender.

2.2.3 CETEN Simplification

In section 1.2 we defined the overall packet loss rate p as being composed of the congestion rate c and the packet error rate e such that $p = c + e$. With the LEAST algorithms, a TCP sender can accurately determine p essentially for free (with no additional help or information that isn't already present). If a CETEN header can be updated by routers along the network path, then the TCP sender will also know e. Since we have an equation capable of yielding c from p and e, there is then no

reason for routers to compute and relay c as in the original CETEN TCP scheme [27]. Eliminating the need for having c encoded in CETEN packet headers effectively halves their required length and removes some processing load from CETEN-supporting routers by saving at least a multiplication operation per packet.

The size of the CETEN packet header required to obtain $\frac{e}{p}$, the percentage of losses due to errors, can be cut in half when used in conjunction with LEAST. The congestion rate fields can be eliminated and only the error rate fields are needed. Alternatively, we could leave the header size the same and double the precision of the survival rates. Either way, we have an improvement over the old CETEN packet header since we remove the redundant information that LEAST can provide. Section 2.2.3.1 explores the differences of these two approaches further.

2.2.3.1 CETEN Error Rate Computation

A major difference between the LEAST-computed packet loss rate and the rates inside the CETEN headers as originally described by Krishnan *et al* [27] is the time scale over which the rates are valid. The original CETEN rates were computed via an exponentially-weighted moving average with weight of 0.5, so it adapted very fast per packet. On the other hand, LEAST generates a loss rate that spans the entire length of a connection and is much less susceptible to rapid fluctuations. For this reason, and for efficiency, a long-range reported error rate on the order of at least many RTTs should be used by routers.

The original CETEN work leaves the exact means of error rate computation largely as future work, showing results for an EWMA with weight of 0.5 computed per packet and suggesting that a rolling average over larger windows of packets might work well too. We find that with that weight, the EWMA moves too fast. Given a high rate of statistical multiplexing between many connections sharing a link, it might be difficult to properly set this value so that proper error rates are reported equally. The advantage of having a quickly moving average is that it can be used more like a per-packet ETEN mechanism since spikes coinciding temporally with a sender perceiving

lost segments likely indicate corruption losses. One problem with this is that during spikes in the corruption rate, it might be difficult to get valid CETEN information through without the information-bearing packet itself being lost due to the corruption.

The problem of the exact means for computing error rates is left as future work by us as well. For our purposes, a steady long-term average is used. If packet errors are uniformly distributed over time, this is a very easy parameter for an administrator to measure and statically configure in a router. An even better solution would be to have the router measure periodically over some suitably long time period and update itself. For different patterns of packet error distribution, another method of error reporting might prove superior.

3. TCP Congestion Control Modifications with CETEN

The methods presented in the previous chapter outline ways that a TCP sender may become aware of its packet loss rate p and packet error rate e. The remaining challenge is then in using this information to intelligently alter congestion control behavior. We analyze the probabilistic algorithm presented by Krishnan *et al* [27] ($CETEN_C$) in section 3.1 and evaluate its performance using simulations. Then, in section 3.2, we present a deterministic algorithm ($CETEN_A$) and present simulation results testing it across a realistic range of error rates. Finally we look into fairness and friendliness implications of these CETEN-aware congestion control modifications.

3.1 Probabilistic CETEN TCP

The probabilistic CETEN algorithm for TCP congestion control we discuss here is that originally described by Krishnan *et al* [27], and is motivated by the idea that TCP should slow down when losses are due to congestion, but shouldn't for error-based losses. Due to our development of an alternative CETEN congestion control approach, we refer to the original Krishnan scheme as $CETEN_C$ for the weighted **c**oin-flip it uses. Each time a loss event occurs, a uniformly distributed random number is drawn between zero and one. If this random number is greater than the ratio $\frac{e}{p}$ then the congestion window and slow start threshold are modified by the standard TCP algorithms and the lost segment is retransmitted, otherwise (if the random number is less than or equal to $\frac{e}{p}$) the segment is retransmitted without taking congestion control actions. Thus over time the congestion window is reduced at the same rate as congestion losses occur, while there is no slowdown at the same rate with which packet errors occur. Although specific losses may not correlate with

the correct action (to reduce or not reduce the congestion window), the aggregate behavior shows a proper number of attempts to be responsible and make a congestion window reduction.

Code implementing this algorithm must be slightly more complex in order to account for the possibility of errors in the combination of LEAST and CETEN estimates, for example if the error rate reported is greater than the overall packet loss rate measured. This can (and would) happen when a TCP connection is too young to have seen enough losses to build up a decent LEAST estimate, while the CETEN reported error rate is already measured over a long time scale. Another reason this might happen is if the packet error rate reported is artificially inflated by either the TCP receiver or some middle-box in an attempt to improve performance. In either case, the estimates are unreasonable and so we may either take the conservative approach and make the stock TCP congestion window reduction, or assume that e is very near p and not reduce. Valid arguments can be made for each case. We will explore the aggressive behavior here in which we assume measured $e > p$ implies error in the LEAST measurement and $e = p$, so no congestion window reduction is made. We choose this for the sake of looking at the maximum potential gain possible via CETEN methods.

Although the idea behind $CETEN_C$ is that over time, congestion window reductions are made at the proper rate, and individual decisions need not be correct, we also examine $CETEN_C$ on a per-decision basis. Table 3.1 lists the combined event probabilities between loss causes and guesses. We can see that correct guesses occur at the rate $2(\frac{e}{p})^2 - 2\frac{e}{p} + 1$. We plot this in figure 3.1 and see that for all values of $\frac{e}{p}$ it is at least 50% or better.

In both cases when a congestion window reduction occurs, $CETEN_C$ behaves exactly as a stock TCP. The two cases where its behavior differs are those when no slowdown is taken. In the case where a loss is due to corruption and no slowdown is taken, there can be a legitimate gain in throughput. When a loss is due to congestion

Table 3.1 Congestion Control Event Probabilities

	Packet Lost to Congestion $\frac{p-e}{p}$	Packet Lost to Corruption $\frac{e}{p}$
Slowdown $\frac{p-e}{p}$	$1-2\frac{e}{p}+(\frac{e}{p})^2$	$\frac{e}{p}-(\frac{e}{p})^2$
No Slowdown $\frac{e}{p}$	$\frac{e}{p}-(\frac{e}{p})^2$	$(\frac{e}{p})^2$

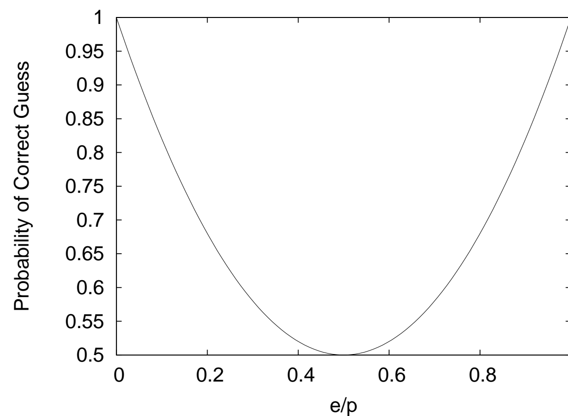

Figure 3.1. $CETEN_C$ Correct Guesses vs e/p

and a TCP doesn't slow down, we may expect further losses to result (depending on the degree of statistical multiplexing of flows at the bottleneck). This results in an inflated packet loss rate p and unchanged e in addition to costly recovery time. With an unchanged numerator and rising denominator, the decreasing ratio $\frac{e}{p}$ makes the only potentially beneficial situation (corruption loss and no slowdown) much less likely, since this occurs at the rate $(\frac{e}{p})^2$. In the long run, these poor guesses could add up to negate the benefit of a TCP using $CETEN_C$.

As an example demonstrating the performance gained by $CETEN_C$ not slowing down for a single corruption loss, figure 3.2 plots the TCP congestion window over

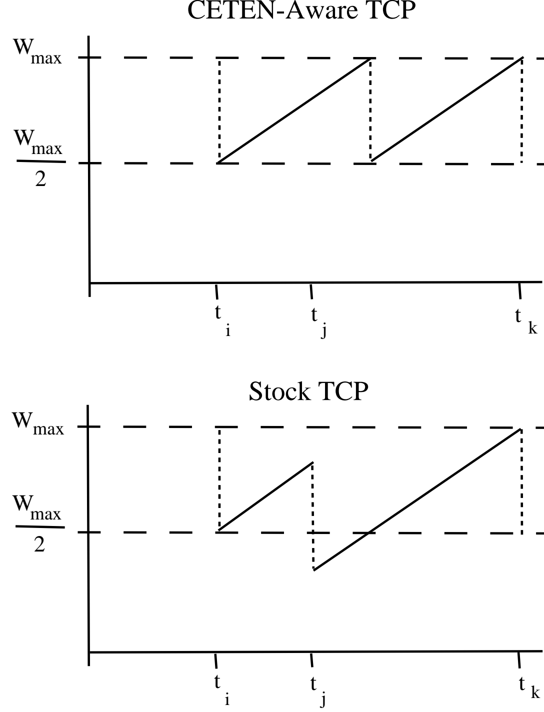

Figure 3.2. Congestion Window Behavior After a Corruption Loss

time between times t_i and t_k in both a stock TCP and a CETEN-aware one when a corruption loss occurs at time t_j and $CETEN_C$ guesses correctly. Here, W_{max} represents the maximum congestion window the network path will accommodate (in terms of MSS) and we assume for the purposes of this example that the TCP sender is able to reach this congestion window both before and after the corruption loss. For convenience, we also choose t_j such that $t_j - t_i = W_{max}(b-1)$ so that each plot covers exactly two full perceived congestion loss periods between t_i and t_k, where b is the delayed acknowledgment threshold.

As an example we can compute the throughput gain of the $CETEN_C$ TCP over stock TCP in figure 3.2. We'll use the abbreviations $\Delta t_1 = (t_j - t_i)$ and $\Delta t_2 = (t_k - t_j)$ so that $(t_k - t_i) = \Delta t_1 + \Delta t_2$. Given the throughput between t_i and t_k as $T_{i,k} = \frac{\int_{t_i}^{t_k} w(t)dt}{t_k - t_i}$,

we can set the percent gain of the CETEN TCP over stock as $\frac{\int_{t_i}^{t_k} w_c(t)dt}{\int_{t_i}^{t_k} w_s(t)dt} - 1$. To compute the TCP throughput, we can use the fact that during congestion avoidance, the congestion window grows linearly to obtain $\frac{1}{b}(t_k - t_i) = W_{max}$, and $\Delta t_2 = bW_{max} - W_{max}(b-1) = W_{max}$.

For the CETEN TCP then:

$$\begin{aligned}
\int_{t_i}^{t_k} w_c(t)dt &= \frac{W_{max}}{2}(\Delta t_1 + \Delta t_2) + \frac{1}{4}W_{max}\Delta t_1 + \frac{1}{4}W_{max}\Delta t_2 \\
&= \frac{3}{4}W_{max}(\Delta t_1 + \Delta t_2) \\
&= \frac{3b{W_{max}}^2}{4}
\end{aligned}$$

And for the stock TCP:

$$\begin{aligned}
\int_{t_i}^{t_k} w_s(t)dt &= \frac{{\Delta t_1}^2}{2b} + \frac{W_{max}}{2}\Delta t_1 + \frac{\frac{\Delta t_1}{b} + \frac{W_{max}}{2}}{2}\Delta t_2 + \frac{\Delta t_2}{2}(W_{max} - \frac{\frac{\Delta t_1}{b} + \frac{W_{max}}{2}}{2}) \\
&= \frac{2{\Delta t_1}^2 + 2bW_{max}\Delta t_1 + 2\Delta t_1 \Delta t_2 + bW_{max}\Delta t_2}{4b} + \frac{\Delta t_2(3bW_{max} - 2\Delta t_1)}{8b} \\
&= \frac{4{W_{max}}^2(b-1)(b-1+b+1) + b{W_{max}}^2 + bW_{max}^2 + 2{W_{max}}^2}{8b} \\
&= \frac{{W_{max}}^2(4b^2 - 3b + 1)}{4b}
\end{aligned}$$

Given the current standard delayed acknowledgment threshold of $b = 2$, the CETEN TCP would gain a factor of about 9.09% in throughput over a stock TCP in this case. In the small time frame this example covers, we see $\frac{e}{p}$ seems to be between 1/3 and 1/4 so if events always played out this way, the CETEN gain would only occur between 1/9 and 1/16 of the time, and in reality less than that since we know that such guesses will actually become increasingly rare over time as the wrong guesses inflate p.

The problem of wrong guesses affecting p is much less severe when several flows share a bottleneck. This is clear in that when capacity is exceeded, it is likely that multiple flows will be notified, in which case the aggregate slowdown may provide enough slack to accommodate CETEN flows which have incorrectly not slowed down. However, this raises concerns about fairness between competing flows, whether they be homogeneously CETEN or some mix of stock TCP and CETEN flows.

3.2 Deterministic CETEN TCP

As an alternative to Krishnan *et al*'s probabilistic CETEN [27], we present a deterministic algorithm. The idea is to choose a congestion window reduction coefficient from between the one-half used by standard TCP where all losses are seen as congestion and one if we assume all losses are due to errors. By basing the choice of this reduction upon $\frac{e}{p}$, we can arrive at a solution whose aggressiveness is suitable to the relative error rate. In a stock TCP stack, the congestion window is reduced by multiplying it by $\frac{1}{2}$, so we redefine this coefficient as a function of $\frac{e}{p}$ that varies between this and one: $\frac{1+\frac{e}{p}}{2}$. This function is plotted in figure 3.3. We call the value computed by this function the **M**ultiplicative **D**ecrease **F**actor (MDF), and since it is used to adapt the congestion window reduction to the relative error rate, we call this adaptive algorithm $CETEN_A$ to differentiate it from $CETEN_C$.

To further motivate use of this decrease factor, we observe that it is the expected value for the window reduction of the $CETEN_C$ algorithm, where the congestion window is multiplied by 1 at frequency $\frac{e}{p}$ and by $\frac{1}{2}$ at rate $\frac{c}{p} = \frac{p-e}{p}$:

$$\begin{aligned} E[X] &= (1 * \frac{e}{p}) + (\frac{1}{2} * \frac{p-e}{p}) \\ &= \frac{e}{p} + \frac{1}{2}(1 - \frac{e}{p}) \\ &= \frac{1}{2} + \frac{1}{2}\frac{e}{p} \\ &= \frac{1 + e/p}{2}. \end{aligned}$$

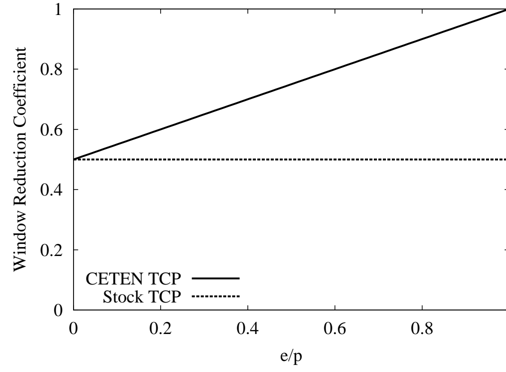

Figure 3.3. Window Reduction Coefficient Function

To theoretically evaluate the throughput gains this CETEN-aware TCP is capable of, we can modify Padhye's TCP model for congestion window reductions with CETEN-generated coefficients rather than 1/2. In this derivation we use the same variables as Padhye *et al* [31], with the addition of r, the window reduction coefficient. These variables are specifically, W_i the maximum congestion window reached during the ith period between loss notifications, X_i the number of rounds of a full congestion window of packets sent during that period, and Y_i the total quantity of packets sent in the period.

We start by changing the definition of W_i in terms of W_{i-1} and X_i to reflect the variable reduction $\frac{1+\frac{e}{p}}{2} = \frac{p+e}{2p}$.

$$W_i = \frac{p+e}{2p} W_i + \frac{X_i}{b}$$

The initial equation for Y_i also remains unchanged, with $\frac{p+e}{2p}$ substituted for 1/2 .

$$\begin{aligned} Y_i &= \sum_{k=0}^{X_i/b-1} \left(\frac{p+e}{2p} W_{i-1} + k\right) + \beta_i \\ &= \frac{X_i}{2}\left(\frac{p+e}{2p} W_{i-1} + W_i - 1\right) + \beta_i \end{aligned}$$

$$E[Y] = \frac{E[X]}{2}(\frac{p+3}{2p}E[W] + E[W] - 1) + E[\beta]$$

We also know that $E[Y] = \frac{1-p}{p} + E[W]$, $E[\beta] = \frac{E[W]}{2}$, and $E[X] = bE[W](\frac{p-e}{2p})$.

$$\begin{aligned} E[Y] &= E[Y] \\ \frac{1-p}{p} + E[W] &= \frac{E[X]}{2}(\frac{p+e}{2p}E[W] + E[W] - 1) + E[\beta] \\ \frac{1-p}{p} + E[W] &= \frac{bE[W]\frac{p-e}{2p}}{2}(E[W]\frac{3p+e}{2p} - 1) + \frac{E[W]}{2} \\ 0 &= (\frac{3p+e}{2p}E[W]^2 - (1 + \frac{2p}{b(p-e)})E[W] - \frac{4(1-p)}{b(p-e)} \end{aligned}$$

We then use the quadratic formula to solve for the value of $E[W]$:

$$E[W] = \frac{1 + \frac{2p}{b(p-e)} + \sqrt{1 + \frac{4p}{b(p-e)} + \frac{4p^2}{b^2(p-e)^2} + 8\frac{3p+e}{p}\frac{1-p}{b(p-e)}}}{\frac{3p+e}{p}} \tag{3.1}$$

Compare this to the value of $E[W]$ obtained from the Padhye model:

$$E[W] = \frac{2+b}{3b} + \sqrt{\frac{8(1-p)}{3bp} + (\frac{2+b}{3b})^2}$$

Figure 3.4 plots the expected size of the congestion window for both TCPs with varying e and $p = 2\%$. The $CETEN_A$ TCP always has a slightly larger window here (although in reality being quantified in full segments, the window is the same between stock and $CETEN_A$ until $e \simeq 0.5\%$), and thus $CETEN_A$ is able to keep more outstanding data and achieve higher throughput. Notice that this is especially true at more severe error rates, as we would wish.

The transformation from the average congestion window $E[W]$ to throughput $B(p, e)$ at given packet loss and error rates p and e is straightforward since $B = \frac{E[Y]}{E[A]}$ where A is the time interval between consecutive loss events and is given by $E[A] = (E[X] + 1)RTT$. Since we know $E[X]$ in terms of $E[W]$ we then have:

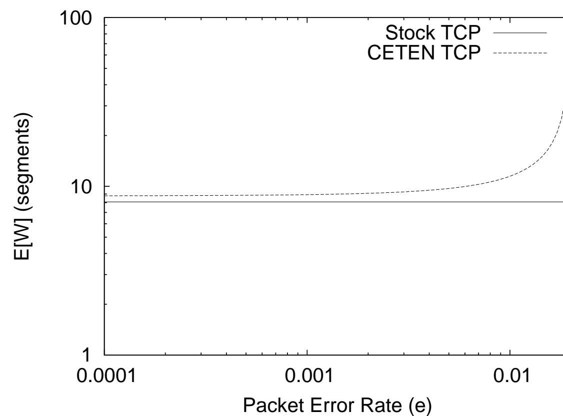

Figure 3.4. Average Congestion Window Size, $p = 2\%$

$$\begin{aligned} B(p,e) &= \frac{E[Y]}{E[A]} \\ &= \frac{\frac{1-p}{p} + E[W]}{E[X]+1} \\ &= \frac{\frac{1-p}{p} + E[W]}{bE[W](\frac{p-e}{2p})} \end{aligned}$$

This function is plotted in figure 3.5 with $p = 2\%$ set constant and the delayed acknowledgment threshold b set to 2. The curve for stock TCP is flat as it has no knowledge of e's contribution to p and is given by equation 1.1, while the CETEN line shows great gains as e approaches p. This clearly shows that in operating environments where packet errors dominate congestion losses, we can expect substantial gains from using $CETEN_A$ TCP over a stock TCP. We also notice that $CETEN_A$ behaves comparably to a stock TCP when such conditions do not hold and the relative packet error rate is low. This implies that it behaves appropriately in low-error environments as well, allowing devices implementing $CETEN_A$ to move seamlessly between wired and wireless networks (or error-free and error-prone networks) without changing con-

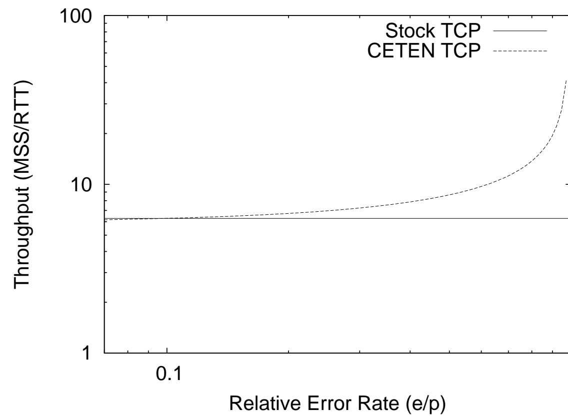

Figure 3.5. Throughput vs Relatative Error Rate, $p = 2\%$

gestion control algorithms. These theoretical results are encouraging, and in the next section we validate them in simulation and look at some fairness issues.

3.3 Simulation Results

This chapter outlines several simulations performed in supplement to our theoretical results showing the throughput gain that CETEN TCPs may have over stock TCP stacks and to explore the interaction between CETEN flows and normal TCP flows sharing a link. We focus entirely on SACK TCP (with DSACK) here for two reasons: scope and relevance. The task of evaluating a new congestion control algorithm is massive enough without the burden of doing so for multiple TCP stacks of differing capability, so we keep the scope down by only exploring CETEN modifications to modern SACK-based stacks and not older Reno-style ones. It can be argued that non-SACK stacks are outdated and that gradually more Internet connections are coming to use the SACK option[1]. Also, since applying CETEN patches to a real world machine would provide the opportunity to add SACK and DSACK capabilities, there isn't much compelling evidence that a Reno-based CETEN is useful. Initial experiences suggested that the gain from using a SACK TCP over a Reno one

[1] http://www.icir.org/floyd/sack-questions.html

was greater than that from using CETEN in the Reno TCP, further decreasing the relevance of exploring a Reno CETEN.

3.3.1 Implementation Details

All of our simulations were conducted using a significantly modified version of the ns-2 simulator[2] version 2.1b9. In addition to fixing several small TCP bugs and adding support for more verbose trace-files, our main modifications were to add support for CETEN packet headers, the modified TCP congestion control algorithms from chapter 4, and the TCP LEAST algorithms for Reno and DSACK TCP. Much of the CETEN code was ported from the ns-2.1b7 patches available from BBN[3] that were used by Krishnan *et al* [27] to originally study CETEN. We modified these patches to apply to the one-way TCP classes in ns-2 which are generally regarded as less buggy and better maintained than the FullTcp class in which $CETEN_C$ was originally implemented in.

The LEAST implementation was written to run in real-time rather than as a post-processing routine as the examples provided in appendix A. We then modified the CETEN TCPs to be able to use the LEAST estimate for p rather than needing the CETEN congestion rate c, as explained in section 2.2.2. Performance of our LEAST implementation in the simulator is given in figure 3.6 where the mean percent error between the least estimate and the actual packet loss rate is shown over 30 trials at various programmed link error rates. This data comes from simulations involving a single long-lived bulk-transfer TCP flow traversing a single lossy 5Mbps bottleneck between two 100Mbps links. In all cases, the observed link error rates were negligibly close to those programmed. We see that the DSACK LEAST algorithm comes within 1% of the actual loss rate across the entire range studied. The only loss of accuracy is caused by dropped DSACKS, after we fixed a bug in ns-2 causing TCP sinks to delay

[2]http://www.isi.edu/nsnam

[3]http://www.ir.bbn.com/projects/pace/eten/index.html

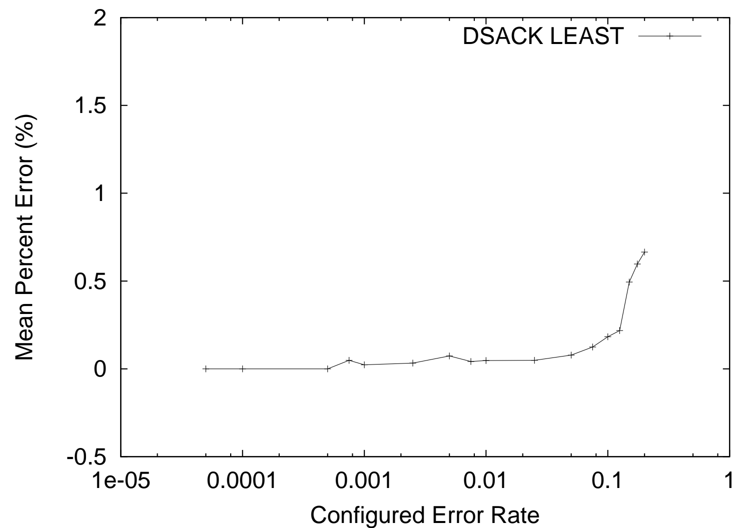

Figure 3.6. Percent Error in LEAST Estimate

acknowledgments that should carry DSACK blocks and then forget about sending the DSACKs.

3.3.2 CETEN Throughput Gain

Since CETEN and LEAST are, by their nature, primarily useful and applicable to long-lived flows with ample data to keep the congestion window constantly full, as in bulk transfer applications, we focus on bulk-transfers in our simulations. The standard ns-2 FTP application agents are used as our traffic sources. We then use the TCP goodput (unique bytes sent per time unit) as a performance measure, since this is the metric that matters to a bulk transfer application. With fixed segment size and RTT, differences come directly from the congestion control decisions and loss rate and distribution. All numbers presented here, unless otherwise noted, come from averaging via arithmetic mean the results of 30 trials at each set of parameters. Where shown, error bars represent a standard deviation of the data.

Our simulations use a simple dumbbell topology, illustrated in figure 3.7 consisting of a pair of routers connected via a lossy link, between traffic source and sink nodes.

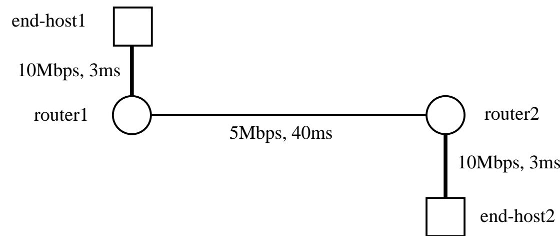

Figure 3.7. Simulation Topology

Traffic sources and sinks are placed on the end-host nodes so that transfers run in both directions and are started at random times within the first 30 seconds of simulation time and run for 3600 seconds. We can place arbitrary numbers of stock TCP, probabilistic or deterministic CETEN TCP, or Constant Bit Rate (CBR) traffic generators on each of the end-hosts. The bottleneck link is 5Mbps (0.625 MBps), with a 40ms one-way delay and queue depths of 150 packets in each direction. The packet error rate of the bottleneck is programmable.

As a simple demonstration of CETEN TCP's advantage over stock TCP in the face of packet errors, we present figure 3.8, plotting the programmed link error rate of the bottleneck against the average measured goodput. Only a very small number of losses are caused by self-congestion of the link since there are no competing flows, with the relative packet error rate $\frac{e}{p}$ approaching unity. Figure 3.8 confirms the intuition that both CETEN TCPs would perform better than stock and that $CETEN_A$ would best $CETEN_C$. The margin of gain between $CETEN_A$ and stock is roughly consistent with that predicted by our theoretical development in equation 3.2. This helps confirm both the validity of our theory and the correctness of our implementation.

Figure 3.9 shows results of the same simulations run with no programmed error rate or cross traffic in the reverse direction (from end-host2 to end-host1), so that there is no loss in the acknowledgment path. This is done to gauge the impact of a perfect LEAST estimate in comparison to a non-perfect estimate caused by lost

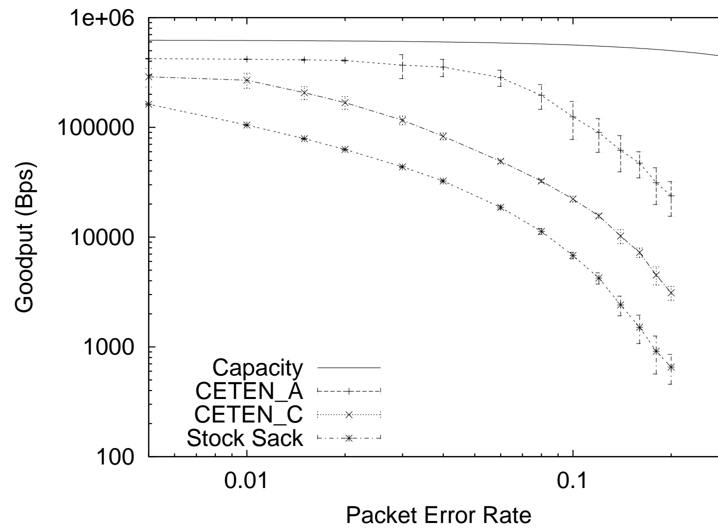

Figure 3.8. Bulk-Transfer Goodput

acknowledgments. Comparing this to figure 3.8 allows us to see how lost ACKs (creeping inaccuracy in the LEAST figure) degrade CETEN performance. We see that both CETEN TCPs perform better when acknowledgments aren't lost, although $CETEN_C$ converges towards stock TCP performance as the error rate grows. This is partially explainable in that stock TCP's goodput doesn't degrade as fast with no acknowledgment loss. Plausible real-world applications for CETEN (mobile wireless devices, satellites, etc) often have highly asymmetric links where one direction may be much lossier than the other, so simulations of this nature show CETEN's usefulness in a range of environments.

Figure 3.10 displays the results of the same experiment of figure 3.8 with the addition of bursts of UDP traffic in order to assess the ability of the CETEN algorithms to maintain performance gains when congestion losses are introduced and not all losses are due to corruption. The UDP traffic starts and stops over randomly generated, exponentially distributed time periods with mean on-time of 2.5s and sends packets at a constant rate of 1 Mbps when on. Five such UDP traffic sources are placed on

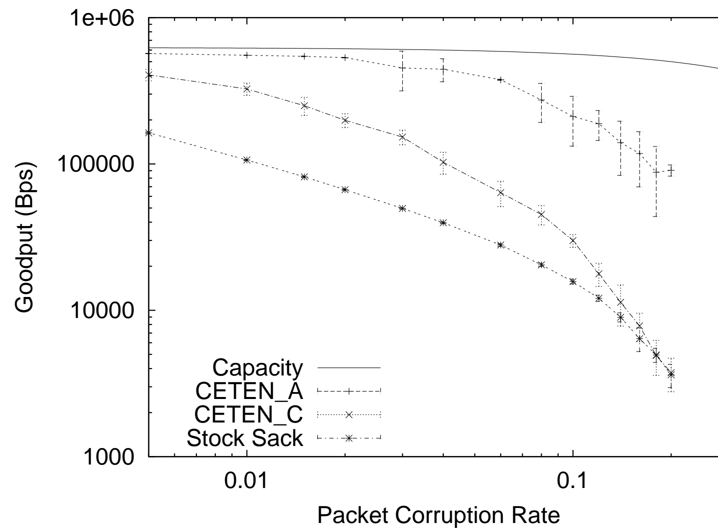

Figure 3.9. Bulk-Transfer Goodput (No Ack Loss)

each end node, in order to completely saturate the bottleneck if all are simultaneously on. Aside from a vertical translation downwards due to the added congestion losses, the shapes of the curves in figure 3.10 are consistent with those in figure 3.8. This shows that even when multiplexed with other non-congestion responsive traffic and e approaching p, CETEN is still able to produce gains.

3.3.3 CETEN Fairness and Friendliness

Ideally, gains in throughput for CETEN TCP flows shouldn't come at the expense of other flows sharing portions of the network path; gains should instead be deducted from the fallow bandwidth that goes unused because of unneeded slowdowns due to packet errors handled as congestion. We'll explore this facet of CETEN behavior under the term *friendliness*. This is not the same as the more common definition of TCP-friendly [18], which means that a non-TCP flow sends approximately as much as a TCP flow under the same network conditions. This definition wouldn't be appropriate in our case, as TCP's throughput is poor and we're trying to overcome that.

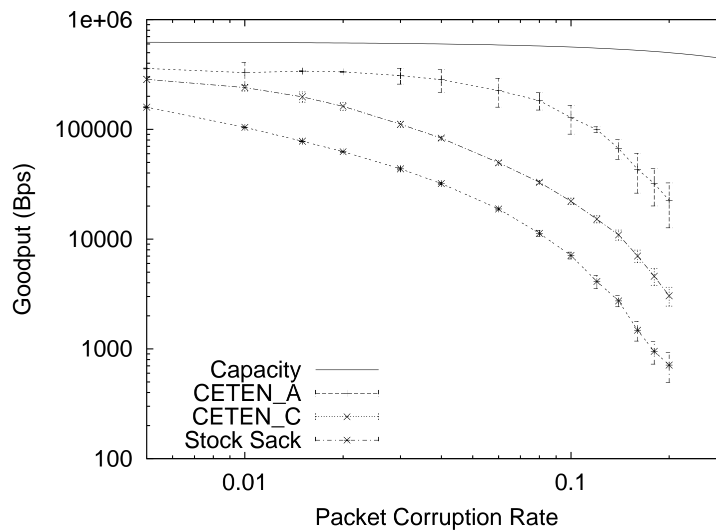

Figure 3.10. Bulk-Transfer Goodput (With Congestion)

In addition, it is important that multiple competing CETEN TCP flows behave fairly to one another by sharing the available capacity equally, or at least as fairly as stock TCPs do. We call this property *fairness*.

3.3.3.1 Fairness

As a measure of fairness, we use Jain's fairness index [23], computed as:

$$f(x_1, x_2, \cdots, x_n) = \frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n \cdot \sum_{i=1}^{n} x_i^2} \tag{3.2}$$

where x_i represents the number of total bytes transferred by a particular connection in a simulation and n is the total number of connections.

We use the same topology from section 3.3.2 adding multiple flows per end-node to generate fairness indices for the two CETEN variants across a range of error rates. We also try two levels of congestion, 20 flows and 100 flows. The flows used are homogeneous in that we run, for example, 20 identically configured stock TCP flows

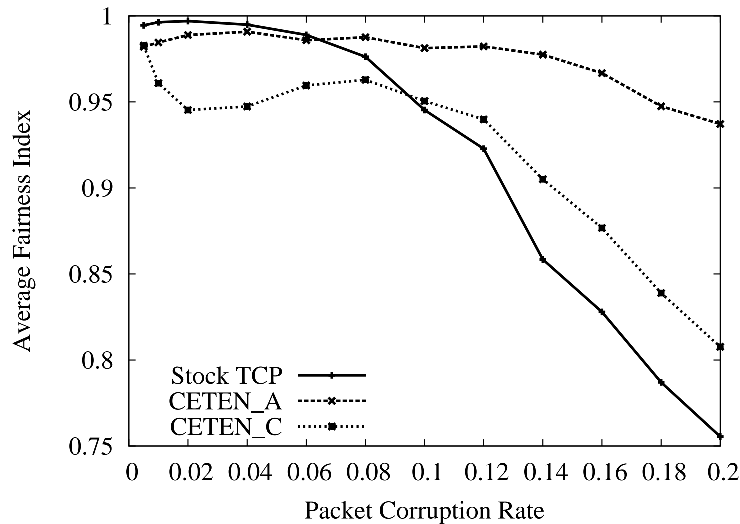

Figure 3.11. Fairness Indices - 20 Flows

in each direction in one simulation, and 20 identically configured $CETEN_A$ flows in another. Along with the fairness index, we also present the aggregate throughput of all the flows. A good fairness index would mean little without a high utilization of the available capacity. A means similar to that devised by Eddy and Allman [14] for evaluating queue management algorithms could be used here to ensure that both properties exist.

Figure 3.11 plots the averaged results over 30 trials at each error rate of a simulation with 20 competing flows in each direction. Examining figure 3.11 it is clear that for 20 competing flows, both $CETEN_A$ and $CETEN_C$ are comparably fair in comparison to the stock SACK TCP. At the lower error rates $CETEN_A$ and stock TCP are both nearly perfectly fair, while $CETEN_C$ is slightly less fair, although it certainly attains very respectable fairness indices. At higher error rates, both stock and $CETEN_C$ fall off rather quickly, while $CETEN_A$ seems to maintain very high fairness regardless of the error rate. In comparison, figure 3.12 showing the same results for 100 flows, also indicates that $CETEN_A$ stays fair through higher p than

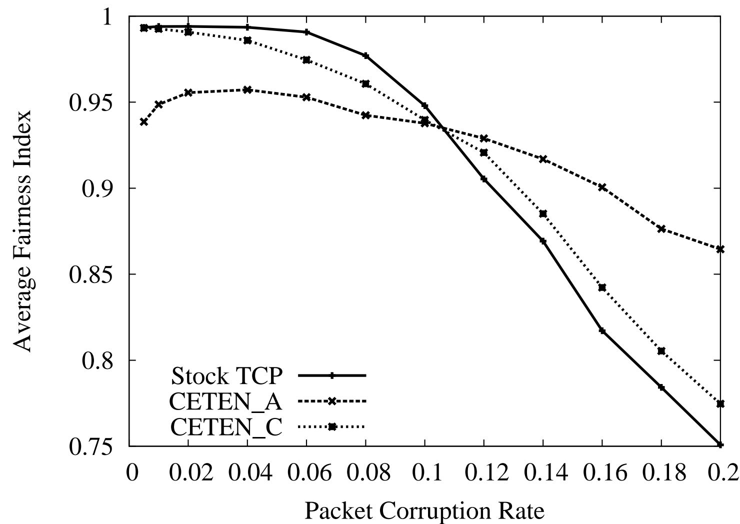

Figure 3.12. Fairness Indices - 100 Flows

the other two TCP variants. Although in this case $CETEN_A$ starts off slightly less fair when the error rate is smaller, it is still well within bounds of adequate fairness. Since one of TCP's strengths lies in its ability to multiplex very fairly with itself, these results for CETEN TCPs showing that the fairness property is preserved (and perhaps even improved) despite a more aggressive congestion control algorithm seem very encouraging.

From an aggregate throughput standpoint, the CETEN results are also quite promising. Figure 3.13 shows average aggregate throughput in both directions for the 20 flow simulations, and figure 3.14 does the same for the simulations with 100 competing flows. In the 20 flow case, even at low error rates, $CETEN_A$ slightly outperforms $CETEN_C$ and stock TCP. As the error rate increases, the combined $CETEN_A$ flows still utilize nearly the full capacity of the link. Meanwhile, higher error rates cause $CETEN_C$ and stock TCP to fall off rapidly, with $CETEN_C$ tolerating packet errors for longer, but eventually showing performance degradation of an order of magnitude. Even with this hit, the combined $CETEN_C$ flows still have

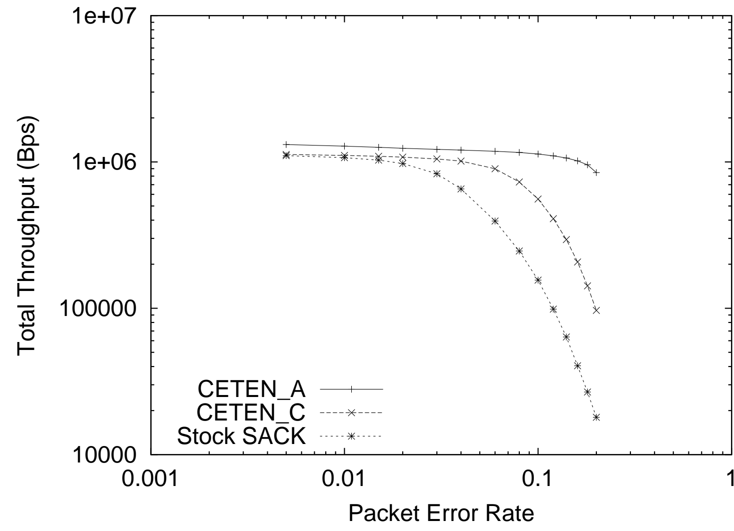

Figure 3.13. Aggregate Throughput - 20 Flows

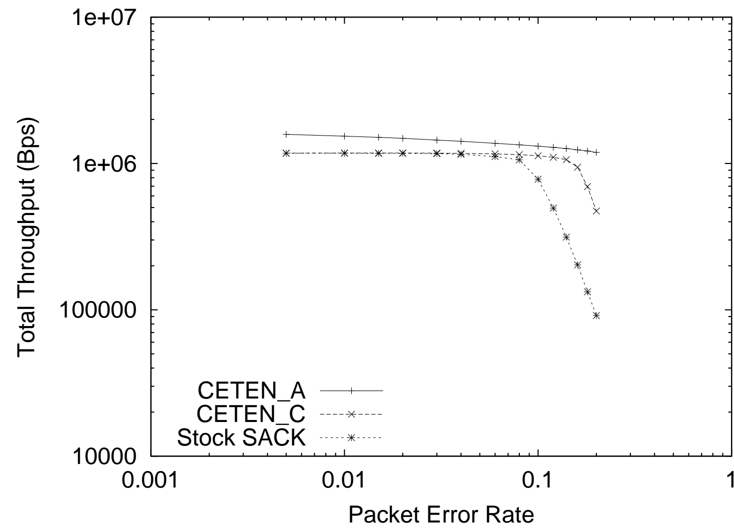

Figure 3.14. Aggregate Throughput - 100 Flows

throughput of nearly another order of magnitude higher than the stock TCP flows. Figure 3.14 tells much the same story, with the main difference being that the statistical multiplexing of even more flows allows the aggregate throughput to remain high longer as the error rate increases. Once again, $CETEN_A$'s capacity utilization is nearly immune to the packet error rate, falling off linearly with it because of the unavoidable losses due to errors.

3.3.3.2 Friendliness

Next, we present an experiment to measure CETEN TCP's friendliness towards competing flows using stock TCP congestion control algorithms. The same topology used in previous experiments is employed using a fixed 1% packet error rate on the bottleneck, with 100 stock TCP flows. Since we've already seen that stock TCP is fair under these conditions in section 3.3.3.1, the arithmetic mean of the number of data packets sent by each flow is a meaningful average of the throughput a single flow can expect over the simulation time. If CETEN is perfectly friendly to stock TCP sources and all gains in throughput come from wasted bandwidth, then we should see that if some of the 100 stock TCP senders are replaced with $CETEN_A$ senders, the average number of packets sent by each stock flow will remain unchanged. Figure 3.15 plots this ideal friendliness, as well as data obtained from simulations where some of the stock sources were replaced with CETEN sources. The results clearly indicate that adding $CETEN_A$ flows causes starvation of the stock flows, while adding $CETEN_C$ flows only negligibly impacts the stock flows.

Investigation into the cause of the stock TCP starvation reveals that it comes as a direct result of a higher congestion loss rate due to the $CETEN_A$ flows less severe congestion response. The $CETEN_A$ flows are so aggressive that they cause congestion losses to start becoming more common. As the congestion rate c rises, stock TCP flows make their window reductions at rate $c + e$ while CETEN flows only make it at $\frac{c}{c+e}$. Since TCP throughput falls off with the square root of the inverse of this rate, even small increases in c cause substantial differences in the stock TCP performance.

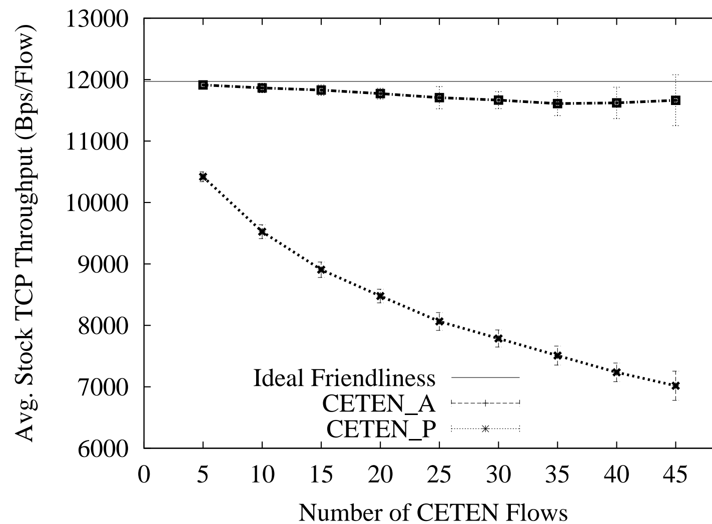

Figure 3.15. CETEN (Un)Friendliness

The $CETEN_A$ flows keep a larger congestion window than the stock TCP flows and so have more outstanding data to generate duplicate acknowledgements and trigger fast retransmissions. In this case, the packet loss rate is high enough that stock TCP's congestion window often isn't large enough for fast retransmissions to be possible, and it resorts to having RTOs generate its retransmissions. The idle time spent waiting for the RTO timer to expire accounts for the majority of the stock TCP degradation in throughput.

This brings up a philosophical question of whether we say the solution is to sacrifice legacy stock TCPs that share a bottleneck with CETEN by saying that if they upgrade to run $CETEN_A$ they'll receive a fair share of the available bandwidth bases on the evidence in section 3.3.3.1, but otherwise their performance is poor anyway so we don't care about making it even worse. Or, we could attempt to cripple the CETEN-aware congestion control algorithms in some way to make them less aggressive, meeting the temporary goal of being friendly to current TCP implementations, while ultimately causing performance in the future (where stock TCP might no longer run over wireless

links) or in custom environments to be less than optimally achievable. We can also consider this a positive quality of $CETEN_C$, which although not showing the same level of gains in the single-flow simulations, still is an improvement over stock TCP, and seems to be much friendlier to stock TCP competition that $CETEN_A$.

One way to temper down $CETEN_A$ would be to use a different, less aggressive function to compute the MDF window reduction factor. Limiting it to the range of 0.5 to 0.75 for example would prevent it from being as aggressive. But if we believe the previous results presented that show $CETEN_A$ to be a good neighbor to itself and use the proper level of aggression needed to fully utilize a link, then such a modification is undesirable. Instead, we propose that this problem of friendliness to traffic using legacy congestion controllers is better fixed through an active queue management (AQM) algorithm in routers. The popular AQM scheme RED [16] is inappropriate here as it shows the same congestion rate to all flows, which is entirely the problem. Instead we need to use an AQM that shows a higher congestion rate to those flows which are contributing most to the congestion and a lower congestion rate to those flows that aren't a factor. In effect, we have to put the "elephant" flows on a diet so that they don't starve the "mice". Such an AQM scheme is provided by CHOKe (**CHO**ose and **K**eep for responsive flows, **CHO**ose and **K**ill for unresponsive flows) [32]. Rather than alter $CETEN_A$, we prefer to recommend the use of CHOKe in CETEN routers. This approach for mitigating the CETEN fairness issues discovered here is being explored as future work.

4. Conclusions and Future Work

4.1 Conclusions

We have introduced and described several changes to the originally proposed CETEN [27] making it both more easily deployable and more effective at high packet error rates. Introduction of the LEAST algorithms make the congestion header fields redundant and unneccessary, so eliminating them allows us to ease both the header overhead and computational overhead of deploying CETEN. We have shown a shortcoming in the probabilistic CETEN algorithm due to its inability to always guess correctly, and we have introduced a deterministic algorithm that addresses this problem. Simulation results show this algorithm to perform substantially better. In addition we have extended previous knowledge of CETEN to include that both proposed congestion control modifications are fair with homogeneous traffic but may be un-friendly to stock TCP traffic when sharing a bottleneck over a lossy link.

In particular, we have shown that $CETEN_A$ both in theory and in practice works well for mitigating the poor interaction between stock TCP and error-prone links. This has been demonstrated in simulation for both single-flow scenarios, with and without loss in the acknowledgement path, alongside non-congestion responsive competing traffic, and in multi-flow environments where the utilization of aggregate flows fully utilized available capacity even at high error rates while preserving an excellent fairness index. We have also shown the original $CETEN_C$ congestion control modification to have merit in fixing TCP for error-prone links, although to a lesser extent than $CETEN_A$.

4.2 Future Work

There are many open issues regarding CETEN. One major question outstanding is how exactly to compute inbound error rates at routers. Whether it is correct to use a fixed or variable granularity of time over which to average the error rate is currently unclear, as well as whether a fixed operator-configured error rate estimate may be sufficient.

Another question still unresolved is how CETEN will respond to more "realistic" error distributions. Packet error distributions in the real world seem to be heavily dependent on the link-layer technology employed on a case-by-case basis. It is unclear whether CETEN will work as well over less uniform distributions of packet errors. Also various real world phenomena like fading channels that cause error rates and distributions to quickly change may effect CETEN's efficiency.

The multiplicative decrease factor (MDF) computing function to use with $CETEN_A$ provides another avenue of research. Alternative MDFs to the linear one we used here should be studied, specifically ones that are less aggressive. The only stipulation is that the function used should be continuous monotonically decreasing with domain $\frac{e}{p}$ from 0 to 1 and range lying between lower bound of $\frac{1}{2}$ and upper bound 1. Some likely candidates are plotted in figure 4.2 which shows MDFs of the form: $MDF = \frac{1+(\frac{e}{np})^k}{2}$. This is a more configurable formula than the one we used ($MDF = \frac{1+\frac{e}{p}}{2}$), where n and k are fixed at one, and thus left out for clarity. Adding the constants n and k allows the MDF to be made less aggressive by raising n and k past one.

A key to the binary representation of the error rate is that it should be easily computed upon by routing hardware, since we would like to impose as little overhead as possible upon busy routers, and computation of the cumulative error survival rate involves updating a packet field at each hop. The CETEN packet header uses 64-bit floats to encode the survival probabilities. The floating point representation used here isn't necessarily the best encoding, but it is easily manipulable in hardware by existing microprocessor instructions.

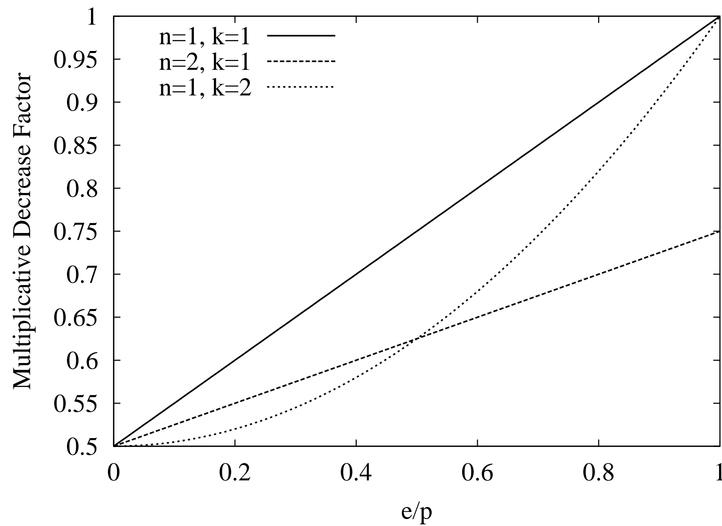

Figure 4.1. Example MDF Function Candidates for Future Work

It is likely that a more compact encoding is possible, and even one with less precision such as 32-bit floating point may suffice, however we do not explore this topic here. At 128 bits (16 bytes), the overhead on a 1500 byte packet is slightly larger than 1%. With this level of overhead, to make sending CETEN information worthwhile, a modified TCP needs to outperform a stock TCP by at least 1%. As we shall show in chapter 4, this is an easily attained goal when even modest amounts of packet errors are present. A tighter encoding scheme would make this threshold even lower. We could also send CETEN headers less frequently than once-per-packet to further make their space and computational overhead vanish. This could be accomplished by only polling for the error rate periodically or after a fixed number of segments, assuming that reports will be representative of the rate over a long time frame.

As discussed thus far, CETEN is vulnerable to an “attack” by the data receiver. If the receiver were to inform the sender that all the drops on the path were caused by corruption and not congestion, then the receiver could induce the sender into transmitting at an inappropriately high rate. Such an attack would allow a receiver to

obtain more than its share of the network resources at the expense of other connections sharing the path. This attack is similar in spirit to the schemes discussed in [38].

Unfortunately, this avenue of attack is fundamental to the design of CETEN since the receiver is the only entity in the path that can characterize the corruption status of the last link in the path. We believe that heuristics can be designed to detect egregious inflation of the corruption rate by the receiver (e.g., by comparing the reported corruption rate with the estimated total loss rate, for instance). However, the system will likely retain a vulnerability to subtle gaming no matter what mechanisms are implemented.

A final more general class of future work in the CETEN domain involves thinking about how much information routing nodes should provide to the network endpoints. Several recent proposals have suggested that internal nodes provide the endpoints with various pieces of information (e.g., ECN [35], XCP [25], QuickStart [22]). An overreaching question is how much of this information *should* be provided? And, if such information is provided, then what else might be useful for the network to provide (e.g., information about packet reordering or asymmetry)? To some degree this is a philosophical question about what level of intelligence to build into the network. While more intelligent networks make things like congestion control easier on endpoints, they may make the global network less scalable or stable in some way.

BIBLIOGRAPHY

[1] Allman, M. On the Generation and Use of TCP Acknowledgements. *Computer Communications Review 28*, 5 (Oct. 1998).

[2] Allman, M., Eddy, W., and Ostermann, S. Estimating Drop Rates With TCP.

[3] Allman, M., Paxson, V., and Stevens, W. TCP Congestion Control, Apr. 1999. RFC 2581.

[4] Arazi, B. *A Commonsense Approach to the Theory of Error Correcting Codes*. MIT Press, 1988.

[5] Avanoglu, E., Paul, S., LaPorta, T. F., Sabnani, K. K., and Gitlin, R. D. AIRMAIL: A Link-Layer Protocol for Wireless Networks. *Wireless Networks 1*, 1 (Feb. 1995), 47–60.

[6] Balakrishnan, H., Padmanabhan, V. N., Seshan, S., and Katz, R. A Comparison of Mechanisms for Improving TCP Performance over Wireless Links. In *ACM SIGCOMM* (Aug. 1996).

[7] Balakrishnan, H., Seshan, S., Amir, E., and Katz, R. Improving TCP/IP Performance Over Wireless Networks. In *ACM MOBICOM* (Nov. 1995), pp. 2–11.

[8] Blanton, E., Allman, M., Fall, K., and Wang, L. A Conservative Selective Acknowledgement (SACK)-based Loss Recovery Algorithm for TCP, Apr. 2003. RFC 3517.

[9] Border, J., Kojo, M., Griner, J., Montenegro, G., and Shelby, Z. Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations. RFC 3135, Internet Engineering Task Force, June 2001.

[10] Braden, R. Requirements for Internet Hosts – Commnication Layers, Oct. 1989. RFC 1122.

[11] Cerf, V., Burleigh, S., Hooke, A., Torgerson, L., Durst, R., Scott, K., Fall, K., and Weiss, H. Delay-Tolerant Network Architecture: The Evolving Interplanetary Internet, Aug. 2002. I-D http://www.ipnsig.org/reports/draft-irtf-ipnrg-arch-01.txt (work in progress).

[12] Chandran, S. R. IP Transport Over Satellite. Kronos Communications White Paper, http://www.kronos.com/iP3_white_paper.pdf.

[13] Dawkins, S., Montenegro, G., Kojo, M., Magret, V., and Vaidya, N. End-to-end Performance Implications of Links With Errors, Aug. 2001. RFC 3155.

[14] Eddy, W. M., and Allman, M. A Comparison of RED's Byte and Packet Modes. *Computer Networks 42*, 2 (June 2003).

[15] Floyd, S., and Henderson, T. The NewReno Modification to TCP's Fast Recovery Algorithm, Apr. 1999. RFC 2582.

[16] Floyd, S., and Jacobson, V. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking 1*, 4 (1993), 397–413.

[17] Floyd, S., Mahdavi, J., Mathis, M., and Podolsky, M. An Extension to the Selective Acknowledgement (SACK) Option for TCP, July 2000. RFC 2883.

[18] Handley, M., Floyd, S., Padhye, J., and Widmer, J. TCP Friendly Rate Control (TFRC): Protocol Specification. RFC 3448, Internet Engineering Task Force, Jan. 2003.

[19] IEEE Computer Society LAN MAN Standards Committee. Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, 1999. In *IEEE Std 802.11-1999*.

[20] Ishac, J., and Allman, M. On the Performance of TCP Spoofing in Satellite Networks. In *IEEE MILCOM* (Oct. 2001).

[21] Jacobson, V., and Karels, M. J. Congestion Avoidance and Control. In *ACM SIGCOMM* (1988).

[22] Jain, A., and Floyd, S. Quick-Start for TCP and IP, Oct. 2002. Internet-Draft draft-amit-quick-start-02.txt.

[23] Jain, R. *The Art of Computer Systems Performance Analysis*, 1st ed. John Wiley and Sons, INC, 1991.

[24] Karn, P., and Partridge, C. Improving Round-Trip Time Estimates in Reliable Transport Protocols. *ACM Transactions on Computer Systems 9*, 4 (1991), 364–373.

[25] Katabi, D., Handley, M., and Rohrs, C. Congestion Control for High Bandwidth-Delay Product Networks. In *ACM SIGCOMM* (Aug. 2002).

[26] Kent, S., and Atkinson, R. Security Architecture for the Internet Protocol, Nov. 1998. RFC 2401.

[27] Krishnan, R., Allman, M., Partridge, C., and Sterbenz, J. P. G. Explicit Transport Error Notification (ETEN) for Error-Prone Wireless and Satellite Networks. Tech. Rep. No 8333, BBN Technologies, March 2002.

[28] Liu, C., and Jain, R. Approaches of Wireless TCP Enhancement and a New Proposal Based on Congestion Coherence. In *The 36th Hawaii International Conference on System Sciences, Quality of Service in Mobile and Wireless Network Minitrack, Big Island, Hawaii* (Jan. 2003).

[29] Mathis, M., Mahdavi, J., Floyd, S., and Romanow, A. TCP Selective Acknowledgement Options, Oct. 1996. RFC 2018.

[30] Mitchell, O. R., and Hall, R. Building TCP Proxies for Layer 5 to 7 Inspection, Sept. 2003.
http://www.commsdesign.com/story/OEG20030916S0025.

[31] Padhye, J., Firoiu, V., Towsley, D., and Kurose, J. Modelling TCP Throughput: A Simple Model and its Empirical Validation. In *ACM SIGCOMM* (1998).

[32] Pan, R., Prabhakar, B., and Psounis, K. CHOKe, A Stateless Active Queue Management Scheme for Approximating Fair Bandwidth Allocation. In *INFOCOM (2)* (2000), pp. 942–951.

[33] Postel, J. Internet Protocol, Sept. 1981. RFC 791.

[34] Postel, J. Transmission Control Protocol, Sept. 1981. RFC 793.

[35] Ramakrishnan, K., Floyd, S., and Black, D. The Addition of Explicit Congestion Notification (ECN) to IP, Sept. 2001. RFC 3168.

[36] Ramakrishnan, K. G., Floyd, S., and Black, D. L. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168, Internet Engineering Task Force, Sept. 2001.

[37] Saltzer, J. H., Reed, D. P., and Clark, D. D. End-To-End Arguments in System Design. *ACM Transactions on Computer Systems 2*, 4 (Nov. 1984), 277–288.

[38] Savage, S., Cardwell, N., Wetherall, D., and Anderson, T. TCP Congestion Control with a Misbehaving Receiver. *Computer Communication Review 29*, 5 (Oct. 1999), 71–78.

[39] Semke, J., Mahdavi, J., and Mathis, M. Automatic TCP Buffer Tuning. In *ACM SIGCOMM* (1998).

[40] Stevens, W. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms, Jan. 1997. RFC 2001.

[41] The Bluetooth SIG. The Bluetooth Core and Profile Specifications. http://www.bluetooth.com.

[42] TIA/EIA/IS-200.1-A. Introduction to CDMA 2000 Standards for Spread Spectrum Systems, Mar. 2000.

A. LEAST Code

The code in this appendix is written in the Python programming language but is high level enough that it may be read as pseudocode. This is the same code used for analysis by Allman, Eddy, and Ostermann [2], and is written for offline-analysis, as opposed to in-stack code operating in realtime. An in-stack implementation for the ns-2 simulator is available from the author of this thesis.

A.1 Reno TCP LEAST Code

```
seqno = ackno = highack = retransmits = dup_xmits = 0
in_rto_event = False

for pkt in snd_trace:
    if pkt.isAck () and (pkt.AckNo () > highack):
        highack = pkt.AckNo ()

    if pkt.IsData ():
        if pkt.SeqNo () > highdata:
            highdata = pkt.SeqNo ()
        else:
            retransmits += 1
            ## an RTO that initiates slow start-based loss recovery
            if not in_rto_event and pt_isRTO ():
                in_rto_event = True
                recovered = recovered_orig = highdata
```

```
            rto_segment = pkt.SeqNo ()
            event_retrans = 1
            event_dupacks = 0
            continue
## in slow start-based loss recovery
if in_rto_event:
    if pkt.IsData ():
        ## count retransmits in the event
        if pkt.IsRetrans () and (pkt.SeqNo () < recovered):
            event_retrans += 1

        ## an RTO within the RTO event; extend the event
        if pkt.IsRTO ():
            recovered = recovered_orig = highdata
            rto_segment = pkt.SeqNo ()

        ## track new packets sent during recovery -- we need to
        ## account for the last few duplicate ACKs
        if not pkt.IsRetrans () and (highack <= recovered_orig):
            recovered = pkt.SeqNo ()

    else:
        ## an ACK that terminates the RTO event
        if pkt.AckNo () > recovered:
            dup_xmits += min (event_dupacks, event_retrans)
            in_rto_event = False

        ## count duplicate ACKs received in the event -- but, not
```

```
            ## any associated with the RTO segment (which are not caused
            ## by needless retransmissions
            elif (pkt.AckNo () == last_ackno) and (pkt.AckNo() >= rto_segment):
                event_dupacks += 1
    ## track the last ACK number
    if pkt.IsAck ():
        last_ackno = pkt.AckNo ()

least = retransmits - dup_xmits
```

A.2 SACK and DSACK TCP LEAST Code

```
highdata = retransmits = dup_xmits = 0

for pkt in snd_trace:
    if pkt.IsData ():
        if pkt.SeqNo () > highdata:
            highdata = pkt.SeqNo ()
        else:
            retransmits += 1
    if pkt.IsACK ():
        if using_DSACK and pkt.DSACK () and WasRexmted (pkt.DSACK ()):
            dup_xmits += 1
        elif not using_DSACK and IsSACKRedundant (pkt):
            dup_xmits += 1

least = retransmits - dup_xmits
```