Improving the Performance of TCP in the Presence of Packet Reordering

by

Peter D. Yang

S.B., Electrical Engineering & Computer Science
Massachusetts Institute of Technology (2000)

Submitted to the Department of Electrical Engineering & Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering & Computer Science

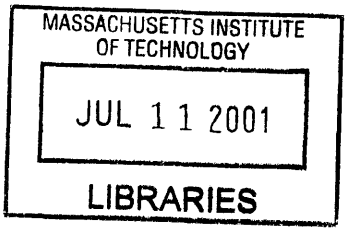
at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 23, 2001

© 2001 Peter D. Yang. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis and to grant others the right to do so.



BARKER

Author .....
Department of Electrical Engineering and Computer Science
May 23, 2001

Certified by .....
Professor Hari Balakrishnan
Thesis Supervisor

Certified by .....
Professor Robert Morris
Thesis Supervisor

Accepted by .....
Professor Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Improving the Performance of TCP in the Presence of Packet Reordering

by

Peter Yang

Submitted to the Department of Electrical Engineering & Computer Science

May 23, 2001

in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering & Computer Science

Abstract

Packet reordering adversely affects the behavior and performance of the Transmission Control Protocol (TCP), the predominant transport protocol on today's Internet. This shortcoming is becoming a problem of increasing importance, as packet reordering is on the rise on some Internet paths. The issue at the heart of TCP's poor performance in the presence of packet reordering is its inability to distinguish well between packet loss and packet reordering. In particular, the fast retransmission algorithm, which was added to TCP to improve its performance in the presence of packet loss, often misinterprets reordered packets as lost packets. This is problematic, as TCP uses packet loss as an indication of congestion. Thus, in addition to retransmitting the packet, a TCP sender performing a fast-retransmission also reduces its sending rate by cutting its congestion window in half. The spurious triggering of these actions when reordering is present needlessly degrades end-to-end throughput.

We have designed and implemented a backwards-compatible extension to TCP that helps to disambiguate packet reordering from packet loss, greatly reducing the number of spurious fast retransmissions even when excessive reordering occurs. Our approach, which modifies only the receiver, involves withholding for a short, adaptively-determined period of time the duplicate acknowledgments produced by out-of-order packets, rather than sending them back to the sender immediately. Doing so provides the receiver with more time to distinguish between loss and reordering before either setting off a fast re-transmission or resuming normal data flow. Experimental results for connections using our modified receiver show significant improvements in TCP performance in the presence of reordered packets, achieving, in cases of heavy reordering, more than five times the transfer throughput of connections using an unmodified receiver. As a result, we believe the introduction of our scheme augurs well for the future of TCP over paths that reorder packets.

Thesis Supervisor: Hari Balakrishnan
Title: Assistant Professor

Thesis Supervisor: Robert Morris
Title: Assistant Professor

Acknowledgments

First and foremost, I would like to thank my advisors, Professor Hari Balakrishnan and Professor Robert Morris, for all their guidance and support as I progressed through this project. They provided a great deal of insight in helping me grapple with the project itself, and their enthusiasm and wealth of knowledge helped both my interest in and knowledge of the field of networking to grow greatly in the past year. In addition, I am grateful for the understanding they showed throughout this project and, more specifically, the flexibility they showed in allowing me to find a thesis that better fit my skills and interests after an abortive attempt at another project fell to the wayside.

I also thank my officemates, Sidney Chang and Nick Feamster, and my pseudo-office-mate, Bodhi Priyantha, for their friendship and support throughout the long hours in lab that this project required.

Finally, I would truly like to thank my family: my father, Dr. Ho-Seung Yang; my mother, Mrs. Myung-Sook Yang; and my brother, the soon-to-be-Dr. Anthony Yang. I thank them in part for their encouragement during this year, but mostly for the constant love and support they have always shown me in everything that I have done.

Contents

Figures

Chapter 1

Introduction

1.1 The Problem

Opinions vary about the extent of packet reordering on paths in today's Internet. A number of studies of Internet packet dynamics have found it to be a rare occurrence and have observed that only 0.3% of all packets on the Internet arrive at their receiver out of order [21, 27]. However, studies have also found that reordering is becoming an increasingly common component of traffic over some Internet paths. As part of the same studies that found reordering to be rare, TCP connections in which up to 36% of the packets exhibited reordering were also observed. More strikingly, a study of the packet dynamics of the routers that make up the MAE-East exchange suggests that the probability that a TCP flow passing through that exchange will experience some degree of reordering is greater than 90% [3]. While one of the original purposes of the TCP was to transparently handle end-to-end reordering of application data in the network, subsequent modifications to the protocol have been added under the assumption that significant degrees of reordering are not common. As a result, TCP has become less robust in its handling of reordered data, and packet reordering has been found to adversely affect the behavior and performance of the TCP protocol itself.

In particular, the fast retransmission algorithm [24], which was added to TCP to improve its performance in the presence of packet losses, causes the protocol to handle reordered packets in the same manner that it does lost packets. Because TCP interprets packet losses to be an indication of congestion in the network, the fast retransmission algorithm will spuriously retransmit a packet that the receiver has already received, and, in a move detrimental to the throughput of the connection, cut the size of the sender's congestion window in half. Together, these actions work to needlessly decrease the end-to-end throughput of a TCP connection that encounters reordering. The purpose of the extensions to TCP proposed in this thesis is to prevent these needless drops in throughput by attempting to eliminate spurious activation of the fast-retransmission algorithm. In turn, this will allow TCP to more robustly handle reordering in the network without degradation of its own performance.

1.2 Our Solution

In this thesis, we implement specific modifications to the 4.4 BSD TCP implementation that prevent the needless triggering of the fast-retransmission algorithm and avoid the corresponding drop in TCP throughput. In particular, we develop a receiver-based scheme in which the sequence of incoming TCP data segments is observed and tested for packet reordering. In cases where reordering is detected, the subsequent duplicate acknowledgments produced at the receiver are not immediately sent back to the TCP sender immediately. Instead, they are withheld at the receiver and are released later in one of two carefully controlled ways. If the number of packet arrivals causes the number of withheld

acknowledgments to exceed an adaptively-determined threshold, packet loss is suspected and the duplicate acknowledgments are released in a controlled manner to trigger a fast-retransmission at the sender. Otherwise, acknowledgments are held until reordering is resolved, to forestall triggering a fast-retransmission. If reordering of the flow is resolved before the threshold is crossed, the correct sequential acknowledgments are sent back to the sender and data transmission continues without a drop in throughput. In effect, our scheme gives the TCP connection greater time to distinguish between packet reordering and packet loss, rather than immediately sending back the duplicate acknowledgments that would set off a spurious fast-retransmission at the sender.

In experiments testing the performance of our mechanism over some specific topologies that cause packet reordering, our modified TCP receiver proved to perform significantly better than a receiver using a normal, unmodified TCP implementation. More specifically, while the throughput of a TCP connection using the unmodified receiver dropped by more than 80% in cases of heavy reordering, we found that connections using our modified receiver suffered almost no drop in performance, with throughput of the connection virtually equalling that of a TCP connection where no reordering was present. Similarly, our results show that, in cases of extreme reordering, the number of spurious fast-retransmissions in connections using our scheme was reduced by almost 95% in comparison to connections using unmodified TCP receivers.

1.3 Research Motivation

TCP is now a mature protocol that is ubiquitous on the Internet, with several modifications and improvements to its original mechanisms made over the years. However, very little work has previously been done to address its poor performance in the face of packet reordering. The reasons for this are twofold, and are the result of a "chicken or egg" problem. On one hand, since TCP is known to perform poorly in the presence of out-of-order packets, much of the Internet has been constructed to reduce the occurrence of reordering. On the other hand, researchers contributing to the design and optimization of TCP observed that little packet reordering occurred in this Internet, and thus saw little need to improve the protocol to handle this rarely-seen case. Thus, as a result of these equal and opposite forces, little has been done to increase TCP's robustness in the face of reordering.

However, recent developments have caused this situation to change and have sparked interest in improving TCP's performance in the presence of reordering. First, previously noted, some studies have found that the incidence of reordering in some paths in today's Internet is on the rise. Thus, because reordering causes current TCP implementations to perform poorly, there is an immediate need to modify TCP to improve its performance in these situations. A second, more proactive reason to pursue research in this area is to enable the use of new network technologies that induce packet reordering. Such disparate technologies as multipath-routing and ad-hoc mobile networking all show promise in increasing the reach of the Internet, but currently have limited applicability due to TCP's inability to handle packet reordering. Modifying TCP to more robustly handle packet reordering would increase both the applicability of and interest in such technologies.

1.4 Research Contributions

The two main contributions of this thesis are the novel solution it provides to the problem of TCP's poor performance in the presence of packet reordering and the ease of deployment that our specific design facilitates.

1) A solution to a problem of growing importance

In this thesis, we introduce a new scheme for increasing TCP's robustness in the face of packet loss that both improves end-to-end throughput for reordered flows and matches the performance of unmodified TCP in cases of in-order packet delivery. Modifying TCP to handle this case is still a relatively unexplored area of research, and we feel that our scheme provides a simple and elegant solution to this problem of growing importance.

2) A solution that is easily deployable in today's Internet

By providing a backwards-compatible solution that is implemented solely in the receiving end of a TCP connection, we provide a mechanism that can be easily deployed by only those users who wish to use it in networks that reorder packets. It does not require the adoption of new TCP options or the cooperation of the sending end of the connection. We believe this is one of the great advantages of our scheme, as it allows any user who is part of a network that induces packet reordering and who wishes to improve their throughput to plug our mechanism into their TCP receiver at their own choosing.

Chapter 2

Background

The current architecture of the Internet has been constructed with the implicit goal of keeping packets in the network in order. In designing networking equipment and protocols, network designers have been able to use as a premise for their designs a model of the Internet in which packets in the network are routed through FIFO queues that keep individual flows in order. One of the results of this practice is that TCP, in its current form, relies heavily on this assumption of in-order delivery in many of its mechanisms, such as fast-retransmit and the use of cumulative acknowledgments. As a result, TCP's performance has been optimized for in-order delivery and is adversely affected by packet reordering.

2.1 Causes of Packet Reordering

However, this paradigm of the Internet as an in-order delivery network has begun to break down, in large part due to the recent phenomenon of exponential growth in network usage. Rather than being a direct result of the increase in Internet traffic, reordering has instead been introduced into the network as a result of practices used by ISPs and router designers at the link and network layers to cope with increasing network loads. In particular, three optimizations introduced by network designers - parallelism in links between routers, per-

packet multipath routing, and the use of routing table caches - are factors contributing to the current increase in packet reordering in today's Internet.

A second source of packet reordering is the introduction of entirely new types of networks onto the Internet that do not rely on the current fixed infrastructure of the Internet. Instead, technologies of this type have dynamic physical infrastructures that are more flexible and easily deployable. However, as a consequence of their dynamism, networks of this type induce reordering of the packets that travel over their paths. Ad-hoc mobile networks are an increasingly popular technology that have this property.

2.1.1 Link-level parallelism

The increased use of parallel links between routers is discussed in depth in [8], and the paper argues that reordering comes as a natural result of this parallelism. In particular, the use of 'hunt groups', or collections of ports that act as a virtual link between routers, in the DEC routers used in the MAE-East exchange is identified as causing packet reordering over paths using those routers. In this scheme, these parallel links are used as a low-cost means of increasing the bandwidth over a single link. A number of lower bandwidth links are grouped together to approximate the bandwidth of a single higher-bandwidth, higher-cost link; for example, ten 100Mbps links may be bundled together to approximate a single Gigabit/sec. link.

However, one result of this practice is that packets from a single flow can be divided up among the parallel links. As a result, head-of-line blocking in the individual ports of a virtual link can result in reordering when packet sizes are variable and a number of flows

are multiplexed through the same virtual link. More importantly, the paper shows that if a fair arbitration scheme is used to determine which port is allowed to transmit a packet, blocking can also occur, resulting in reordering.

2.1.2 Per-packet multipath routing

A second cause of reordering is discussed in [22]. As described in this paper, per-packet multipath routing (or 'route fluttering', as it is termed in the paper) is a network layer mechanism that, like link-level parallelism, is deliberately used to mitigate the effects of increased network load on an overburdened route. This is done by splitting the load on a packet-by-packet basis between two distinct routes to the destination. While multipath routing does fulfill a load balancing function, fluttering between two routes with differing round-trip times has the problematic effect of causing out-of-order packet arrival for an individual TCP flow, unless the round-trip times for the two links are identically matched. As with parallel links, the degree of reordering can vary widely, as the delay over each link can fluctuate with congestion.

While a number of traffic dispersion schemes of this type have been proposed [13], the very fact that TCP performs poorly in the presence of packet reordering has resulted in the general non-deployment of per-packet multipath routing in the global Internet. As a result, most multipath routing today is performed at the granularity of individual TCP flows, to keep the packets of each flow in-order. Part of the purpose of this project is to increase TCP's robustness to allow greater use of schemes like per-packet multipath routing that cause packet reordering but would allow for more efficient use of network

resources.

2.1.3 Routing cache misses

A third possible cause of reordering is an artifact of some earlier router architectures. The use of routing table caches in fast-path line cards was a practice that helped routers cope with increased load on high-speed links. While router designers are increasingly using schemes like Cisco's Express Forwarding [6] that store entire routing tables in the forwarding path, older router designs placed small routing table caches in the fast-path to allow them to directly forward packets without performing a full slow-path route lookup.

However, in certain cases, route cache misses can cause packets to be reordered within a router. In particular, if a long string of packets from a single flow arrive at a line-card input port and their destination address is not in the cache, all arriving packets from that flow are forwarded to the slow-path CPU in order to do a route lookup. However, once the forwarding entry for the first packet in the flow is found, it will also be written to the cache and any succeeding packets in the flow will be automatically forwarded through the fast-path using the route cache. However, a full route-lookup will need to be performed for the remaining packets that are queued in the slow-path. The greater delay for these packets in comparison to ensuing packets in the flow that are forwarded through the fast path will cause the packets stuck in the slow-path to be reordered within the flow. While this scenario will only affect a few packets at the beginning of flows whose destination addresses are not included in the cache, in routers that serve a large number of destination addresses and exhibit little locality in caching addresses, a large number of cache misses could cause

reordering in a significant number of flows.

2.1.4 Ad-hoc mobile networking

Ad-hoc mobile networks such as MIT's CarNet/Grid system [20] are networks that do not rely on any fixed network infrastructure to route data [5, 17]. Instead, constantly moving mobile nodes use one another, rather than dedicated routers, to create networks paths from a source to a destination. While such a scheme increases the flexibility and deployability of mobile networks, the constantly changing nature of the ad-hoc network paths between a mobile source and a mobile destination results can result in significant reordering of packets travelling in such a network. As a result, current versions of TCP will not work well with ad-hoc mobile networks, and increasing TCP's performance in the presence of reordering will help to more seamlessly integrate such networks into the infrastructure of the Internet.

2.2 Effects of Packet Reordering on TCP

In order to understand the effects of packet reordering on the performance of TCP, it is important to have an understanding of the specific mechanisms that are affected and the strong packet ordering assumptions these mechanisms rely upon. In particular, the fast-retransmission mechanism [24, 25, 26, 14, 15] relies heavily on TCP's implied assumption of packet ordering to make its prediction about which packets have been lost due to congestion in the network and, as a result, is adversely affected by packet reordering.

2.2.1 Cumulative and duplicate acknowledgments

One of the key features of TCP is its use of acknowledgments (ACKs) for each segment of data sent. The receiving end of a TCP connection sends back an acknowledgment to the sender telling the TCP sender that a segment has been received and which segment it expects to receive next. TCP communicates this information by including in the ACK an acknowledgment number that corresponds to the sequence number of the next packet expected by the receiver. TCP uses a system of cumulative acknowledgments to set this acknowledgment number, with the value set to indicate to the sender that the receiver has received all packets with sequence numbers less than the acknowledgment number. Such a system provides a level of redundancy that makes the TCP protocol more robust to the loss of ACKs in the network.

In TCP using cumulative acknowledgments, the acknowledgment numbers sent from the receiver to sender should monotonically increase if data packets are not being lost or reordered in the network. However, if a data segment is received that is not the next expected in-sequence segment, the receiver will not return the usual ACK for the next-expected segment after the incoming segment. Instead, a duplicate acknowledgment will be generated for the next-expected segment after the most recently received in-order segment. Thus, a duplicate acknowledgment tells the sender that a segment has been received; however, this received segment is not the next-expected packet in the sequence. A duplicate acknowledgment also tells the sender that the expected packet has still not been received at the other end.

2.2.2 Effects due to TCP's fast-retransmission algorithm

Originally, TCP used only a timeout-based system to determine if packets were lost and subsequently retransmit them. However, as the detrimental effects of waiting more than a round-trip time to retransmit a packet became apparent, the fast-retransmission algorithm [24, 25, 26, 14, 15] was added to TCP to allow TCP senders to more quickly

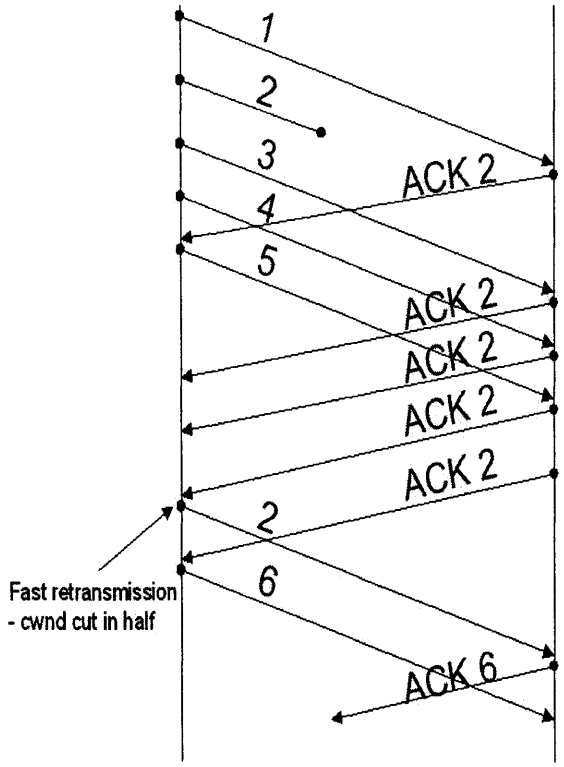


Figure 2.1: *TCP's fast-retransmission algorithm: a packet is retransmitted upon reception of three duplicate ACKs at the sender. The sender's cwnd is also cut in half*

detect and retransmit segments that were assumed to be lost. Rather than wait for an entire retransmission timeout before assuming that a packet has been lost, this mechanism makes an educated guess about which segments have been lost by observing the stream of

acknowledgments that arrives at the TCP sender. Relying on the fact that ACKs from the receiver should be monotonically increasing for an in-order data stream, the fast retransmit mechanism assumes at the sender that a segment has been lost when three duplicate ACKs arerereceived. It then proceeds to retransmit the missing segment to the receiver. In addition, TCP also cuts the congestion window *cwnd* to half its previous value and sets the slow-start threshold *ssthresh* equal to the value of *cwnd*. These actions, called the fast-recovery mechanism, are performed in conjunction with fast-retransmission under the assumption that packet losses, as detected by fast-retransmit, are indications of congestion in the network. The use of the fast-retransmission mechanism significantly improves TCP's performance in the presence of packet losses, as the connection is not required to wait an entire costly retransmission time-out before retransmitting a packet that is presumed to be lost.

However, in the case where reordering exists in the network, the presence of the fast-retransmission algorithm causes TCP to handle reordered packets in the same manner it does lost packets. If the next-expected packet arrives more than three packets out-of-order, at least three dupACKs will have already been sent by the time that the packet actually does arrive. As a result, because TCP interprets three dupACKs as a packet loss and packet losses are interpreted as an indication of congestion in the network, the fast retransmit and fast recovery algorithms will spuriously retransmit a packet and erroneously cut the size of the sender's window in half. TCP's throughput is needlessly decreased as a result of its incorrect assumption of loss.

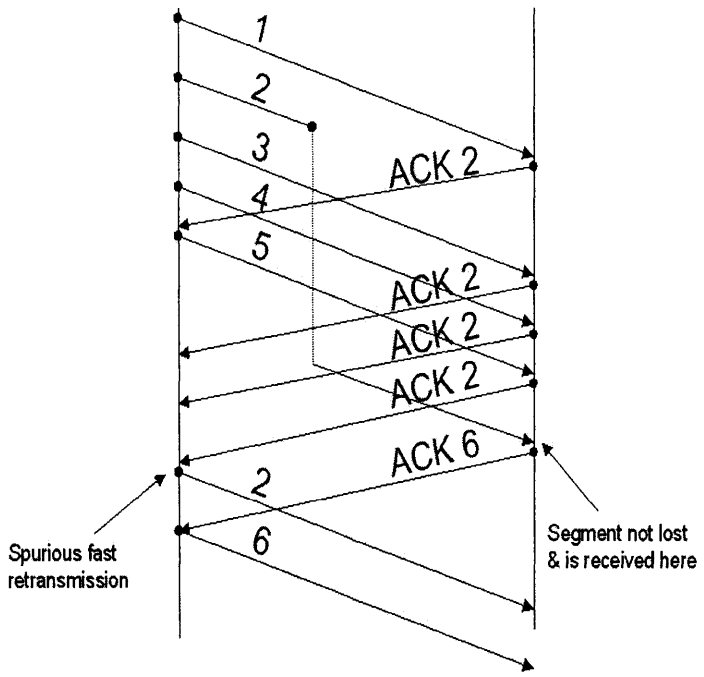


Figure 2.2: *A diagram of a spurious fast-retransmission due to reordering. Because the packet arrives more than three packets out-of-place, three dupACKS are sent to the sender before the packet arrives and a fast-retransmission is incorrectly triggered.*

The particular source of this problem is the small three duplicate ACK threshold used to trigger fast-retransmissions, which is a value based on TCP's bias towards in-order delivery. RFC 2001 [24] actually codifies this bias, stating:

> Since TCP does not know whether a duplicate ACK is caused by a lost segment or just a reordering of segments, it waits for a small number of duplicate ACKs to be received. It is assumed that if there is just a reordering of the segments, there will be only one or two duplicate ACKs before the reordered segment is processed, which will then generate a new ACK. If three or more duplicate ACKs are received in a row, it is a strong indication that a segment has been lost.

Thus, the fast retransmission threshold was set under the premise that any reordering in the network results in packets arriving no later than three segments after the segment with the previous sequence number. While this is an accurate assumption for an Internet that keeps packets in-order, in an Internet where parallel links exist or multipath routing is used, this is a faulty assumption and will cause TCP to regard reordered packets as lost packets, resulting in spurious fast retransmissions and corresponding drops in throughput. If packets are greatly reordered within a single window, it is possible that multiple retransmits will occur and that the value of *cwnd* will be decreased to a very low value, a problem discussed in [3]. If this continues over the lifetime of a flow, TCP will not be able to properly open its congestion window, growth of *cwnd* will be stunted, and will not correctly utilize its allocated bandwidth [11].

2.2.3 Other effects

In addition to the detrimental effect on throughput caused by the fast retransmission mechanism's spurious reduction of the congestion window, reordering affects TCP in a number of other ways. Most prominently, the header prediction algorithm [16, 26] processes packets in a fast path through the TCP code in the case where "TCP is receiving data [and] the next expected segment for this connection is the next in-sequence data segment." Any packets that do not fit this profile, including out-of-order packets, are processed more slowly through TCP's general slow-path, resulting in greater processing time at the receiver. Thus, reordering affects performance by adding to the per-packet processing time at the TCP receiver. Similarly, additional processing overhead is incurred by the

receiver in retrieving packets in the correct order from TCP's reordering buffer. In addition, bandwidth is wasted through the unnecessary retransmission of packets that are received but are more than three packets out of order.

Reordering can also occur in the reverse path, and TCP acknowledgments can experience the same type of reordering that data segments do, resulting in a loss of self-clocking and bursty TCP transmitter behavior [3]. Thus, reordering has a number of negative effects on TCP performance in addition to its impact on the fast-retransmission algorithm. However, as we believe the unnecessary decrease in throughput caused by spurious fast-retransmissions is by far reordering's most detrimental effect on TCP's performance, that is the problem focused on in this project.

2.3 Prior Work

The incidence of reordering in today's Internet has been explored in a number of studies [3, 21, 27]. The examination of the packet dynamics at MAE-East [3] also includes an in-depth exploration of the effects of reordering on TCP. While the problem has been well-defined in this manner, almost no published work existed that discusses implementations of possible changes to TCP that would allow it to more robustly handle packet reordering and improve its performance. Recently, however, three schemes have been proposed that tackle the problem in different ways.

2.3.1 D-SACK and the Eifel algorithm

The first scheme, outlined by Sally Floyd in [10], proposes extensions to SACK TCP that would allow spurious fast-retransmissions that occur due to reordering to be detected and essentially be "undone". The main idea behind this scheme is that a spurious fast-retransmission caused by reordering will result in the TCP receiver receiving a duplicate data segment -- the original out-of-order packet and the packet sent by the fast-retransmission mechanism when it predicted that the original packet was lost. This prediction is made based on the fact that packet duplication in the network itself is rare (this is supported by experimental data in [21]). Thus, duplicate packets can be used as an indication of spurious fast-retransmission. If news of this reception of a duplicate segment can then be relayed back to the sender, the sender can determine *a posteriori* that a fast-retransmission was unnecessary and undo the reduction of the congestion window, doubling it and returning it to its proper size.

This scheme requires a mechanism by which the receiver can notify the sender that it has received a duplicate segment. Floyd's email discusses a proposed change to SACK which, when a TCP data segment that has already been covered by the cumulative ACK (ie. is a duplicate of a previously received segment) arrives at the receiver, would then use the first SACK block of the resulting duplicate acknowledgment to tell the sender which duplicate segment has been received.

This recommendation was documented in July 2000 in RFC 2883 [12], which outlines a SACK extension called D-SACK (with the D standing for 'duplicate') that incorporates the changes Floyd outlines in her email. It also includes a description of the fast-retransmission 'undo' scheme. However, while both documents propose these changes to SACK

TCP, no known implementation is known of at this time. Because this scheme requires the adoption and implementation of the new D-SACK option at all TCP senders and receivers, it is less desirable than a scheme implemented only at the sender or receiver. In addition, in 'undoing' the reduction of *cwnd* caused by the fast-retransmission, *cwnd* is instantaneously doubled in size. This causes a burst of packets to be transmitted at once, which is detrimental to TCP performance and congestion control.

The Eifel algorithm [18] uses the same scheme that Floyd proposes for "undoing" spurious reductions of the sender's *cwnd*. However, rather than requiring the use of a new option, this design uses either timestamps or bits in the TCP header's reserved fields to notify the sender of reception of duplicate segments. This scheme in its implemented form is subject to the same shortcomings as the D-SACK scheme, namely the need for modifications to both the sender and receiver and burstiness due to sudden inflation of the congestion window. While Reiner Ludwig, Eifel's designer, does acknowledge the second problem in his paper, he does not address it in his actual implementation.

However, in addition to undoing false reduction of the congestion window due to spurious fast-retransmissions, both the D-SACK scheme and the Eifel algorithm are also able to detect and undo spurious retransmissions caused by timeouts. This is by virtue of their use of duplicates segments as an indication of unnecessary retransmission, as spurious retransmissions of both the timeout and fast-retransmit variety will result in duplicate packets at the receiver. While spurious timeouts are currently a rare occurrence on the Internet, Ludwig postulates in his paper that they will become more common as hosts that use packet-radio networks and that, as a result, can become disconnected for seconds at a

time without losing data become more ubiquitous on the Internet. Thus, the Eifel algorithm and D-SACK cover a greater number of cases than our scheme, which only addresses the specific problem of spurious fast-retransmissions due to reordering.

In her email, Floyd also states that notification from receiver to sender that a duplicate segment has been received could also "be used to modify [the sender's] dupACK threshold." Such a modification could forestall future fast retransmissions by increasing the amount of time that TCP has to distinguish between packet losses and packet reordering. By increasing the threshold, TCP can prevent losses from being confused with reordering by the fast-retransmission logic. However, she fails to go into detail about how such a scheme would be implemented.

2.3.2 Adjusting the fast-retransmission threshold

Such a scheme is discussed as part of Vern Paxson's examination of end-to-end packet dynamics and the corresponding network pathologies [21], which suggests adjusting the value of the fast-retransmission threshold to better handle the effects of packet reordering. The paper distinguishes between 'good' sequences of duplicate ACKs (those that correspond to actual packet losses) and 'bad' sequences (those that cause unnecessary retransmissions due to packet reordering). The ratio of good to bad retransmissions is then used to quantify the effect of adjustments made to both the duplicate ACK threshold and the waiting time, which is defined as a delay that the receiver observes before sending out a duplicate ACK.

Paxson's examination illustrates the trade-off that comes as a result of adjusting the duplicate ACK threshold. For example, raising the threshold to four duplicate ACKs

increases the good/bad ratio 2.5 times, but at a cost of 30% fewer actual fast-retransmission opportunities (as a greater number of time-outs will occur before the threshold is reached). Lowering the threshold to two dupACKs gains approximately 70% more opportunities for fast-retransmissions, but also causes the good/bad ratio to decrease by a factor of three as more reordering events are interpreted as losses. However, adding a wait time of 20 ms to the receiver before generating a second dupACK keeps the fast-retransmit opportunities the same while keeping the good/bad ratio nearly the same as for the three-ACK threshold. While adding this optimization would require modifications to both the sender and receiver (thus complicating the implementation and deployment of the solution), this problem can be dealt with by implementing the wait time as part of the sender by delaying the triggering of the fast-retransmission algorithm by 20 ms after the duplicate ACK threshold has been surpassed.

Paxson's scheme incorporates many of the same concepts that will be used in this project. In particular, adjusting the threshold value to reduce spurious retransmissions and implementing the modifications solely in either the sender or receiver are features also used, in modified form, in our strategy. However, the scheme Paxson describes in [21] statically sets the threshold value, requiring users to manually tune their TCP connections and live with the trade-off between decreased performance caused by reordering or decreased performance caused by a greater number of RTOs. In contrast, we propose a scheme that will in effect dynamically adjust the retransmit threshold. By doing so, we intend to both equal TCP's throughput in situations where little reordering occurs and improve its performance in the presence of reordering by limiting the number of spurious

fast-retransmissions. In doing so, the hope is that a more robust TCP will give the user the best of both worlds when it comes to TCP throughput.

Chapter 3

Design

3.1 Design Goals

A number of design criteria were considered when determining how to modify TCP to handle packet reordering, including:

1) Improved TCP throughput for flows that experience reordering in the network

For reasons described previously, the main goal of this project is to make TCP more robust in the face of packet reordering in the network.

2) Minimal effects on TCP in the absence of reordering (i.e., throughput equal to unmodified TCP when no reordering is present)

While the problem being explored is improving TCP's performance when reordering is present, any modifications to TCP must also ensure that throughput in the normal case where no reordering is present is not adversely affected. This is because it is expected that the bulk of the data flow will consist of correctly ordered packets. Any mechanisms we introduce should also behave robustly in cases of actual packet loss and approximate as closely as possible TCP's performance in those situations. We believe that these goals are of great importance, as normal performance must be unaffected in order for any proposed changes to TCP to be adopted by users.

3) Implementation in the TCP stack of either the sender or receiver, not both

Another decision that was made was to implement any changes to TCP solely in either the receiver or sender. The basis for this decision was the fact that any solution that was implemented in both the sender and receiver would require that the modified TCP be deployed by the independent parties administering each of the ends of a TCP connection.

While common sense would dictate that any modification to TCP that clearly provides performance benefits would immediately be added to existing implementations of TCP/IP stacks, this is not necessarily the case. For example, the SACK option was presented to the Internet community in an RFC in 1996 and has been shown to clearly improve TCP's performance [8]. Despite this fact, in a December 1999 study of an AT&T modem pool where 58% of SYN packets contained 'SACK permitted' messages, only 5% of corresponding TCP senders responded with ACKs where SACK was also enabled. Thus, only 5% of the resulting TCP connections actually used SACK [9]. While a dialup modem pool is clearly a biased sample, this example does serve as an indication of the difficulties in deploying a change to TCP that requires both sender and receiver modifications.

In this specific case, while SACK has become standard issue in the TCP stacks of most OS's used in the personal computers that initiate most TCP connections and are the primary TCP data receivers on the Internet, few of the high-volume servers that comprise most of the TCP senders chose to use it at the time of the study. This may have been because the administrators of servers have little incentive to implement an update that primarily benefits the other end of the connection at an increased processing cost for the server. Thus, the case of SACK shows that, in cases where a desired modification to TCP

must be implemented at both ends of a connection to work, differing incentives may cause uneven implementation of such features. Because cooperation of both ends of the connection is required for such a modification to work, the level of its acceptance will track the end-user group that more slowly implements the change.

It should be noted that high throughput and reordering of segments are an issue that greatly affects the user at the TCP receiver that is receiving data but have little impact on the TCP sender transmitting that data. Because of this, implementing a solution in the receiver would allow those users who know that they will be connected to a network that causes a great deal of packet reordering to use the modified receiver at their own choosing, without requiring the cooperation of the party at the other end. Thus, a solution that does not rely on cooperation between the TCP sender and receiver and preferably can be implemented solely in the receiver was decided upon as one of the essential design criteria.

3.2 Design Description

The solution described in this thesis to address TCP's decreased performance in the face of packet reordering is based on a single observation. TCP's degradation in throughput in the presence of reordered packets is a direct result of spurious triggering of the fast-retransmission algorithm at the TCP sender. In particular, the algorithm's use of the low three dupACK threshold results in an erroneous fast-retransmission when reordering in the network causes a packet to arrive more than three packets out of sequence. Thus, the desired goal of our solution is to eliminate these throughput-reducing spurious retransmissions caused by reordering.

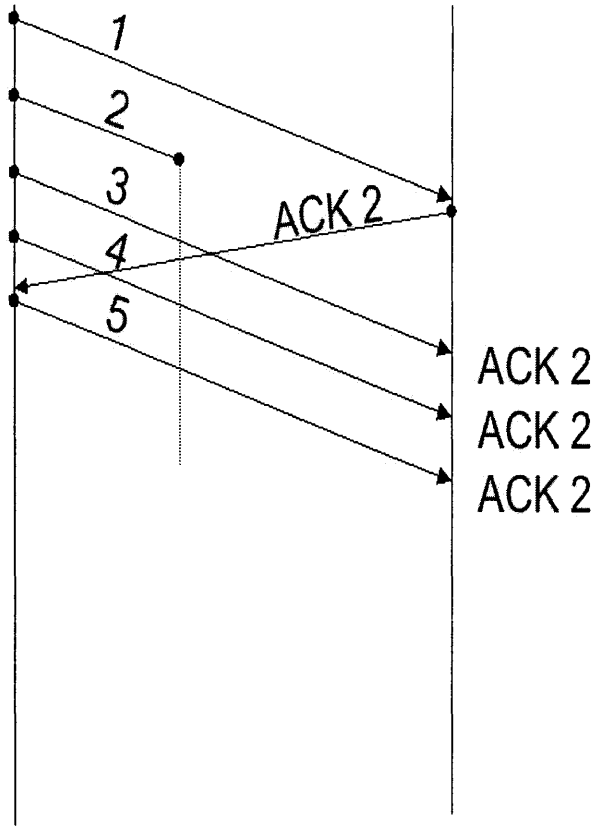


Figure 3.1: *Our solution: Provide receiver with additional time to disambiguate packet loss and packet reordering before setting off the fast-retransmission mechanism at the sender. We accomplish this at the receiver by withholding dupACKs for a dynamically-determined amount of time, rather than sending them back to the sender immediately*

To accomplish this goal, we propose enhancements to TCP that give the receiver additional time to distinguish between loss and reordering before sending the dupACKs that trigger a fast-retransmission at the sender. As such, the operation of our mechanism can conceptually be separated into three parts. In the first, reordering of incoming segments is detected and duplicate ACKs are withheld at the receiver, rather than being sent back immediately to the sender. At the same time, for each packet that arrives out of order, the

receiver determines how far out of order the packet is and uses that value to possibly update the value of the reordering threshold, which is used as the cutoff for determining the number of reordered packets that can arrive before we assume that a packet is lost rather than reordered. Finally, when either the next-expected packet arrives or the number of incoming reordered segments exceeds the threshold, indicating loss, the appropriate duplicate or sequential acknowledgments are sent back to the TCP sender.

3.2.1 Main idea: Withholding duplicate ACKs

To forestall throughput-reducing spurious fast-retransmissions and adhere to the design criteria, a receiver-based solution was explored in which the receiver holds onto duplicate ACKs rather than immediately sending them back to the sender. In this scheme, the receiver observes the sequence numbers of incoming data segments and determines if packets are arriving out of order. In cases where reordering is detected, the receiver withholds the dupACKs until either the reordering is resolved and the correct in-order segment arrives, or until reordering is not resolved and the number of arriving segments increases the number of withheld dupACKs beyond an adaptively-determined threshold. If reordering is resolved before the number of incoming packet arrivals exceeds this threshold, the withheld dupACKs are discarded and the correct sequential ACKs for the data segments corresponding to the withheld dupACKs are paced out in their place. As a result, a spurious fast-retransmission and its corresponding erroneous halving of the sender's *cwnd* are prevented, and the throughput of the connection is maintained. The cost to the connection in this scheme is the increase in the round-trip time incurred when ACKs are withheld at the receiver, rather than being immediately sent back to the sender. However, the argument

at the heart of this project is that the reduction in throughput-reducing spurious fast-retransmissions far outweighs this added delay.

3.2.2 Adaptively-determined threshold

In order for this scheme to work on the global Internet, a threshold is required to account for packet losses. In a network with no losses, it would be sufficient to simply withhold all duplicate ACKs until the next-expected packet arrives and reordering is resolved. However, packet losses are an inherent part of the Internet and retransmission timeouts are expensive operations on a high-bandwidth link. Thus, though it is impossible to unequivocally determine that a packet which has not yet arrived at the receiver has been lost and not reordered, a method for roughly differentiating between the two conditions is an important part of this scheme. The adaptively-determined threshold used in our mechanism fulfills that role, as it is used by the receiver to determine the number of dupACKs to hold on to before assuming that the next-expected packet has been lost rather than reordered.

One of the key design considerations for our scheme is the manner in which this threshold is dynamically set at the receiving end of the TCP connection, as well as the underlying metric used as its basis. We wish to set the threshold high enough to allow almost all cases of reordering to be resolved before the number of packet arrivals crosses the threshold, while not setting it so high as to cause a large delay before dupACKs are sent back in cases of actual packet loss. As such, we believe that an approximation of this ideal threshold value can be found by setting it to the maximum amount of packet reorder-

ing that the TCP flow has experienced over many packet arrivals. If it is assumed that the level of reordering is not continually increasing, which is a valid assumption for the causes of reordering discussed previously, this value provides a reasonable estimate of a maximum bound on the amount of reordering the flow is experiencing and will encounter in the future. As a result, setting the threshold to this value, and subsequently withholding ACKs for packet arrivals until the number of arrivals crosses the threshold or reordering is resolved, should provide the receiver with enough time in almost all cases to make the decision about whether a packet has been lost or has been delayed due to reordering.

The decision to use this method to set the threshold leads to the question of how to measure and quantify the amount of packet reordering observed by a TCP receiver. In our scheme, we wish to use a metric that accurately reflects the degree of reordering observed within the flow. An examination of the manner in which TCP handles packet reordering reveals a natural way to measure this. When reordered TCP segments (i.e., segments with sequence numbers greater than that of the next-expected packet) arrive at the receiver, they are not discarded and instead are buffered in a structure called the reordering queue. We first define a metric called the *degree* of reordering of each of the segments in the queue. This corresponds to the difference between the starting sequence number of the next-expected packet after the last in-order packet received and the next-expected packet after each out-of-order packet that is placed in the queue. We then use a metric called the *stride* of reordering to quantifiy the maximum degree of reordering that a packet in the flow is currently experiencing. It does so by measuring the degree of the most out-of-order packet, which is the segment in the queue that begins with the highest sequence number.

In doing so, we measure the greatest amount within the sequence number space that a packet is currently out of order in the flow.

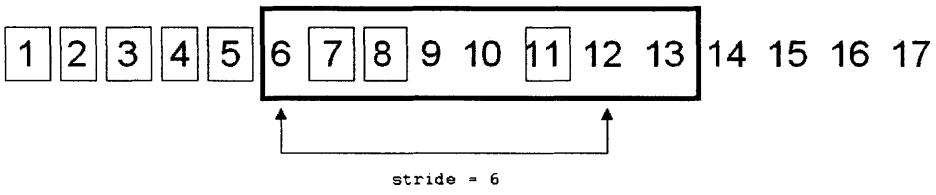


Figure 3.2: *Value of the stride of reordering in one example of out-of-order arrival at the receiver. Sequence numbers of packets that have already been received are marked with squares. In this example, the stride equals 6. This is the difference between the sequence numbers of next-expected segment after segment 5 (the highest in-order segment received) and segment 11 (the highest out-of-order segment received).*

The reordering stride accurately quantifies the current, instantaneous level of reordering in a flow in the manner we desire. If no packets are being reordered in the network, the reordering stride will equal zero, correctly indicating that packets are arriving in order. If packets arrive greatly reordered and a packet from much later in the flow arrives much earlier than the next-expected segment in the flow's sequence, the difference between the sequence numbers of the next-expected packet and the packet expected after the arriving packet will be large, correctly indicating a large amount of reordering in the flow.

By setting the threshold to equal the maximum stride seen over a large number of reordered packet arrivals, an educated guess can then be made that a packet loss has occurred

if the number of reordered packet arrivals exceeds the threshold, as we have not seen levels of reordering of greater values than that in the past. In this case, the withheld duplicate acknowledgments are then paced out from the receiver at a rate that approximates that with which the corresponding packets arrived at the receiver. In the event of actual packet loss, the transmission of these withheld dupACKs will correctly trigger the fast-retransmission algorithm at the sender.

In addition, the throughput of the connection is only minimally affected by this mechanism. The sender, as part of the fast-recovery algorithm, will continue to send new segments upon reception of each dupACK, and the only cost incurred will again be the increased delay caused by the withholding of the ACKs. In the case where the next-expected packet arrives after the threshold is exceeded and the pent-up dupACKs have already been released (ie. where the amount of reordering is greater than the threshold), the throughput of the connection suffers due to the spurious halving of *cwnd*. However, as this spurious retransmission would have occurred if an unmodified TCP receiver were being used, there is no loss of performance over normal TCP. Thus, because the number of spurious fast-retransmissions that are avoided as a result of this mechanism far outweighs the number it causes, the withholding of dupACKs and the use of a reordering threshold work together to greatly increase the throughput of a TCP connection that experiences packet reordering.

While the use of this threshold-based scheme works well in general, care must be taken to handle a number of special cases. In particular, while one of the great advantages of our scheme is that it is implemented solely in the receiver, a drawback is that our mech-

anism is not privy to pertinent information about the flow that is known only to the TCP sender. As a result, there exist two degenerate cases that our design has been modified to handle. In the first, since the receiver has no information about the size of the sender's window, *cwnd*, it is possible that the number of packet arrivals used as the threshold exceeds the window size. In such a case, if a packet is actually lost but the threshold is not crossed, the number of packet arrivals would not exceed the threshold and the receiver would stall without releasing any ACKs. To handle such cases, a timeout was introduced that approximates the time it takes for a packet with a degree of reordering equal to the maximum stride value to arrive at the receiver. If no packets arrive at the receiver before the timeout expires, we can assume with a fairly strong degree of certainty that no more packets will be arriving at the receiver as, in the past, we have not waited for longer than this time for any packet to arrive. The timeout value was calculated to be the threshold value times the value of a weighted-moving average of packet-interarrival times calculated at the receiver.

The source of a second degenerate case in our scheme is the fact that the receiver cannot unequivocally determine if, when a segment arrives and resolves reordering, that segment is the reordered segment it has been waiting for or the retransmission of a lost segment. If the segment that arrives and resolves the reordering is in fact a retransmitted segment rather than an out-of-order segment, we do not wish to use its stride value in our determination of the maximum stride used as the threshold, as the threshold will not provide an accurate measure of the past level of maximum reordering. Instead, in this case, it will incorrectly be set to a full window size. In addition, all other stride values calculated for all packet arrivals that arrive after the lost packet should not be counted in the threshold

calculation, since the failure of that packet to arrive was in fact due to loss and not reordering and our threshold is intended to be a measure of reordering only. To handle such cases, the time interval between when the next-expected packet failed to arrive in order and the time of its actual arrival is measured. It is then compared to the round trip time estimator kept at the TCP receiver and, if the measured time interval is greater than this RTT, it is assumed that the packet that resolves the reordering is a retransmission. In such cases, all stride values calculated for all packets in this interval are disregarded when determining the threshold value.

Another related design decision involves determining which stride values to use in the calculation of the threshold. In previous versions of our designs, the reordering stride was calculated for every packet arrival, no matter if it was an in-order packet or an out-of-order packet. The design was later changed to only record the stride for out-of-order packet arrivals only. The reason for doing this was that by choosing the threshold from among only stride values produced by reordered packets increases the robustness of our scheme in the presence of intermittent reordering (for example, reordering caused by routing cache misses). In such scenarios, long periods of in-order delivery would result in a threshold value of zero. As a result, each transitory period of reordering would result in a spurious fast-retransmission as the threshold was being set. In addition, once the transitory period of reordering ends, the threshold is again driven down to zero by the in-order packet flow, and the receiver will be unprepared for the next instance of reordering that occur. In contrast, by only recording the value of the stride for actual reordered packets, we maintain a viable threshold between instances of reordering that gives the mechanism a better chance to resolve the reordering when such transient cases occur.

A side-effect of this change was that it resulted in our code performing no actions during in-order packet flow. As a result, it ensures that our scheme for improving TCP's performs in the face of reordering as well as normal TCP in cases of normal, in-order packet flow, which was one of our original design goals. Because only duplicate acknowledgements are withheld and the next expected packet arrives as anticipated in cases of correct packet ordering, the corresponding sequential cumulative ACK is sent upon each packet arrival (or delayed appropriately if delayed acknowledgments are being used). Neither the dupACK withholding mechanism or threshold come into play, and operation at the modified receiver is identical to that of an unmodified receiver. Thus, throughput for the connection is not affected.

3.2.3 Pacing out acknowledgments

Once reordering is resolved or the threshold is crossed, the appropriate acknowledgments, either sequential or duplicate, are released from the receiver back to the sender. To maintain the self-clocking nature of ACK stream back to the sender and prevent ACK compression, the withheld acknowledgments must be paced out to the sender at carefully spaced intervals, rather than being burst back at once. In addition, in the case of sequential ACKs, the acknowledgments being sent must be relabelled with the correct cumulative acknowledgment numbers corresponding to the sequence numbers of the data segments that arrived at the receiver.

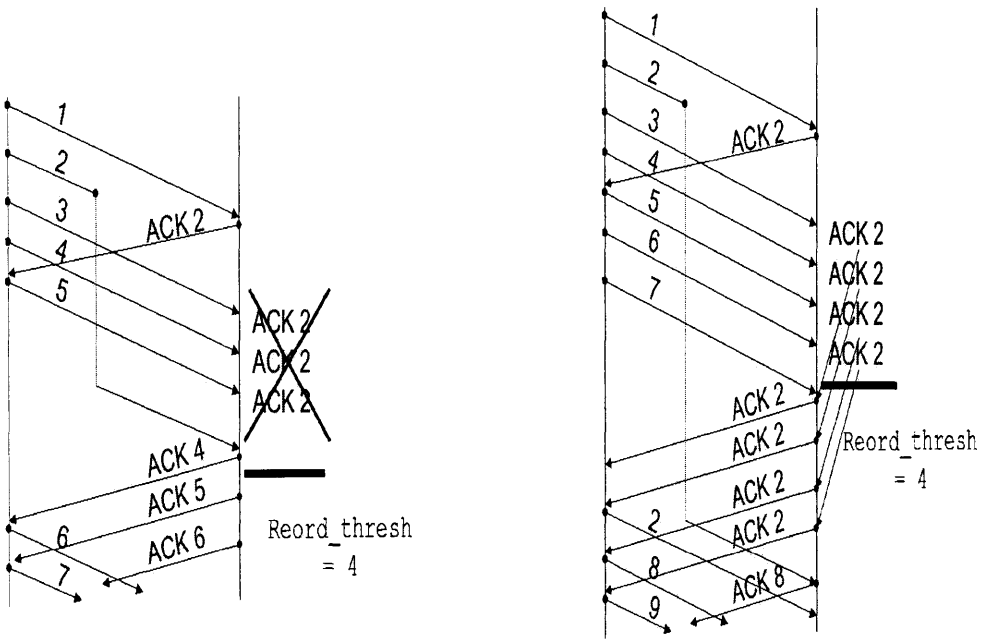


Figure 3.3: *The two scenarios for releasing ACKs:In the first, reordering is resolved before the threshold is crossed and sequential ACKs are sent back. In the second, the threshold is crossed and packet loss is assumed. The withheld dupACKs are sent back to the sender. In both cases, the ACKs being sent are paced out at calculated intervals.*

The ideal behavior in this case would be to pace the acknowledgments out at the same packet interarrival times with which their corresponding data packets arrived at receiver and with acknowledgment numbers corresponding to the exact sequence numbers of the arriving packets. However, the wide variation in the number of reordered packets that can be withheld, coupled with the desire to limit the amount of state used to implement this scheme, dictated that an approximation be used. Rather than record the sequence number of and interarrival time between each reordered packet, the total time interval between the arrival of the first out-of-order packet and the time at which either reordering is resolved or the threshold is exceeded is kept. In addition, in cases where reordering is resolved, the

interval of ACK numbers between the value of the next-expected packet after the reordered packet and the next-expected packet for the flow after reordering is resolved are also recorded. The withheld ACKs are then paced out at regular intervals, with the interpacket-spacing calculated to be the total time interval over which the packets arrived divided by the number of packets to be paced out. In addition, when sequential ACKs are being paced out, the ACK numbers of the acknowledgments being sent out are evenly spaced out over the interval between the sequence number of the next-expected segments before and after reordering is resolved. While not perfect, we believes these approximations are sufficient to maintain the self-clocking behavior of the TCP connection.

Two additional features were instituted as part of the mechanism for pacing out acknowledgments. First, if the delayed acknowledgments [4, 2] are in use, the number of acknowledgments sent back to the sender is set to equal the ceiling of half the number of withheld acknowledgments. Second, to optimize performance, the duplicate acknowledgments corresponding to the first two reordered packet arrivals are sent back to the sender immediately, rather than being withheld. The rationale behind this is that we wish to trigger the fast-retransmission mechanism as quickly as possible is cases where the number of packet arrivals at the receiver exceeds the threshold and packet loss is assumed. Thus, if the threshold is exceeded and the withheld dupACKs are sent back, the fast-retransmission mechanism will be triggered upon reception of the first paced-out dupACK, rather than having to wait for the third. In this case, the number of dupACKs sent back is adjusted to reflect the fact that two dupACKs have already been sent.

In our design, the first two dupACKs are sent back immediately upon reception of the first two reordered data segments, before it is known if reordering is resolved or the threshold is crossed. Thus, the two dupACKs will be sent even in cases where reordering is resolved and sequential ACKs are sent back. In this case, the number of sequential ACKs sent back does not reflect the fact that two dupACKs were sent and equals the original number of ACKs withheld at the receiver (or half that number, in the case of delayed ACKs). The reason this is possible is that the TCP sender, upon reception of the first two dupACKS, performs no actions other than to note the arrival of those dupACKs and then wait for the arrival of either the third dupACK or the resumption of normal data flow. Thus, in the case where reordering is resolved and sequential ACKs are sent back, the transmission of the two dupACKs has no effect on the sender (other than wasted bandwidth) and the full number of sequential ACKs can and should be sent back. Note that if sender has implemented Limited Transmit [1], which causes the sender to send additional segments upon reception of the first two dupACKs, this feature of our design should not be used. However, as most Internet hosts do not yet support Limited Transmit and since the sending of the first two dupACKs optimizes performance for our scheme and is trivial to remove, it remains a part of our current design.

Chapter 4

Implementation

The mechanisms described in the previous chapter were implemented and tested in the TCP stack of version 4.1 of the FreeBSD operating system, which is based on the 4.4 BSD release. Extensions to the existing TCP implementation included the addition of a number of variables to the TCP protocol control block, which is a structure used to keep information about a TCP connection over its lifetime. In addition, modifications were made to both the `tcp_input()` and `tcp_output()` routines, which are called upon reception and transmission of a segment, respectively.

4.1 Additions to Protocol Control Block

A number of variables and structures were added to the TCP protocol control block (PCB) to modify it to handle reordering. Though this increased the amount of per-connection state required for hosts using the modified TCP receiver, it was necessary to add these structures to the PCB due to the fact that out mechanism requires the use of certain elements of state over the entire lifetime of a flow. Thus, the important variables added to the PCB include a counter used added to keep track of the number of dupACKs being withheld, the reordering threshold for the connection, and a circular buffer to hold the reorder-

ing stride values for recent reordered packet arrivals. The value of the threshold variable was set to be the maximum stride value in this circular buffer.

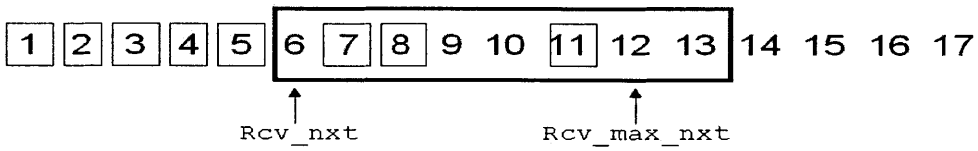


Figure 4.1: *Values of rcv_nxt and rcv_max_nxt in one example of packet reordering. Sequence numbers of packets that the receiver has received are marked with a square.*

An examination of the specific manner in which BSD TCP handles reordered packets revealed a straightforward method to calculate the actual values of the reordering stride for each reordered packet arrival, using existing a single existing PCB variable together with others we introduce. A TCP receiver does not discard any out-of-order segments that it receives while it waits for the next-expected segment to arrive. Instead, these out-of-order segments are buffered in the reordering queue, and a duplicate acknowledgment is sent to the sender upon reception of all packets that are not the next-expected segment. The starting sequence number of the next-expected segment is indicated by the PCB variable `rcv_nxt`, and reordering is resolved when the `rcv_nxt` arrives at the receiver. All consecutively numbered segments in the reassembly queue up to the newly updated value

of rcv_nxt are then passed on to the application, and a non-duplicate ACK is sent to the transmitter to tell it the next expected segment using this new value of rcv_nxt. To calculate the stride, we first institute a new PCB variable, rcv_max_nxt, that indicates the sequence number of the next expected segment after the highest received segment to this point in the same way that rcv_nxt indicates the next-expected segment after all consecutively received segments to this point. The highest-received segment corresponds to the most highly reordered segment for this flow, and the difference between rcv_max_nxt and rcv_nxt provides exactly the stride value we are looking for.

4.2 Modifications to tcp_input()

The majority of the modifications to the FreeBSD TCP segment processing code occurred in the tcp_input() module, which is run upon reception of a TCP segment by a host. An examination of this code reveals the existence of the TCP_REASS macro. The purpose of this macro is to determine if a segment arriving at the receiver is out-of-order. If the segment is out-of-order, it is placed on the reassembly queue and an immediate dupACK is generated. In-order segments cause a delayed ACK to be scheduled and are passed to the application through the socket receive buffer. In addition, their arrivals cause a number of statistics to be updated, including rcv_nxt.

The existence of the TCP_REASS macro in the tcp_input() code was leveraged in the implementation of the modifications. Almost all the necessary code was implemented within the macro, and it also provided pre-existing tests to determine if a packet

was out-of-order or not. No code was modified in the code that handles in-order packet arrivals, which makes up the remainder of the `tcp_input()` module. Thus, since no actions are performed on correctly ordered packets by our code, it is safe to assume that in-order performance of the connection is virtually unaffected.

If the `TCP_REASS` macro deems a packet to be out of order, additional operations are performed. While, conceptually, our scheme involves withholding ACKs at the receiver, in practice this would require to much memory to in the receiver's PCB. Instead, a reordering counter, which measures the number of reordered packet arrivals and, consequently, the number of dupACKs to transmit back, is incremented. Then, checks are performed to see if reordering has been resolved (by checking to see if the value of `rcv_nxt` has changed since the last packet arrival) or the threshold has been exceeded. If either of these actions has occurred, a function named `tcp_reorder_release()` is called that generates and paces out the correct number of sequential or duplicate ACKs, based on the value of the reordering counter. If reordering has not been resolved and the threshold has not been exceeded, no additional actions are performed, as the mechanism is in the process of accumulating dupACKs and the counter has already been updated to reflect the number of withheld ACKs.

The `tcp_reorder_release()` function was implemented in `tcp_input()` and is responsible for the pacing out of ACKs from the receiver upon resolution of reordering or exceeding of the reordering threshold. This task is accomplished in practice by using the BSD timeout queue [7], which allows calls to functions to be executed a specified amount of time in the future. Calls to the `tcp_output()` function, which is

responsible for the generation and transmission of ACKs, are placed on the timeout queue to allow ACKs to be paced out at pre-specified time intervals.

The fact that the timeout queue requires that a pointer to the argument for the function placed in the queue be used (rather than the argument itself) posed a problem, since the argument to `tcp_output()` is the TCP PCB, the values of which constantly change over the lifetime of the connection. As a result, an additional data structure, the TCP PCB snapshot, was created to hold the values of PCB variables like `rcv_nxt` whose values change by the time the call to `tcp_output()` is finally executed off of the timeout queue. Because it is conceivable that additional instances of reordering could occur while the ACKs corresponding to another reordering event are still being paced out, a circular buffer of snapshots was created to hold the state necessary to pace the ACKs corresponding to each event one after the other. In addition, ACKs for other data segments that arrive while withheld ACKs are being paced out are also placed on the timeout queue and the requisite data for those ACKs is placed in the circular buffer of snapshots. This is necessary so as to prevent our receiver from causing ACK reordering, as the ACKs for the arriving data segments will have acknowledgment numbers greater than those of those waiting to be paced out on the timeout queue.

4.3 Modifications to tcp_output()

While less extensive than those made to `tcp_input()`, modifications also were made to the code responsible for the transmission of acknowledgments, the

`tcp_output()` module. The main modifications to this module were twofold. First, the `tcp_output()` function was modified so as to include the correct acknowledgment numbers in the ACKs that are released when reordering is resolved, as `tcp_output()` is the function that is placed on the timeout queue by `tcp_reorder_release()` In the case where the threshold is exceeded, it is sufficient to include the current value of `rcv_nxt` as the duplicate ACK number, since this is the packet that needs to be retransmitted. Since this is the default behavior of TCP, no modification is needed. However, in the case where reordering is resolved, code was added to renumber the ACKs with increasing ACK numbers equally spaced within the interval between the old, pre-resolution value of `rcv_nxt` and the current value. The last ACK corresponds to the segment that resolved the reordering (i.e., the last of the released sequential ACKs should have the value of `rcv_nxt` when reordering was resolved as its ACK number).

Additionally, the ACK flag in an outgoing segment is turned off if a call is made to `tcp_output()` while dupACKs are being accumulated. This will occur only if the receiver has a data segment of its to send during the time that dupACKs are being accumulated. In this case, the ACK flag must be turned off because, by default, all segments are sent with an ACK. If a large number of data segments are sent from the receiver for some reason while reordering is unresolved and the ACK flag is left on, duplicate ACKs would be sent back to the sender in the data segments, triggering the very fast-retransmission that the withholding of dupACKs seeks to avoid.

Chapter 5

Results

The exact effects of reordering on the performance of a TCP connection are heavily dependent on the characteristics of that connection. The bandwidth and delay of the path being used by the connection, as well as the packet sizes used, all play a role in determining the degree of reordering and its resultant effect on TCP. In addition, the exact timing and distribution of the spurious retransmissions caused by packet reordering during the duration of the connection factors heavily in determining the exact level of throughput degradation that the connection experiences. These factors make it difficult to quantify both the exact amount of degradation of throughput that reordering causes in TCP and the effect of our scheme in improving that performance. In his paper describing the Eifel algorithm [18], Ludwig notes this and argues that it is impossible to definitively evaluate the throughput improvement that his scheme provides in the presence of reordering.

However, to examine the effectiveness of our modified TCP in improving transfer throughput in the face of reordering, we chose to test its performance in specific reordering scenarios in an actual network. In particular, a Dummynet [23] was set up between two hosts and used to simulate a multipath network topology that would induce reordering of packets between the two end of a TCP connection. Performance of our modified receiver was compared to that of an unmodified TCP receiver by recording the throughput

of data transfers performed over the network using the Test TCP (TTCP) tool in conjunction with the modified and unmodified receivers. In addition, the numbers of spurious fast-retransmissions that occurred in connections using each of the receivers were compared. This was measured using the number of completely duplicate packets received at the receiver, which is an accurate count of the number of spurious fast-retransmissions when packet duplication is rare, as it is in today's Internet.

5.1 Test Setup

A series of tests were run over a number of networks with various bandwidth-delay products. Different multipath topologies, corresponding to different levels of reordering, were used to produce the necessary levels of packet reordering needed to test the effectiveness of our scheme in the presence of reordering. All links had bandwidths of 1.5Mbps, and a fixed packet size of 500 bytes was used. Topologies using three different sets of delays were used to provide the different pipe-sizes required to test the effects of different degrees of reordering. They included:

1) 50 ms path tests:

a) 50 ms: A single link with a 50 ms delay between sender and receiver, corresponding to normal network conditions with no reordering. The pipe size (bandwidth-delay product) of a 1.5Mbps link with a delay of 50 ms is 9375 bytes. As such, a single link in this topology is capable of holding 18.75 500-byte packets when completely full.

b) 50 ms/37.5 ms: Two parallel paths from sender and receiver, with one path configured to have a delay of 50 ms and the other having a 37.5 ms delay. Packets had an equal probability of being sent over either link to the receiver. The difference in the bandwidth-delay products of the two links corresponds to the maximum reordering stride possible for a TCP connection using this reordering-inducing network. In this case, the difference in the bandwidth delay products was 2704 bytes, or a quarter of the largest possible window size for this connection. This value also corresponds to a maximum possible reordering stride of approximately five 500-byte packets; in other words, the most a single packet could get out of order in the flow was five packets, and degrees of reordering for individual packets in the flow could range from zero to five. Thus, in this topology, the levels of reordering high enough such that, in most instances of reordering, more than enough dupACKs are generated to trigger a spurious fast-retransmission at the sender.

Only a single path with a delay of 50 ms was created between the receiver and sender, so as to not induce ACK reordering and its resulting detrimental effects.

c) 50 ms / 25 ms: Two paths from sender to receiver, one having a delay of 50 ms, the other having a delay of 25 ms, with a single 50 ms return path. The maximum reordering stride for this topology was equal to nine packets, which corresponded to 50% of the largest window size for this connection. Thus, the 50 ms multipath topologies were used to test relatively low levels of reordering, where most cases of reordering generated by the multipath configuration are of degrees such that they will barely produce enough dupACKs to cause a spurious fast-retransmission

(i.e., the degree of reordering will be of levels such that around three dupACKs will be generated in a single instance of reordering).

2) 100 ms path tests:

a) 100 ms single path, providing a non-reordering link between sender and receiver and corresponding to a pipe size of 18,750 bytes (37.5 packets).

b) 100 ms / 75 ms, providing a multipath topology that reordered packets with a maximum reordering stride of nine packets (25% of window).

c) 100 ms / 50 ms, corresponding to a maximum possible reordering stride of nineteen packets (50% of window) for connections over these paths. Thus, these tests correspond to heavier levels of reordering, where degree of most instances of reordering will be high enough to generate more than three dupACKs and trigger a spurious fast-retransmission.

3) 200 ms path tests:

a) 200 ms single path, supplying a non-reordering network and having a pipe size of 37,500 bytes. Thus, 75 packets can fit in the pipe when full.

b) 200 ms / 150 ms, inducing reordering on packets passing over this topology, with a maximum reordering stride of nineteen packets (25% of window).

9) 200 ms / 100 ms, inducing reordering levels with a maximum reordering stride of thirty-seven packets (50% of window). Thus, the 200 ms path delay topologies corresponded to cases of heavy reordering, where specific instances of reordering

will result in high threshold values in the dupACK withholding mechanism at the receiver.

In addition to testing our modified receiver's performance in the presence of packet reordering, we also tested its performance in the face of packet loss. Dummynet was used to induce packet loss rates of 0.5%, 1%, and 5% on the each of the topologies and their resultant levels of reordering.

5.2 Performance

To evaluate the true effectiveness of our scheme, its performance under the three different conditions mentioned above was tested: normal in-order packet delivery, packet reordering, and packet loss. In addition, the throughput of connections using our modified receiver under conditions of combined packet reordering and packet loss was examined to determine the robustness of our scheme under such conditions.

5.2.1 In-order packet delivery

For each of the single link topologies where no packet reordering occurred, the connections using the modified receiver achieved throughputs virtually equal to those of connections using the unmodified TCP receiver. For instance, connections using the modified receiver over the network with a 50-ms path delay achieved an average throughput of 1217.02 kbps (Figure 5.1). This was virtually equal to the average throughput of 1217.04 kbps for the connections using the unmodified TCP receiver over the same topology. Similar performance numbers were seen over each of the other delays. Thus, the modified

receiver met the design goal of minimally impacting TCP under normal, in-order operation and achieving a level of throughput equal to that of an unmodified receiver subject to the same conditions. This is to be expected, as the implementation of our mechanism in the receiver does little to affect on the code that handles in-order packet processing at the receiver.

5.2.2 Packet reordering

In each of the multipath topologies that caused reordering of packets, the detrimental effects of reordering on TCP performance could clearly be seen in connections using the unmodified receiver (Figure 5.1). For the topologies corresponding to the 50 ms delay, throughput dropped by almost 38%, from 1217.04 kbps when no reordering was present. Performance suffered even more for the larger delay topologies, as the combination of larger possible reordering stride resulting in more fast-retransmissions and larger absolute reductions in window sizes when *cwnd* was halved by a spurious fast-retransmission worked together to drive down the throughput of the connection. Most strikingly, for the 200 ms topology, throughput dropped almost 80% in the case where the maximum possible reordering stride was on the order of half of a window size.

In stark contrast, for all delays, connections using the modified TCP receiver achieved throughputs virtually equal to those of normal TCP when no reordering was present. For example, for the 100-ms delay, the connection averaged 1163.77 kbps for the heaviest

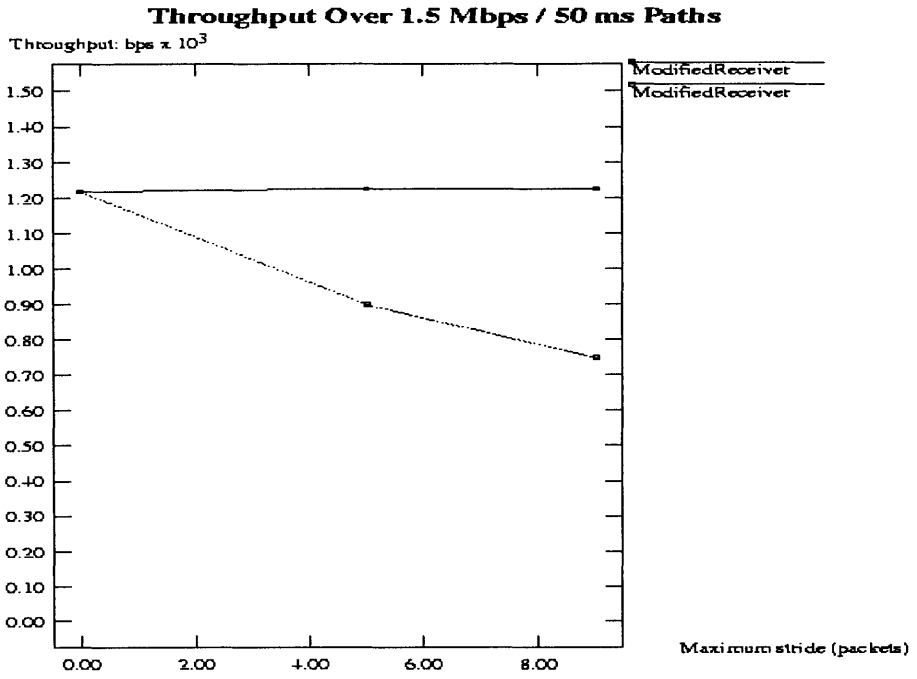


Figure 5.1: *Plots of throughput versus maximum reordering stride for the topologies based on 50-ms path delays. A maximum stride of zero corresponds to the case where no reordering is present. The benefit of our modified receiver can clearly be seen as, in the case of heaviest reordering, throughput increases more than 50% over the unmodified receiver and equals that of the case when no reordering is present.*

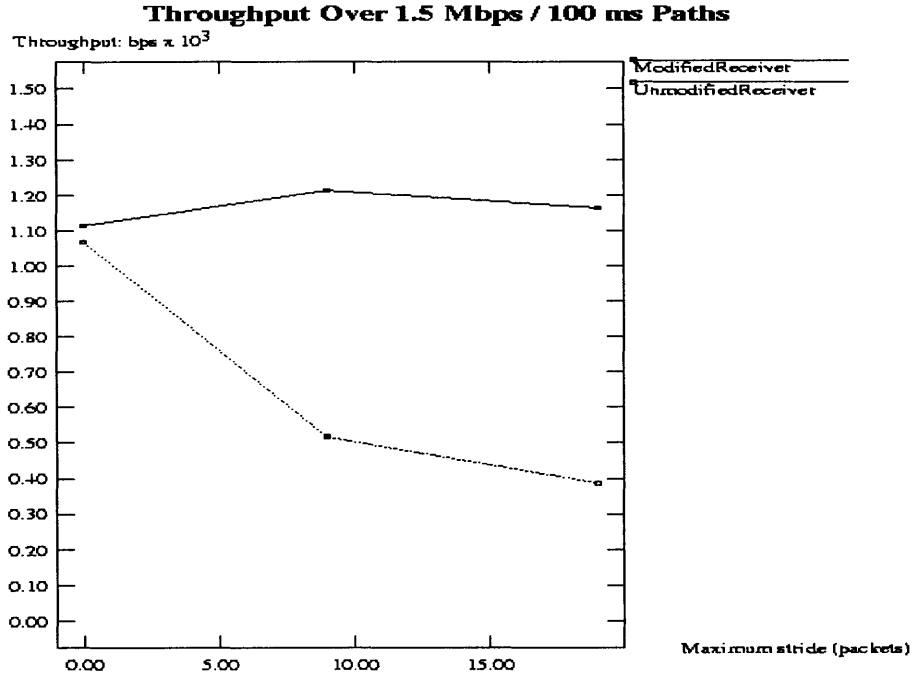


Figure 5.2: *Throughput -vs- maximum reordering stride for 100-ms path delays. Reordering in this topology results in a larger number of spurious fast-retransmissions, and the modified receiver offers almost three times the throughput of the unmodified receiver.*

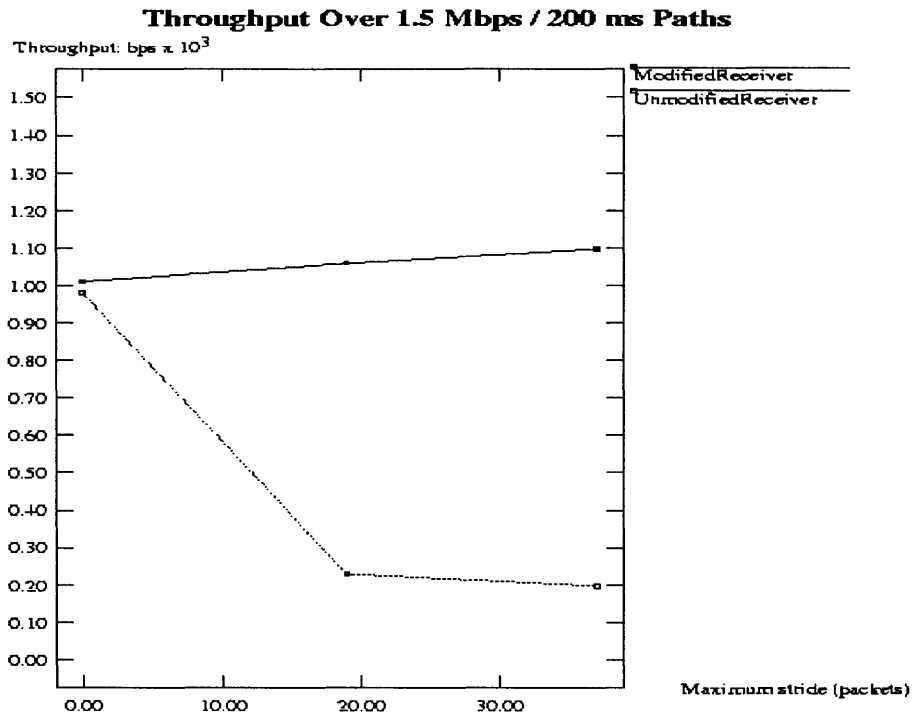


Figure 5.3: *Throughput -vs- maximum reordering stride for 100-ms path delays. The benefits of our modified receiver can most clearly be seen in this case of heavy reordering, as it increases throughput more than five times over that of the unmodified receiver.*

An examination of the number of spurious fast retransmissions brings the effectiveness of the modified receiver into sharper focus. Over the 10,000 packets sent in each test, connections using the unmodified receiver on reordered networks averaged between 2.68 and 4.95 *cwnd*-halving spurious fast-retransmissions per 1000 packets sent (Figure 5.2). In contrast, connections using the modified receiver averaged anywhere between 0.16 and 1.12 spurious fast-retransmissions per 1000 packets (Figure 5.3). This corresponds to reductions of more than 95% in the number of spurious fast retransmissions for the heaviest levels of reordering. Thus, the effectiveness of the modified receiver and its withholding of dupACKs in reducing the number of spurious fast-retransmissions, and the resultant gain in throughput, can clearly be seen in the results of these tests.

Topology	# of Spurious Fast-Retransmissions (per 1000 packets)
50 ms single link	0
50 ms / 37.5 ms	2.68
50ms / 25 ms	4.08
100 ms single link	0
100 ms / 75 ms	2.06
100 ms / 50 ms	3.74
200 ms single link	0
200 ms / 150 ms	3.18
200 ms / 100 ms	4.95

Figure 5.4: *Number of spurious fast-retransmission per 1000 packets for connections using the unmodified TCP receiver*

Topology	# of Spurious Fast-Retransmissions (per 1000 packets)
50 ms single link	0
50 ms / 37.5 ms	0.16
50ms / 25 ms	0.16
100 ms single link	0
100 ms / 75 ms	0.20
100 ms / 50 ms	0.20
200 ms single link	0
200 ms / 150 ms	0.89
200 ms / 100 ms	1.12

Figure 5.5: *Number of spurious fast-retransmissions per 1000 packets for connections using the modified TCP receiver. The use of the modified receiver decreases the number of these throughput-decreasing retransmissions anywhere from 80% to more than 95%.*

5.2.3 Packet loss

In addition to testing our scheme's effectiveness in its primary purpose of improving performance in the face of packet reordering and its effectiveness in cases of in-order delivery, it was also important to verify its robustness in cases of packet loss. This is because the trade-off in our design is that the receiver is given greater time to disambiguate packet reordering from packet loss, but at the cost of increased delay and increased time to identify and react to actual packet loss. Thus, it is important that our scheme both show improved performance in the presence of reordering and, in the presence of packet loss, offer performance comparable to that of a connection using an unmodified TCP receiver. As such, the performance of our modified receiver was tested in the same topologies previously tested, but with packet loss rates of 0.5%, 1%, and 5%. In addition, our original tests can be seen as tests in the presence of no packet loss, as loss was negligible in our Dummynet.

The 0.5% packet loss rate corresponds to a case where the number of fast-retransmissions caused by packet losses is nearly equal to the number of spurious fast-retransmissions caused by reordering in our topologies (ie. 50 fast-retransmissions caused by loss per 1000 packets sent vs. ~26 to ~49 spurious retransmissions per 1000 packets caused by reordering). Both the 1% and 5% loss rates corresponded to cases where retransmissions due to loss greatly outnumbered spurious retransmissions due to reordering.

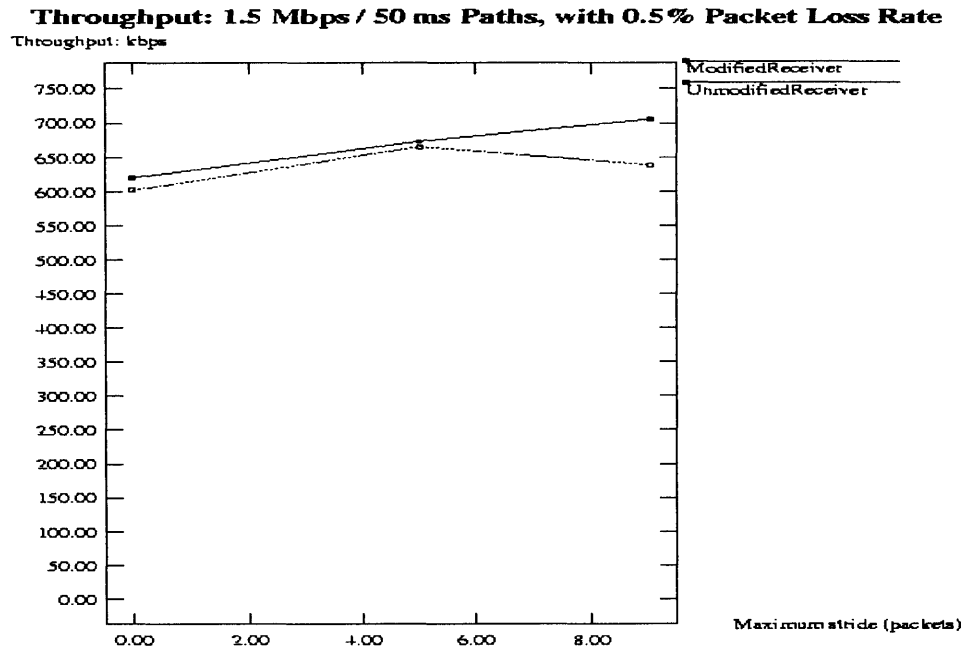


Figure 5.6: *Throughput -vs- maximum reordering stride for connections with 0.5% packet loss rates over 50-ms path delays. The modified receiver also offers increased throughput in this case, where the number of fast-retransmissions due to packet loss is similar to the number of spurious fast-retransmissions.*

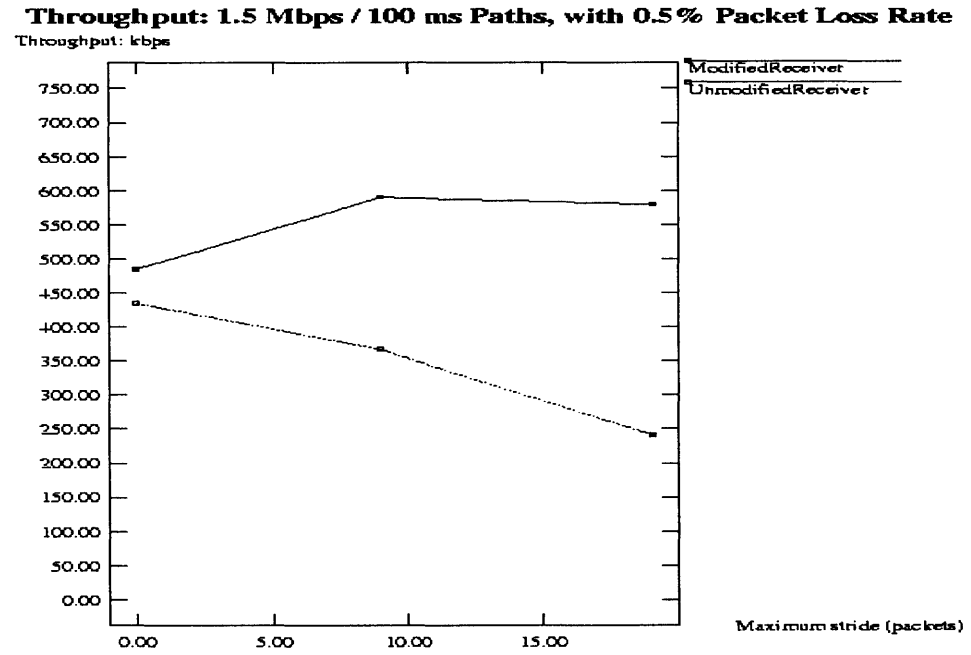


Figure 5.7: *Throughput -vs- maximum reordering stride for connections with 0.5% packet loss rates over 100-ms path delays. Again, the benefit of our scheme is apparent for comparable levels of packet loss and reordering. In this case, throughput increases more than five times in the worst case.*

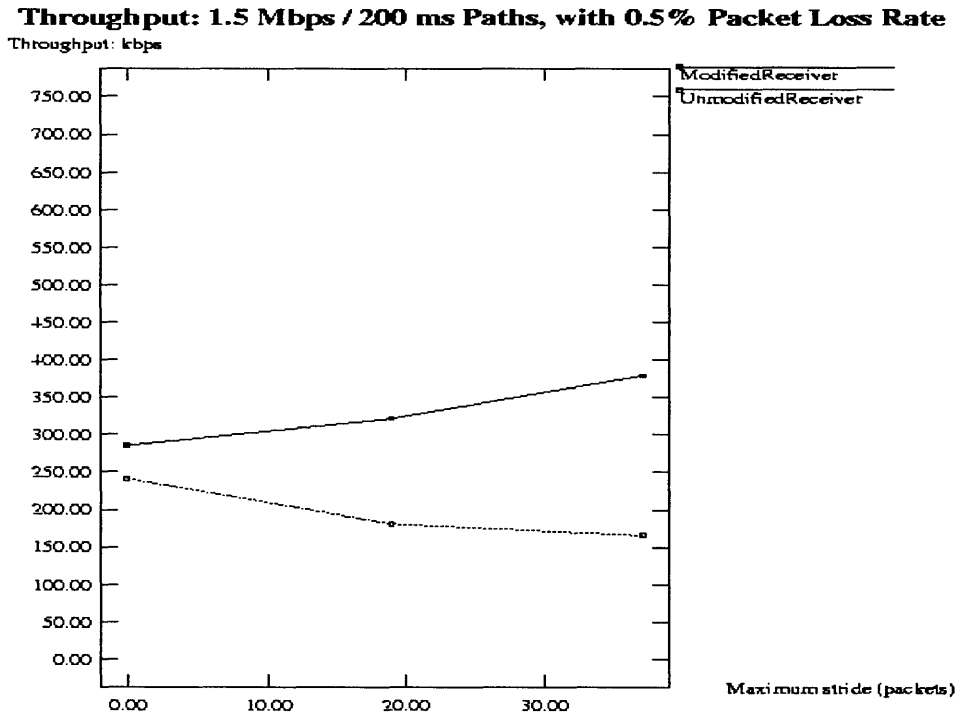


Figure 5.8: *Throughput -vs- maximum reordering stride for connections with 0.5% packet loss rates over 200-ms path delays. Again, the benefit of our scheme is apparent here.*

For topologies experiencing 0.5% packet loss rates (where the number of retransmissions due to packet loss were roughly of the same magnitude as those caused by reordering), our modified receiver produced higher throughputs across all levels of reordering (Fig. 6.6. - 6.8). For example, for the maximum level of reordering, corresponding to a maximum possible reordering stride of thirty-seven 500-byte packets over a 200-ms link, connections using our scheme averaged 379.17 kpbs, while connections using the unmodified receiver over the same topology averaged only 165.52 kbps. In these cases, the performance benefit that our dupACK withholding scheme provides in the face of reordering far outweighs the performance loss caused by the additional withholding delays.

In tests over topologies that induced 1% packet loss rates (and the number of possible retransmissions due to loss outnumbered those due to reordering by a 2-to-1 or 3-to-1

ratio), our scheme had performance poorer than unmodified TCP in almost all cases. This is to be expected, as for each packet actually lost, higher thresholds due to the presence of reordering causes the connection to wait for a longer period of time before identifying actual packet loss and triggering a retransmission. In fact, this is the configuration that would cause our scheme to perform least favorably, when only some reordering is present, but the incidence of reordering is greatly outstripped by the incidence of packet loss. In those cases, for each actual packet loss, the connection will have to wait for an additional time equal to delay caused by the withholding mechanism. In addition, for cases of infrequent reordering, the benefits of preventing spurious retransmissions due to reordering will not be experienced enough by the connection to outweigh these additional delays. However, even for packet losses of this level, our scheme still improved on the performance of unmodified TCP in the case of heaviest reordering. In a 200ms/100ms delay multipath topology where reordering could be on the order of up to 39 packets, our scheme once again outperformed unmodified TCP (Figure 5.11), as the performance benefit derived from preventing spurious fast-retransmissions once again outweighed the drop in throughput due to additional delays.

In the tests for 5% packet loss rates, levels of throughput were uniformly low, across all configurations of delay and reordering and no matter whether the modified or unmodified receiver was used. This is most likely due to the fact that, since 1 in every 20 packets were being lost, windows sizes were driven to consistently low values due to repeated retransmissions and their resultant decrease in the size of *cwnd.*

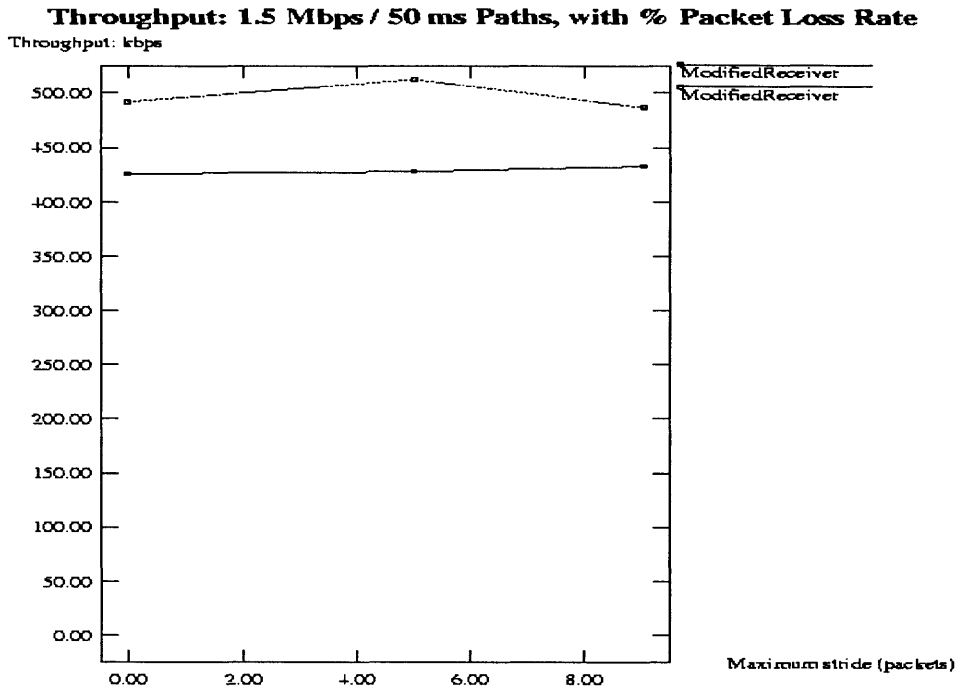


Figure 5.9: *Throughput -vs- maximum reordering stride for connections with 1% packet loss rates over 50-ms path delays. In situations where packet loss far outstrips reordering, our scheme shows a slight degradation in performance, resulting in throughput 10% lower than connections using an unmodified receiver.*

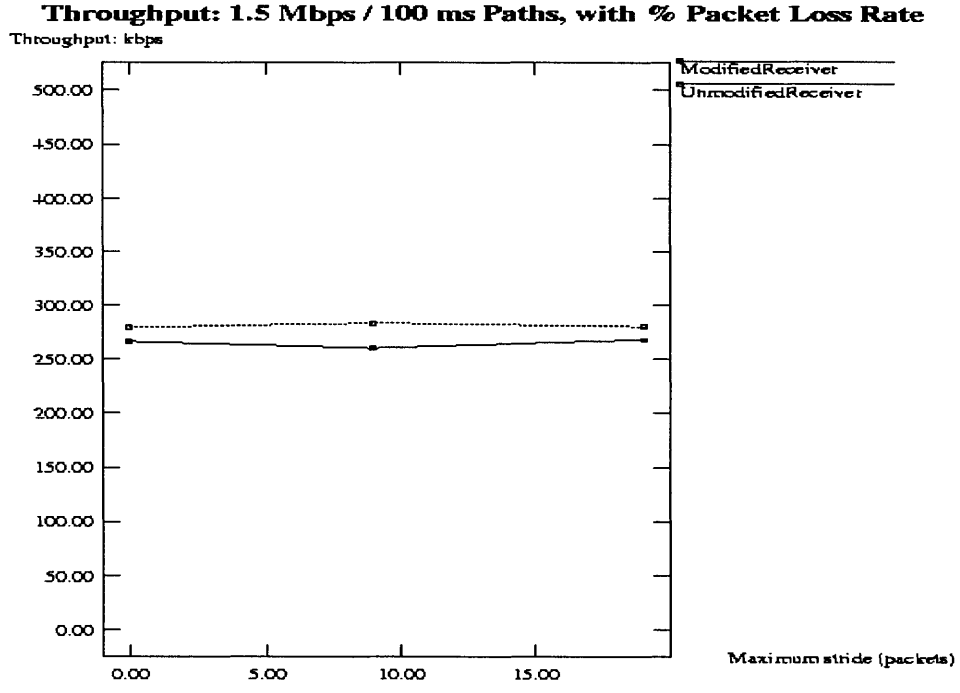


Figure 5.10: *Throughput -vs- maximum stride for connections with 1% packet loss rates over 100-ms paths. Performance again tails that of the unmodified receiver, but by less than 5% in this case, as negating the effects of even low relative levels of reordering results in performance gains.*

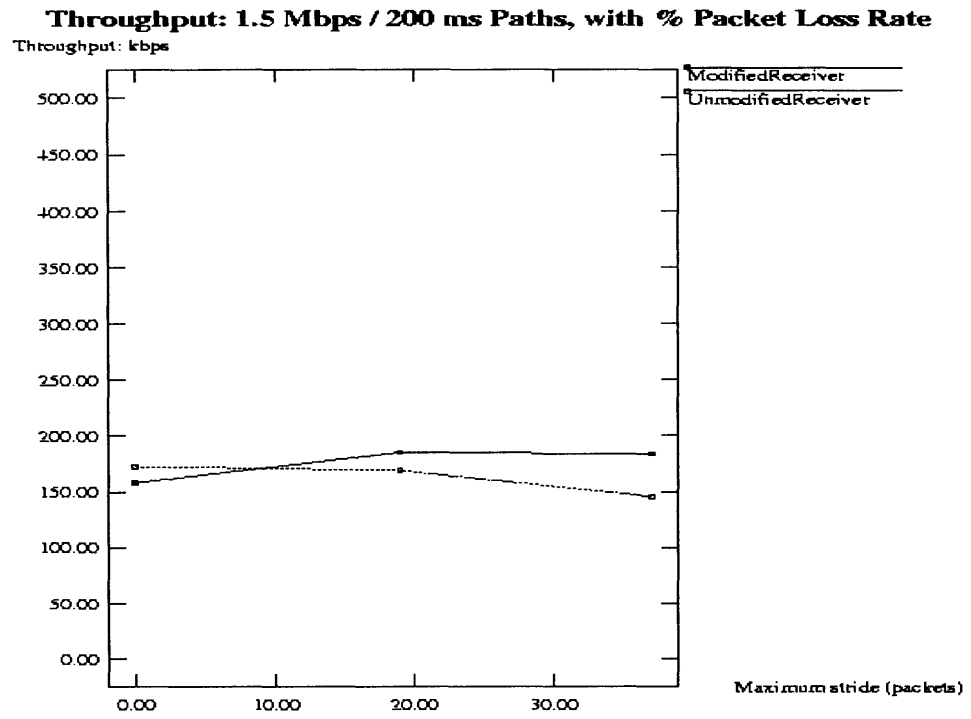


Figure 5.11: *Throughput -vs- maximum reordering stride for connections with 1% packet loss rates over 200-ms path delays In this case, as reordering increases, the performance gain of our scheme in the face of reordering begins to outpace the degradation in performance due to loss as the level of reordering increases and the effects of spurious fast-retransmissions begin to take their toll on the connection.*

Thus, from the results of our experiments, it is possible to conclude that receivers implementing our dupACK withholding scheme offer significantly improved performance in cases where the incidence of reordering greatly outstrips the level of packet loss and in cases where the incidence of both anomalies are equal, while having slightly degraded performance in cases where the incidence of packet loss is much greater than that of reordering. However, one of the strengths of our scheme is that it can be implemented solely in the receiver and at the choosing of the individual user. Thus, users themselves can evaluate the dynamics of the networks they are using and, if reordering is the main pathology they observe in their network, can implement our receiver and benefit from the significant gains in throughput it provides.

Chapter 6

Conclusion

Our scheme for improving TCP's decreased performance in the face of packet reordering is based on a single observation. The degradation in TCP transfer throughput in the presence of packet reordering is a direct result of spurious triggering of the fast-retransmission algorithm at the TCP sender. Thus, the desired goal of our solution was to eliminate these throughput-reducing spurious retransmissions caused by reordering.

The mechanism we implemented for improving TCP's throughput in the presence of packet reordering achieves this goal, as evidenced by the fact that connections using a modified receiver that implements the duplicate ACK withholding mechanism exhibit almost no drop in throughput in the face of large amounts of packet reordering. This can be contrasted with the performance of unmodified TCP, which experiences a significant drop in throughput due to a large number of reordering-induced spurious fast-retransmissions. As such, our experimental results have found that, in cases where extreme levels of reordering exist, our scheme can provide up to five times the throughput when compared to connections using the unmodified receiver. In addition, in cases where both reordering and packet loss exist, our scheme offers more than twice the performance of the unmodified receiver in cases where reordering and loss are equally likely while only showing a degradation in throughput of less than 10% in cases where the incidence of loss far outstrips that of reordering.

In addition to achieving the chief goal of improving the performance of TCP in the face of packet reordering, the implemented solution also adheres to the other design criteria we set out to meet. The dupACK withholding scheme allows the entire mechanism for eliminating fast retransmissions to be implemented purely in the receiver. In addition, the throughput for a TCP connection over a path that causes no packet reordering and which uses our dupACK-withholding modified receiver is equal to that of a connection using an unmodified receiver.

As stated previously, there is a current need to modify TCP to be able to robustly handle packet reordering, both to allow it to adapt to current levels of reordering and also to allow the protocol to be used with networking technologies that induce packet reordering. The receiver-based modifications to TCP discussed in this thesis have been shown to improve TCP's performance in the presence of reordering without affecting its performance under normal conditions. As a result, this solution is a viable option for users who wish to improve the performance of their TCP connections in the face of reordering.

References

[1] Allman, M., Balakrishnan, H., and Floyd, S. Enhancing TCP's Loss Recovery Using Limited Transmit. RFC 3042. January 2001.

[2] Allman, M. On the Generation and Use of TCP Acknowledgments. Computer Communication Review, Vol. 28, No. 5, October 1998.

[3] Bennett, J., Partridge, C., and Shectman, N. Packet Reordering is Not Pathological Network Behavior. IEEE/ACM Transactions on Networking, Vol. 7, No. 6, Dec. 1999. pp. 789-798.

[4] Braden, R. Requirements for Internet Hosts - Communications Layers. RFC 1122. October 1989.

[5] Broch, J., Maltz, D., Johnson, D., Hu, Y., and Jorjeta, J. A Performance Comparison of Multi-Hop Wireless Ad Hoc Network Routing Protocols. In Proc. of the 4th ACM/IEEE Mobicom. October 1998

[6] Cisco Systems. Cisco Express Forwarding Feature Module. http://www.cisco.com/univercd/cc/td/doc/product/software/ios112/ios112p/gsr/cef.htm. Copyright 1998-99.

[7] A. Costello and G. Varghese. Redesigning the BSD Callout and Timer Facilities. Technical Report WUCS-95-23, Washington University in St. Louis, November 1995. http://www.cs.wustl.edu/cs/techreports/1995/wucs-95-23.ps.gz.

[8] Fall, K. and Floyd, S. Simulation-based Comparisons of Tahoe, Reno, and SACK TCP. Computer Communication Review, Vol. 26. No. 3. July 1996. pp. 5-21.

[9] Floyd, S. Questions: SACK TCP Deployment: What Fraction of the Bytes/Packets/TCP-Flows in the Internet are SACK-capable? http://www.aciri.org/floyd/questions.html

[10] Floyd, S. Re: TCP and Out-of-Order Delivery. end2end-Interest mailing list. Feb. 2, 1999.

[11] Floyd, S. TCP and Successive Fast Retransmits. http://www.aciri.org/floyd/papers/fastretrans.ps. May 1995.

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

[13] Gustafsson, E and Karlsson, G. A Literature Survey on Traffic Dispersion. IEEE Network. March/April 1997. pp. 28-36.

[14] Hoe, J. Improving the Start-up Behavior of a Congestion Control Scheme for TCP. In Proceedings of ACM SIGCOMM '96, August 1996.

[15] Hoe, J. Startup Dynamics of TCP's Congestion Control and Avoidance Schemes. MIT Master's of Science Thesis. June 1995.

[16] Jacobson, V. 4BSD TCP Header Prediction. Computer Communication Review, Vol. 20, No. 2, April 1990. pp. 13-15.

[17] Johnson, D. Routing in Ad Hoc Networks of Mobile Hosts. In Proc. of the IEEE Workshop on Mobile Computing Systems and Applications. December 1994.

[18] Ludwig, R. and Katz, R. The Eifel Algorithm: Making TCP Robust Against Spurious Retransmissions. ACM Computer Communications Review, Vol. 30, No. 1. January 2000.

[19] Mathis, M., Mahdavi, J., Floyd, S., and Romanow, A. TCP Selective Acknowledgment Options. RFC 2018. October 1996.

[20] Morris, R., Jannotti, J., Kaashoek, F., Li, J., and DeCouto, D. CarNet: A Scalable Ad- Hoc Wireless Network System. Proceedings of the 9th ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System. September 2000.

[21] Paxson, V. End-to-End Internet Packet Dynamics. IEEE/ACM Transactions on Networking, Vol. 7, No. 3, June 1999. pp. 277-292.

[22] Paxson, V. End-to-End Routing Behavior in the Internet. IEEE/ACM Transactions on Networking, Vol. 5, No. 5, October 1997. pp. 601-615.

[23] Rizzo, L. Dummynet: A Simple Approach to the Evaluation of Network Protocols. ACM Computer Communication Review, Vol. 27, No. 1. January 1997. pp. 31-41.

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

[25] Wright, G. and Stevens, W. TCP/IP Illustrated, Volume 1: The Protocols. Addison-Wesley Publishing Company, 1994.

[26] Wright, G. and Stevens, W. TCP/IP Illustrated, Volume 2: The Implementation. Addison-Wesley Publishing Company, 1995.

[27] Zhang, Y., Paxson, V., and Shenker, S. The Stationarity of Internet Path Properties: Routing, Loss, and Throughput. www.aciri.org/vern/papers/stationarity-May00.ps.gz. May 2000