A Simulation Study of Reordering-Resilient TCP Enhancements

by

Todd Nightingale

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

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2002

© Todd Nightingale, MMII. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.
A Simulation Study of Reordering-Resilient TCP Enhancements

by

Todd Nightingale

Submitted to the Department of Electrical Engineering and Computer Science on February 1, 2003, in partial fulfillment of the requirements for the degree of Masters of Engineering in Computer Science and Engineering

Abstract

TCP traffic makes up a large portion of the Internet's load. The throughput TCP connections are able to obtain depends heavily on the underlying network providing in-order packet delivery. IP networks do not guarantee in-order delivery, but the design of hardware, networks, protocols have been influenced by TCP's in-order requirement. Despite this the Internet today does reorder packets on some links. More importantly, more throughput could be achieved if techniques such as multi-path routing could be used. Unfortunately, the parallelism in these schemes results in packet reordering and a resulting TCP performance loss.

This work examines methods for allowing TCP connections to obtain high throughput in the presence of packet reordering. We review the existing proposals, describe a new, receiver based proposal, and provide a detailed simulation-based evaluation.

In this thesis we present results which show that our modified receiver with an unmodified Reno sender was able to perform as well or better than any of the other proposed solutions. In addition, Eifel is able to consistently out perform DSACK despite using much less packet overhead and internal state.

Thesis Supervisor: Hari Balakrishnan
Title: Associate Professor
Acknowledgments

I would like to express my gratitude to all those who made this thesis possible. Primarily I’d like to thank Hari Balakrishnan for his inspired teaching, academic guidance, and sharing his excitement for the field of computer networks.

Working with Michel Goraczko, Nick Feamster and Jonathan Salz has been a fulfilling experience. I am indebted to them for their technical insight and assistance.

On a personal note I’d like to mention some of the people who have helped me through my time at MIT. Many thanks to Michael Bonnet, Dennis Gregorovic, Richard Hanna, Kevin Schmidt, Daniel Parker, Joshua Merok, Douglas Creager, Brendan Cox, Jonathan Salz, and Jennifer Farver.

Lastly, many thanks to my parents and brothers who have supported all my endeavors, academic and otherwise.
# Contents

1 Introduction
   1.1 Motivation ................................................. 11
   1.2 Packet Reordering on the Internet ...................... 12
      1.2.1 Low Level Reordering .......................... 13
      1.2.2 Ad-hoc Routing ................................. 13
      1.2.3 Multipath Routing ............................... 14
   1.3 TCP and Reordering ...................................... 16
      1.3.1 RFC 793: TCP .................................. 16
      1.3.2 Current TCP .................................. 18
      1.3.3 Self-clocking Characteristics of TCP .......... 21
   1.4 Contributions ........................................ 23

2 Previous Work
   2.1 Eifel .................................................. 26
   2.2 SACK/DSACK ......................................... 27
   2.3 New Eifel ............................................. 28
   2.4 Sender Adaptation ...................................... 30
   2.5 Receiver Adaptation .................................. 31

3 Reordering Resilient Receiver (RRR) .................. 33
   3.1 Overview .............................................. 33
   3.2 Threshold Management ................................. 35
      3.2.1 Recovery Strides ............................... 36
3.2.2 Filtering Stride Measurements ........................................ 37
3.3 Acknowledgment Generation ............................................. 37
  3.3.1 RELEASE.QUEUE ................................................ 38
  3.3.2 Sequential Release ................................................ 38
  3.3.3 Duplicate Release ............................................... 40
3.4 Early Duplicate Release ................................................. 40
3.5 Acknowledgment Smoothing ............................................. 41

4 Simulation ........................................................................... 43
  4.1 Implementation ......................................................... 43
  4.2 Simulations .................................................................. 44
    4.2.1 TCP Configuration ............................................... 45
    4.2.2 Queue Configuration ............................................ 45
  4.3 Microtest ................................................................. 46
  4.4 Multipath Test ........................................................... 47
    4.4.1 Multipath Throughput ........................................... 50
  4.5 Cross-traffic Test ....................................................... 52

5 Conclusion .......................................................................... 55
  5.1 Future Work .............................................................. 55
List of Figures

1-1 Sliding Window ........................................ 14
1-2 Sliding Window ........................................ 17
1-3 TCP (Reno) Example .................................. 21
1-4 TCP Self Clocking Flow Control ....................... 21
2-1 Eifel Example .......................................... 27
2-2 DSACK Example ........................................ 29
2-3 New Eifel Example ..................................... 29
3-1 Receiver Adaptation Diagram .......................... 34
3-2 Reorder Resolution Example ............................ 35
3-3 Complete Recovery ..................................... 39
3-4 Partial Recovery ....................................... 40
3-5 Ack Smoothing ......................................... 41
4-1 Micro-Test Topology ................................... 46
4-2 Micro-Test ............................................... 47
4-3 Multipath Test Topology ................................ 48
4-4 TCP Multipath Test ..................................... 48
4-5 Eifel Multipath Test .................................... 49
4-6 New Eifel Multipath Test ............................... 49
4-7 DSACK Multipath Test .................................. 50
4-8 Receiver Reorder Multipath Test ....................... 50
4-9 Multipath Throughput Comparison ..................... 51
Chapter 1

Introduction

Current TCP sources require in-order packet delivery to maintain high throughput. Congestion control mechanisms mistakenly see reordering as congestion and backoff. This thesis examines existing proposals to modify TCP, describes a new proposal, and analyzes the different techniques through simulation.

1.1 Motivation

On the Internet (and other networks) packets can arrive at the receiver in a different order than they were sent. Studies have shown that packet reordering is prevalent on some Internet paths [19] and perhaps not as infrequent as many have believed [11]. The IP layer does not guarantee, nor does it attempt to provide, in-order delivery of packets. So, any transport layer protocol providing reliable, in-order delivery (such as TCP) must have the capacity to reconstruct packet order.

Many transport protocol mechanisms have been designed using a FIFO queue to model the underlying network layer. This has led to protocols that rely on in-order delivery of packets to achieve good performance. In turn, these protocols have led to the design of network hardware and network topologies that conform to the FIFO model. In many cases, in-order delivery is achieved at the expense of bandwidth utilization.

The design of the Transport Control Protocol (TCP) assumes that packets will
arrive in approximately the same order they were sent. TCP can recover from very limited reordering without a loss in throughput, but if any significant reordering is present on the network, TCP performance dramatically suffers. This is well known behavior [16, 11] and has a huge impact on the development of networks. This work demonstrates TCP’s poor performance in the presence of reordering and explores proposals for TCP modification.

If it were possible for TCP to perform well in the presence of packet reordering on the network, it would be possible to relax the in-order requirement on the network layer. Without the in-order requirement multipath and ad-hoc routing become a much simpler task.

For the most part, today’s Internet utilizes unicast routing. All packets from a source-destination pair are routed through the same path. Routing packets down multiple routes could (and probably would) cause packet reordering since different routes have different latencies. However, if multipath routing were possible, routers could split traffic down multiple paths, effectively summing the available bandwidth. See section 1.2.3 on multipath routing.

To accommodate the growing demand for mobile communication ad hoc network protocols such as Bluetooth have been developed. These networks generate their topologies and routing in an ad hoc manner. Frequent ad hoc routing and topology changes can cause packets to be reordered, influencing TCP performance. See section 1.2.2 on ad hoc networking.

1.2 Packet Reordering on the Internet

The Internet does not guarantee in-order packet delivery and was not designed to do so [5]. Conventional TCP can not differentiate between reordering and packet loss. This leads TCP to see reordering as a sign of congestion and to backoff its sending rate. See section 1.3 for a more complete explanation. This behavior leads TCP to perform poorly in the presence of substantial reordering, and consequently, the network layer has informally taken on the responsibility of providing in-order
delivery. Despite this, some Internet connections experience high levels of reordering [19].

Bennet et al. [11] argue that this is not pathological behavior, but a necessary artifact of parallelism at the link layer and network layer which must be corrected for at a higher layer such as TCP. Placing the responsibility for adapting to packet reordering at the end hosts instead of demanding in-order packet delivery from the underlying network may allow networks to provide data connections with more bandwidth and less latency.

1.2.1 Low Level Reordering

Parallelism between routers at the physical layer has been shown to cause packet reordering. Some routers employ multiple lower bandwidth links between the same source-destination pair to simulate one larger link. This method, referred to as shunt groups, has been shown to cause packet reordering [11] in the presence of variably sized packets. By routing packets from the same flow through different links, head-of-line blocking causes packets to be reordered.

In addition, router cache misses can cause one packet to be delayed for longer than its successors. Under certain conditions, this can cause some reordering at the link layer which will be observed by the TCP receiver.

1.2.2 Ad-hoc Routing

Ad-hoc routing protocols and wireless networking modifications are becoming more prevalent. Ad-hoc routing doesn’t rely on any stationary node; instead, it passes packets through a number of mobile hosts, dynamically routing packets from source to destination. Even small scale host mobility (possibly not even end hosts) can easily lead to routing changes and packet reordering.

In an ad-hoc network peer nodes which can be extremely unreliable do routing for each other. If a node which is on the path from sender to destination goes down, another node will be chosen to take its place. It is possible that the path through
the second node could be faster than the path through the first node, and packets on that path would arrive early.

1.2.3 Multipath Routing

It is possible for the network to accommodate a greater load by using traffic dispersion to better utilize its own resources. Traffic dispersion, using multiple paths to deliver the traffic to the same destination, is made possible by multipath routing.

![Unicast Routing](image1) ![Multipath Routing](image2)

Figure 1-1: Sliding Window

At a multipath router, traffic may be dispersed on the level of flows or packets; the former allows less flexibility in adapting to network conditions, while the latter can dramatically degrade TCP performance. Experiments with flow level and packet level dispersion have been conducted showing them both to be valuable [9].

Many schemes have been proposed to decide which and how many different paths routers should use to disperse traffic to a specific destination. Bahk and Zarki [4] proposed a scheme that reduced the workset of candidate paths to those at most $k$ hops longer than the shortest path or having less congestion than the shortest path. The Dynamic Threshold Multipaths algorithm by Su and de Veciana [21] forwards on only those paths whose "goodness" is within a certain factor of the best path. Work by Vutukury and Garcia-Luna-Aceves [23] presents a modified Bellmand-Ford
algorithm to build loop-free multipath topologies.

**Flow Level Multipath Routing**

Flow level routing preserves flow packet order, but requires additional network state. In a flow level multipath routing scheme, routers always route packets from a TCP flow to the same path. Different flows, however, are forwarded down different paths achieving the multipath effect. In order to achieve this, routers can store information about which path to forward which flows. Alternatively, end hosts can include path information in packets. Both of these solutions add significant complexity to the routing decisions.

Mechanisms for end hosts to encode path information into packets would have to accompany mechanisms for those hosts to learn about changing path information. This additional complexity seems unnecessary.

Keeping per-flow state is impractical for large routers forwarding millions of packets a second. In addition, per-flow dispersion can not always be effective in relieving congestion. When a single flow produces a large packet burst, per-flow multipath routers will be forced to send them all down the same, possibly already congested link. Finer granularity is needed to relieve congestion caused by a single flow.

Large routers often deal with enough flows to allow per-flow multiplexing to be an effective means of distributing load, but the per-flow state needed to maintain consistent per-flow decisions may be prohibitive. Smaller routers can maintain per-flow state, but servicing fewer hosts makes it more likely that flow level granularity will not adequately distribute the network load.

**Packet Level Multipath Routing**

Because per-packet multipath routing would cause packet reordering in the network, little research has been done into its possible deployment. Packet level multipath schemes may scramble packet order, but, with finer granularity, they can provide better load balancing than their flow level counterparts. By routing packets from the same flow down different links, packet bursts from a single source can be serviced
more efficiently, lessening packet loss and latency[22]. Almost no additional network state is needed to implement packet level multipath routing.

If TCP sources were able to perform well in the presence of reordering, this type of dispersion would become a viable option, catalyzing the deployment of lightweight multipath routing protocols without excessive route state..

1.3 TCP and Reordering

Most traffic on the Internet is TCP, so any deployable, network-layer mechanism must allow TCP to perform well. For that reason, the way in which TCP flows react affects the design of routers, queuing protocols, transfer protocols, and network topologies. This has led to packet reordering has lead to routing protocols that ensure in-order delivery of TCP flows and network hardware which minimizes link-layer reordering.

1.3.1 RFC 793: TCP

TCP provides reliable, in-order delivery to clients regardless of network loss rate. It achieves this through cumulative acknowledgments and a transmission window.

TCP receivers are responsible for transmitting an acknowledgment for the next expected byte in the stream each time a packet is received. (This requirement was eventually relaxed with Delayed Acknowledgments [14].) The acknowledgment is cumulative, meaning that it acknowledges complete, correct reception of all data up to, but not including, the byte acknowledged. These acknowledgments allow senders to be sure data was correctly received, and are used to signal to senders which data should be retransmitted.

TCP receivers must also advertise a window. This window is the largest amount of unacknowledged data the receiver will accept. (This value is usually determined by hardware constraints.) This window is used by the sender to control its transmission rate.

Senders are responsible for respecting the receiver's advertised window. If the receiver is advertising a window of $Win_R$, then the sender may only send $Win_R$
unacknowledged bytes into the pipe at any time. If the sender has received an acknowledgment for byte $S_{ack}$, then the highest bit that can be transmitted by the sender is $S_{ack} + Win_R$. The sender also maintains $S_{next}$, the next packet to be sent.

$$S_{next} \leq S_{ack} + Win_R$$  \hspace{1cm} (1.1)

![Diagram of sliding window](image)

**Figure 1-2: Sliding Window**

This example shows a sender with six packets queued and a receiver advertising a four packet window. Four packets are transmitted right away, but before the next can be sent acknowledgments must be received.

This mechanism is called a sliding window. As acknowledgments are received, the window slides forward, always keeping packets in the pipe. The sliding window scheme allows TCP to be self-clocking. It can adapt to network conditions because its sending rate is determined by how long it takes to receive acknowledgments. This is also known as "Bandwidth Limited", and is a property which all future modifications attempt to preserve. See the 1.3.3 section for more detail.

Packet loss is a normal part of network operation. The dropping of packets by a router can indicate congestion on the network path. Some routing protocols such as Random Early Detection (RED) [8] drop packets before their queues fill in the hope of avoiding TCP Synchronization, which could cause under-utilization of an overloaded path. TCP senders must detect packet loss and retransmit data accordingly.

The original TCP specification [20] calls for a TCP sender to retransmit a packet after a time-out interval. This interval is calculated from an exponentially weighted
moving average (EWMA) of the round-trip time. Originally, samples used to calculate this average were taken at most once every round trip time. Additions to TCP, namely the time-stamp option [14] [15], have increased the frequency of these samples dramatically. In this scheme congestion is only signaled by a time-out, so it is fairly resilient to packet reordering.

Unfortunately, it does not recover from packet loss in a timely way. The sender does not register a packet loss and retransmit until a timeout occurs after it sent its last packet. According to the specification the timeout is 1.3 to 2.0 times the exponentially weighted moving average of the recorded round trip time. Using this timeout as the only method of retransmission, connections could remain idle for unacceptable periods of time after a packet loss.

1.3.2 Current TCP

In later revisions of TCP [2], congestion control mechanisms such as slow start, congestion avoidance, Fast Retransmit, and Fast Recovery were adopted. These modifications were designed to improve TCP’s reaction to packet loss in the underlying network. Unfortunately, this performance improvement was at the expense of performance in the presence of reordering.

Fast Retransmit calls for a packet retransmission after three duplicate acknowledgments are received, falling back on the timeout in some cases of multiple loss or reordering. When a TCP sender executes a Fast Retransmit, it also halves the congestion window, effectively halving its sending rate. In networks which reorder packets, the Fast Retransmit behavior unnecessarily slows transmission by halving the congestion window. Waiting for three duplicate acknowledgments is TCP’s attempt to resolve reordering. When a packet arrives out of order by three or fewer packets a TCP running Fast Retransmit will recover seamlessly.

Current TCP senders maintain an congestion window (\(Win_c\)) and a slow start threshold (\(ss\_thresh\)). They operate in slow start mode while the \(Win_c\) is less than \(ss\_thresh\), and in congestion avoidance mode otherwise. During slow start, \(Win_c\) is increased by a packet length for every received acknowledgment. During congestion
avoidance $Win_c$ is increased by a packet length divided by $Win_c$ for every received acknowledgment. (This approximates one packet length per round trip.)

*Fast Recovery* allows TCP sources to keep the pipe full after detecting a loss. Each duplicate acknowledgment that arrives, this could be as many as a full window, will expand the window by one packet size. This allows TCP to keep the pipe full and maintain its self-clocking pattern of sending packets in response to acknowledgments.

**Tahoe TCP**

Tahoe TCP is an implementation of these congestion control protocols. It always uses the maximum of the $Win_c$ and $Win_r$ values as its sending window. Tahoe TCP can quickly recover from many packet losses in a window, but it may retransmit packets which are already buffered at the receiver. When a loss is detected:

$$ ss\_thresh = \frac{Win_c}{2} $$

$$ Win_c = 1mss(maximum\_segmentsize) $$

$$ S_{next} = S_{ack} $$

$$ Win_{send} = max(Win_c, Win_r) $$

Tahoe forces the source to recenter slow start with a window of 1 at the packet which must be retransmitted. Congestion avoidance will commence once the source’s window is half of what its value was at the time the loss was detected. A loss of a solitary packet will lead to retransmission, a round trip pause and then slow start from the old $S_{next}$.

**Reno TCP**

Tahoe’s round trip pause causes the pipe to completely empty out before the transmission will continue. Reno was designed to prevent this pathology. Reno stores $DupAck$, the number of duplicate acknowledgments received. The *Fast Recovery* algorithm was added to enable the source to keep the pipe full while recovering from
the loss. When Fast Retransmit detects a loss (three duplicate acknowledgments), $S_{ack}$ is retransmitted, but $S_{next}$ is not modified. From then on, each received acknowledgment inflates the window by one packet. This can be seen in the $Win_{send}$ calculation.

$$ss_{thresh} = \frac{Win_c}{2}$$

$$Win_c = ss_{thresh}$$

$$Win_{send} = \max(Win_c + DupAck, Win_r)$$

If a solitary packet is lost, Reno TCP will resend it and continue in congestion control mode. $Win_c - 1$ duplicate acknowledgments will eventually arrive, increasing the window to $Win_c/2 + Win_c - 1$. Of course, $Win_c - 1$ packets have left the pipe (and generated those acknowledgments), so $Win_c/2$ packets are put into the pipe, specifically, packets $S_{ack} + Win_c$ to $S_{ack} + Win_c/2 + Win_c - 1$. This is approximately one half of the old sending rate, and is what congestion control permits. When the recovery acknowledgment arrives, $DupAck = 0$, $Win_{send} = Win_c/2$ and $S_{next} = S_{ack} + Win_c + Win_c/2$, which is the very next unsent packet.

TCP senders trigger the Fast Retransmit algorithm after receiving three packets acknowledging the same segment (four identical acknowledgments total). This allows some reordering to be resolved seamlessly without a spurious retransmission and detrimental transmission rate decrease. The current algorithm keeps this threshold constant, instead of controlling it based on the traffic patterns of the connection. After three duplicate acknowledgments, the transmission rate is halved regardless of past reordering on the connection. Allowing this threshold to react to observed traffic could dramatically improve throughput on reordering links. Some modification to TCP to allow this threshold to adapt or another threshold to be created is clearly needed before connections over reordering paths will function efficiently.
1.3.3 Self-clocking Characteristics of TCP

In figure 1.3.3, the sender sent a window of packets through its pipe. As the packets travel from the high bandwidth link to the low bandwidth link, they spread out in time.

The top pipe represents the forward path with three segments: a high bandwidth pipe near the sender, a low bandwidth pipe, and a high bandwidth pipe near the receiver. The bottom pipe represents a similar, reverse pipe. The vertical dimension represents bandwidth and the horizontal represents time.

\footnote{Figure 1.3.3 is similar to figure 1 in [12].}
The packets are shaded gray. The area of packets in the figure represents bytes of data. As packets enter different pipes with different bandwidths their area (and length in bytes) stays constant. A packet going from a high bandwidth link (long vertically) to a low bandwidth link (short vertically) must spread out in time (horizontally).

We define the time it takes a packet to enter the link the packet time ($Pt$). The time it takes a packet to enter the bottleneck link is $Pt_{bb}$. These packets exit the bottleneck and enter the high bandwidth link near the receiver still spaced out. At the receiver the packets arrive $Pt_{bb}$ apart.

\[
Pt = \frac{\text{packetsize}}{\text{bandwidth}}
\]

\[
Pt_{bb} = \frac{\text{packetsize}}{BB}
\]

The packet time spacing is preserved by the receiver in its acknowledgments. Acknowledgments arriving at the sender arrive spaced out according to the lowest bandwidth link, $Pt_{bb}$ apart. If the sender’s window is large enough and packets are only sent in response to acknowledgments, then the sender’s packet spacing would exactly match the packet time on the bottleneck link. The sender’s transmission rate would be equal to the bottleneck bandwidth ($BB$).

The optimum value for the sender’s window is the round trip time ($rtt$) times $bb$. Packets arriving at the sender are always going to arrive at $Pt_{bb}$ intervals. If the sender always has $BB \times rtt$ bytes in the pipe, no queuing is required, but the bottleneck link will never be idle.

Determining the optimum window value is non-trivial for sources that don’t know the path’s parameters or the constantly changing cross traffic. TCP’s congestion control mechanisms probe for bandwidth by expanding the congestion window.

This acknowledgment clocking scheme forces TCP to pace out its transmissions, preventing packet bursts. Burstiness can increase packet loss and hurt performance [3].

This self-clocking behavior depends on the uninterrupted flow of acknowledgments.
Packet reordering causes a disturbance in the acknowledgment flow. Reordering could cause the receiver to generate many duplicate acknowledgments followed by a single recovery acknowledgment acknowledging many packets. This would cause TCP’s window to jump allowing it to transmit a large packet burst.

1.4 Contributions

TCP performs poorly in the presence of packet reordering. A reordering-resilient TCP connection is needed to effectively handle reordering already present on the Internet and provide an environment in which routing protocols are not bound by in-order delivery constraints.

The two main contributions of this thesis are a modification to TCP which allows the receiver to maintain high throughput in the presence of reordering and a study of the comparative performance of this and other proposals.

In chapter 3, we present a modified TCP receiver. A TCP connection with our receiver is able to perform in the presence of packet reordering without substantial losses in throughput. It is able to adapt to changing reordering in the network and prevent unnecessary performance loss. Without packet reordering it performs virtually identically to an unmodified TCP.

Our solution, unlike others, requires modification only to the receiver, and requires no additional TCP option (nor bandwidth overhead). We hope this will ease incremental deployability.

In chapter 4, we compare standard TCP, DSACK [6], Eifel [16], our own New Eifel presented in chapter 2 and our reordering resilient receiver. We obtain comparative results from simulations run on the Network Simulator [10].
Chapter 2

Previous Work

In recent years, many modifications to the core TCP protocol have been proposed to alleviate the problems posed by packet reordering. Many of these solutions require a TCP option, negotiated between sender and receiver. The systems then use the added functionality to improve performance. Most of the previous work involves two stages: detecting a spurious retransmission, and reacting to it (possibly by “undoing” congestion control actions).

We believe that modifications to TCP to handle packet reordering should strive toward the following goals (in order of importance):

1. Maintain High Throughput
   The presence of packet reordering should not substantially lower the throughput of TCP flows. Packet reordering can be mistaken for congestion by sources causing needless backoffs that lead to underutilization.

2. Prevent Spurious Retransmission
   New protocols must prevent spurious retransmissions by the sender. Spurious retransmission injects useless packets into the network, causing the load to be higher than the demand. This can lead to wasted bandwidth and contribute to network collapse.

3. Prevent Packet Bursts
   Packet bursts can cause congestion effecting all other sources on the path.
Bursts also increase the chance of packet loss. In standard TCP, bursts are caused by packets acknowledging large byte ranges (more than one packet).

4. **Fast Network Adaptation**

   New algorithms must adapt to changing network conditions rapidly. The nature of reordering on Internet paths is dynamic, so sources should be able to adapt to changes in conditions.

5. **Minimize Impact to TCP**

   Any modification to TCP should maintain the self-clocking, bandwidth-limited nature of TCP, causing little or no change to TCP’s behavior on a FIFO network. Modifications should minimize the impact to TCP’s latency and throughput for in-order packet delivery.

6. **Minimal Additional TCP State**

   In order to allow systems to simultaneously maintain many TCP flows, additional per-flow state and intensive computation must be kept to a minimum.

The following are existing proposals for modifications and additions to TCP which would improve performance in the presence of packet reordering.

**2.1 Eifel**

The Eifel algorithm [16] was proposed January 2000. It uses the TCP time-stamp option [14] to detect spurious retransmissions. At the initiation of the TCP connection, the time-stamp option must be negotiated in the three-way handshake. The time-stamp contains enough information to disambiguate between spurious retransmissions and an actual packet loss.

The first time a source retransmits any segment, it stores its time-stamp as $\text{ts\_first\_remit}$. It also stores the values of the congestion window and the slow start threshold. Upon receiving an acknowledgment for a segment that was retransmitted, the source compares the acknowledgment’s time-stamp with $\text{ts\_first\_remit}$. If
*ts_first_rexmit* is greater than the received time-stamp, it can be assumed that the received packet is acknowledging the original transmission and the retransmission was spurious. If the time-stamp is properly echoed, and no acknowledgments are lost, every spurious retransmission can be detected.

Upon detection of a spurious packet retransmission, the slow start threshold and the congestion window are restored to their value before the retransmission. This effectively undoes the congestion control actions taken during fast-retransmission. Unfortunately, this will cause an unwanted packet burst since the congestion window will instantaneously increase by many packet lengths (it could as much as double). For this reason, it is suggested that a burst limiter be added to the TCP as well. The Eifel designers hope that by returning to the state before congestion was mistakenly detected, the sender will be able to perform better in the presence of reordering.

Eifel is designed to keep throughput high in the face of reordering, but it does not attempt to prevent spurious retransmissions or packet bursts.

### 2.2 SACK/DSACK

The Selective Acknowledgment (SACK) [17] option can give the sender information about what out-of-order packets are arriving at the receiver. Bennet et al. [11] suggest that this information can be used to determine if reordering is taking place. That work goes on to suggest that DSACK, a further addition to TCP, could be used to
detect spurious retransmissions.

An addition to the SACK protocol, duplicate-SACK (DSACK) has been designed to inform the sender of duplicate segments arriving at the receiver [6]. This scheme uses these duplicate segments to detect spurious retransmission. Assuming no acknowledgments are lost, every spurious retransmission will eventually be detected.

In addition to standard SACK options, a DSACK enabled receiver will send back a SACK option for duplicate packets which it receives. If a sender receives a packet with a SACK option for a packet less than the acknowledgment sequence number, it can deduce that a duplicate segment has been received. Assuming that no packet duplication occurred on the Internet, a valid assumption in many cases, this means that that packet was spursiously retransmitted.

Instead of directly resetting the congestion window, as in Eifel, DSACK sets the slow start threshold to its old congestion window. This will allow the sender to return to its old congestion state in slow start. Blanton et al. [18] use this approach with DSACK and find that is very effective, but it does not limit the number of retransmissions and is more aggressive than a Reno TCP source. Of course, packet duplication could cause this protocol to behave strangely.

Figure 2.2 shows this scheme recovering from a spurious retransmit. There is reordering at $t = 6 \text{sec}$ which is interpreted as congestion at $t = 6.5 \text{sec}$. Finally, at $t = 6.7 \text{sec}$, a DSACK option is received at the sender and the spurious retransmission is detected. Slow start handles recovery and the reordered stream resumes just slightly behind the FIFO stream.

2.3 New Eifel

Eifel and Stuck both use a congestion state recovery mechanism to allow senders to revert to a high sending rate if it is determined that a retransmission and backoff were spurious. Since DSACK relies on the receiver to acknowledge the duplicate reception of the retransmitted packet, the soonest time at which it can recover is one round-trip time after the retransmission. That retransmission was either triggered by a timeout
or a fast-retransmission. Either way, a full round trip time is lost in addition to the time it took to trigger the retransmission.

Eifel is able to recover as soon as it receives the first packet acknowledging the original transmission. Not having to wait for the second transmission and acknowledgment (DSACK) saves one round-trip time. Because of this, we saw it as valuable to combine the slow start recovery mechanism outlined in [6] with the Eifel detection scheme [16] into New Eifel.

New Eifel uses the standard Eifel approach to detect spurious retransmissions (time-stamp comparison). However, instead of directly resetting the congestion window and slow start threshold, New Eifel borrows a technique from Floyd, et. al.[6] and sets the slow start threshold (ss_thesh) to the old congestion window. This will allow the source to recover to its old congestion state using TCP’s slow-start
algorithm. The recovery is slightly slower than the pure Eifel approach, but it limits the unwanted sender packet burst. Using the existing TCP algorithms to re-open the window may allow easier integration into current systems.

Figure 2.3 shows the congestion window of a TCP connection over time. A standard TCP connection, an Eifel TCP connection and a New Eifel TCP connection are shown. There is a reordering event at 4.5 seconds. The standard TCP connection cannot disambiguate the reordering from a loss, assumes congestion, and halves its window from 26 to 13. The Eifel connection is able to recover after receiving an acknowledgment with a time-stamp which precedes its ts_first_retransmit and it returns its congestion window to 26. Upon sensing a spurious retransmission New Eifel opens ss.thresh to 26 allowing the source to use slow start to recover to the old window size.

2.4 Sender Adaptation

The TCP specification [2] defines a duplicate acknowledgment threshold of three. Should a sender receive three duplicate acknowledgments (four total acknowledgments for the same sequence number), it fast-retransmits the packet. It is this value which allows TCP to withstand some reordering, but being a constant, it does not adapt to network conditions. If a connection never experiences any packet reordering, waiting for three duplicate acknowledgments uselessly adds latency. If a connection's packets are often reordered by a substantial amount, spurious fast retransmits will cause the connection to underutilize its bandwidth.

There has been some work done on adapting the duplicate acknowledgment threshold in order to prevent spurious retransmission while managing latency. Blanton et al. [18] compare techniques to manage DUP.THRESH. They propose a number of schemes for managing this value.

One scheme explored is constant increase/decrease scheme in which DUP.THRESH is increased by a constant value when a spurious retransmission is sensed and reset to three when a timeout is experienced. Unneeded retransmission in their scheme
would be detected using DSACK, but it could just as easily be done using Eifel. The detected retransmission indicates the threshold is too low, while timeouts could indicate the threshold is too high. This scheme allows the threshold to adapt using these signals as feedback.

Another scheme presented is to calculate the `DUP_THRESH` as an exponentially weighted moving average (EWMA) of the size of detected reordering events. Their findings indicated that this was not an optimal scheme. This was an expected result; since they were setting the upper limit with an average, it would always be too low.

They also compare adding a timeout to the standard three acknowledgment limit. In this method, the timeout must expire before the three duplicate acknowledgments to trigger a retransmission. The length of the timeout is set as “the amount of time that would have been required to obtain enough duplicate ACKs to disambiguate reordering from loss in previously experienced reordering events.” This could be done using a circular buffer of timeout values and DSACK. Of course this method requires substantially more overhead than the others.

### 2.5 Receiver Adaptation

All of the previously proposed solutions require that modifications or TCP options be present on both sides of the connections. SACK and time-stamp are both TCP options which must by negotiated in TCP’s SYN handshake. DSACK runs on top of SACK and in order to be effective both ends of the connection must support it. We believe a system which could operate on only one side of the connection (allowing any one system to increase its performance without requiring foreign TCP modifications) would be much more useful on the Internet today since many nodes do not or will not support these TCP modifications.

A proposal by Peter Yang [24] outlines a system for controlling the threshold at the receiver. The receiver withholds duplicate acknowledgments until a variable threshold is met. After the threshold is exceeded, the receiver releases the acknowledgments to the sender. If reordering is resolved before the threshold is met, the withheld duplicate
acknowledgments are replaced with sequential ones, emulating in-order reception. The threshold is defined to be the maximum amount of reordering the flow has seen recently. Releasing the acknowledgments in a burst would disturb the self-clocking nature of the acknowledgment stream. Instead, released acknowledgments are paced out to simulate the timing with which they arrived.

We expand this proposal into a packet reordering protocol which only requires modifications at the receiver, allowing senders to be unaware of any change. See chapter 3.
Chapter 3

Reordering Resilient Receiver (RRR)

3.1 Overview

Here we propose our new reordering resilient TCP. It is designed as a modified receiver capable of working with any standard sender. It is our hope that requiring only modifications to the receiver will ease deployment. We take advantage of the fact that in standard TCP connections, the receiver has sufficient information to detect reordering and can influence the sender by modifying the acknowledgment stream. Our solution can provide an effective option for users who require high performance in the face of underlying reordering, but do not have access to the sender’s TCP configuration. In addition, this is the only protocol presented which does not rely on negotiating a TCP option (time-stamp, SACK, DSACK, etc.).

As out-of-order packets arrive at our receiver, it will temporarily withhold the duplicate acknowledgments in the hope that the reordering will be resolved. Should reordering resolve itself, the withheld duplicate acknowledgments will be replaced with sequential ones and released in order to emulate in-order reception. After detecting a loss, the receiver will release its withheld duplicate acknowledgments.

Unmodified TCP sources are “self-clocked” by the incoming acknowledgment stream. Each regular in-order acknowledgment received will slide TCP’s window
(and possibly expand the window) triggering the next data to be sent. See section 1.3.3 for more detail. In order to preserve the self-clocking nature of TCP, the packet releases (duplicate and sequential) will be spaced out in an attempt to mimic regular acknowledgment generation. Figure 3.1 shows a successful reordering recovery.

The following state variables were added to the TCP receiver in order to execute our algorithm:

- **WITHHELD ACKS** The number of duplicate acknowledgments currently being withheld.

- **STRIDES** A circular buffer of stride values used to determine the duplicate
acknowledgment threshold.

- **RELEASE_QUEUE** A list of acknowledgment number, timeout time pairs representing the acknowledgment to be spaced out, and when.

- **LAST_ARRIVAL** Time the last packet with new data arrived.

- **REORDER_TIME** An array of the times at which packets first failed to arrive in order.

- **IGNORE** The last duplicate release sequence number.

### 3.2 Threshold Management

Standard TCP senders detect a loss when **DUP_ACK** duplicate acknowledgments arrive at the sender (typically **DUP_ACK** is set to three). Our proposal will effectively modify this value, but instead of increasing the **DUP_ACK** threshold at the sender, we will withhold the acknowledgments which would exceed the threshold at the receiver. The receiver takes on the responsibility to set its own threshold and release stored duplicate acknowledgments when it detects a loss.
In order to avoid congestion detection at the sender, we withhold duplicate acknowledgments at the receiver. In a lossless environment withholding enough duplicate acknowledgments would eventually resolve reordering. In actuality, we must be able to resolve reordering without substantially interfering with loss detection. In order to do this, our receiver withholds duplicate acknowledgments up to a threshold and then, assuming we observed a loss and not reordering, it releases the duplicates.

3.2.1 Recovery Strides

We propose a variable threshold designed to estimate the largest amount of reordering the connection has experienced in recent history. We define the reordering "stride" to be the difference between the last in-order packet sequence number and the highest received packet sequence number. (So a connection experiencing regular, lossless, in-order reception always has a stride of zero.) We believe this value is a good quantifiable estimate of the degree of reordering the connection is experiencing at any given time (assuming there has not been a loss). The connection in figure 3.1 is experiencing reordering with a stride of 4.

We choose to record strides only when a recovery packet is received. In this way, the largest stride value is recorded for each recovery event. This keeps the size of the STRIDES buffer small while maintaining a good estimate of the nature of reordering on the network. Whenever reordering is resolved (the next expected packet arrives) the stride (before reception) is stored in the STRIDES circular buffer.

We define the threshold for the number of acknowledgments which can be withheld at the receiver as the maximum stride value in STRIDES. This way, the threshold will represent the largest amount of reordering the receiver has observed recently and how much time the history represents can be controlled by varying the size of the STRIDES buffer. In order to allow a connection to react to a network which has stopped reordering packets, we also push a zero into the STRIDES buffer for every OBSERVE_NO_INORDER packets which arrive in order. In our tests \( \text{size(STRIDES)} = 10 \) and \( \text{OBSERVE\_NO\_REORDER} = 1000 \).
3.2.2 Filtering Stride Measurements

Not every recovery packet received is a good measure of reordering. Recovery packets can be retransmissions from either the Fast Retransmit algorithm or a sender timeout. If a retransmission is responsible for the recovery packet it is quite likely that the stride will be the entire window of packets.

In order to disambiguate a reordered packet from a Fast Recovery retransmission, we use the IGNORE value. Each time a duplicate release is executed, IGNORE is set to the acknowledgment number for that release. When a recovery packet is received its stride is stored in STRIDES only if its sequence number is higher than IGNORE. Care will have to be taken in actual TCP implementation to insure that sequence number wrapping does not cause a problem.

Timeouts can also cause the sender to send a retransmission. The TCP standard [1] sets one second as the minimum for a retransmission timeout regardless of connection characteristics. Since there must be one second before a packet can be retransmitted, we conservatively set our timeout to one half of one second. If a recovery packet arrives more than half a second after its sequentially previous packet, the recovery stride is ignored.

3.3 Acknowledgment Generation

Our receiver generates an acknowledgment whenever an in-order packet arrives, as an unmodified receiver would (ignoring Delayed Acknowledgments). If a non-duplicate out-of-order packet is received the duplicate acknowledgment which would be generated is withheld. Instead, WITHELD_ACKS is incremented.

Should WITHELD_ACKS ever exceed \( \max(\text{STRIDES}) \), WITHELD_ACKS are generated with duplicate acknowledgment numbers and released. If reordering is resolved before \( \max(\text{STRIDES}) \) is exceeded then sequential acknowledgments are generated to emulate in-order delivery and released. Should high strides value be recorded on the connection, it is likely that many packets will be released at once.
3.3.1 RELEASE_QUEUE

Instantaneously releasing many acknowledgments could have negative effects on the throughput of the connection as well as other connections. If the sender receives many sequential acknowledgments in a very short period it would cause its window to slide very quickly resulting in a packet burst. Packet bursts increase loss rate and hurt performance [3]. A sender using the Fast Recovery mechanism would send a similar packet burst in response to a large duplicate packet release.

To keep this from happening, packets which are released will be placed into the RELEASE_QUEUE with a timeout value chosen to space out their release, emulating in-order delivery. If the queue is empty, the first acknowledgment is sent immediately. A packet is released from the RELEASE_QUEUE whenever a packet is received. Each time a packet is sent from the queue a timer is started; should the timer reach the timeout of the first packet in the RELEASE_QUEUE before the next arrival, that packet is released anyway.

3.3.2 Sequential Release

In a sequential release, it is important to choose a timeout that estimates the inter-arrival time an in-order connection would have experienced. In order to do this we maintain an array of time values indicating when a packet failed to arrive in order: REORDER_TIME. When a packet arrives out-of-order and it is the highest packet so far, REORDER_TIME(x) is set to now for $prev\_highest < x < new\_highest$. In this way, REORDER_TIME(x) stores the first time that packet $x$ failed to arrive in-order, or an out-of-order packet arrived ahead of it.

Upon a sequential release the timeout for packets placed in the RELEASE_QUEUE is set to the interval between the REORDER_TIME of the recovery packet and the REORDER_TIME of the highest in-order packet, divided by the number of packets in that range. This estimates the inter-arrival time to be equal in that range and conservatively averages over the largest packet range which is fully received.
\[ t_{seq} = \frac{t_{reorder}(next - 1) - t_{reorder}(recovery)}{next - recovery} \] (3.1)

Figure 3-3 shows a complete recovery. As you can see, a packet which arrives out-of-order sets its own \textit{REORDER.TIME} (packet 3) and the \textit{REORDER.TIME} of all packets before it remaining unset (packet 2) with its arrival time. When packet 2 finally does arrive in this example the reordering interval will be calculated as seen in equation 3.2. Then acknowledgments for packet 3, 4, 5, 6 and 7 will be placed in the \textit{RELEASE.QUEUE} each with a timeout of 3/5.

<table>
<thead>
<tr>
<th>Arrival Time:</th>
<th>1</th>
<th>6</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>REORDER.TIME:</td>
<td>N/A</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Packet Num:</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Figure 3-3: Complete Recovery

\[ \text{timeout} = \frac{t_{reorder}(current) - t_{reorder}(recovery)}{current - recovery + 1} \]

\[ \text{timeout} = \frac{t_{reorder}(6) - t_{reorder}(2)}{6 - 2 + 1} \]

\[ \text{timeout} = \frac{5 - 2}{6 - 2 + 1} \]

\[ \text{timeout} = \frac{3}{5} \]

In our second example 3-4 there is a partial recovery followed by a full recovery. When packet 2 arrives at time 5, the interval is calculated to be 2/5. Acknowledgments for packets 3 and 4 are pushed onto the \textit{RELEASE.QUEUE} with a 2/5 timeout. This is the partial recovery. When packet 4 arrives we complete the recovery and acknowledgments for packet 5, 6 and 7 are pushed onto the queue with timeouts of 1/3.
3.3.3 Duplicate Release

Should $\text{WITHHELD_ACKS}$ exceed $\max(\text{STRIDES})$ a duplicate release would be triggered. In this case $\text{WITHHELD_ACKS}$ acknowledgments are generated, each requesting the next expected packet. The timeout for these packets is calculated as the average time between the arriving packets causing the release 3.2.

$$t_{dup} = \frac{t_{\text{reorder}}(\max) - t_{\text{reorder}}(\text{next})}{\max - \text{next}}$$  \hspace{1cm} (3.2)

3.4 Early Duplicate Release

Our receiver is designed to be indistinguishable from a standard receiver in the presence of in order delivery. In out-of-order delivery, we have made an effort to ensure that the only change in behavior is a small additional latency added to duplicate acknowledgments. This could cause congestion feedback information to arrive a little later than it would ordinarily, so in an attempt to mitigate this effect we introduce Early Duplicate Release.

The Reordering Resilient Receiver will immediately release the first two duplicate acknowledgments. Whenever the receiver does a duplicate release it will have two less acknowledgments to send and the sender will begin congestion control when it receives the first released acknowledgment. This will decrease the added latency by approximately two packet spacings.

This optimization depends on the fact the most senders set their duplicate acknowledgment threshold to 3. If it were higher than more duplicate acknowledgments could be sent early, if it was lower, than this optimization would render the rest of our system useless (actually harmful in the presence of reno Fast Retransmit).
3.5 Acknowledgment Smoothing

It is possible that a sequential release will occur while acknowledgments from a previous duplicate release are waiting in the release queue. In this case, the acknowledgment smoothing mechanism smooths the acknowledgments in the release queue in an attempt to provide the sender with a smooth ack stream mimicking in-order reception. Figure 3-5 shows an example of a packet trace which would cause a retransmission using a standard TCP.

<table>
<thead>
<tr>
<th>Arrival Time</th>
<th>1</th>
<th>5</th>
<th>2</th>
<th>6</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>REORDER_TIME</td>
<td>N/A</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Packet Num</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Figure 3-5: Ack Smoothing

Should the $\text{max(STRIDES)} = 3$ at $time = 5$ a duplicate release would send 4 duplicate acknowledgments for packet 2 to the $\text{RELEASE.QUEUE}$ with a timeout of 3/4. If the $\text{RELEASE.QUEUE}$ is empty one will be release immediately at $t = 5$ and another at $t = 5.75$, but at $t = 6$ reordering will be resolved and a sequential release will append 1 acknowledgment for packet 7 to the $\text{REORDER.QUEUE}$. The ack-smoothing algorithm will then smooth the acknowledgments in the $\text{RELEASE.QUEUE}$ changing the $[2, 2, 6]$ queue to $[3, 4, 6]$. If the sending TCP is implementing a standard fast-retransmission algorithm this would prevent the spurious retransmission and backoff.

With acknowledgment smoothing, even duplicate releases can be caught in time to prevent spurious retransmissions. In addition, it will smooth the acknowledgment stream, causing the sender’s window to slide more evenly, preventing packet bursts.
Chapter 4

Simulation

Our hope is that through simulation we will be able to examine the reordering resilient properties of TCP and some of its modifications. Simulation allows us to completely control experimental conditions and easily collect detailed, precise data.

The most straightforward application of a reordering TCP connection is per-packet multipath routing. Unfortunately, there are not any large, per-packet, multipath routers on the Internet today (partly because they cause TCP to perform poorly). Simulation allows us to explore TCP behavior under these conditions which do not currently exist.

4.1 Implementation

We chose the Network Simulator 2 (NS-2) [10] as our simulation platform. It provides the base for many of the protocols used to run our tests including Reno and SACK TCP implementations, routing and queueing modules, and a multipath framework.

To accommodate our tests a few NS modules were added and modified.

1. ReorderSink The module implementing our Reordering Resilient Receiver (RRR).
   It is built as an extension of TcpSink and can run as the sink of any TCP sender.

2. EifelTcpAgent The Eifel sender: implements the Eifel protocol as an extension of RenoTcpAgent. It requires the time-stamp option be enabled. Setting the
neweifel parameter to true will cause the EifelTcpAgent to perform New Eifel instead of standard Eifel.

3. Sack1TcpAgent The provided NS module implementing SACK and DSACK. It has been modified so that when enabled, DSACK responses will trigger the congestion control “undo” as specified in the DSACK RFC [6].

4. ReorderQueue The Reorder Queue, it implements a queue which accepts reordering commands such as: $queue_\_reorder RANDOM. This will randomly reorder the packets in the queue.

Additionally some modules have been altered for data collecting purposes.

An extension to the standard DropTail queue, the ReorderQueue, was developed in order to provide reordering in a simple network. The ReorderQueue allows us to carefully control how much reordering connections see. It supports the void reorder(ReorderType t) command which will reorder the packets in its queue using either reversal, randomness, or end-swapping.

The ReorderQueue also has the capacity to selectively reorder some flows but not others. With packets from many different sources going through the router, a reordering command will reorder only the packets from sources which have been explicitly enabled in the ReorderQueue. This way an Eifel TCP can be tested with three TCP sources as cross traffic. The Eifel connection experiences reordering, but the cross traffic TCP sources see regular in-order delivery.

In order for each reordering to have approximately the same relative affect on the flows, we add a couple requirements. Once given a reorder command the ReorderQueue waits until it has a minimum number of packets queued (we used seven). Also, no packet is ever reordered more than once.

4.2 Simulations

It has been well documented that the heterogeneity of traffic, network resources, congestion and applications on the Internet make it difficult to simulate effectively
Here we have not made an effort to simulate the entire Internet or even certain paths. Instead, we simulate a few controlled conditions in order to show how some of these schemes can be effectively used. Our simulations provide a set of data which we use to compare the protocols, but no attempt was made to simulate the exact conditions of any single Internet path.

### 4.2.1 TCP Configuration

All of the presented tests have the maximum TCP window set high enough (1000 packets) to ensure that the connections were limited only by the congestion window. It is possible that in reality a TCP’s maximum window would keep it from utilizing the available bandwidth, but in this case throughput is limited by hardware constraints, not protocol mechanisms. In our simulation we allow TCP’s congestion control to manipulate its own window by setting the maximum to a level which will never be reached.

Goodput is the amount of data successfully transmitted from sender to receiver. It does not include duplicate receptions or packet overhead.

All of our simulated TCP sources produce 1000 byte data payloads. The TCP’s start times are randomized over a 1 second interval. That way competing TCP sources won’t burst slow start packets at the same time.

### 4.2.2 Queue Configuration

Although many Internet paths are poorly configured, we choose to configure our networks with optimal queue sizes. For TCP connections, the optimal queue size is equal to the pipe size. Our queues are all set to this value, the bottleneck bandwidth times the round trip time, (see equation 4.1).

\[
Q_{size} = BW_{bn} \times ((2 \times Delay_{bn}) + (2 \times Delay_{node}))
\]  

(4.1)
4.3 Microtest

![Micro-Test Topology Diagram]

Figure 4-1: Micro-Test Topology

We set up a simple topology (see figure 4-1), and added our reordering element on the data path. Adding it on the return path would have no effect since that queue never holds more than one acknowledgment. Queue sizes were set as the pipe size, as in equation 4.1. The sources were run for 6000 seconds to determine their achieved goodput.

Our reordering element ReorderQueue will never reorder the same packet twice. This would leave the reordering stride a packet could experience unbounded. In addition, the queue will never execute a reordering event unless there are at least seven packets in the queue. These two constraints can lead to reorder requests to be made faster than they can be serviced. In this case there will be a backlog of requests.

In figure 4-2 it is clear that as the reordering becomes more frequent the throughput the TCP connection is able to obtain diminishes drastically. The flat part of the throughput graph is due to the reordering backlog. The flat portions of the TCP and the New Eifel graph represent the maximum amount of reordering these simulation parameters were able to produce.

It should be noted that New Eifel and DSACK both obtain far less throughput than the other reordering schemes, Reorder and Reno, at extremely high reordering rates. As the reordering becomes more moderate New Eifel and DSACK eventually
become the most effective protocols. They both use the slow-start mechanisms to recover from congestion control changes made spuriously because of congestion. The slow-start is not able to recover in time when reordering is too frequent. In moderate conditions, however, slow-start is able to more gradually slide the senders window, causing fewer packet bursts and a higher overall throughput.

Eifel and our RRR both perform very well even in the presence of a reordering event every 0.2 seconds. Eifel's ability to immediately recover to the previous congestion state (no slow start) allows it to perform at very high rates of reordering. RRR is able to trick the sender into performing almost no spurious retransmissions. This performance comes at a latency cost which prevents it from performing as well as DSACK and New Eifel when there is less reordering.

4.4 Multipath Test

Multipath routing is the most straightforward motivation for equipping TCP sources with reordering resilient capabilities. If TCP sources were able to perform in the
presence of persistent packet reordering, per-packet multipath routing (with minimal router state) could be a reality. To this end we’ve simulated TCP sources in a multipath environment. Using a modified “dumbbell” architecture (seen in figure 4-3). Here the multipath router chooses from the three best paths and shares traffic in a round robin fashion. In addition to the source being tested we include other sources to provide cross traffic.

In figure 4-4 you can see that the TCP source is never able to open its congestion window (\( \text{CWND} \)). The source mistakenly sees the reordering from the network as losses indicating congestion and responds by halving its window. These numerous spurious backoffs lead to an average usable throughput (goodput) of 28 Kb/sec.

In figure 4-5, we see that Eifel is able to open its congestion window to more than 60 packets. There is a less populated line of \( \text{CWND} \) values at half the persistent
value. These values are the backoffs which are later “undone” by the Eifel algorithm. Reverting to the old value allows the source to maintain a high sending rate. Eifel obtains $89Kb/sec$ usable throughput, more than three time TCP’s.

In figure 4-6 we see that New Eifel is not able to substantially open its window. The slow-start recovery is too slow and doesn’t return the source to its previous state fast enough. At the cost of recovery speed, the slow-start better distributes the sender’s packet burst. With the constant reordering multipath routing causes New Eifel to operate in the left region of figure 4-2. New Eifel obtains $59Kb/sec$ usable throughput (goodput).

In figure 4-7 we see that DSACK is able to open its window, but not as effectively as Eifel. DSACK suffers from the same problem that Eifel does: the benefit gained from smoothing the acknowledgement stream using the slow-start mechanism has less of an impact than the slowed recovery. DSACK is able to achieve $67Kb/sec$ usable
throughput (goodput).

Figure 4-8: Receiver Reorder Multipath Test

In figure 4-8 we see RRR's reaction to multipath routing. This is the only adaptive protocol which we observe. As the flow begins, reordering is sensed and the receiver's activity changes (it begins withholding acknowledgements). For the first few seconds of the transmission RRR acts very similar to an unmodified TCP, but as it senses the reordering in the network, RRR is able to achieve much higher bandwidth. In fact, the sender doesn't observe any reordering and produces a very smooth congestion window sawtooth. RRR obtains 81Kb/sec usable throughput.

4.4.1 Multipath Throughput

Our multipath throughput test consists of a single source/destination pair and a varying number of possible paths. Each additional path has the same bandwidth,
but increased latency. The one path test shows TCP performance with a \( d \) latency path and \( bw \) bottleneck bandwidth. The two path test adds a path with \( bw \) bandwidth and \( 1.5d \) latency. The third has \( 2d \) latency, and so on.

![Figure 4-9: Multipath Throughput Comparison](image)

Our multipath throughput test in which the bottleneck bandwidth is constant at 1Mb for all paths, but path latencies increase with the number of paths. 1 path = 20ms, 3 paths = 20ms, 30ms, 40ms. 5 paths = 20ms, 30ms, 40ms, 50ms, 60ms.

In figure 4-9 we see the results from our multipath throughput test.

For one path (no reordering) standard TCP, Eifel and RRR perform identically. DSACK, using SACK’s scoreboard optimizations is able to perform better. This result confirms that RRR doesn’t affect TCP’s performance at all for in-order delivery.

As the additional paths are added and routed through, TCP Reno’s performance dramatically decreases. It is able to cope with only one additional path fairly well (its 3 dupack threshold is able to compensate for most of the reordering), but more than that and the TCP source actually sees less goodput as more available throughput is added.

DSACK performance is slightly better. It is able to perform with three paths (and a maximum delay difference of 20ms), but with four or more paths it also fails to cope
with reordering and its performance drops. DSACK’s slow-start recovery scheme does not re-inflate the congestion window fast enough to cope with the constant reordering. As in figure 4-7 the window is never able to open.

New Eifel is able to detect the congestion slightly faster, but like DSACK it is slow to re-inflate the window and is not able to recover from reordering fast enough.

As more paths are introduced into the system, RRR takes advantage of the additional bandwidth. Its goodput is almost linearly increasing with the available bandwidth. The source is able to use all of the additional bandwidth regardless of delay variation.

The Eifel source also performs well in the multipath test. It is able to perform equally well to the standard TCP sources with only one path and as the number of paths increases it is able to use the additional bandwidth (almost as well as our RRR protocol).

4.5 Cross-traffic Test

![Cross Traffic Test Topology](image)

Figure 4-10: Cross Traffic Test Topology

In our cross-traffic test we compare our experimental TCP implementations operating in the presence of other sources. We use the topology in figure 4-10. Here the multipath topology is routing packets from a number of different sources. The com-
peting sources (running standard TCP) will interfere and congest the network. Using this simulation we can study how the modified TCP’s will affect standard TCP’s on the network.

Figure 4-11: Cross Traffic Test

Figure 4-11 shows, as we expect, that all the sources perform similarly in the one path (no reordering) case. As the number of paths (and frequency of reordering) increase, Reno TCP is not able to use the additional bandwidth. In fact, for five paths, the reno source isn’t able to produce any better goodput than with one path.

DSACK is able to cope with up to three paths, but more than that and the reordering becomes too frequent. The source does not have enough time to recover from a reordering event and it is not able to open its congestion window to fill the entire aggregate pipe.

RRR and Eifel are able to fill the aggregate pipe regardless of how many paths there are. In fact, they perform very similarly.
Chapter 5

Conclusion

Reordering on the Internet should not be considered pathological behavior. In order for the Internet to provide sources with the full available bandwidth, parallelism leading to packet reordering is required. Packet ordering should be the responsibility of end hosts.

In the presence of packet reordering, the current TCP algorithm performs inadequately. Of the proposed sender-based solutions, Eifel performed the best under most conditions. It requires very little additional resources: four TCP state variables and the TCP time stamp option.

Receiver Reordering performed as well if not better than Eifel in our simulations. It requires no TCP options, but requires some substantial state in the receiver (which is probably less heavily loaded in a traditional server/client architecture). Not depending on TCP options and only requiring modifications in the receiver will make this option easily deployable.

5.1 Future Work

If a receiver sends duplicate acknowledgments for a packet, it ignores the stride when that packet is eventually received. This stride value is ignored because it is assumed that the packet received was a retransmission. See section 3.2.2 for a more complete example. If another copy of this packet is received than it is likely that the orig-
inal reception was not a retransmission and its recovery stride should be added to \textit{STRIDES}. It is unclear how to precisely make the distinction between retransmission and original packet since it is possible that the packet was retransmitted more than once, but perhaps a combination of timers and packet logic could detect valid stride values from duplicate reception. This would give the receivers many more valid stride values allowing them to track reordering much faster.

Since most TCP sources have \texttt{DUP.THRESH} set to three packets there is no reason the first two duplicate acknowledgments need to be withheld. Releasing them right away would allow the connection to respond faster to a loss (though probably not significantly). This would remove some of the positive effects of acknowledgment smoothing, but future work could be done to determine which optimization is preferred under which conditions.

In [18] it is proposed to add a variable timeout to the \texttt{DUP.THRESH} at the sender. In addition to waiting for three duplicate acknowledgments before backing off and retransmitting, the source waits for a timer to expire. This gives the source an additional chance to resolve reordering. Our receiver could use a similar method and wait for a timeout as well as \( \text{max}(\text{STRIDES}) \) duplicate acknowledgments. As with the [18] scheme, this timeout would be dynamically chosen based on the amount of reordering previously observed.
Bibliography


[18] Ethan Blanton Ohio. On making tcp more robust to packet reordering.


